

UNIVERSITÀ DEGLI STUDI DI PARMA
FACOLTÀ DI INGEGNERIA
Corso di Laurea Specialistica in Ingegneria Informatica

PROGETTAZIONE DI UN'ARCHITETTURA
CONFIGURABILE PER LA LOCALIZZAZIONE
IN TEMPO REALE DI ROBOT MOBILI

Relatore:
Chiar.mo Prof. STEFANO CASELLI

Correlatori:
Dott. Ing. FRANCESCO MONICA
Dott. Ing. DARIO LODI RIZZINI

Tesi di laurea di:
LUCA DOMENICHINI

15 Marzo 2007

*A Palmo, Anna, Paolo e Elena.
Per avere creduto in me.*

Sono passati ormai sei mesi da quando iniziai a lavorare su questo progetto. Ricordo che, prima di iniziare i lavori, il pensiero della tesi quinquennale evocava in me una sensazione simile a quella che si proverebbe con un grosso macigno sulla schiena. Invece, le ore e le giornate spese in laboratorio a muovere il robot e a condividere i propri pensieri e le proprie idee con le persone che lo affollavano si sono rivelate sempre più spesso divertenti ed educative.

Colgo quindi l'occasione per ringraziare tutte le persone che in qualche modo mi hanno assistito durante i lavori.

Ringrazio il professor Caselli per la sua serietà impeccabile, che mi ha permesso di svolgere il lavoro in modo lineare e senza interruzioni alcune.

Un grazie a Francesco perchè ha dato un contributo fondamentale a questa tesi; probabilmente senza di lui non avrei ancora finito.

Un grazie anche a Dario perchè senza la sua mente che progetta, io sarei partito a testa bassa a scrivere codice a caso, nonchè per l'aiuto manuale nelle fasi finali del lavoro.

Ringrazio tutta la mia famiglia perchè mi ha sempre spronato a darcela tutta fin dalle elementari.

Alla mia Zanna perchè mi ha sempre fatto ridere quando ne avevo bisogno e perchè ha sopportato per tutto questo tempo il mio stress. Ringrazio in parallelo anche la Chiara perchè con la sua ansietà mi ha spinto ad accelerare quando io mi sarei accasciato.

Non può mancare, di certo, un vero grazie agli abitanti di viale Osacca (la Rossa compresa), perchè hanno movimentato il soggiorno. Grazie anche per non avermi mai fatto lavare i pavimenti, la mia camera, la cucina, il bagno.. insomma, tutto.

Grazie a tutti quelli che non cito per motivi di spazio.

Infine, un grazie speciale a Beppe Grillo e al blog.

“Un giorno le macchine riusciranno a risolvere tutti i problemi,
ma mai nessuna di esse potrà porne uno.”

Albert Einstein

Indice

| | | |
|----------|--|-----------|
| 1 | Introduzione | 1 |
| 1.1 | Localizzazione e robotica | 1 |
| 2 | Il problema della localizzazione | 3 |
| 2.1 | Introduzione alla localizzazione | 3 |
| 2.1.1 | La rappresentazione della mappa | 6 |
| 2.1.2 | Il modello di stato | 7 |
| 2.2 | Approcci alla localizzazione | 9 |
| 2.2.1 | Filtraggio Bayesiano | 9 |
| 2.2.2 | Filtri particellari | 12 |
| 2.2.3 | Real Time Particle Filter | 17 |
| 3 | Architettura del robot mobile | 24 |
| 3.1 | Caratteristiche hardware | 24 |
| 3.1.1 | Il sistema di elaborazione | 24 |
| 3.1.2 | Attuazione | 26 |
| 3.1.3 | Sensorialità | 27 |
| 3.2 | Software di controllo del robot | 28 |
| 3.2.1 | Il demone robotd | 28 |
| 3.2.2 | Il framework Smartsoft | 29 |
| 3.2.3 | Il framework YARA | 32 |
| 4 | Progettazione e realizzazione del sistema | 37 |
| 4.1 | Una visione di insieme | 37 |
| 4.2 | Progettazione del sistema base | 40 |

| | | |
|----------|---|------------|
| 4.2.1 | Sensorialità | 40 |
| 4.2.2 | Movimento | 44 |
| 4.2.3 | Operatività | 54 |
| 5 | Test del sistema base | 59 |
| 5.1 | Test di controllo | 59 |
| 5.1.1 | Reattività | 59 |
| 5.1.2 | Navigazione | 61 |
| 5.2 | Test delle prestazioni | 64 |
| 5.2.1 | Modulo sensoriale | 64 |
| 5.2.2 | Modulo di movimento | 66 |
| 5.2.3 | Moduli comportamentali | 68 |
| 6 | Integrazione del sistema di controllo con il localizzatore | 71 |
| 6.1 | Integrazione del sistema con LSOFT | 71 |
| 6.1.1 | Il progetto LSOFT | 71 |
| 6.1.2 | Il modulo Localizer | 77 |
| 6.1.3 | Verifica delle prestazioni | 81 |
| 6.1.4 | Il ClusterGridContainer | 86 |
| 6.2 | Progettazione dell'architettura di tracking | 90 |
| 6.2.1 | ARToolkit | 90 |
| 6.2.2 | Il NomadTracker | 91 |
| 6.3 | Risultati sperimentali | 94 |
| 6.3.1 | Tempi di esecuzione | 94 |
| 6.3.2 | Accuratezza nella localizzazione | 99 |
| 7 | Conclusioni | 103 |
| | Appendici | 106 |
| A | Realizzazione dei behaviours per il framework Smartsoft | 106 |
| B | Interfacciamento remoto del robot | 109 |
| | Bibliografia | 109 |

Capitolo 1

Introduzione

1.1 Localizzazione e robotica

La rapida evoluzione dei sistemi di calcolo ha consentito di progettare software sempre più complessi e articolati, in cui numerose entità autonome, variamente denominate processi, *task*, *thread* o agenti, collaborano al fine di raggiungere un obiettivo comune.

Anche la robotica sta rivolgendo la sua attenzione agli strumenti che consentono di progettare software in modo semplice e veloce, al fine di costruire architetture dall'alto grado di riusabilità ed efficienza. In particolare, è di crescente interesse il settore della robotica di servizio e della domotica, dove la presenza di robot mobili si affianca all'intervento umano; tuttavia, in alcuni contesti l'agente mobile deve necessariamente agire come un sistema indipendente, con un aiuto minimo o nullo da parte dell'uomo. È questo il caso dei sistemi robotici pensati per il supporto ai disabili, dove l'incapacità della persona di svolgere determinate azioni viene compensata dall'azione del robot.

Il primo requisito che un robot mobile di questo tipo deve presentare è la capacità di localizzarsi in un ambiente molto eterogeneo. In altre parole, l'ambiente in cui il robot opera fianco a fianco con le persone, non può in generale essere totalmente predisposto per facilitare al robot il compito di individuazione della propria posizione. A ciò si aggiunge il fatto che, nella maggior parte dei casi, l'ambiente è dinamico e soggetto a cambiamenti rapidi. Il robot, pertanto, si trova a gestire

una quantità elevata di informazioni sensoriali affette da forte incertezza. Per questo motivo, le tecniche che hanno ottenuto i maggiori successi nella localizzazione e nel controllo dei robot mobili sono quelle che considerano l'incertezza nel modello anziché eliminarla; si tratta, quindi, di applicare i metodi della teoria della probabilità ai modelli matematici che descrivono l'ambiente. Un approccio di questo tipo, notoriamente, richiede risorse di calcolo notevoli, soprattutto se l'ambiente di riferimento è molto vasto.

Il progetto sviluppato durante questa tesi ha avuto come obiettivo principale l'integrazione di un algoritmo di localizzazione all'interno di una architettura software per robot mobili. Il lavoro è stato suddiviso in due fasi sostanzialmente distinte. In primo luogo si è provveduto alla progettazione e al collaudo dell'architettura di controllo di base. Attraverso una serie di moduli comportamentali è stato stabilito il grado di reattività e di accuratezza nella navigazione assicurati dalla architettura di base. In secondo luogo, una volta disponibile l'architettura di controllo, è stata effettuata l'integrazione vera e propria del localizzatore e ne sono state valutate le prestazioni. Unitamente all'integrazione è stata effettuata una ottimizzazione delle prestazioni del software di localizzazione consentendo, in ultima analisi, la sua esecuzione in tempo reale.

La tesi è organizzata nel modo seguente.

Il capitolo 2 discute il problema della localizzazione e descrive gli algoritmi principali per la sua risoluzione.

Il capitolo 3 fornisce una descrizione dei componenti hardware e software presenti sul robot di riferimento e rilevanti ai fini della progettazione dell'architettura.

I capitoli 4 e 5 presentano il processo di progettazione dell'architettura di controllo del robot e l'insieme dei test effettuati per valutarne le prestazioni.

Il capitolo 6 presenta, invece, il lavoro svolto sul software di localizzazione e descrive il procedimento utilizzato al fine di realizzare l'integrazione con l'architettura di controllo. Viene anche presentato l'insieme dei risultati sperimentali ottenuti nei test svolti sul sistema completo.

Infine, nel capitolo 7 sono espone le considerazioni finali sul progetto realizzato e sugli sviluppi futuri.

Capitolo 2

Il problema della localizzazione

2.1 Introduzione alla localizzazione

Il problema della localizzazione nasce dall'esigenza di permettere una migliore interazione tra il robot e l'ambiente che lo circonda. Affinché un robot sia in grado di svolgere compiti anche complessi, occorre, in primo luogo, che sia in grado di navigare al suo interno. La capacità di navigazione di un robot è, però, interconnessa con la conoscenza della propria posizione rispetto a un sistema di riferimento solidale con l'ambiente.

L'obiettivo che un sistema di localizzazione si pone è la determinazione della posizione e dell'orientamento del robot. Il primo problema che si pone è stabilire quali siano le informazioni che si possono impiegare nel determinare la posizione del robot. È da notare che sull'ambiente non è possibile formulare ipotesi particolari, in quanto l'obiettivo che ci si pone è la possibilità di localizzazione all'interno di ambienti non strutturati¹. In realtà, per semplificare il problema, si suppone che il mondo in cui si va a interagire sia statico: ipotizzarne l'immutabilità non è comunque una ipotesi del tutto irragionevole, poichè la maggior parte delle caratteristiche di un ambiente rimangono immutate e nella maggioranza dei casi i cambiamenti sono più lenti del movimento del robot. Infine, si suppone che il mondo di interazione del robot sia chiuso, ovvero, è totalmente conosciuto e ben delimitato: possiamo

¹Un ambiente si definisce strutturato quando è stato fisicamente predisposto in modo da consentire l'esecuzione di task robotici.

quindi rappresentarlo tramite una mappa.

Per potersi localizzare, quindi, l'insieme delle informazioni che vengono catturate dall'ambiente vengono combinate al fine di ottenere la stima della posizione. Generalmente, le sorgenti di informazioni vengono classificate in due tipi in base alla loro provenienza [1]:

- *Sorgenti propriocettive*
- *Sorgenti eterocettive*

Le sorgenti *propriocettive* forniscono informazioni riguardanti lo stato interno del robot, ovvero, sulle variabili che è in grado di manipolare autonomamente. In questa categoria rientrano tutte quelle informazioni che il robot è in grado di rilevare a partire da se stesso, come le velocità dei motori, la rotazione di un asse rispetto a un altro e, nel caso di un manipolatore, le angolazioni dei giunti di controllo. In linea teorica, sarebbe possibile, con il solo ausilio di questi dati, ricostruire la posizione relativa del robot semplicemente integrando nel tempo le velocità di traslazione e rotazione: questa pratica prende il nome di *odometria*. Tuttavia, con la sola odometria si può effettuare solo una localizzazione *dead-reckoning*, che però è destinata all'insuccesso a causa dell'incertezza, della risoluzione limitata dei dati e del rumore.

Le sorgenti eterocettive completano il quadro mettendo il sistema in relazione con il contesto in cui opera il robot. Esse ci permettono di arrivare a determinare la posizione assoluta [2] cogliendo le caratteristiche dello spazio circostante, come la presenza di oggetti in un certo punto e ad una certa distanza, *pattern* e *feature* che emergono dall'analisi di immagini, oppure proprietà globali dell'ambiente. Se le informazioni prodotte dai sensori eterocettivi fossero molto accurate, sarebbe teoricamente possibile, disponendo di una *pre-conoscenza* della conformazione del mondo esterno, associare direttamente le osservazioni ad una posizione assoluta; in ambienti non strutturati un approccio di questo genere risulta tuttavia insufficiente.

Un metodo di localizzazione adatto ai nostri scopi, indipendentemente dal principio adottato, ha la necessità di integrare le due sorgenti di informazioni secondo lo schema rappresentato in figura 2.1.

Il funzionamento di un sistema di localizzazione è, generalmente, organizzato in due fasi ben distinte.

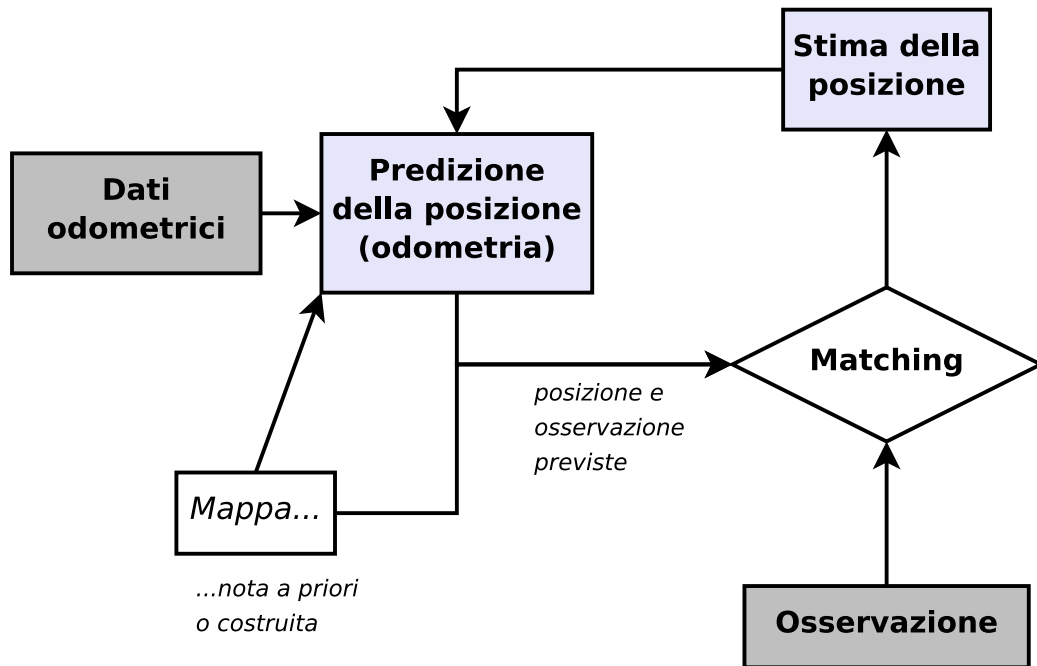


Figura 2.1: Schema della localizzazione.

- **Predizione:** è causata dal movimento del robot. Si basa sui dati odometrici prodotti e consiste in un avanzamento delle possibili stime di posizione sulla base dei dati rilevati. L'uso delle informazioni odometriche non è sufficiente a localizzarsi anche quando la posizione iniziale è nota; infatti, il progressivo accumulo degli errori commessi dall'odometria impedisce anche l'inseguimento di posizione.
- **Correzione:** Attraverso le osservazioni misurate dai sensori eterocettivi, il dato posizionale viene validato, ottenendo una stima più accurata della precedente. Questo è in genere valido se il procedimento algoritmico riesce a convergere alla soluzione, ma in alcuni casi potrebbe essere che l'elevata incertezza permane fino a quando non si presenta una osservazione altamente discriminante, come la curva di un corridoio, una rientranza di una porta o altro.

Queste caratteristiche mettono in evidenza che l'insieme delle osservazioni deve

essere confrontato con qualche rappresentazione certa dell'ambiente; questo può essere fatto con l'ausilio di una mappa.

Presupponendo di conoscere a priori la mappa dell'ambiente in cui il robot si trova, non siamo però a conoscenza della sua posizione iniziale. In letteratura ci si riferisce a questo tipo di problema parlando di *lost robot problem*; le sue maggiori difficoltà si manifestano in fase iniziale: occorre generare un numero di ipotesi sufficientemente ampio da coprire tutto il mondo accessibile. Poichè il sistema possiede una mappa, ipotizzando che si muova al suo interno, lo spazio che deve essere considerato è in qualche modo delimitato. Una volta che è stata ottenuta una stima corretta entro un ragionevole errore è sufficiente effettuare *position tracking*².

2.1.1 La rappresentazione della mappa

La mappa è la rappresentazione spaziale dell'ambiente che il robot mantiene al suo interno per poter svolgere i propri *task*. Nel procedimento di localizzazione la mappa è usata come *union trait* fra l'ipotesi di posizione predetta e le caratteristiche che si dovrebbero osservare in quel punto dello spazio. Astruendo la si può considerare come una tabella o funzione in cui la chiave di ricerca è lo spazio ed il valore restituito dipende dall'impiego per il quale è stata predisposta.

A seconda del *dominio* e della struttura dati che la implementa essa viene tradizionalmente classificata come *topologica* o *metrica* oppure, equivalentemente, come *discreta* o *continua* [1].

Nelle mappe topologiche lo spazio è rappresentato da un numero finito di punti nello spazio, legati da relazioni di prossimità; logicamente, lo spazio è rappresentato come un grafo i cui nodi sono le posizioni e i lati sono le connessioni tra posizioni vicine. Il processo di localizzazione si riduce alla associazione della posizione del robot ad uno dei nodi della mappa.

Le mappe metriche, invece, si propongono di rappresentare lo spazio in cui si trova il robot facendo uso di un sistema di riferimento cartesiano. Questo è ovviamente raggiunto tramite una rappresentazione spaziale della geometria del mondo di riferimento; l'approccio più tipico è la *occupancy grid map* [3, 4]. Il limite ine-

²Se la mappa è nota e così pure la posizione iniziale, sia pure con qualche incertezza, allora è sufficiente inseguire la posizione, ossia basta aggiornarla sulla base dell'odometria e validarla con i dati sensoriali.

liminabile di questa tipologia di mappa è inevitabilmente la sua risoluzione: più è alto il numero di celle rappresentate, più possiamo sperare che il risultato del procedimento di localizzazione sia accurato.

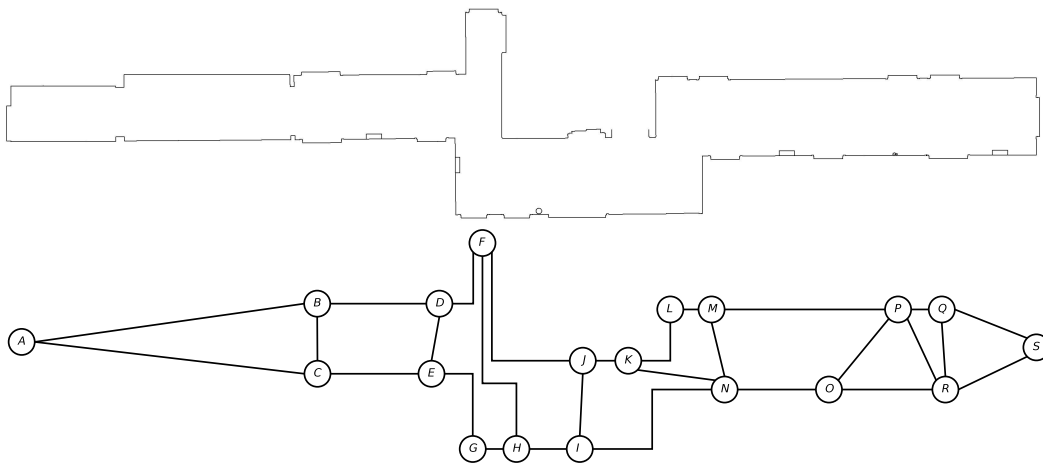


Figura 2.2: Mappe metrica e topologica della Palazzina 1 del Dipartimento di Ingegneria dell'Informazione.

Vista la loro semplicità, le *occupancy grid map* sono state per lungo tempo le mappe più utilizzate, specialmente per rappresentazioni bidimensionali dello spazio; al fine di stabilire la porzione del mondo nella quale il robot si può muovere senza incorrere in collisioni, le uniche operazioni da svolgere sulla mappa si riducono a una discriminazione tra le celle libere e quelle occupate.

2.1.2 Il modello di stato

Ciò che è emerso dai paragrafi precedenti ha sottolineato la necessità di descrivere il nostro sistema come un modello di stato. Quello che caratterizza un modello di questo tipo è sicuramente l'insieme delle variabili controllabili; possiamo pensare che le informazioni propriocettive siano i valori dei controlli, ovvero, gli ingressi che impostano il *set-point* dello stato del sistema. La misurazione dello stato, ovvero la posizione e l'orientamento del robot, corrisponde, inoltre, all'incognita che si desidera stimare. Infine, l'insieme delle osservazioni effettuate dai sensori eterocettivi corrisponde alle uscite del nostro sistema.

Come in ogni sistema reale, inoltre, è impossibile pensare che non sia presente alcuna sorgente di rumore; ad esempio, lo slittamento delle ruote sulla superficie di appoggio favorisce l'insorgere di una certa incertezza nelle misurazioni odometriche e nei tempi di esecuzione dei comandi ricevuti dagli attuatori. Non solo, anche le condizioni di illuminazione dell'ambiente giocano un ruolo importante nella stima delle osservazioni, così come i vari materiali di cui sono composte le pareti su cui si riflettono i raggi dei sensori; tutti questi elementi possono introdurre un errore nella stima della distanza effettiva, determinando una incertezza che nel modello di stato deve essere considerata.

Il metodo più comune per integrare l'incertezza al modello esatto, consiste nell'utilizzo dei metodi della teoria della probabilità e di strumenti come i processi stocastici.

Mettendo insieme queste premesse, il sistema può essere rappresentato tramite due equazioni, dove x_t , u_t e z_t sono le variabili che rappresentano rispettivamente lo stato, gli ingressi e le uscite al tempo t :

$$\begin{cases} x_t = f(x_{t-1}, u_{t-1}) + w_t \\ z_t = h(x_t, u_t) + v_t \end{cases} \quad (2.1)$$

È da notare che nelle equazioni 2.1 compaiono anche i termini dei processi del rumore w_t e v_t ; nella pratica, il rumore viene rappresentato con una funzione densità di probabilità o PDF. Senza perdere di generalità, il rumore viene in genere considerato additivo.

La prima equazione in 2.1 rappresenta il modello dinamico del sistema, in quanto esprime in termini degli ingressi l'evoluzione dello stato x_t . Generalmente, questo modello viene anche detto modello odometrico, in quanto questo calcolo viene effettuato considerando lo spostamento rilevato dal robot tramite l'odometria. La seconda relazione è invece detta modello sensoriale; è da notare che la dipendenza da u_t dalla z_t in genere non sussiste, ma per completezza è stata indicata.

Nella teoria dei sistemi il localizzatore è un particolare esempio di *osservatore dello stato*; noti gli ingressi e le uscite deve riuscire a stimare lo stato. Le equazioni 2.1 riportano formalmente le fasi di predizione e correzione come già visto al paragrafo 2.1. L'equazione del modello dinamico costituisce la predizione del movimento del sistema, che viene corretta tramite l'osservazione e l'applicazione del

modello sensoriale.

2.2 Approcci alla localizzazione

2.2.1 Filtraggio Bayesiano

La discussione al paragrafo 2.1.2 ha messo in evidenza l'opportunità di impiegare modelli in grado di rappresentare l'incertezza dei dati disponibili e delle inferenze ricorrendo alla teoria della probabilità. Lo scopo è quello di riuscire a stimare lo stato del sistema usando le tre fonti di informazioni disponibili: i controlli, le osservazioni e la mappa. Formalmente, la conoscenza al tempo t è composta dalla successione degli ingressi u_0, \dots, u_{t-1} , dalle osservazioni z_0, \dots, z_t e dalla mappa m , che supponiamo nota; a partire da questi dati si richiede di ricavare lo stato x_t . La notazione u_t , z_t e x_t verrà impiegata in riferimento sia ai processi stocastici sia alle loro uscite sperimentali.

I filtri bayesiani sono una classe di algoritmi per la stima probabilistica dello stato di un sistema. Applicando la formula di Bayes è possibile formulare l'algoritmo sotto forma di stima ricorsiva della PDF dello spazio degli stati condizionata ai dati acquisiti fino a quel particolare istante. Conseguentemente, lo stato x_t non sarà rappresentato da un semplice valore numerico, ma dalla sua distribuzione; per ottenere un valore sintetico occorrerà introdurre un criterio ulteriore, ad esempio la massima verosimiglianza (*maximum likelihood*).

La PDF che rappresenta la posizione fisica del robot è spesso denominata *belief*, in quanto rappresenta la credenza di trovarsi in una certa posizione. Al fine di determinare con precisione il *belief* per lo stato corrente x_t , vanno considerate tutte le informazioni sulla storia passata inerenti al movimento e alle osservazioni del robot. Pertanto si può scrivere che

$$Bel(x_t) = p(x_t | z_t, u_{t-1}, z_{t-1}, \dots, u_0, z_0, m) \quad (2.2)$$

La densità di probabilità condizionata 2.2 è generalmente denominata "a posteriori" poichè rappresenta una stima che dipende da tutte le informazioni sensoriali precedenti. Il termine a priori è, in questo contesto, riferito alla densità di proba-

bilità non condizionata dall'ultima osservazione z_t , ossia alla densità predetta sulla base del controllo u_t .

Se ipotizziamo che il sistema dinamico è markoviano, cioè che le osservazioni z_t e le misure di controllo u_t sono indipendenti dalle misurazioni passate, ma dipendono solo dallo stato x_t e dalla mappa m , allora la funzione di densità di probabilità a posteriori può essere aggiornata semplicemente applicando due semplici regole sequenziali:

- ogni volta che viene ricevuto un nuovo campione di controllo u_{t-1} lo stato del sistema viene predetto secondo la seguente equazione:

$$Bel^-(x_t) = \int p(x_t|x_{t-1}, u_{t-1})Bel(x_{t-1})dx_{t-1} \quad (2.3)$$

- ogni volta che viene ricevuta una misurazione sensoriale, lo stato viene corretto secondo la seguente equazione:

$$Bel(x_t) = \alpha p(z_t|x_t)Bel^-(x_t) \quad (2.4)$$

dove α è un coefficiente di normalizzazione.

Il termine $p(x_t|x_{t-1}, u_{t-1})$ descrive il modello dinamico, ovvero, come il sistema evolve a fronte del controllo u_{t-1} . Il termine $p(z_t|x_t)$ è, invece, il modello sensoriale che descrive la probabilità di osservare il dato z_t supponendo di trovarci allo stato x_t .

Nell'equazione 2.3 il termine Bel^- è denominato credenza predetta (*predictive belief*), in quanto la stima dello stato non tiene conto delle osservazioni effettuate ma solo della sua probabile evoluzione dinamica.

I filtri bayesiani non vogliono fornire un metodo concreto per realizzare una applicazione di localizzazione, ma forniscono solamente la struttura per poter calcolare l'evoluzione del *belief*; in ultima approssimazione, il *belief* sarà sempre più equivalente allo stato x_t allo scorrere del tempo cioè per $t \rightarrow \infty$, $Bel(x_t) = x_t$. Per implementare correttamente un filtro bayesiano è sufficiente descrivere il proprio modello sensoriale $p(z_t|x_t)$, il modello dinamico $p(x_t|x_{t-1}, u_{t-1})$. Inoltre, è

necessario dare una rappresentazione concreta al modello che descrive il *belief* $Bel(x_t)$.

Classificazione dei filtri bayesiani

In questo paragrafo vengono brevemente discussi i diversi approcci al filtraggio bayesiano senza pretendere di esaurire l'argomento. Per una più approfondita discussione si rimanda a [5, 6, 7]. Il nostro interesse è, infatti, principalmente rivolto al filtro particellare, ma per comprenderne la specificità occorre inquadrarlo in una più vasta classe di algoritmi.

La figura 2.3 mostra una prima classificazione in metodi che rappresentano la densità di probabilità con una funzione continua e in metodi che la rappresentano in forma discreta. È opportuno ribadire che la continuità non riguarda il dominio di appartenenza dello stato, ma la rappresentazione della densità di probabilità, che è data da una funzione.

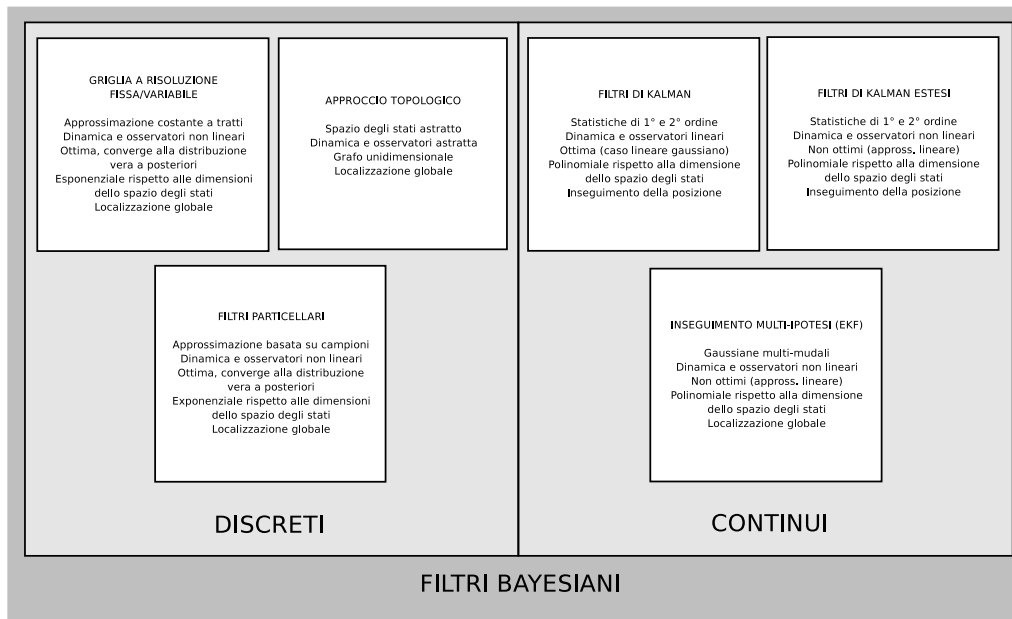


Figura 2.3: Classificazione dei filtri bayesiani.

I tre metodi continui raffigurati sono filtri gaussiani derivati dal filtro di Kalman. Essi sono ottimi per i sistemi lineari, approssimano il *belief* attraverso la sua media e la sua covarianza. Siccome i sistemi a cui facciamo riferimento non sono lineari,

si ricorre ad una approssimazione lineare dello stato: in questo caso si parla di filtro di Kalman esteso (EKF).

Nel caso dei metodi che operano nel discreto, gli approcci di tipo topologico sono basati sulla rappresentazione simbolica dell'ambiente attraverso un grafo strutturato. Il vantaggio di questo approccio risiede nella sua efficienza e nel fatto che è in grado di rappresentare qualunque distribuzione sullo spazio degli stati. Tuttavia, la stima posizionale fornita soffre, generalmente, di molta incertezza, senza contare che il sistema richiede la presenza di una serie di *marker* associati a posizioni globali.

Un altro tipo di approccio suddivide lo spazio disponibile in griglie di dimensioni costanti (oppure variabili), utilizzando una rappresentazione metrica dell'ambiente (vedi 2.1.1). Questo metodo si è rivelato essere molto più robusto e preciso di quello topologico, tuttavia, la rappresentazione dell'ambiente tramite questo set di griglie contribuisce in maniera significativa alla richiesta di risorse computazionali che, nel caso lo spazio degli stati sia di dimensioni elevate, può assumere un andamento esponenziale.

Infine, il metodo che ha riscosso maggior successo nella risoluzione del problema della localizzazione è il *particle filter* (PF), che costituisce l'argomento del prossimo paragrafo.

2.2.2 Filtri particellari

I filtri particellari prendono il loro nome dalla modalità con cui la PDF dello stato viene rappresentata: si utilizza, infatti, un insieme di campioni o particelle, ciascuno dei quali caratterizzato da un set di coordinate nello spazio degli stati, rappresenta una ipotesi di localizzazione. Il vantaggio principale di questo approccio è la possibilità di rappresentare qualunque tipo di densità di probabilità discretizzandola all'elemento base della particella. Confrontando questo metodo con quello basato su griglie, i filtri particellari risultano molto più efficienti perchè impiegano le risorse di calcolo per l'elaborazione dei campioni, i quali tendono a concentrarsi nelle regioni dello spazio degli stati dove è più alta la probabilità che sia presente il robot. Il corretto funzionamento e l'efficacia dei PF dipendono, come vedremo, dal numero dei campioni.

Formalizzando quanto detto, i filtri particellari rappresentano la funzione $Bel(x_t)$ con un set S_t di N campioni pesati, distribuiti secondo la funzione *belief*:

$$S_t = \left\{ \langle x_t^{(i)}, w_t^{(i)} \rangle \mid i = 1, \dots, N \right\}$$

La convenzione adottata in precedenza è ancora valida, quindi, $x_t^{(i)}$ rappresenta lo stato mentre $w_t^{(i)}$ è l'*importance weight*, un numero non negativo che rappresenta la verosimiglianza del campione. La metodologia che è all'origine del PF, è l'*importance sampling* [8], che prevede di valutare la distribuzione di una v.a. Y , funzione di un'altra v.a. X , campionando una distribuzione detta *proposal* $\pi(\cdot)$, diversa dalla distribuzione di X , $f_x(\cdot)$. L'*importance weight* è definito formalmente come rapporto tra $f_x(x)/\pi_x(x)$.

Il *sequential importance sampling* (SIS) è l'adattamento dell'*importance sampling* per consentire l'applicazione al problema di filtraggio.

Detta $f_t(\cdot)$ una funzione della variabile aleatoria x_t è possibile calcolare la funzione $Bel(x_t)$ come la media di tutte le possibili evoluzioni del modello di Markov:

$$I(f_t) = \int_{D^t} f_t(x_t) p(x_{0:t}|z_{0:t}) dx_t \quad (2.5)$$

Definendo la funzione di importanza come $\pi(x_{0:t}|z_{0:t})$, otteniamo che

$$I(f_t) = \int_{D^t} f_t(x_t) \frac{p(x_{0:t}|z_{0:t}, u_{1:t})}{\pi(x_{0:t}|z_{0:t}, u_{1:t})} \pi(x_{0:t}|z_{0:t}, u_{1:t}) dx_t \quad (2.6)$$

In realtà, la funzione di importanza è approssimata da una somma di funzioni di Dirac poste in corrispondenza dei vari $x_{0:t}^{(i)}$, quindi, possiamo ottenere una approssimazione della 2.6 come

$$\widehat{I}_N(f_t) = \sum_{i=0}^N f_t(x_t^{(i)}) w_t^{(i)} \quad (2.7)$$

$$w_t^{(i)} = \frac{p(x_{0:t}^{(i)}|z_{0:t}, u_{1:t})}{\pi(x_{0:t}^{(i)}|z_{0:t}, u_{1:t})} \quad (2.8)$$

Nell'ipotesi di poter decomporre anche $\pi(\cdot)$ in modo ricorsivo si ha un diverso

modo di calcolare il peso:

$$w_t^{(i)} = w_{t-1}^{(i)} \frac{p(z_t | x_t^{(i)}) p(x_t^{(i)} | x_{t-1}^{(i)}, u_{1:t})}{\pi(x_t^{(i)} | x_{t-1}^{(i)}, z_{0:t}, u_{1:t})} \quad (2.9)$$

L'algoritmo così ottenuto è detto anche *sequential importance sampling* (SIS). Applicandolo alla localizzazione si dovrà aggiungere anche il controllo u_t , quindi il secondo termine a numeratore diventa $p(x_t^{(i)} | x_{t-1}^{(i)}, u_t)$. In realtà vengono introdotte anche alcune semplificazioni per rendere lineare la sua implementazione.

La prima approssimazione avviene nella fase di predizione: per ciascun campione $x_{t-1}^{(i)}$ ne viene generato uno aggiornato $x_t^{(i)}$ campionando la funzione di distribuzione $p(x_t | x_{t-1}^{(i)}, u_{t-1})$, ovvero, applicando al modello matematico una incertezza riprodotta dal calcolatore come un numero pseudo casuale secondo la distribuzione del rumore scelto.

Particolare importanza assume la fase di ricampionamento. Questo procedimento viene utilizzato per ottenere un nuovo insieme di campioni che rispecchi l'importanza dei campioni. Durante il ricampionamento il peso di ogni campione viene aggiornato, tuttavia, non si tratta di una procedura deterministica poichè avviene tramite il campionamento pseudo-casuale della distribuzione formata dalle coppie $[x_t^{(i)}, w_t^{(i)}]$, in modo tale che la probabilità che la particella i -esima sia ricampionata sia proporzionale al peso $w_t^{(i)}$. L'algoritmo 1 è chiamato *Residual Systematic Resampling* (RSR) [9] e mostra uno dei possibili modi per fare quanto detto.

Algorithm 1 Algoritmo di ricampionamento

Require: N numero di campioni in ingresso, M numero di campioni in uscita,

un array dei pesi $\{w_n\}_1^N$

- 1: Genera un numero casuale $\Delta U^{(0)} \sim U[0, \frac{1}{M}]$
 - 2: **for** $m = 1, \dots, N$ **do**
 - 3: $i^{(m)} = \lfloor (w_n^{(m)} - \Delta U^{(m-1)}) \cdot M \rfloor + 1$
 - 4: $\Delta U^{(m)} = \Delta U^{(m-1)} + \frac{i^{(m)}}{M} - w_n^{(m)}$
 - 5: Copia $i^{(m)}$ copie del campione attuale nella distribuzione
 - 6: **end for**
-

Risulta possibile, a questo punto, presentare l'algoritmo di base del filtro partecellare (algoritmo 2).

Algorithm 2 Filtro particellare

Require: $S_{t-1} = \{(x_{t-1}^{(i)}, w_{t-1}^{(i)}) | i = 1, \dots, n\}$ che rappresenta $Bel(x_{t-1})$, misura del controllo u_{t-1} , osservazione z_t

- 1: $S_t := \emptyset, \alpha := 0$
- 2: **for** $i := 0, \dots, N$ **do**
- 3: Campiona un indice j dalla distribuzione discreta data dai pesi di S_{t-1}
- 4: Campiona $x_t^{(i)}$ da $p(x_t | x_{t-1}, u_{t-1})$ dove il condizionamento è dato dal campione $x_{t-1}^{(j)}$ e da u_{t-1}
- 5: $w_t^{(i)} := p(z_t | x_t^{(i)})$ // Calcola l'importance weight
- 6: $\alpha := \alpha + w_t^{(i)}$ // Somma dei pesi totali
- 7: $S_t := S_t \cup \{(x_t^{(i)}, w_t^{(i)})\}$ // Inserisce il campione in S_t
- 8: **end for**
- 9: **for** $i := 1, \dots, n$ **do**
- 10: $w_t^{(i)} := \frac{w_t^{(i)}}{\alpha}$ // Normalizza il peso dei campioni
- 11: **end for**
- 12: **return** S_t

A ogni iterazione, l'algoritmo riceve un set di campioni S_{t-1} che rappresentano il precedente *belief* del robot, una stima del movimento u_{t-1} e una osservazione z_t . Le righe dalla 2 alla 8 illustrano l'elaborazione di N campioni in una iterazione del localizzatore. Alla riga 3 si determina quale campione estrarre dal set S_t , ovvero, viene estratto un indice j con probabilità proporzionale al peso di ciascun campione nella distribuzione. Successivamente lo stato di tale campione viene aggiornato attraverso il controllo u_{t-1} , portando a predire $x_t^{(i)}$ (riga 4). Questo è fatto ricampionando dalla funzione di densità di probabilità $p(x_t | x_{t-1}, u_{t-1})$ che rappresenta la dinamica del sistema. Infine, viene ricalcolato il peso del campione estratto e la densità di probabilità viene normalizzata a 1 (da riga 5 a 10).

Applicando questo modello al problema della localizzazione di robot mobili, la posizione del robot è rappresentata nello spazio cartesiano bidimensionale, mentre le misurazioni sensoriali z_t possono includere misurazioni di distanze tramite scanner laser o sonar, ma anche immagini digitalizzate acquisite da telecamera. L'informazione di controllo u_t riguardante il movimento del robot viene solitamente ricostruita dal calcolo odometrico presente nel software del robot.

Aspetti implementativi del filtro particellare

Un sistema che vuole fare uso di filtri particellari deve necessariamente prevedere che il filtro paga la propria flessibilità in termini di complessità computazionale. Questo carico viene generato perchè nel filtro si ha a che fare con un numero elevato di campioni; ogni singola fase dell'algoritmo (predizione, correzione, ricampionamento) deve essere ripetuta per ogni campione.

Entrando nello specifico, appare evidente che la complessità dell'algoritmo dipende in modo diretto dal numero di campioni impiegati per rappresentare lo stato. In particolare, la fase più problematica coinvolge l'applicazione del modello sensoriale ai campioni. Uno dei più diffusi modelli è il *beam model* [7], pensato per essere applicato ai sensori di prossimità. In questo caso la fase di correzione viene eseguita tramite un confronto, fatto per ogni campione, tra i dati osservati e le misure prelevate direttamente dalla mappa a disposizione. Questo confronto è basato su procedure di *ray-tracing*; sostanzialmente, il metodo consiste nel rilevare la cella della mappa corrispondente all'ostacolo rilevato. La figura 2.4 mostra come può essere raggiunto.

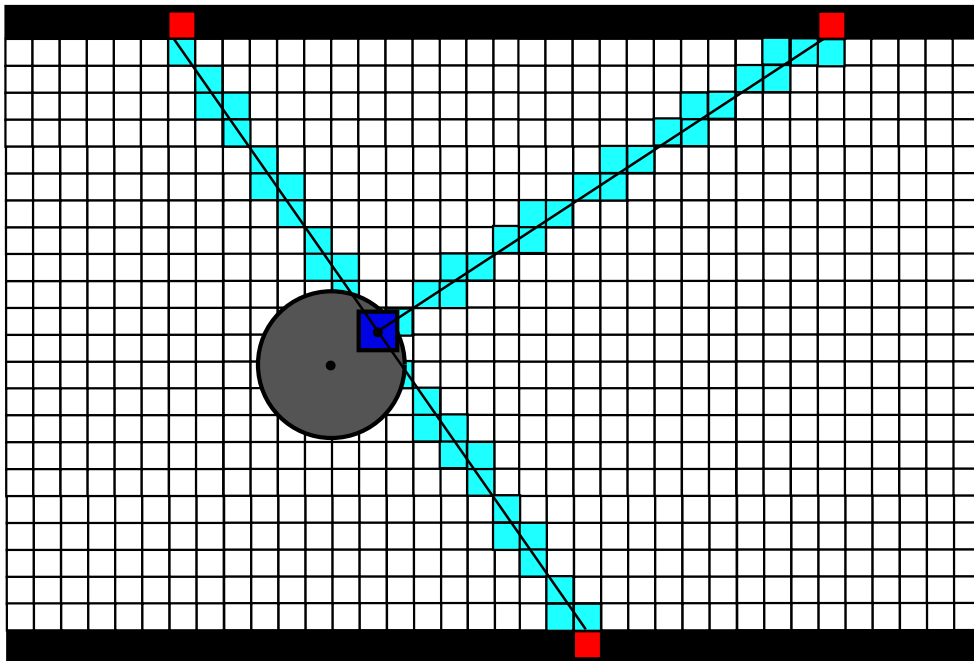


Figura 2.4: Esempio di *ray-tracing* tracciato da un robot che naviga nel corridoio.

Queste procedure richiedono, infatti, operazioni geometriche sulla figura che devono necessariamente iterare fino all'individuazione dell'ostacolo più vicino; senza una conoscenza a priori della posizione in cui si trova il robot, non è possibile avere una stima sulla complessità di tale algoritmo.

Infine, per produrre una stima concreta, osservabile dall'esterno, dobbiamo poter fornire un dato sintetico che esprima l'interezza della distribuzione. Non ci si può, semplicemente, accontentare del campione a peso maggiore, perchè questo potrebbe essere solo una ipotesi isolata e priva di valore. La soluzione, consiste nell'applicazione di alcune tecniche di *clustering*, al fine di ottenere l'aggregazione di tutti i campioni che vengono considerati vicini.

In entrambe le situazioni si deduce che il numero di campioni di cui è formata la distribuzione è di fondamentale importanza al fine della buona riuscita dell'algoritmo di localizzazione. Bisogna notare, però, che se pochi campioni sono di beneficio per la velocità di esecuzione, difficilmente riusciranno a garantire la convergenza del filtro. Viceversa, un numero maggiore di campioni riesce a garantire una convergenza migliore, ma a un costo computazionale da tenere in considerazione.

2.2.3 Real Time Particle Filter

Il problema dell'esecuzione in tempo reale

Il carico computazionale che un modulo di localizzazione introduce all'interno di un'architettura di controllo complessa deve essere certamente tenuto in considerazione. Innanzi tutto occorre assicurarsi che i diversi *task* robotici possono essere completati correttamente e nei tempi loro assegnati, incluso il *task* di localizzazione. In secondo luogo, la perdita di dati sensoriali dovuta ai limiti della capacità di elaborazione ha un impatto negativo sulle prestazioni del localizzatore: l'acquisizione dei dati dei sensori pone quindi dei vincoli temporali.

Si tratta, di fatto, di riuscire a rispettare dei vincoli temporali imposti dalla complessità algoritmica del filtro particellare. Tuttavia, il problema non riguarda la realizzazione di uno scheduler di sistema per consentire l'esecuzione temporizzata delle attività di localizzazione; il problema è semmai come sia possibile migliorare le prestazioni del filtro particellare in modo che sia in grado di sostenere vincoli temporali anche stretti, operando a livello algoritmico.

Supponiamo che il tempo di arrivo delle misurazioni sensoriali z_t sia T_z , mentre il tempo per il calcolo della correzione della stima per gli N campioni sia T . In questo caso, per avere un sistema in grado di funzionare in tempo reale senza alcuna perdita di dati, è necessario che l'algoritmo venga eseguito $k = \left\lceil \frac{T}{T_z} \right\rceil$ volte in un periodo T . La figura 2.5 mostra alcuni metodi per cercare di utilizzare tutte le misurazioni sensoriali in un'unica iterazione [6].

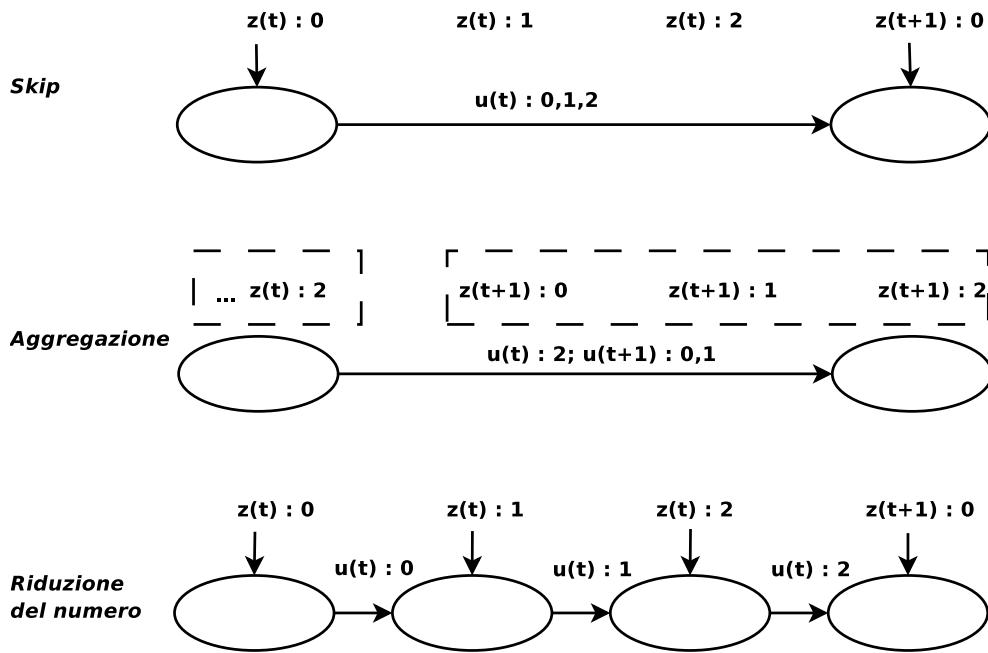


Figura 2.5: Strategie per compensare le limitazioni sulle risorse di calcolo.

Il primo approccio è quello più semplice, ma anche il più impreciso, in quanto, le informazioni che arrivano tra una finestra di esecuzione e la successiva vengono scartate. Il secondo approccio consiste, invece, in una aggregazione delle informazioni sensoriali che sono arrivate fino a quell'istante in un unico dato. In un caso ideale questa soluzione sarebbe la migliore, tuttavia, l'aggregazione di più informazioni sensoriali distanti sia spazialmente che temporalmente, risulta piuttosto complicata, senza contare che introdurrebbe ulteriori carichi computazionali all'elaboratore. Infine, l'aggregazione non sempre può essere fatta in modo ottimo. L'ultimo approccio in figura 2.5, invece che ridurre il numero dei dati sensoriali acquisiti, riduce il numero dei campioni su cui effettuare l'elaborazione; non appena giunge una nuova informazione sensoriale, il sistema arresta l'elaborazione dei campioni attua-

li e ricomincia con nuovi campioni sulla base della nuova osservazione. Purtroppo, questo metodo può portare a un divergenza del filtro, a causa dell'insufficienza del numero dei campioni considerati.

Il partizionamento

In questo scenario si inquadrano i *Real Time Particle Filters*. L'idea centrale degli *RTPF* è di includere nella computazione della posizione tutti i rilevamenti sensoriali distribuendo gli N campioni del set tra le osservazioni ricevute. Questo risultato viene raggiunto partizionando il set dei campioni S_t in un certo numero di partizioni, ciascuna associata a una delle osservazioni che si susseguono all'interno di una finestra temporale di aggiornamento; supponendo di ricevere tre aggiornamenti sensoriali in una finestra temporale, avremo al più $k = \lceil \frac{N}{3} \rceil$ campioni in ciascuna delle 3 partizioni. La figura 2.6 presenta la modalità con cui vengono ricampionati i campioni.

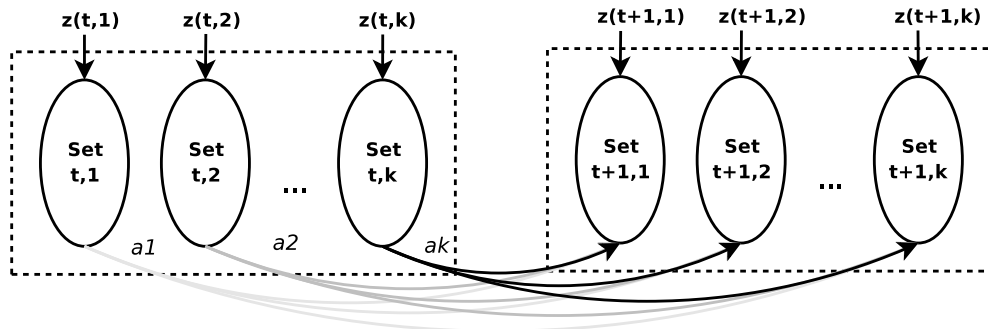


Figura 2.6: Schema del ricampionamento del RTPF.

Come si può vedere, invece di utilizzare un solo set di campioni S_t , si dispone di k partizioni S_{t_1}, \dots, S_{t_k} , associate agli istanti temporali t_1, \dots, t_k . Al fine di determinare il peso dei campioni, questo viene ricalcolato, al termine della fase di correzione, solo sulla base della partizione di appartenenza; in questo modo, il costo computazionale di un RTPF è equivalente a quello di un normale filtro che mantiene N/k particelle a ogni iterazione. Se dal punto di vista dei tempi di calcolo sono equivalenti, cambia la rappresentazione dello stato x_t . Infatti, nel RTPF, la distribuzione rappresenta lo stato al tempo t_k , mentre $Bel(x_{t_k})$ è definito dalla combinazione pesata di tutte le partizioni appartenenti a quella finestra temporale;

in questo modo, vengono di fatto mantenute k traiettorie che vengono pesate al fine di ottenere la stima migliore.

Cerchiamo di capire in che misura il RTPF si avvicina alla precisione di un ipotetico filtro in grado di funzionare in tempo reale per tutti i campioni N . Il termine di paragone è il *belief ottimo* della finestra temporale t , ossia quella distribuzione che si otterrebbe applicando iterativamente un filtro bayesiano non soggetto a vincoli sulla capacità computazionale del sistema. Lo si può calcolare espandendo la formula ricorsiva dal tempo t_k sino a t_1 .

$$Bel_{opt}(x_{t_k}) \propto \int \cdots \int \prod_{i=1}^k p(z_{t_i}|x_{t_i}) \cdot p(x_{t_i}|x_{t_{i-1}}, u_{t_{i-1}}) \cdot Bel(x_{t_0}) dx_{t_0} \cdots dx_{t_{k-1}} \quad (2.10)$$

La 2.10 è stata ottenuta integrando ricorsivamente le densità di probabilità del modello odometrico e sensoriale su tutto l'insieme delle possibili traiettorie dal punto x_{t_0} al tempo t_0 .

Eseguiamo lo stesso calcolo nel caso del RTPF; in questo caso si calcola l'integrale per ogni partizione, partendo sempre dal punto x_{t_0} . Tuttavia, ogni partizione i -esima conosce solamente l'osservazione z_{t_i} , cioè quella pervenuta all'istante t_i . In questo modo, l'inferenza sullo stato rappresentata da $Bel_i(x_{t_k})$ è il risultato della sola fase di predizione eccetto l'unico *feedback* del dato sensoriale.

$$Bel_i(x_{t_k}) \propto \int \cdots \int p(z_{t_i}|x_{t_i}) \cdot \prod_{j=1}^k p(x_{t_j}|x_{t_{j-1}}, u_{t_{j-1}}) \cdot Bel(x_{t_0}) dx_{t_0} \cdots dx_{t_{k-1}} \quad (2.11)$$

Combinando insieme i diversi $Bel_i(x_{t_k})$ si ottiene allora una approssimazione di $Bel_{opt}(x_{t_k})$ che chiameremo $Bel_{mix}(x_{t_k}|\alpha)$:

$$Bel_{mix}(x_{t_k}|\alpha) \propto \sum_{i=1}^k \alpha_i Bel_i(x_{t_k}) \quad (2.12)$$

dove $\alpha_i \geq 0$ rappresentano i pesi associati a ciascuna partizione. Infine, deve essere verificato che $\sum_i \alpha_i = 1$.

Ottimizzazione dei coefficienti di mix

Cerchiamo, ora di determinare il vettore ottimo dei coefficienti α_i . Il singolo coefficiente rappresenta il peso della sua traiettoria; tutti insieme i coefficienti determinano l'andamento della traiettoria $Bel_{opt}(x_t)$. Per raggiungere l'ottimizzazione, l'idea è di impostare i coefficienti in modo da minimizzare l'errore di approssimazione introdotto dal partizionamento. Equivalentemente, possiamo minimizzare il vettore α che minimizza la divergenza KL tra la Bel_{mix} e la Bel_{opt} [5, 10]. La quantità che deve essere minimizzata è la seguente:

$$J(\alpha) = \int Bel_{mix}(x_{t_k}|\alpha) \log \frac{Bel_{mix}(x_{t_k}|\alpha)}{Bel_{opt}(x_{t_k})} dx_{t_k} \quad (2.13)$$

I metodi maggiormente utilizzati per risolvere problemi di questo tipo sono i metodi di discesa del gradiente. Calcoliamo le derivate parziali della 2.13 rispetto a α :

$$\frac{\partial J}{\partial \alpha_i} = 1 + \int Bel_i(x_{t_k}) \log \frac{Bel_{mix}(x_{t_k}|\alpha)}{Bel_{opt}(x_{t_k})} dx_{t_k} \quad (2.14)$$

Tuttavia l'impiego di questa formula è sconsigliato, vista la presenza del termine $Bel_{opt}(x_{t_k})$ che non vogliamo calcolare. Ciò che si cerca di ottenere dal RTPF è una limitazione del costo computazionale di $Bel_{opt}(x_{t_k})$, perciò viene utilizzata una approssimazione del gradiente. La procedura di calcolo usata in [6] vuole che i *belief* siano rappresentati da un valore numerico da sostituire all'interno della 2.14. Questi valori sono calcolati nel modo seguente:

$$Bel_i = \sum_{m=1}^{N_p} w_{im} \quad (2.15)$$

$$Bel_{mix} = \sum_{p=1}^k \alpha_p Bel_p \quad (2.16)$$

$$Bel_{opt} = \sum_{m=1}^{N_p} \prod_{i=1}^k w_{im} \quad (2.17)$$

Indichiamo con w_{im} il peso di una particella m -esima appartenente alla partizione i -esima, mentre con N_p il numero di campioni per ogni partizione. Siccome ogni partizione è rappresentata dalla somma dei pesi delle particelle che ne fanno parte,

risulta che il gradiente privilegia per lo più gli insiemi di campioni che rappresentano ipotesi più probabili, quindi, di maggior peso.

Sono ora disponibili tutti i dati necessari per il calcolo del gradiente approssimato:

$$\frac{\partial J}{\partial \alpha_i} \simeq 1 + Bel_i \log \frac{\sum_{i=1}^k \alpha_i Bel_i}{Bel_{opt}} \quad (2.18)$$

Muovendosi nello spazio degli α_i , nella direzione del gradiente con passo di lunghezza fissata e per un numero di iterazioni predefinito, si dovrebbe arrivare a stimare il minimo di 2.13. In realtà si incorre nel problema del *bias*, determinato dalla distribuzione dei campioni sulla finestra temporale. Questo è causato proprio dalla presenza delle partizioni del RTPF, poichè, siccome ogni partizione è legata alle precedenti dalla fase di ricampionamento, accade che nelle ultime fasi della finestra temporale i campioni tendono a essere più sparsi. Tendenzialmente, si viene a formare uno squilibrio in favore di S_{t_k} , cioè l'ultima partizione della finestra, sia nell'espressione del gradiente sia nel vettore dei pesi del mix α . In questo modo, la $Bel_k(t_k)$ diventa un *bias* che condiziona l'evoluzione della finestra temporale successiva.

Una possibile soluzione, proposta in [6], consiste nella clusterizzazione dei campioni, in modo da calcolare la 2.15 e la 2.17 separatamente su ciascuno dei campioni. Questo porta, intuitivamente, a spezzare l'uniformità che il gradiente tende a favorire, in modo che solo i campioni che formano il cluster di maggior peso vengono favoriti e non tutti quelli della partizione S_{t_k} .

Un'altra soluzione è stata proposta in [11]; questo metodo abbandona l'idea del confronto fra distribuzione empirica e distribuzione teorica, ma cerca di costruire un modello del processo di ricampionamento che produce simultaneamente concentrazione e redistribuzione dei campioni sulla nuova finestra temporale. Ogni nuova partizione, costituita in seguito al *resampling*, possiede una quota di particelle provenienti da ciascun insieme della precedente finestra temporale. Per concentrazione si intende il ricompattamento delle partizioni in un'unica densità di probabilità a posteriori di N campioni, mentre per redistribuzione si intende il procedimento opposto, secondo il quale gli N campioni raggruppati vengono nuovamente sparsi in k partizioni. Il problema del *bias*, in questo modo, viene risolto imponendo che nes-

suna partizione riesca a prevalere sulle altre; il ricampionamento avviene, quindi, in modo da mantenere l'equilibrio tra le distribuzioni nei vari istanti temporali.

Capitolo 3

Architettura del robot mobile

3.1 Caratteristiche hardware

Il robot di riferimento per il sistema di localizzazione sviluppato in questa tesi è costituito da una base mobile *Nomad 200*[12], prodotta dalla società statunitense *Nomadic Technologies Inc.* nella prima metà degli anni '90, alla quale sono stati apportati significativi miglioramenti alla sensorialità e all'unità di elaborazione. L'utilizzo del robot *Nomad* è particolarmente frequente in ambito accademico e di ricerca, come si evince dalla letteratura robotica e dell'intelligenza artificiale. Si tratta di un robot mobile di forma cilindrica di dimensioni relativamente contenute (figura 3.1), la cui dotazione di sensori e attuatori lo rende particolarmente adatto a svolgere attività in un ambiente chiuso e strutturato. In questo capitolo viene descritta la dotazione attuale del robot presente nel laboratorio del Dipartimento di Ingegneria dell'Informazione, che comprende alcune componenti originali e alcune periferiche che sono state aggiunte in seguito.

3.1.1 Il sistema di elaborazione

La potenza di calcolo a disposizione sul robot è attualmente fornita da un processore *Intel Pentium III* con frequenza di clock di 1 GHz. Il PC di bordo monta un modulo RAM PC133 da 256MB e un hard disk da 40GB.

Per conciliare i vincoli di ingombro, dati dal diametro del robot di 50 cm, il processore alloggia su una scheda madre *Soyo SY-7VEM* [13], che presenta due



Figura 3.1: Il robot mobile Nomad 200.

canali IDE Ultra ATA/66, due porte USB e una scheda video Trident Blade 3D, oltre a un codec audio compatibile con lo standard AC97. Sono presenti anche tre slot PCI e un solo slot ISA. Lo slot ISA viene utilizzato per alloggiare la scheda *Galil DMC 630*[14], preposta al controllo dei motori del Nomad. La connessione wireless è garantita dalla presenza di una scheda *PCI D-Link AirPlus G+*, aderente allo standard 802.11g, in grado quindi di offrire una banda massima di 54Mbps.

Il sistema di elaborazione è alimentato da due batterie alloggiare nella parte inferiore della torretta, che forniscono ognuna 17 Ah, più una terza batteria da 12 Ah posta alla base del robot, per un totale di 46 Ah. L'alimentazione viene fornita da un alimentatore DC-DC da 250 W, collegato alle batterie del Nomad. Tale sistema di alimentazione consente all'elaboratore del robot di essere operativo per più di

cinque ore consecutive.

3.1.2 Attuazione

Tra le caratteristiche di interesse per lo svolgimento del lavoro esposto nel capitolo precedente, risulta senz'altro indispensabile la mobilità del robot stesso, che viene garantite da un sistema *synchro drive* costituito da tre ruote, di uguali dimensioni, posizionate simmetricamente alla base della struttura del Nomad. Le ruote sono collegate insieme da un unico sistema di cinghie, in grado di ruotarne i tre assi in modo sincrono (figura 3.2). Un analogo sistema di cinghie è preposto alla traslazione del robot, e un terzo motore permette di ruotare la torretta in modo indipendente dagli altri due assi. Questo consente di muovere il Nomad in una direzione mentre la torretta è libera di ruotare a piacimento.

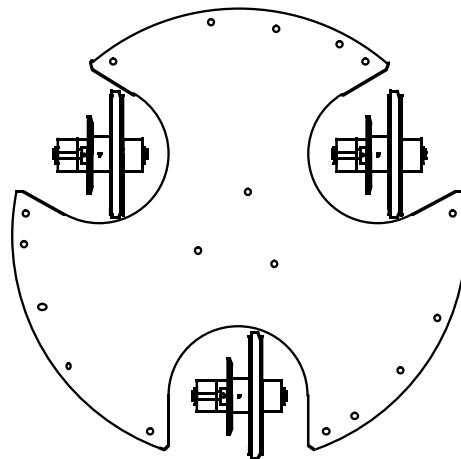


Figura 3.2: Le ruote (vista dal basso).

I tre assi del robot sono controllati da una scheda dedicata *Galil DMC 630*[14], che consente di raggiungere velocità di traslazione e di rotazione rispettivamente di 50 cm/s e 45 °/s. Nonostante il robot non possa essere considerato pienamente *olonomo*¹ è comunque possibile fare ruotare su se stesso il Nomad in modo da non occupare altro spazio aggiuntivo oltre al suo normale perimetro. La scheda DMC, collocata sul bus ISA della scheda madre, fornisce un protocollo di comunicazione

¹Un robot *olonomo* è in grado di muoversi in qualunque direzione indipendentemente dalla direzione verso cui è orientato.

a caratteri, tramite il quale è possibile richiedere l'esecuzione di operazioni di moto relativamente complesse. La scheda fornisce due modalità di funzionamento: comandi in velocità e comando in posizione. La prima modalità consente di impostare direttamente le velocità degli assi di rotazione: in questo modo le velocità di rotazione vengono mantenute per un tempo indefinito o fino a quando non si imposta una velocità differente. La seconda modalità permette di impostare uno spostamento relativo alla posizione attuale del robot: gli encoder posti sui motori del Nomad provvedono a tenere traccia dello spostamento effettuato e a controllare la distanza percorsa, sia per l'asse di traslazione sia per gli assi di rotazione.

Agli organi di moto di un robot mobile sono applicati degli encoder che misurano l'entità dei movimenti realizzati; le informazioni generate da questi sensori sono usate per il controllo a basso livello del robot: i valori misurati sono necessari per le funzioni svolte a un più alto livello dell'architettura di controllo del robot per effettuare i calcoli odometrici, che sono fondamentali per i task di localizzazione e di navigazione.

La potenza necessaria al robot per muoversi è fornita da due batterie da 12 V, collegate in serie e alloggiata alla base del robot stesso, che forniscono 12 Ah. Con queste batterie gli organi di attuazione hanno una autonomia di circa cinque ore.

3.1.3 Sensorialità

L'insieme dei sensori in dotazione al Nomad 200 ha subito un'evoluzione considerevole per dare spazio a componenti hardware più recenti e con prestazioni migliori che consentono al robot di percepire ambienti complessi. È il caso della periferica laser Sick LMS 200; il suo principio di funzionamento consiste nell'invio di un treno di impulsi laser infrarossi e nella stima del tempo di volo, che fornisce una misura della distanza dell'ostacolo incontrato. Tramite il movimento di uno specchio rotante il raggio viene emesso nella direzione voluta, in modo da effettuare la scansione di un'area di 180°. La risoluzione massima, ovvero la minima distanza angolare tra un raggio e il suo successivo, è di appena 1/4°, tuttavia in questa modalità non è possibile scansionare l'intera area di 180°, in quanto il laser limita la ricerca a soli 100°. Nel caso si utilizzino risoluzioni inferiori (1° oppure 0.5°) diventa possibile coprire l'intera area di 180°. A seguito di diversi test per misurare i

tempi di rilevazione delle distanze, sono stati dedotti i dati in tabella 3.1. I tempi di acquisizione si riferiscono a tempi medi in una serie da 100 scansioni consecutive.

| Risoluzione (°) | Tempo di acquisizione (ms) | σ (m) |
|-----------------|----------------------------|--------------|
| 1.00 | 13 | 0.01 |
| 0.50 | 26 | 0.01 |
| 0.25 | 52 | 0.01 |

Tabella 3.1: Tempi di acquisizione del laser Sick LMS 200.

Il laser è collegato al PC di bordo tramite una porta seriale di tipo RS422 a 500Kbps, più che sufficiente a sostenere il carico di dati del laser. L'interfaccia utilizzata consiste in un convertitore *EasySync USB-COMi* con chip FTDI, collegato a una delle due porte USB.

Il laser scanner Sick LMS 200 rappresenta uno standard *de facto* per i sensori di prossimità di cui sono equipaggiati solitamente i robot mobili. Il laser in questione è stato montato sulla sommità della torretta del Nomad in modo da non intralciare il posizionamento degli altri componenti, come il braccio manipolatore *Manus* (figura 3.1).

3.2 Software di controllo del robot

3.2.1 Il demone robotd

Il software originale *robotd* fornito dalla Nomadic, presenta le caratteristiche tipiche dei demoni UNIX: il *server* viene lanciato durante lo *start up* del sistema operativo per consentire ai *client*, che eseguono codice *custom*, di connettersi localmente o da macchina remota, mediante un meccanismo di IPC basato su *UNIX Socket*.

Il demone *robotd* fornisce le interfacce di accesso a tutti i componenti che erano presenti nella configurazione originale del Nomad 200, in particolare, viene fornita una libreria C che consente di controllare ogni componente hardware del Nomad.

Il limite principale del software *robotd* risiede nella totale assenza di meccanismi di comunicazione di tipo *push*: le letture dei sensori avvengono infatti tramite una richiesta effettuata dal client sul server, introducendo un tempo di latenza significativo e scarsamente misurabile. Questo intervallo può essere ridotto soltanto se si

aumenta la frequenza con cui il dato viene richiesto (quindi introducendo overhead sulle comunicazioni), e comunque non può mai essere annullato completamente.

Infine, risulta decisivo il fatto che il software di controllo non utilizza nessuna funzionalità real-time del sistema operativo, con la conseguenza che le prestazioni decadono progressivamente quando l'unità di elaborazione è sovraccarica.

3.2.2 Il framework Smartsoft

Il framework *Smartsoft* nasce con l'obiettivo di fornire una interfaccia comune verso architetture di tipo robotico, realizzando pattern di comunicazione di tipo generale [15]. Il framework prevede la creazione di una serie di componenti, ognuno dei quali mette a disposizione uno o più servizi, rappresentati da oggetti *server*, che forniscono dei dati. Nel caso di un'applicazione robotica concreta, un dato prodotto da un servizio può essere, ad esempio, una misura di distanza effettuata dal laser scanner. I componenti si registrano ai server tramite la creazione di oggetti *client*, specificando a quale server e a quale servizio si intende registrarsi. In figura 3.3 si vede un esempio di tre componenti che forniscono e usano vicendevolmente i servizi offerti dagli altri due.

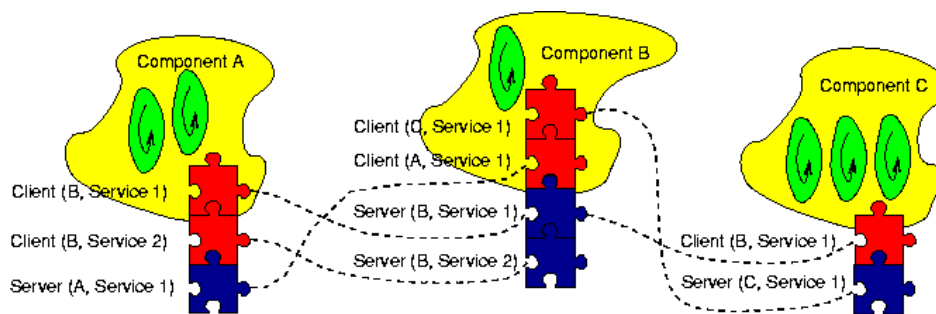


Figura 3.3: Esempio di client e server in Smartsoft.

Ciascun server stabilisce il modo in cui i dati prodotti verranno notificati ai client, ovvero aderisce a uno dei 4 *pattern* di comunicazione proposti, che sono illustrati in tabella 3.2. Questi *pattern* costituiscono il *core* del framework, in quanto consentono di progettare il sistema desiderato a un livello di astrazione più elevato.

Scendendo a un livello di astrazione più basso, si può notare che i *pattern* di comunicazione mettono in collegamento componenti che di fatto vengono imple-

| Pattern | Descrizione |
|----------------|---|
| Send | Modello: <i>client/server one-way</i> Trasferisce dati dal client al server senza la ricezione di una conferma dal server. Rappresenta una comunicazione <i>monodirezionale</i> utile per inviare comandi e settare configurazioni. |
| Query | Modello: <i>client/server two-way</i> Il client effettua una richiesta al server ed attende la risposta. Rappresenta una comunicazione <i>bidirezionale</i> tra un solo client ed un solo server. Può essere utile quando un'informazione viene utilizzata ad una frequenza molto bassa rispetto alla velocità con cui viene prodotta: è più ragionevole che il client la richieda mediante una query piuttosto che venga prodotta una quantità eccessiva di aggiornamenti non necessari. |
| Autoupdate | Modello: <i>publisher/subscriber 1-to-n distribution</i> Uno o più client si registrano presso un server richiedendo un'informazione che viene inviata non appena nuovi dati sono disponibili. Rappresenta una comunicazione di tipo <i>PUSH</i> . È possibile ridurre il traffico, se il client richiede l'informazione ad una frequenza minore rispetto a quella con cui viene prodotta, mediante una richiesta che prevede l'invio dei dati soltanto ad ogni ennesimo aggiornamento, o ad ogni intervallo temporale (<i>autoupdate timed</i>). |
| Event | Modello: <i>client/server asynchronous notification</i> Il server individua un evento avvenuto sullo stato del sistema ed informa in modo asincrono il client che ne assicura la gestione. Gli eventi sono utilizzati principalmente per notificare modifiche nello stato che sono rilevanti per coordinare i task in esecuzione. |

Tabella 3.2: *Pattern* di comunicazione forniti dal framework Smartsoft.

mentati come singoli processi di sistema (a ogni componente viene associato un processo Unix). A questo livello la comunicazione viene effettuata mediante un'architettura di tipo CORBA, quindi, l'esecuzione dei componenti non è delegata a una sola macchina, ma può essere distribuita agevolmente su più unità di calcolo, in virtù del fatto che è il sistema CORBA a occuparsi della distribuzione delle informazioni.

All'interno del Dipartimento di Ingegneria dell'Informazione di Parma è stata

sviluppata una serie di classi che consente di avere il completo accesso alle risorse hardware sensomotorie del Nomad.

Queste classi sono disponibili in due versioni:

- **Server Robotd:** utilizza il software *robotd* per controllare l'hardware originale del robot.
- **Server Direct:** accede direttamente alle periferiche presenti sul robot interfacciando l'hardware di basso livello senza l'uso del software originale robotd.

In entrambe le versioni il software garantisce l'accesso a ogni periferica presente originariamente sul robot. Al contrario, non è presente alcun supporto alla periferica laser Sick.

Limiti di Smartsoft

Ora che abbiamo presentato *Smartsoft* nelle sue principali funzionalità possiamo cercare di trarne qualche conclusione, per capire se il *framework* è adatto a raggiungere gli scopi presentati nel capitolo 1, con particolare riferimento alle prestazioni che il *framework* può conseguire in conseguenza delle scelte progettuali.

L'utilizzo di *CORBA* per realizzare i *pattern* di comunicazione porta grandi vantaggi in termini di portabilità dell'intera architettura e compatibilità con ogni altro sistema che ne faccia uso. Tuttavia, questo porta ad alcune inefficienze: la serializzazione dei dati attraverso lo stack di *CORBA* può essere più pesante rispetto a una comunicazione locale implementata *ad hoc*. Infine, bisogna considerare anche che *CORBA* costringe a un notevole impiego di risorse per l'apprendimento dei suoi meccanismi, che in *Smartsoft* non sono sufficientemente mascherati.

Un altro problema di cui ci si è accorti durante lo sviluppo di alcune applicazioni è l'assenza del supporto per l'esecuzione di *task* di tipo *real-time*: il framework non prevede la creazione di *thread* con priorità e il controllo dell'esecuzione a questo livello è completamente assente. Questo comporta che se il sistema raggiunge livelli di carico elevati, allora le prestazioni complessive dell'applicazione non sono più garantite. Per fare un esempio, l'esecuzione continua di un'applicazione di *collision avoidance* non è garantita, ovvero, se lo scheduler del sistema operativo

è sovraccarico esiste una certa probabilità che il robot finisca per urtare l'ostacolo che ha di fronte. L'obiettivo del progetto mira a costruire un'architettura che sia in grado di garantire le sue prestazioni, in modo che il robot operi in un ambiente in modo sicuro, escludendo che possa avvenire qualche inconveniente.

3.2.3 Il framework YARA

YARA [16] è un progetto realizzato presso l'Università degli studi di Parma e significa, ironicamente, *Yet Another Robotics Architecture*. Lo sviluppo di YARA ha avuto come obiettivo la costruzione di un sistema software che fornisca allo sviluppatore gli strumenti necessari per realizzare un'applicazione robotica a un livello sufficientemente elevato di astrazione, nascondendo i dettagli legati allo scheduling *real-time* dei task, contrariamente a quanto è stato fatto con *Smartsoft*, e fornendo un set completo di primitive di comunicazione.

Il design dei singoli componenti ha preferito focalizzarsi in modo tale da consentire lo sviluppo immediato di applicazioni *behaviour based*, ma, al contempo, fornisce gli strumenti per realizzare sistemi di tipo deliberativo dal carico computazionale maggiore. La caratteristica fondamentale che caratterizza YARA è il suo scheduler interno di tipo *real-time*, che si appoggia su uno strato software denominato *TODS*[17]. *TODS* fornisce un'interfaccia orientata agli oggetti per supportare la programmazione in tempo reale all'interno di un'applicazione C++ per il sistema operativo Linux. È stato sviluppato presso il Dipartimento di Ingegneria dell'Informazione dell'Università di Parma e consente di schedulare task periodici e *one-shot* assegnando a ognuno di essi una thread di sistema, prelevata da un *thread-pool*, che viene sottoposta allo scheduler *FIFO* del sistema operativo Linux. La libreria permette di realizzare un algoritmo di scheduling *real-time* variando dinamicamente la priorità delle thread poste in esecuzione. La politica di scheduling che viene fornita di default con *TODS* è di tipo *EDF*.

TODS interviene laddove il kernel del sistema operativo manca di fornire un set soddisfacente di primitive per la realizzazione di applicazioni *real-time*. In particolare, mancano gli strumenti per la gestione di regioni critiche che permettano di risolvere i problemi della *priority inversion*. Esistono alcuni metodi per prevenire questo problema, il più semplice dei quali prende il nome di *priority inheritance*

protocol. Tra le altre cose, *TODS* fornisce un insieme di strumenti di sincronizzazione (mutex, semafori, monitor) che realizzano il protocollo di *priority inheritance* a livello di libreria, garantendo il corretto funzionamento dello scheduling *EDF* anche in presenza di task che utilizzano risorse condivise.

Funzionamento di YARA

L'elemento fondamentale che caratterizza il *framework* è il *Modulo*, inteso come componente attivo e autonomo che opera all'interno dell'architettura: il sistema in esecuzione è composto da un insieme di moduli che svolgono attività indipendenti e cooperano tra loro scambiandosi informazioni e comandi (figura 3.4).

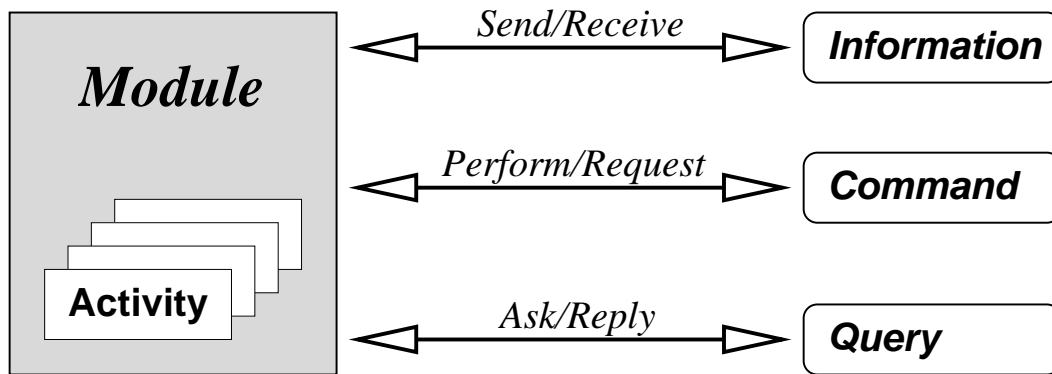


Figura 3.4: Strumenti per lo sviluppo: il *Modulo* ed i suoi componenti.

Un modulo concreto è un blocco funzionale che svolge compiti elementari. L'uso combinato di più moduli consente di creare sistemi con architetture di controllo sia reattive sia deliberative. La libreria consente di realizzare i moduli a partire dalla classe astratta `Module`, che rappresenta il principale *hot-spot* del *framework*. Similmente a quanto avveniva con *Smartsoft*, i vari moduli comunicano tra loro grazie all'utilizzo di alcuni *pattern* che consentono di realizzare comunicazioni dalla semantica differente (figura 3.4). Ciascun modulo è contenuto all'interno di una *unit*, che rappresenta l'istanza completa dell'architettura.

Il modulo non espone nessun metodo pubblico poichè realizza la sua trama di esecuzione in modo autonomo; a questo scopo vengono inserite al suo interno una serie di attività concorrenti, che si occupano di eseguire le funzioni del modulo. Ogni at-

tività associata ad un modulo viene posta in esecuzione tramite i meccanismi interni del framework: l'esecuzione può essere periodica, nel qual caso il metodo che vi è associato viene lanciato ad intervalli regolari, oppure può dipendere dall'interazione con gli altri moduli presenti nel sistema (ad esempio può essere attivata quando viene ricevuta un'informazione). Ad ogni attività, tramite il costruttore è assegnato un parametro che, espresso in secondi, ne indica il periodo nominale di funzionamento: esso rappresenta l'esatta cadenza temporale dell'esecuzione periodica e l'intervallo minimo che deve intercorrere tra due attivazioni aperiodiche consecutive. In entrambe le situazioni, operando in ambito *real-time*, il valore indicato avrà anche il significato di deadline per l'attività, cioè il tempo massimo entro cui l'esecuzione del metodo deve giungere a termine per essere giudicata corretta.

La classe template `Activity` consente di specificare le attività del modulo e fornisce i metodi necessari al loro utilizzo da parte degli altri componenti della libreria. Il codice che realizza concretamente le operazioni legate all'attività viene specificato fornendo al costruttore dell'oggetto `Activity` il puntatore alla funzione membro corrispondente; quest'ultima dovrà essere dichiarata nel corpo della classe derivata da `Module` compatibilmente con i parametri template della *activity* a cui viene associata. L'esecuzione periodica delle attività viene scatenata dal metodo `startPeriodicRun()`, mentre viene fermata con `stopPeriodicRun()`. Generalmente questi metodi vengono chiamati durante la fase di setup del modulo di cui fanno parte e vengono fermati in fase di distruzione. Nel listato 3.1 è presentato un esempio di modulo con una attività periodica di controllo dei motori.

I moduli che vengono realizzati per un certo sistema sono fisicamente nello stesso spazio di indirizzamento, quindi è possibile creare canali di comunicazione solo tra di essi. *YARA*, a differenza di *Smartsoft*, non fornisce un supporto per lo scambio di dati in un ambiente distribuito, che deve essere realizzato *ad hoc* dallo sviluppatore, infatti il modello di scambio di informazioni utilizzato è quello del paradigma a memoria condivisa. In caso di architetture distribuite bisogna prevedere in questo caso la presenza di un oggetto *proxy* che espleti le funzionalità di invio e ricezione remota delle richieste.

I tre *pattern* di comunicazione disponibili con *YARA* consentono di dare risposta a tutte le principali esigenze di comunicazione. Il primo fra questi è il *pattern Information*: esso consente ai moduli di inviare informazioni di tipo *one-way*, sen-

```
1 class MotorModule : public YARA::Module
2 {
3 private:
4   YARA::Activity<void, void> myActivity;
5   void do_something();
6 public:
7   MotorModule(YARA::Unit& u, std::string name) :
8     Module(u, name),
9     myActivity(this, &MotorModule::do_something, 0.02)
10  { }
11 };
```

Listato 3.1: Scheletro del modulo per il controllo dei motori in YARA.

za che, a priori, ci sia nessun tipo di richiesta. Questo pattern è appropriato, per esempio, per l'invio di informazioni riguardanti misurazioni sensoriali. La sua implementazione prevede l'utilizzo di due classi template *Sender* e *Receiver* che costituiscono gli estremi del canale di comunicazione; tramite il parametro template è possibile specificare il tipo del dato da scambiare. Dal punto di vista dell'utilizzatore è sufficiente invocare il metodo `send()` della classe *sender* per inviare il dato, mentre da lato ricevente una chiamata a `get()` recupera il dato trasportato dal *framework*.

Il secondo *pattern* disponibile è *Command*, pensato appositamente per l'invio di comandi. La sintassi di utilizzo è simile a quella del *pattern* precedente², ma in fase di costruzione dell'oggetto *Requester* è possibile specificare una priorità del comando, ovvero un *rank*. In ambito robotico questa funzionalità può essere utile per realizzare architetture reattive i tipo *subsumption* [18], nelle quali tra i comandi impostati da moduli differenti viene considerato solamente quello a priorità più elevata.

Il terzo pattern prende il nome di *State*, in quanto le sue funzionalità sono destinate alla realizzazione di *macchine a stati*, eventualmente distribuite tra i moduli. Il pattern *State*, simile come semantica al pattern *Event* di OROCOS (tabella 3.2), ser-

²Gli oggetti *Sender* e *Receiver* si sostituiscono con *Requester* e *Performer*. I metodi invocati invece non sono più `send()` e `get()`, ma è presente solo il `request()` in quanto alla richiesta del comando viene attivato il metodo associato al *Performer*.

ve a distribuire informazioni sullo stato di funzionamento dei moduli e sul verificarsi di eventi sporadici. L'invio di dati di questo tipo sarà generalmente caratterizzato da una varianza temporale piuttosto elevata. Esempi di informazioni che possono essere inviate con questo meccanismo di comunicazione sono lo stato di moto di un attuatore (*moving, stopped, stalled*), lo stato di funzionamento di un modulo di localizzazione (*localized, lost*) o la situazione percepita da un behaviour di *collision avoidance* (*safe, danger*).

Nell'esempio 3.2 è presentato l'utilizzo dei pattern *Information* e *Command* in un semplice behaviour di collision avoidance. Quando viene ricevuta l'informazione che rappresenta la distanza frontale, viene verificato che il robot non si trovi a meno di 50cm da un ostacolo, nel qual caso viene impostato il comando di velocità nulla.

```
1 class StopRobot : public Module
2 {
3     private:
4         Requester<float> vel;
5         Receiver<float> distance;
6         Activity <void, void> test_collision_act;
7
8         void test_collision() {
9             float front_distance = distance.get();
10            if (front_distance < 0.5) freeze();
11            else revoke();
12        }
13        void freeze() { vel.request(0.0); }
14        void unfreeze() { vel.revoke(); }
15
16    public:
17        StopRobot(Unit& u) : Module(u, "StopRobot"),
18            vel( this, "velocity", Requester<float>::MAX_RANK ),
19            distance( this, "front_distance"),
20            test_collision_act( this, &StopRobot::test_collision, 0.05) {}
21    };
```

Listato 3.2: Esempio di utilizzo dei pattern *information* e *command*.

Capitolo 4

Progettazione e realizzazione del sistema

4.1 Una visione di insieme

Il punto centrale del lavoro di tesi è il localizzatore, che racchiude al suo interno un cuore algoritmico molto complesso (vedi capitolo 1). Riassumendo quanto esposto in precedenza, il localizzatore necessita di due modelli matematici, costruiti sulla base di modelli probabilistici; l'esecuzione dell'algoritmo richiede l'ingresso di due controlli:

- controllo posizionale; fornisce una stima dello spostamento compiuto dall'ultima iterazione. Viene utilizzato per aggiornare il modello dinamico del sistema.
- controllo sensoriale; rappresenta una misurazione delle distanze dal punto di osservazione del robot. Viene utilizzato per aggiornare il modello sensoriale del sistema.

Possiamo prevedere che, nell'architettura complessiva del sistema, il modulo che si occuperà del localizzatore dovrà essere collegato da canali di comunicazione alla parte sensoriale e agli organi di movimento.

Il *framework YARA* è stato principalmente progettato al fine di ottenere una semplice e rapida progettazione di architetture di tipo reattivo. Benchè non siano

stati esclusi a priori i sistemi deliberativi, c'è comunque una spiccata preferenza per le architetture che fanno del loro cavallo di battaglia i *behaviours*. Per questo motivo, è realistico pensare che accanto al modulo di controllo motori ci siano alcuni comportamenti che definiscono i movimenti del robot sulla base degli dati sensoriali.

In figura 4.1 viene presentato uno schema a blocchi del sistema complessivo. Si

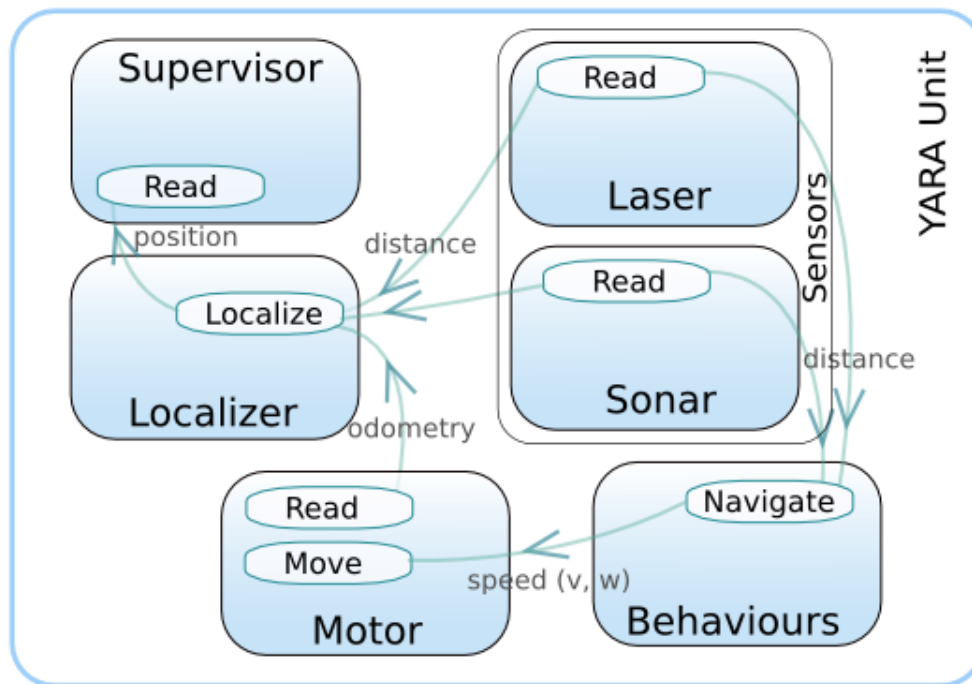


Figura 4.1: Schema a blocchi del sistema.

può notare come le serie di informazioni prodotte dai vari moduli che compongono il sistema finisca inevitabilmente per essere convogliate nel modulo *Localizer*. Questo scambio di messaggi avviene tramite l'utilizzo dei *pattern* di comunicazione che *YARA* fornisce.

È da sottolineare, inoltre, che il sistema presentato in figura 4.1 è comprensivo di ogni caratteristica progettata. Tuttavia, la modularità tipica delle architetture implementate con *YARA* consente di rimuovere a piacimento uno o più moduli non desiderati; al fine di ottenere un semplice movimento del robot, ad esempio, è possibile utilizzare solamente i moduli di movimento e sensorialità, escludendo temporaneamente l'utilizzo del modulo *Localizer*.

Vediamo come il sistema deve funzionare nel suo complesso. Supponiamo che venga assegnato al robot un task di movimento di un qualche tipo, ad esempio un behaviour che gli consenta di esplorare l'ambiente. Lo scambio di informazioni segue l'ordine suggerito dalle frecce in figura 4.1. Vediamo di riassumere le funzioni che i moduli devono assolvere:

- **Moduli Sensoriali:** I sensori sono sempre la fonte di informazioni principale. I sensori controllano il comportamento del *behaviour* in esecuzione e forniscono le stime delle distanze rilevate al modulo del localizzatore. È necessario prevedere una molteplicità di sensori ampia perchè il localizzatore pratica la *fusione sensoriale*; con questo termine si indica una tecnica usata per riunire diversi tipi di informazioni sensoriali in un'unica misurazione omogenea.
- **Modulo Motore:** Oltre a svolgere la semplice funzione di pilotaggio del movimento degli assi, deve anche eseguire un calcolo che consente di stimare l'odometria del robot. Questo dato deve necessariamente essere prodotto a livello software da questo modulo, l'unico a conoscere tutti i movimenti che vengono imposti al Nomad. Al fine della localizzazione, il dato odometrico è importante quanto la stima sensoriale, perchè il localizzatore deve disporre nella fase di predizione.
- **Modulo Localizzatore:** Implementa l'algoritmo del filtro particellare visto al capitolo 1. Per funzionare necessita dell'ingresso dei dati odometrici, forniti dal modulo Motor, e dei dati sensoriali, forniti dal modulo Laser. In uscita produce una stima della posizione anche complessa, che difficilmente può essere riutilizzata così com'è.
- **Modulo Supervisor:** La sua funzione risiede in un meccanismo di controllo del localizzatore, che, in prima analisi, potrebbe limitarsi a prelevare l'output prodotto dal modulo *Localizer* e visualizzarlo in qualche modo; in alternativa, sarebbe possibile costruire un sistema più complesso che consenta di elaborare i dati prodotti dal localizzatore al fine di manipolare in modo diverso il robot. Un modulo di questo tipo potrebbe interfacciarsi con un algoritmo di decisione per stabilire il tipo di controllo da effettuare sul sistema.

- **Moduli Comportamentali:** L'impiego di questi *behaviours* consente di far muovere il robot per parecchio tempo senza l'intervento umano. Un esempio di *behaviour* frequentemente realizzato è il *wall following*, ma ce ne sono molti altri. Si può anche pensare di coordinare due o più *behaviours* al fine di ottenere un comportamento che assomiglia il più possibile a quello di un sistema deliberativo.

Il progetto è stato spezzato in due parti al fine di ottenere, in primo luogo, una architettura di controllo funzionante e affidabile, mentre in seguito il localizzatore è stato integrato nell'intera architettura. Alla fine dell'implementazione del sistema base è stata eseguita una serie numerosa di test per verificare il corretto funzionamento e valutare le prestazioni del sistema di base (capitolo 5).

Le prestazioni sono effettivamente il punto critico del nostro sistema. Le tipiche implementazioni di localizzazione [11] richiedono in genere parecchia potenza di calcolo per la loro esecuzione, con il risultato che la stima della posizione può essere prodotta con un ritardo di qualche secondo.

4.2 Progettazione del sistema base

La progettazione del sistema parte da quella dei suoi moduli fondamentali. Come illustrato in figura 4.1, è possibile suddividere l'architettura di controllo in due parti: la prima che si occupa della sensorialità e la seconda che permette di utilizzare la scheda motori del robot. In seguito, occorre collaudare il funzionamento dell'architettura tramite una serie di *behaviours* comportamentali, al fine di ottenere una architettura di cui sono note le prestazioni e il grado di affidabilità.

4.2.1 Sensorialità

I moduli sensoriali che si vuole implementare devono fornire accesso alla periferica e rendere disponibili i dati che il dispositivo produce. Tuttavia, occorre pensare al modulo come a un oggetto attivo, privo il più possibile di una interfaccia che espone metodi pubblici, in quanto esso possiede un suo stato interno e gli unici eventi che possono cambiarlo devono essergli notificati tramite i pattern *YARA*. Pertanto, i

moduli sensoriali non possono presentare una interfaccia che permette di accedere direttamente alla periferica hardware, ma devono incapsulare tutta l'implementazione di basso livello al loro interno. Un modulo sensoriale può essere visto come una *black box* che, una volta avviata, fornisce ininterrottamente i dati misurati e dall'esterno non deve essere possibile accedere all'interno del componente.

Tra i moduli sensoriali presenti nello schema 4.1, il Nomad è grado di eseguire effettivamente solo quello relativo alla periferica laser. Il modulo *Sonar* avrà comunque uno scheletro analogo (lo stesso procedimento è valido per un modulo *Infrared*); la differenza tra un modulo e un altro risiede unicamente nell'implementazione delle comunicazioni di basso livello con l'hardware.

Pensando a una possibile implementazione in *YARA*, il modulo dovrà disporre di una *Activity* dove interroga l'hardware tramite il driver di basso livello. Non appena il driver ritorna il dato misurato, questo deve essere passato a livello del modulo e, da lì, deve essere copiato all'interno del *framework* per essere reso disponibile agli altri moduli che lo richiedono.

La figura 4.2 schematizza per passaggi logici i compiti di un modulo sensoriale teorico, che invia dati di tipo *Data*. I moduli concreti dovranno implementare la classe *driver_hardware* a seconda della periferica a cui sono collegati.

In questo modo otteniamo un modulo che aggiorna costantemente il *framework* non appena le nuove misurazioni diventano disponibili. È logico che, a seconda della periferica per cui si implementa il modulo sensoriale, il tempo di recupero delle informazioni sia variabile; l'*activity* dovrà impostare un periodo proporzionale a quel particolare dispositivo (ricordiamo che il sonar impiega solo 4ms per ottenere una scansione mentre il laser impiega almeno 13ms).

Il modulo laser

Come evidenziato al paragrafo precedente, la struttura del modulo laser è del tutto analoga a quella di un modulo sensoriale generico; l'unico elemento che cambia è l'implementazione del substrato software di interfacciamento con l'hardware.

A questo scopo, il modulo del laser utilizza al suo interno un oggetto di una classe appositamente creata per manipolare dispositivi di questo tipo. Questa classe è stata modificata in diverse occasioni al fine di ottenere un oggetto in grado di

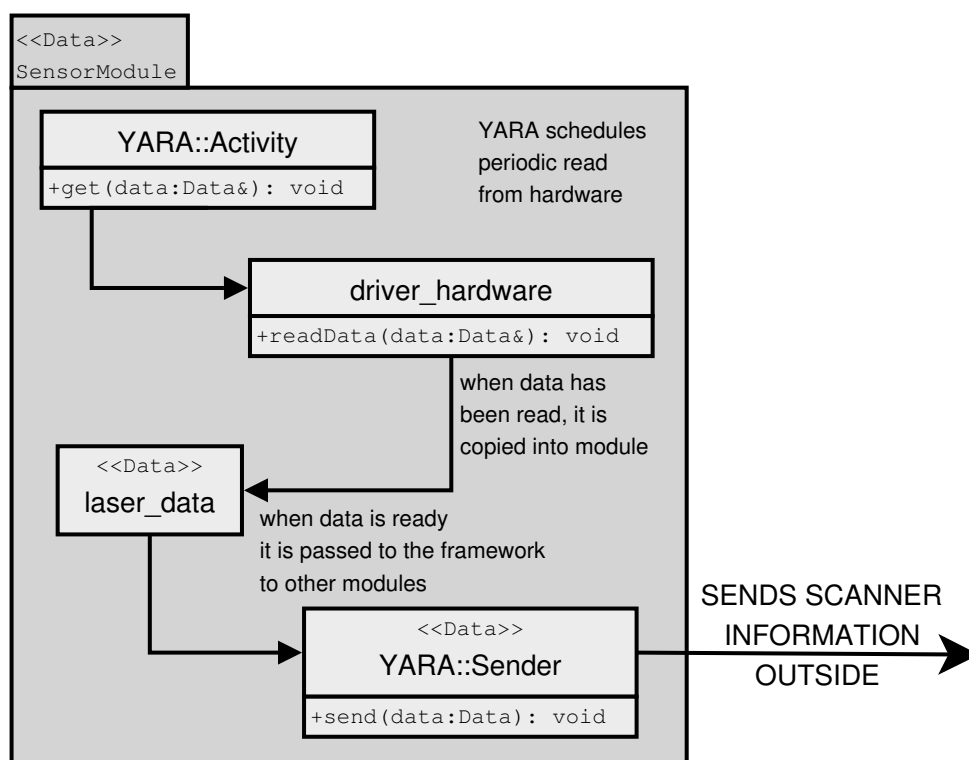


Figura 4.2: Schema di un modulo sensoriale.

interfacciarsi alla periferica secondo la configurazione hardware desiderata. Il dispositivo laser, infatti, possiede una serie di caratteristiche configurabili durante il suo *start up*, che consentono di specificare in che modo deve effettuare le scansioni (vedi paragrafo 3.1.3).

La classe originale presentava diversi bug, dovuti a erranee interpretazioni del protocollo di comunicazione tra il device laser e il software. In un primo tempo, perciò, è stato svolto un lavoro di pulizia sulla classe del driver laser, eliminando alcuni bug e semplificando l'interfaccia di utilizzo; le modifiche che si sono susseguite sulla classe hanno consentito di esporre a costruttore tutti i parametri di configurazione che la periferica può fornire. L'interfaccia attuale della classe *SickLMS200* è illustrata nel listato 4.1.

Il metodo di accesso principale è `readData()`; esso consente di passare al driver un oggetto della classe `vector` al cui interno vengono copiate le informazioni recuperate dal dispositivo.

```
1 class SickLMS200 {
2
3     public:
4
5     SickLMS200(std::string device_name, speed_t port_rate,
6               unsigned int resolution, unsigned int range_res)
7     throw (NotReadyException);
8
9     // scan data and stores it in millimeters
10    void readData( std::vector<unsigned int>& data );
11    // scan data and stores it in meters
12    void readData( std::vector<double>& data );
13    // scan data and stores it in raw format
14    void readData( std::vector<raw_laser_data>& data );
15
16    void flush(); // flushes serial port, to make sure data is current.
17 };
```

Listato 4.1: Interfaccia del laser Sick LMS 200.

Nel contesto di una applicazione *real-time*, l'invio delle informazioni sensoriali è sicuramente una attività svolta in modo molto frequente. Un sistema che desiderasse leggere tutte le informazioni prodotte dal laser dovrebbe dotarsi di una applicazione in grado di trasferire ed elaborare dati alla velocità di 13ms che, se sommata al resto delle attività in esecuzione, potrebbe appesantire eccessivamente la velocità di elaborazione generale. Per questo motivo l'implementazione della classe è stata modificata per minimizzare il numero di copie consecutive necessarie alla produzione del dato; mentre il driver originale eseguiva 4 copie sequenziali dello stesso dato prima di restituirlo all'utente, le modifiche introdotte hanno permesso di ridurre il numero di queste copie a 1.

Analizzando la configurazione hardware del Nomad 200, si nota che il dispositivo è connesso alla *main board* tramite porta seriale RS422 che viene poi convertita in porta USB tramite un convertitore con chip FTDI. Questa soluzione causa la presenza di un buffer di limitate dimensioni tra la periferica e la CPU, di cui bisogna necessariamente tenere conto durante la progettazione. Per questo motivo l'interfaccia del driver presenta un metodo `flush()` che consente lo svuotamento del

buffer in modo asincrono. Questo metodo risulta necessario quando una applicazione legge le informazioni prodotte dal laser in modo più lento di quanto vengano scritte sul buffer; infatti, se si verifica questa situazione accade che i dati prodotti dal laser saturano la capacità di contenimento del buffer, causando errori in lettura.

Il listato 4.2 presenta l'implementazione dell'interfaccia del modulo sensoriale.

```
1 class LaserModule : public YARA::Module {
2   private:
3     laser::SickLMS200 laser_driver;
4     std::vector<double> myData;
5
6     YARA::Activity<void, void> laser_activity;
7     void laser_activity_body();
8
9     YARA::Sender< std::vector<double> > laser_data_sender;
10
11   public:
12     LaserModule(YARA::Unit& u,
13                speed_t speed = B500000,
14                unsigned int resolution = 50,
15                unsigned int range_res = 1,
16                const char * device_path = "/dev/ttyUSB0"
17                );
18 };
```

Listato 4.2: La classe LaserModule.

4.2.2 Movimento

La seconda parte fondamentale del sistema di controllo del robot è il modulo che realizza il movimento del robot. In realtà, questo modulo risulta più complesso in quanto non deve occuparsi solamente del movimento, ma deve anche procedere a una stima odometrica dello stesso.

Il modulo *Motor* possiede una struttura completamente diversa da quella utilizzata per il modulo sensoriale, perciò non è solo necessaria l'implementazione delle

classi di interfacciamento hardware di basso livello, ma anche una riprogettazione della sua struttura a livello più alto.

Facendo riferimento alla figura 4.1, si può notare che questo modulo possiede due attività. La prima, denominata *Move*, prende in ingresso un comando di velocità in traslazione e in rotazione. Il suo compito è essenzialmente quello di chiamare le primitive di basso livello per pilotare i motori.

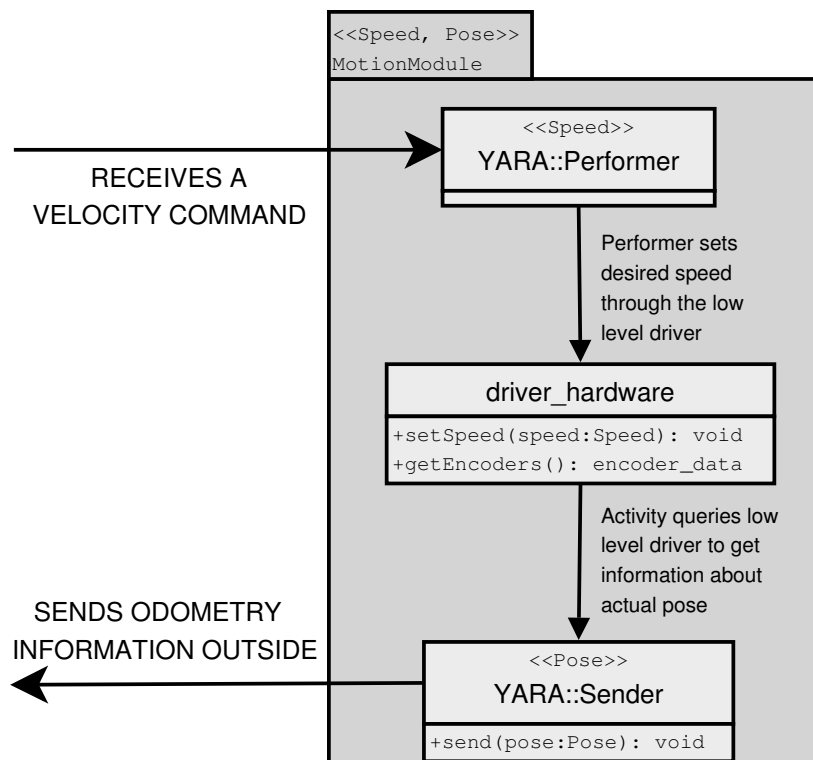


Figura 4.3: Schema di un modulo di movimento.

La seconda attività, invece, è chiamata *Read*; il suo ruolo consiste nell'esecuzione periodica di un calcolo odometrico al fine di produrre una stima del movimento compiuto.

La figura 4.3 presenta la realizzazione con YARA del modulo di movimento. Da notare che il driver che si interfaccia con l'hardware è presente anche in questo caso, ma, a differenza di prima, ora viene interfacciato a due attività diverse che si contendono concorrentemente l'accesso alla risorsa. Questa contesa, comunque, viene risolta a un livello software inferiore, poichè è YARA a occuparsi dell'accesso alle

risorse condivise proteggendo il loro accesso con un *mutex*. È da notare che questa protezione è necessaria poichè un ordine pseudo casuale dei comandi inviati alla scheda potrebbe determinare un comportamento non prevedibile dei motori. Inoltre, è da sottolineare che l'esecuzione delle due attività non è necessariamente sequenziale, anzi, l'attività di controllo dei motori interviene esclusivamente alla ricezione di un comando di velocità, mentre l'attività di recupero delle informazioni odometriche ha un suo periodo impostabile a piacimento, a seconda dell'accuratezza con cui si vuole ottenere la misurazione.

Al fine di massimizzare il riuso di quanto è stato fatto, si è scelto di progettare una classe che contenesse tutte le caratteristiche specifiche del Nomad, e di lasciare al di fuori di essa tutti gli oggetti che possono essere riutilizzati. Per questo motivo, il *porting* del sistema su un robot diverso dal Nomad non dovrebbe presentare troppe difficoltà, richiedendo di adattare una sola classe tra quelle realizzate.

La classe *Nomad200*, perciò, consente di interfacciare una qualunque applicazione robotica con la scheda motori del robot e, al contempo, di eseguire l'elaborazione odometrica di cui si è parlato. Ne consegue che la sua interfaccia è molto semplice, in quanto delega l'esecuzione delle sue attività ad altri oggetti specifici. Il listato 4.3 mostra l'elenco dei metodi pubblici della classe, che consentono di svolgere le operazioni sui motori e sull'odometria.

```
1 class Nomad200 {
2     public:
3         Nomad200(std::string device_path);
4
5         // ODOMETRY MEMBERS
6         void update(double deltatime);
7         Pose getPose() const;
8         STATE getState() const;
9         void reset(double x, double y, double theta, double thetaTurret);
10        // MOVEMENT MEMBERS
11        void setVelocity(const motion::CommandVelocity& cmd);
12    };
```

Listato 4.3: La classe Nomad200.

La scheda motori

Gli organi di movimento del Nomad sono pilotati da una scheda di controllo che espone un linguaggio di tipo dedicato al programmatore. Nel progetto originale *robotd* queste comunicazioni erano gestite interamente a livello software *user space*; successivamente, presso il Dipartimento di Ingegneria dell'Informazione è stato sviluppato un driver di livello kernel *ad hoc*. Questo progetto ha creato un *kernel driver* in grado di gestire le comunicazioni tra il software e l'hardware della scheda a livello *kernel space*. Il risultato è che ora sono disponibili un device `/dev/nmotor0` e una classe *Dmc* che consentono di gestire la comunicazione con la scheda in modo molto più semplice.

Anche in questo caso, si è svolto un lavoro di ottimizzazione sulla classe in questione, cercando di rendere l'interfaccia più agevole all'integrazione del sistema di localizzazione. L'interfaccia della classe è relativamente complessa perchè presenta tutti i metodi possibili di accesso alla scheda. Nel listato seguente 4.4 sono mostrati i metodi di maggiore interesse.

```
1 class Dmc {
2     public:
3         // position command mode
4         bool setRelPos(int x, int y, int z);
5         bool setRelPos(int axis, int value);
6         // velocity command mode
7         bool setJog(int x, int y, int z);
8         bool setJog(int axis, int value);
9
10        // encoder reading methods
11        bool getPosition(motion::encoder_data& data);
12        bool getError(motion::encoder_data& data);
13        bool getStatus(motion::encoder_data& data);
14    };
```

Listato 4.4: Interfaccia della classe Dmc.

Odometria

Il Nomad 200 fa parte della categoria di robot la cui configurazione prende il nome di *synchro drive* a 3 ruote [19]. Le ruote sono accoppiate meccanicamente in un modo da ruotare tutte contemporaneamente nella stessa direzione e alla stessa velocità. Questa sincronizzazione tra gli assi è raggiunta tramite un sistema di cinghie che pilotano contemporaneamente tutti gli assi interessati. Queste informazioni risultano di grande importanza al fine di realizzare un algoritmo di calcolo odometrico preciso e affidabile. Infatti, a partire dalla conoscenza del numero di rotazioni rilevate dagli encoder, è possibile calcolare la posizione del robot istante per istante. Questo calcolo può essere approssimato, in prima analisi, da un movimento in linea retta secondo la direzione di avanzamento media. Questa approssimazione vale tanto più quanto più frequentemente viene fatto l'aggiornamento del calcolo odometrico, ma commette comunque un errore. In figura 4.4 è presentato il tipico caso in cui si commette l'errore maggiore.

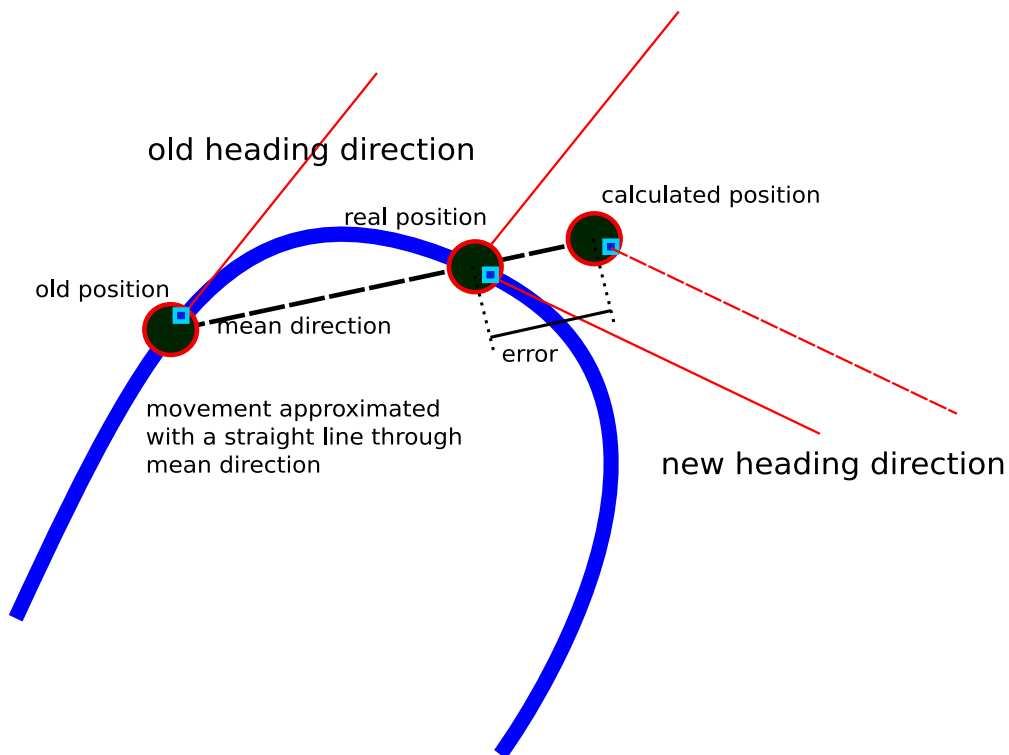


Figura 4.4: Approssimazione lineare della traiettoria.

Nel caso il robot sterzi in modo brusco, allora la retta mediana che approssima il movimento (linea grigia tratteggiata) conduce fuori dalla traiettoria reale (linea curva). Questo perchè la lettura dei valori degli encoder di traslazione riporta un movimento molto esteso, che se sviluppato lungo una retta può generare errori anche considerevoli.

Per migliorare l'algoritmo di calcolo odometrico è stato progettato un nuovo modello matematico che tiene conto di più elementi per sostituire l'approssimazione rettilinea con una più precisa del 2° ordine.

Sappiamo che, una volta impostato il comando, le velocità di traslazione e di rotazione rimangono costanti fino all'arrivo del comando successivo. Approssimiamo a zero il tempo di salita e di discesa delle velocità, in altre parole, supponiamo che il tempo per portarsi da velocità zero alla velocità di regime sia nullo. In questo modo otteniamo che la velocità si porta a regime seguendo il profilo del gradino. Sotto queste ipotesi approssimanti, un comando di velocità di traslazione e rotazione costanti fa muovere il robot lungo un arco di cerchio.

A partire dalle coordinate (x_0, y_0) è possibile ricavare la nuova posizione (x_1, y_1) . Infatti, sappiamo che:

$$\dot{\theta} = \text{cost.} \quad (4.1)$$

$$v = \text{cost.} \quad (4.2)$$

Allora è possibile ottenere il raggio del cerchio come l'inverso della sua curvatura:

$$K = \frac{1}{R} = \frac{\dot{\theta}}{v}$$

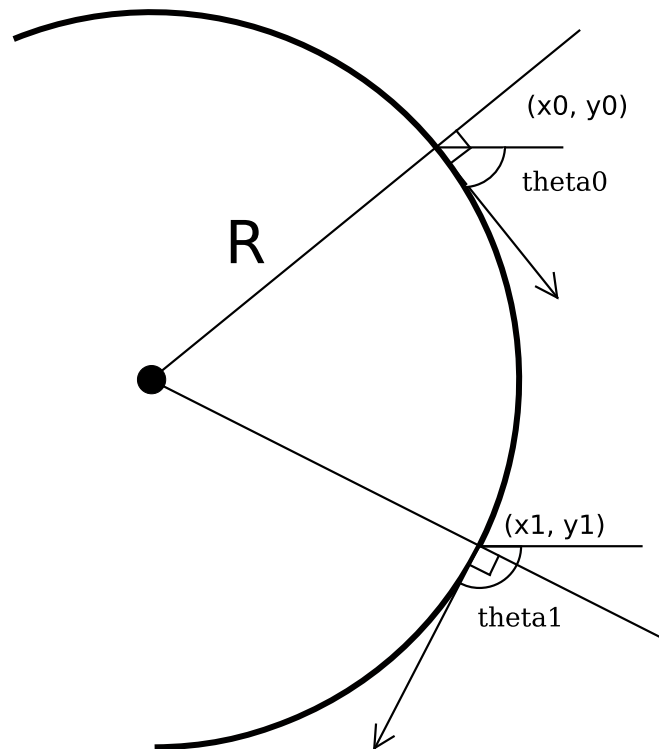


Figura 4.5: Esempio di movimento su un cerchio.

Per convenzione, il raggio del cerchio è sempre positivo, quindi prendiamo per semplicità $\dot{\theta} \geq 0$ e $v \geq 0$ (ai fini della dimostrazione è equivalente se vengono presi entrambi negativi). Applichiamo al punto (x_0, y_0) un vettore di modulo pari alla velocità v e di argomento pari a θ_0 . Calcoliamo il versore normale alla sua direzione come $(-\sin \theta_0, \cos \theta_0)$. Conoscendo la misura del raggio, possiamo ricondurci alle coordinate del cerchio, ottenendo che

$$\begin{cases} X_r &= X_0 - R \sin \theta_0 \\ Y_r &= Y_0 + R \cos \theta_0 \end{cases}$$

Ripetendo il procedimento per il punto (x_1, y_1) , ricaviamo il suo versore normale come $(\sin \theta_1, -\cos \theta_1)$. A questo punto, partendo dal centro ormai noto possiamo ricavare il punto (x_1, y_1) come

$$\begin{cases} X_1 = X_r + R \sin \theta_1 \\ Y_1 = Y_r - R \cos \theta_1 \end{cases}$$

Combinando le ultime due e risistemando i membri si ottiene infine

$$\begin{pmatrix} X_1 \\ Y_1 \end{pmatrix} = \begin{pmatrix} X_0 \\ Y_0 \end{pmatrix} + \left[\begin{pmatrix} -\sin \theta_0 \\ \cos \theta_0 \end{pmatrix} + \begin{pmatrix} \sin \theta_1 \\ -\cos \theta_1 \end{pmatrix} \right] R \quad (4.3)$$

Tramite la 4.3 possiamo quindi effettuare un calcolo odometrico che approssima al secondo ordine il movimento del robot, ottenendo così una maggiore precisione. Questa formula vale solo se ci accontentiamo di considerare valide la 4.1 e la 4.2. In caso contrario il modello deve essere ulteriormente aggiornato a un grado superiore.

Si è pensato di incorporare il calcolo odometrico all'interno di una classe *Integrator* il cui unico scopo è quello di eseguire questo tipo di calcoli. All'interno di questa classe vengono implementate sia la formula 4.3 che l'approssimazione rettilinea classica, principalmente per un motivo. La formula 4.3 prevede una moltiplicazione finale per il raggio del cerchio R . Tuttavia, l'algoritmo implementato nella classe deve obbligatoriamente risalire alla misura del raggio tramite l'inverso della curvatura, ovvero

$$R = K^{-1} = \frac{v}{\dot{\theta}}$$

Se il valore di $\dot{\theta}$ rilevato è prossimo allo zero, questo tende a far esplodere il raggio del cerchio all'infinito. Matematicamente la formula è corretta, tuttavia un calcolatore potrebbe, senza troppa sorpresa, fare divergere il calcolo a un valore infinito. Per questo motivo si è scelto di impostare una minima velocità di rotazione entro la quale viene utilizzato il metodo rettilineo che non soffre di alcun problema numerico. Questo valore minimo è stato stimato essere circa 0.5° .

La classe *Integrator* fornisce sostanzialmente un unico metodo di accesso con il quale vengono impostati i valori di traslazione e rotazione del robot. La classe

mantiene al suo interno tutte le informazioni ricevute fino a quel momento e opera con l'algoritmo presentato al fine di ottenere la stima della posizione. Il listato 4.5 mostra l'interfaccia della classe.

```
1 class Integrator {
2     public:
3         Integrator();
4
5         void reset(double x, double y,
6                 double theta, double thetaTurret);
7         Pose integrate(double delta_trans, double delta_steer,
8                 double delta_turret, double deltatime);
9     };
```

Listato 4.5: Interfaccia della classe Integrator.

Il modulo Motor

Il modulo *Motor* deve occuparsi dell'esecuzione dei comandi che provengono dall'esterno e del calcolo odometrico. Le due classi che abbiamo visto nei paragrafi precedenti servono proprio per svolgere questi due compiti di base. Per poter inserire il tutto all'interno del *framework* dobbiamo operare in modo analogo a quanto già fatto per il modulo del laser.

Mentre l'*activity* per il calcolo odometrico è piuttosto simile a quella già descritta per il laser, e di fatto ne conserva quasi tutte le caratteristiche, l'*activity* destinata al controllo dei motori ha invece una particolarità in più. Non è certamente possibile stabilire a priori un periodo di esecuzione di questo task, in quanto i comandi possono essere inviati anche a grandi distanze di tempo. Quello che si richiede, invece, è che il robot risponda immediatamente ogni volta che gli arriva un nuovo comando. Il *framework YARA* mette a disposizione del programmatore il *pattern command* che consente di assegnare a una attività l'esecuzione del comando in entrata. Siccome a ogni attività è associata una *deadline*, in questo caso essa rappresenta il minimo tempo che deve intercorrere tra una esecuzione di un comando e il suo successivo.

In questo caso diventa importante stimare il miglior tempo associato alla *deadline* dell'attività di esecuzione comandi. Un tempo troppo prolungato potrebbe negare la possibilità di eseguire comandi a una frequenza elevata; in un controllo di *collision avoidance* questo potrebbe portare a malfunzionamenti e, eventualmente, a incidenti. Viceversa, un tempo troppo ridotto rischia di sovraccaricare l'architettura; se il tempo è talmente basso che il *framework* non è in grado di trasportare il comando fino alla scheda e ritornare, il sistema genera una *deadline missed*.

L'altra attività invece, similmente al laser, si pone semplicemente come periodica e legge i dati prodotti dalla scheda, rendendoli disponibili a chi li richiede.

Al fine di chiarire meglio quanto descritto, nel listato 4.6 vengono mostrati non solo i metodi pubblici di avvio del modulo (`start()` e `stop()`), ma anche i membri privati.

```
1 class MotorModule : public YARA::Module {
2     public:
3         static const double INTEGRATION_PERIOD = 0.050; // seconds
4         static const double COMMAND_PERIOD = 0.025; // seconds.
5
6         MotorModule(YARA::Unit& u);
7         void start() {
8             motor_performer.start();
9             odometry_activity.startPeriodicRun();
10        }
11        void stop() {
12            motor_performer.stop();
13            odometry_activity.stopPeriodicRun();
14        }
15
16        private:
17            motion::Nomad200 nomad; // Nomad200 class
18            // ODOMETRY MEMBERS
19            YARA::Activity<void, void> odometry_activity;
20            void odometry_activity_body();
21            YARA::Sender< motion::Pose > odometry;
```

```

22     // PERFORMING MOVEMENT MEMBERS
23     YARA::Performer< motion::CommandVelocity > motor_performer;
24     YARA::Activity<void, motion::CommandVelocity > performer_activity;
25     void motor_performer_activity_body(motion::CommandVelocity& cmd);
26 };

```

Listato 4.6: Il modulo Motor.

4.2.3 Operatività

In questa sezione sono descritti i modelli da cui si è partiti per progettare i behaviours e, in seguito, ne verrà fornito un modello implementativo basato su *YARA*.

I tipici algoritmi che la letteratura presenta sono i classici *wall following*, *center following*, *collision avoidance* e altri ancora. La maggior parte dei behaviours implementati fanno riferimento a [20], dove viene introdotto un modello matematico per la coordinazione di differenti behaviours collaborativi. L'idea principale che sta alla base di questi comportamenti è il concetto di *attrattore* e *repulsore*. Possiamo descrivere un sistema dinamico con un semplice equazione:

$$\dot{\vec{x}} = \vec{f}_b(\vec{x})$$

La funzione \vec{f}_b può essere interpretata come la forza che agisce sulle variabili comportamentali. La forza viene diretta in modo tale da formare un *attrattore* verso la direzione desiderata, oppure un *repulsore* che allontana il robot dagli ostacoli. La funzione \vec{f}_b dipende dalla posizione relativa tra il robot e l'ambiente su cui opera; il coordinamento tra behaviours multipli, viene raggiunto assegnando un peso alla forza impressa da ognuno di essi. In genere un attrattore si presenta nella forma

$$\dot{\theta} = f_b(\theta) = -\lambda \sin(\theta - \psi_b)$$

dove $\lambda > 0$ rappresenta la forza dell'*attrattore* e ψ_b la sua direzione.

Viceversa, un *repulsore* è della forma

$$\dot{\theta} = f_b(\theta) = \lambda \sin(\theta - \psi_b)$$

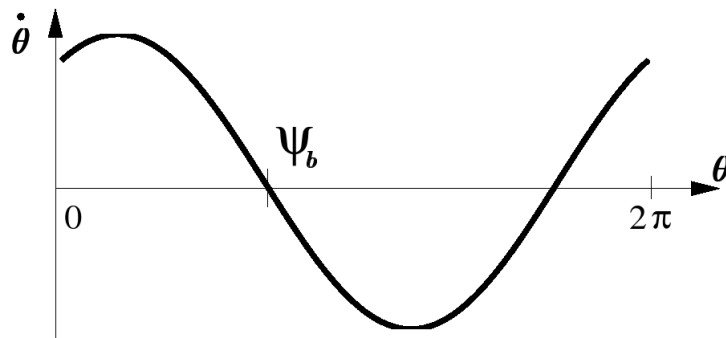


Figura 4.6: La funzione dell'attrattore.

in quanto deve respingere la posizione angolare θ dalla direzione del *repulsore* ψ_b .

La composizione di *attrattori* e *repulsori* può dare luogo a comportamenti più complessi, come vedremo nei paragrafi seguenti.

Center Following

Il primo behaviour progettato e testato è stato il *center following*. Il comportamento che implementa consente al robot di navigare in un interno non strutturato e seguire in ogni istante il centro dell'area in cui si trova. Questo comportamento è ideale per fare seguire al robot la direzione di un corridoio, mentre funziona un po' peggio quando il robot si trova in aree più vaste, come una stanza. Il suo funzionamento è abbastanza semplice: il sensore laser misura la distanza dal muro alla sua destra e alla sua sinistra; il behaviour agisce sui motori in modo da rendere queste distanze uguali.

Parlando in termini di *repulsori* e *attrattori*, è necessario impostare una dinamica per ciascuno dei due muri laterali. In particolare, se impostiamo un repulsore nella direzione normale a ciascun muro e la moduliamo in modo che la forza repulsiva sia uguale e contraria esattamente nel centro del corridoio, allora il robot non potrà fare altro che allinearvisi.

Moduliamo inoltre la forza repulsiva secondo una legge esponenziale negativa, avente come parametro la distanza dal muro: in questo modo, se il robot si sta dirigendo verso la direzione del muro verrà respinto con maggior violenza rispetto a un robot che sta andando dritto. Chiamiamo d_{wall1} e d_{wall2} le distanze dai muri

destro e sinistro. Dette, inoltre, ψ_{wall1} e ψ_{wall2} le normali ai muri destro e sinistro, e detti c_{wall1} e c_{wall2} i fattori di modulazione che determinano la ampiezza della forza, possiamo scrivere

$$f_{b1}(\theta) = \lambda_1 \sin(\theta - \psi_{wall1}) e^{-c_{wall1} d_{wall1}} \quad (4.4)$$

$$f_{b2}(\theta) = \lambda_2 \sin(\theta - \psi_{wall2}) e^{-c_{wall2} d_{wall2}} \quad (4.5)$$

Per far sì che il robot rimanga al centro del corridoio, è sufficiente imporre che $c_{wall1} \equiv c_{wall2}$, poichè in questo modo le forze repulsive si eguagliano al centro. Sommando la 4.4 e la 4.5 al fine di ottenere un sistema unico, si ottiene

$$f_b(\theta) = f_{b1} + f_{b2} = \lambda_b \sum_{i=1}^2 [\sin(\theta - \psi_{wall_i}) e^{-c_{wall} d_{wall_i}}] \quad (4.6)$$

che è la formula finale del behaviour *center following*. Le due costanti introdotte sono parametri che determinano la reattività del behaviour in fase di funzionamento. La costante λ_b determina un incremento generale della forza repulsiva, rendendo potenzialmente più instabile il behaviour; la costante c_{wall} , invece, influisce sulla rapidità dell'esponenziale negativo, andando a determinare lo smorzamento progressivo della forza repulsiva all'allontanarsi dal muro (più è grande, meno si sente la forza di repulsione del muro più vicino).

A riprova del funzionamento del behaviour, la figura 4.7 mostra il risultato di questi due repulsori sovrapposti: la loro somma, dà luogo a un attrattore esattamente nel centro del corridoio, permettendo al robot di navigare come desiderato.

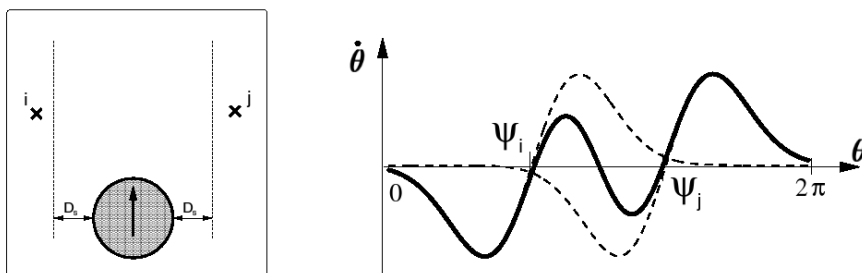


Figura 4.7: Attrattore generato dalla somma di due repulsori.

Wall Following

Il suo funzionamento si basa generalmente sull'acquisizione dei dati sensoriali dal fianco del robot, in modo da mantenere una distanza dal muro preimpostata e costante. Un behaviour di questo tipo è probabilmente il più adatto a scopi di navigazione prolungata, in quanto consente al robot di seguire indefinitamente ogni superficie e di esplorare tutto il perimetro di una costruzione.

L'algoritmo su cui si è scelto di implementare il behaviour è equivalente a quello visto al paragrafo precedente per il *center following*. Tuttavia, mentre nel caso appena visto la distanza dal muro viene regolata in automatico, ora è necessario impostarla in modo manuale. Per fare questo si è ricorsi a un espediente matematico, modificando la formula 4.6. Detta $wall_distance$ la distanza da mantenere tra il robot e il muro e detta d_{wall_1} la distanza rilevata dal muro di riferimento, imponiamo nella 4.6 che la distanza rilevata dal muro opposto sia pari a

$$d_{wall_2} = 2 \cdot wall_distance - d_{wall_1} \quad (4.7)$$

Unendo la 4.6 con la 4.7 si ottiene

$$f_b(\theta) = \lambda_b \left\{ \begin{aligned} & [\sin(\theta - \psi_{wall_1}) e^{-c_{wall} d_{wall_1}}] + \\ & [\sin(\theta - \psi_{wall_2}) e^{-c_{wall} (2 \cdot wall_distance - d_{wall_1})}] \end{aligned} \right\} \quad (4.8)$$

Supponiamo, ad esempio, che d_{wall_1} sia la distanza dal muro alla nostra destra. Se imponiamo la 4.7, in pratica stiamo facendo credere al calcolatore che il corridoio sia largo esattamente $2 \cdot wall_distance$; in questo modo il robot, seguendo il solito behaviour di *center following*, si allinea perfettamente con il centro di questo corridoio virtuale, cioè esattamente a $wall_distance$ dal muro alla nostra destra.

Aspetti implementativi

I due behaviours indicati nei paragrafi precedenti sono stati i più usati durante la serie dei test dell'architettura. In effetti, sono stati realizzati anche altri piccoli behaviours, ma ai fini di questa trattazione risulta superfluo indicarne gli aspetti algoritmici, in quanto il loro utilizzo è servito unicamente a scopo di test dell'architettura

(capitolo 5). La figura 4.8 mostra uno schema delle classi da utilizzare nel caso di un generico behaviour.

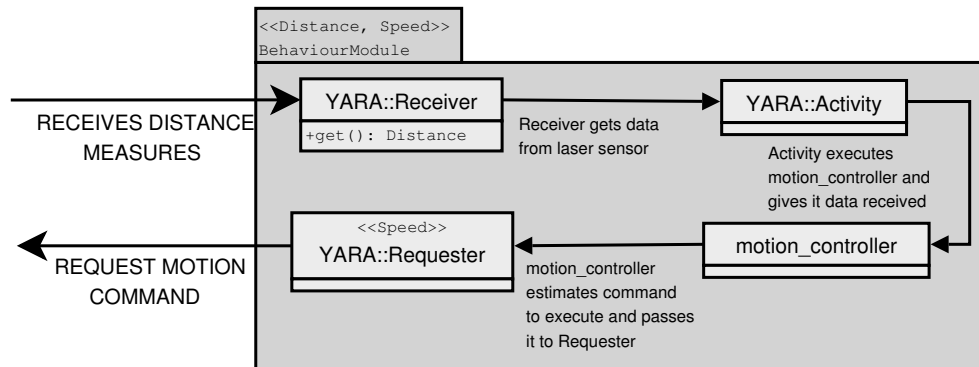


Figura 4.8: Schema delle classi di un behaviour generico.

L’algoritmo che i due behaviours visti implementano, secondo la 4.6 e la 4.8, richiede la conoscenza dell’angolo di rotazione θ del robot rispetto all’ambiente. Poichè in questo caso non possiamo fare affidamento ad altro che non sia il sensore laser, è necessario ottenere la misura dell’angolo dai suoi rilevamenti. In questo modo siamo vincolati alla precisione del laser, e siccome vogliamo sfruttare tutti i 180° che la periferica può fornire, possiamo ottenere una precisione massima dell’angolo θ pari a 0.5° .

Il risultato viene ottenuto, in entrambi i behaviours, considerando l’angolazione della misurazione minima effettuata dal laser. In questo modo otteniamo l’indice del beam corrispondente alla misurazione minima che corrisponde, a meno di qualche aggiustamento, alla rotazione del robot rispetto alla direzione del corridoio.

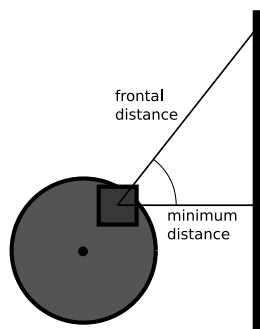


Figura 4.9: Calcolo dell’angolo di rotazione el robot.

Capitolo 5

Test del sistema base

Questo capitolo descrive l'insieme delle verifiche sperimentali che sono state effettuate per valutare il livello di *affidabilità* che è stato possibile ottenere dall'architettura software di controllo realizzata.

5.1 Test di controllo

5.1.1 Reattività

I test di reattività si pongono l'obiettivo di studiare il comportamento del sistema complessivo, con uno sguardo particolare al funzionamento del *framework* sottostante. I test che seguono sono stati condotti utilizzando entrambe le soluzioni disponibili, ovvero Smartsoft e YARA, in modo da effettuare delle misurazioni quanto più possibile oggettive e realistiche. I dettagli implementativi dei behaviours realizzati con *Smartsoft* sono disponibili in appendice [A](#).

Il primo test effettuato ha lo scopo di valutare la latenza di esecuzione di un comando di moto, che viene richiesto dall'architettura di controllo quando le letture sensoriali rilevano una condizione anomala e potenzialmente pericolosa per il robot o l'utente che si trova nelle vicinanze. La riduzione della latenza dovuta all'architettura software aumenta la reattività del robot e migliora la sua sicurezza. La prova eseguita consiste in una sorta di crash test, realizzato muovendo il robot in direzione di una parete frontale e impostando una velocità nulla agli assi del robot quando la distanza dalla parete diventa inferiore a un valore di soglia. Lo spazio

di arresto è stato misurato mediante lo scanner laser. Inoltre, si è scelto di porre a diretto confronto lo stesso behaviour eseguito dai due *framework* presentati ai paragrafi 3.2.2 e 3.2.3. In questo modo, è possibile verificare se la scelta di progettare l'intero sistema su YARA è stata positiva oppure no. La tabella 5.1 mostra i risultati dell'esperimento eseguito in assenza di carico, e traccia un confronto tra le due architetture, evidenziando una leggera superiorità a favore di Smartsoft.

| Framework | speed | η | σ | <i>min</i> | <i>max</i> |
|-----------|-------|--------|----------|------------|------------|
| YARA | 20 | 5.7 | 0.9 | 3.9 | 6.8 |
| YARA | 30 | 9.9 | 0.4 | 9.5 | 10.6 |
| YARA | 40 | 14.9 | 0.7 | 14.1 | 16.1 |
| YARA | 50 | 22.7 | 0.9 | 21.4 | 23.9 |
| Smartsoft | 20 | 3.9 | 0.4 | 3.3 | 4.7 |
| SmartSoft | 30 | 8.0 | 0.7 | 6.8 | 9.0 |
| SmartSoft | 40 | 13.4 | 0.9 | 12.3 | 15.0 |

Tabella 5.1: Spazio medio di arresto (in cm) misurato a differenti velocità (in cm/s) su 10 prove consecutive.

La ragione è da ricercarsi nel fatto che, mentre il task *real-time* è dotato di temporizzazione interna, che non è correlata con la lettura dei dati dal sensore, l'assenza di meccanismi di scheduling all'interno di Smartsoft permette di elaborare in modo immediato le letture che provengono dal dispositivo hardware, con un leggero beneficio sulla latenza media.

Si è proceduto, quindi, alla ripetizione dello stesso test simulando un carico di CPU mediante un processo a bassa priorità che esegue alternativamente fasi di *sleeping* ed di *busywait*; tale processo può essere considerato equivalente ad un task non reattivo dell'architettura, che svolga ad esempio un'attività di localizzazione o di pianificazione dei compiti. In questo caso, valutando il comportamento dell'architettura al variare del carico di disturbo (tabella 5.2), si evidenzia il beneficio in termini di robustezza e ripetibilità reso possibile dallo scheduling in tempo reale di YARA.

Si è dimostrato così che l'utilizzo di una piattaforma che implementa meccanismi di scheduling *real-time* consente di mantenere un alto grado di affidabilità impossibile da ottenere ricorrendo a tecniche differenti.

| Framework | Load | η | σ | <i>min</i> | <i>max</i> |
|-----------|------|--------|----------|------------|------------|
| YARA | ○ | 6.0 | 0.3 | 5.3 | 6.6 |
| YARA | ●● | 6.5 | 0.3 | 6.0 | 7.0 |
| Smartsoft | ○ | 5.6 | 0.4 | 4.6 | 6.0 |
| SmartSoft | ● | 6.2 | 0.5 | 5.6 | 7.1 |
| SmartSoft | ●● | 16.4 | 13.8 | 6.3 | 45.3 |

Tabella 5.2: Spazio medio di arresto (in cm) alla velocità di 25 cm/s e con differenti livelli di carico su 10 prove consecutive.

5.1.2 Navigazione

Se i test di reattività hanno permesso di dimostrare che l'impiego di una tecnica di controllo real-time garantisce una maggiore robustezza all'applicazione, le cui componenti reattive non vengono influenzate dallo stato di funzionamento delle altre parti del software, i test di navigazione illustrati in questa sezione intendono mostrare che lo scheduling *real-time* permette di migliorare anche l'accuratezza della navigazione eseguita dal robot. A tal scopo sono stati utilizzati i due behaviours proposti al paragrafo 4.2.3. Per produrre una stima della qualità del moto, invece, si è fatto uso dello stesso laser scanner utilizzato per la navigazione del robot.

Il primo test effettuato ha riguardato l'impiego del behaviour di *center following*. Il grafico in figura 5.1 mostra come lo stesso behaviour implementato sopra il *framework YARA* risulti più reattivo rispetto all'esecuzione su Smartsoft.

Dall'andamento del grafico si osserva come il valore di *set point* viene raggiunto molto più rapidamente nel caso di *YARA*, e come questo valore viene mantenuto in modo migliore per tutta la durata del test. Le misurazioni in questione non si sono avvalse di un meccanismo di *tracking* esterno, fatto che ne mette parzialmente in discussione la validità, ma sono state calcolate sulla base delle misurazioni del laser, semplicemente sottraendo la misura della distanza del *beam* destro a quella del *beam* sinistro. Il fatto che il behaviour sia stato eseguito in un ambiente strettamente rettilineo rende questo test comunque valido e significativo, in quanto la misure eseguite dal laser, e da queste le misure di distanza del grafico, soffrono solamente degli errori riportati dal dispositivo stesso.

Come ulteriore verifica, è stato condotto un test con behaviour di *wall following*. Si è preferito, in questo caso, ricondurre l'esperimento sulla base dei tempi,

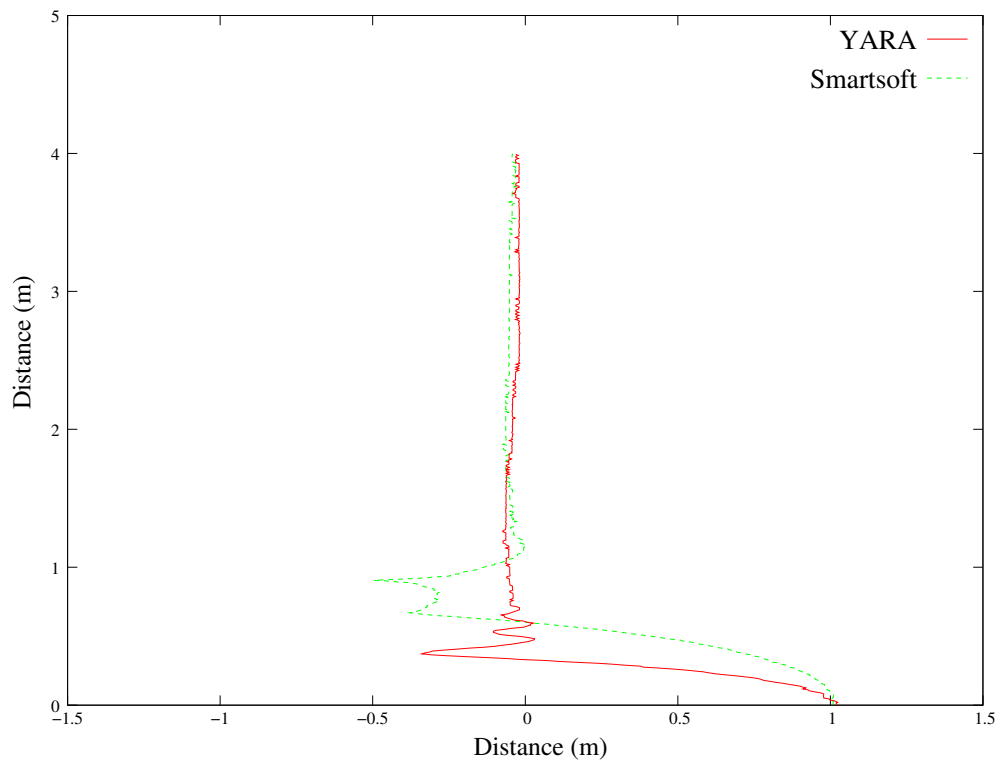


Figura 5.1: Tracking del comportamento *center following*

in modo da avere una stima anche su lunghi periodi e permettere, in questo modo, di svincolarsi dall'ipotesi di corridoio rettilineo presente nel test precedente. Il comportamento di *wall following* è stato così eseguito per una lunghezza totale di 25m, trovandosi di fronte anche a parecchi ostacoli. Il behaviour è stato impostato in modo da mantenere una distanza costante dal muro alla sua destra pari a 50cm, muovendosi alla velocità di 25cm/s.

Gli andamenti visualizzati in figura 5.2 mostrano che lo stesso algoritmo di controllo, anche in assenza di carico di disturbo, porta ad un risultato di navigazione più accurato se viene applicato tramite un'architettura che fa uso di scheduling real-time. La traiettoria disegnata dal robot controllato con YARA presenta variazioni più contenute della distanza dal muro, un comportamento meno oscillante (osservabile anche visivamente durante la navigazione) e in genere tende a svolgere il compito in modo più rapido; infatti il movimento oscillante che viene realizzato dall'applicazione non real-time tende a produrre rallentamenti frequenti, che il

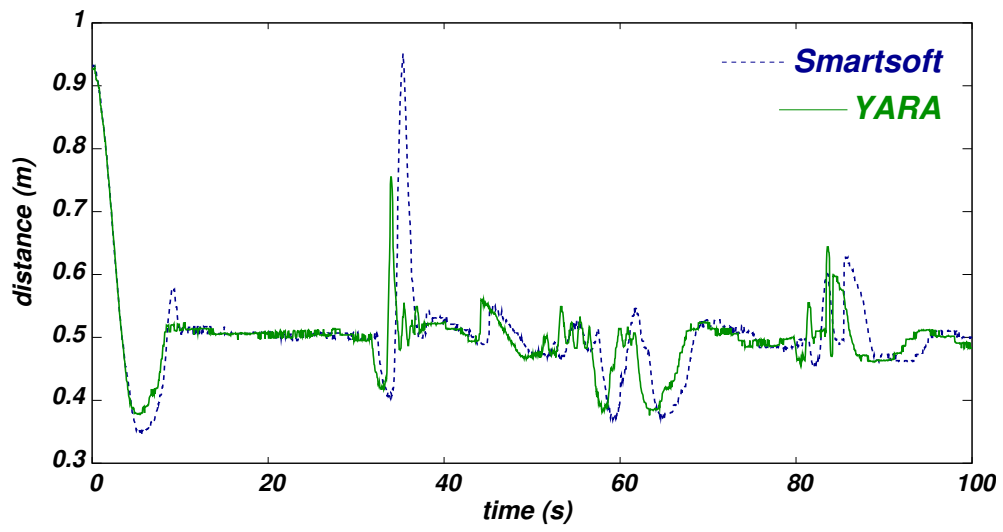


Figura 5.2: Esecuzione di un comportamento di *Wall Following* guidato dal sensore laser, usando alternativamente i framework *YARA* e *Smartsoft* in assenza di carico.

behaviour imposta quando lo spazio libero frontale scende sotto una certa soglia.

5.2 Test delle prestazioni

Noti i risultati sulla reattività, è possibile rivolgere l'attenzione a modellare il funzionamento dell'intero sistema, in modo da ottenere una architettura che sia conforme ai requisiti del localizzatore. Come evidenziato più volte in precedenza, le applicazioni di localizzazione presentano elevati requisiti computazionali che, il più delle volte, rendono impossibile il rispetto di vincoli *real-time*. Inoltre, se si aggiunge il fatto che il sistema realizzato prevede la presenza anche di altri moduli operativi (e che in futuro se ne potranno aggiungere molti altri), il carico totale dell'architettura rischia di diventare troppo pesante. Occorre, perciò, un lavoro di analisi e di riduzione, laddove possibile, del costo computazionale dei singoli elementi del sistema. Nel nostro caso, questi elementi sono rappresentati proprio dai moduli di cui si è parlato nel capitolo 4.

5.2.1 Modulo sensoriale

Il modulo del laser non richiede una analisi molto approfondita. Le sue specifiche presentano il funzionamento in tre modalità differenti, per cui, è sufficiente misurare quali sono i suoi consumi e agire di conseguenza. I test sono stati effettuati considerando l'interazione che avviene tra il modulo laser e un suo possibile ricevitore. In questo modo vengono misurati tutti i tempi provocati dall'utilizzo del modulo sensoriale ed è stato ricavato che la percentuale di utilizzo della CPU, sotto ipotesi ragionevoli di esecuzione, è veramente irrisoria, rimanendo confinata entro lo 0.5% della disponibilità totale.

Questo deriva dal fatto che le primitive di accesso alla periferica sono, in ultima analisi, delle chiamate a `read()` bloccanti. Pertanto, l'effettiva lettura dei valori avviene in pochissimi millisecondi, quelli necessari per trasferire le informazioni dal laser al *framework*. Da una serie di verifiche, atte a misurare i tempi delle scansioni, è emerso che la maggior parte del tempo richiesto viene persa nella attesa che la periferica elabori al suo interno le informazioni acquisite.

Per tutte queste ragioni, si può scegliere di impostare il periodo della attività di lettura dei dati del laser in prossimità del suo periodo di lettura fisico, rimanendo sicuri del fatto che il suo utilizzo non provoca praticamente alcun sovraccarico della

unità di calcolo. Per la precisione, il tempo dell'*activity* di lettura è stato volutamente impostato a un valore leggermente inferiore al tempo dichiarato dalla periferica; infatti, se il periodo di lettura fosse superiore al periodo di acquisizione della periferica, questo causerebbe un accumulamento di dati mai letti nel buffer (vedi 4.2.1), con la conseguenza che le informazioni recuperate possono non essere le ultime disponibili. Nel grafico seguente vengono mostrati i due possibili casi.

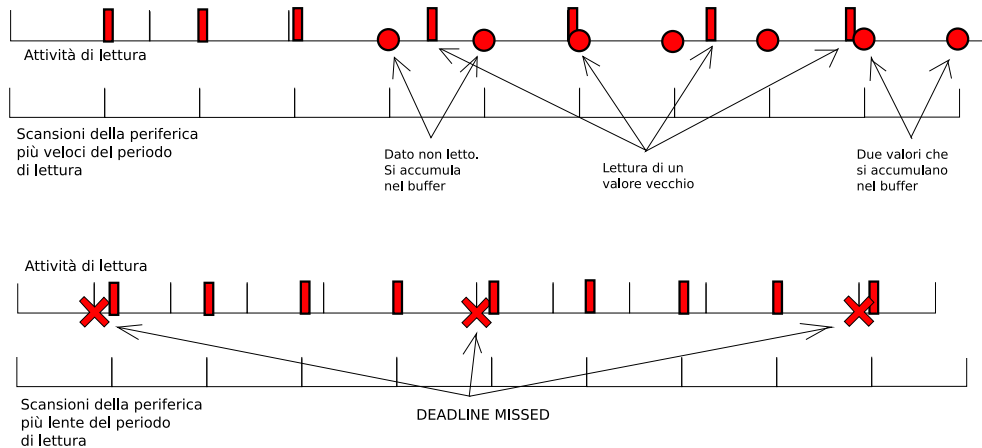


Figura 5.3: *Time line* dei due possibili casi di schedulazione dell'attività di lettura dati dal laser scanner.

Se il periodo dell'*activity* è superiore a quello del laser, allora, dopo alcune letture andate a buon fine, un dato non viene letto in tempo e viene accumulato nel buffer della seriale. Con il trascorrere del tempo, sempre più dati vengono accumulati nel buffer, con il risultato che i valori che vengono letti dall'attività si riferiscono a informazioni passate. Se invece il periodo dell'*activity* è inferiore a quello del laser, allora tutte le misurazioni vengono effettivamente recuperate alla stessa frequenza secondo cui vengono prodotte. Tuttavia, adottando questa scelta, si presenta periodicamente una *deadline*, a causa del fatto che il laser è più lento dell'attività di lettura; questa *deadline* mancata non è segno di cattivo funzionamento, poichè il valore mancato viene comunque letto immediatamente alla rischedulazione dell'attività, ovvero pochi millisecondi dopo.

5.2.2 Modulo di movimento

Lo studio del modulo di movimento ha portato a suddividerlo in due attività distinte: la prima addetta all'esecuzione dei comandi e la seconda, periodica, addetta al calcolo odometrico. In realtà, la prima attività è analoga a quella presente nel modulo laser; come si è visto al paragrafo precedente, la sua presenza causa un minimo *overhead* computazionale che può facilmente essere ignorato. Diversamente avviene per quanto riguarda il calcolo odometrico.

L'attività che controlla l'odometria è di tipo periodico e agisce sulla scheda motori per ottenere la posizione degli *encoders* istante per istante. Nonostante i calcoli relativi non siano particolarmente complessi, una eccessiva diminuzione del periodo che controlla l'attività potrebbe facilmente aumentare la quantità di risorse richieste al funzionamento del modulo.

I test svolti sono stati ripetuti al fine di stimare in che modo la variazione del periodo dell'attività influisse sul totale utilizzo della CPU da parte del modulo. Nel grafico 5.4 si vede come questa relazione sia rappresentata da una curva tendente al 100% se il periodo tende a essere eccessivamente piccolo. In realtà, per periodi molto ridotti, il sistema viene sovraccaricato causando un cattivo funzionamento di tutta l'architettura.

La tabella 5.3 mostra qualcuno dei dati espressi graficamente.

| Periodo (ms) | Utilizzo cpu (%) |
|--------------|------------------|
| 0.300 | 2.5 |
| 0.200 | 3.5 |
| 0.100 | 7 |
| 0.075 | 9 |
| 0.050 | 14 |
| 0.025 | 28 |
| 0.010 | 70 |
| 0.009 | 78 |
| 0.008 | 90 |

Tabella 5.3: Percentuale di utilizzo di CPU da parte del modulo motori, al variare del periodo di integrazione.

I risultati riportano, con una certa sorpresa, che il semplice calcolo odometrico, a una frequenza di 20Hz, consuma il 14% della CPU totale di sistema. È una

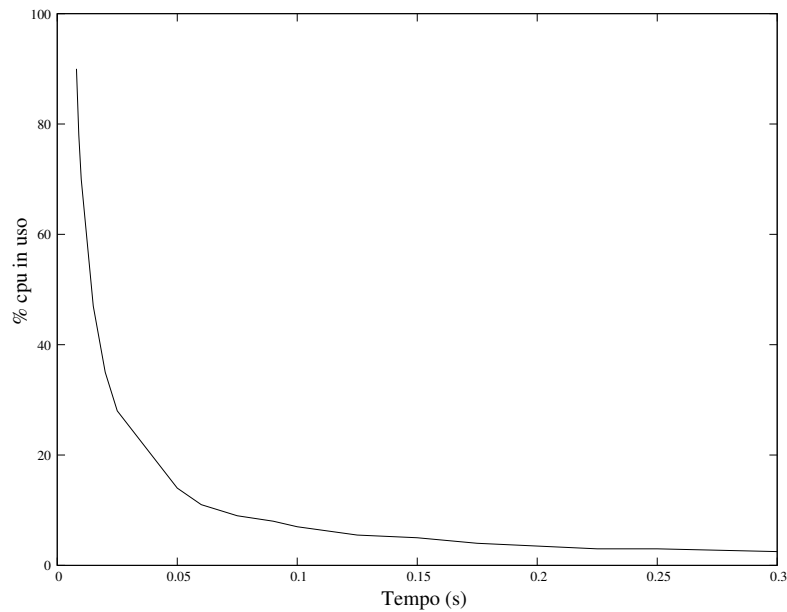


Figura 5.4: Relazione tra il periodo dell'attività di calcolo odometrico e l'uso dell'unità di calcolo del modulo motore.

cifra davvero importante se si pensa che il calcolo odometrico è espresso, nel caso peggiore, da

$$\begin{pmatrix} X_1 \\ Y_1 \end{pmatrix} = \begin{pmatrix} X_0 \\ Y_0 \end{pmatrix} + \left[\begin{pmatrix} -\sin \theta_0 \\ \cos \theta_0 \end{pmatrix} + \begin{pmatrix} \sin \theta_1 \\ -\cos \theta_1 \end{pmatrix} \right] R$$

Una serie successiva di test ha messo in luce come gli alti tempi richiesti sono causati dall'utilizzo della scheda DMC 630. Collocandosi sul bus ISA, la scheda mantiene la CPU in utilizzo *busy wait* durante ognuna delle sue operazioni; il calcolo odometrico, infatti, richiede che la scheda ritorni i valori letti dagli *encoders* al fine di produrre una stima del movimento, e proprio in quel punto l'attività rimane bloccata in attesa attiva, consumando inutilmente parecchi cicli di CPU.

Il *tuning* del periodo di calcolo odometrico può giocare un ruolo importantissimo al fine di stabilire quali possono essere le prestazioni del localizzatore. Infatti, mentre nell'ultimo tratto la curva vista in figura 5.4 si comporta quasi come un esponenziale, per periodi più alti è praticamente lineare. In questo modo, raddoppiando il periodo di integrazione, ad esempio da 50ms a 100ms, possiamo risparmiare circa il 7% di CPU.

Tuttavia, se vogliamo effettivamente permetterci di aumentare il periodo di integrazione, dobbiamo valutare in che quantità ne andremo a perdere dal punto di vista della precisione numerica. Un aumento sproporzionato del periodo potrebbe portare, infatti, a una perdita considerevole di informazioni, risparmiando sì sul tempo di localizzazione, ma andando a compromettere i dati in modo irrecuperabile.

Per effettuare una stima dell'errore commesso, consideriamo il caso peggiore in cui la misurazione viene sempre prodotta attraverso il metodo lineare. Per peggiorare ulteriormente le misure, si è impostato il robot alla massima velocità possibile, dove il metodo linearizzante soffre in modo particolare di errori di precisione. Abbiamo impostato al robot una traiettoria circolare, in modo che il metodo del secondo ordine risulti esatto; quello che è emerso dai test ha mostrato che, a un periodo di 50ms, l'errore massimo commesso dalla stima linearizzante è pari a $4 \cdot 10^{-5}$ m. Riscalando l'errore in un periodo di 100ms valutiamo l'errore totale commesso in $8 \cdot 10^{-5}$ m. Eseguendo lo stesso test con un periodo pari a 100ms si ottiene la stima dell'errore di linearizzazione massimo pari a $3 \cdot 10^{-4}$ m. L'errore prodotto è praticamente di un ordine di grandezza superiore, tuttavia si mantiene al di sotto della soglia del millimetro; la periferica laser utilizzata per produrre le misure sensoriali, infatti, non può produrre una stima delle distanze così raffinata, perciò, anche se il calcolo odometrico risulta così inaccurato, non incide per niente sulla probabilità di localizzare al meglio il robot. La scelta di impostare il periodo del calcolo odometrico a 100ms risulta, quindi, efficiente, permettendo di risparmiare ben il 7% della CPU totale disponibile.

5.2.3 Moduli comportamentali

I moduli comportamentali costituiscono la base di movimento necessaria per ottenere una stima di localizzazione. La loro struttura prevede il calcolo delle velocità da impostare sulla base dei valori letti dalla periferica laser. Analogamente a quanto visto in precedenza, la lettura dei valori sensoriali è stata stimata essere praticamente nulla al paragrafo 5.2.1; in questo modo, possiamo occuparci esclusivamente della parte di controllo motori, essendo questa l'unica che può sovraccaricare la CPU.

In effetti, si presenta una situazione simile a quella vista per il calcolo odometrico: un periodo troppo basso è destinato a far aumentare di molto il carico sulla CPU,

poichè l'impostazione dei comandi a motore è pilotato sempre dalla scheda DMC 630, la quale non prevede l'utilizzo di primitive alternative a quelle *busy waiting*.

I test svolti si sono concentrati sul behaviour di *wall following* ma sono pienamente compatibili con tutti gli altri comportamenti, perchè l'architettura di comunicazione verso la scheda motori è sempre la stessa. A seguito di una serie di prove ripetute, si è configurato il valore del periodo dell'attività di controllo in modo da produrre un grafico che stimasse quale fosse la percentuale utilizzata. Il risultato è riportato in figura 5.5.

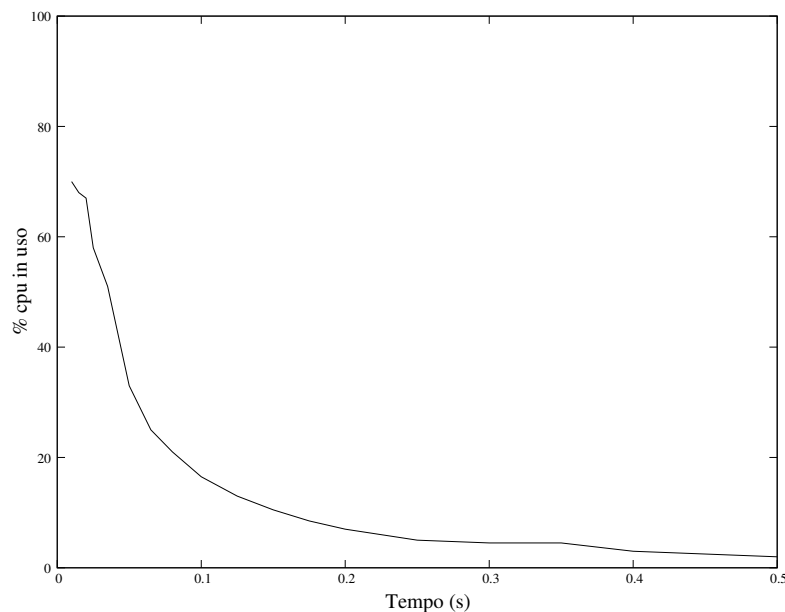


Figura 5.5: Relazione tra il periodo dell'attività di controllo di un modulo comportamentale e l'uso dell'unità di calcolo.

La tabella 5.4 riporta inoltre alcuni dei valori presenti in figura.

A 10ms si ha la massima frequenza possibile, oltre la quale il sistema diventa instabile a causa del sovraccarico computazionale. Similmente a quanto visto in precedenza, la curva si comporta in modo più lineare quando il periodo tende ad allungarsi. In questo caso, però, ciò che conta ai fini della localizzazione è che il comportamento funzioni; non importa se il controllo non è ai massimi della reattività, ma l'importante è che il robot riesca nel compito di navigazione assegnatogli. Per questo motivo, dopo alcuni test mirati a valutare la sicurezza dell'applicazione, si è impostato il periodo del behaviour sui 500ms: il comportamento è ancora in

| Periodo (ms) | Utilizzo cpu (%) |
|---------------------|-------------------------|
| 0.010 | 70 |
| 0.050 | 33 |
| 0.100 | 16.5 |
| 0.200 | 7 |
| 0.300 | 4.5 |
| 0.400 | 3 |
| 0.500 | 2 |

Tabella 5.4: Percentuale di utilizzo di CPU da parte dei moduli comportamentali al variare del periodo di controllo.

grado di arrestarsi qualora si presenti davanti al robot un ostacolo improvviso, tuttavia, non mostra quella reattività che si vede chiaramente qualora si imposti il valore del periodo a 50ms, cosa che accadeva nei test di reattività visti in [5.1.1](#). In questo modo si ottiene un modulo comportamentale che riesce nell'intento di far navigare il robot in un ambiente, occupando solamente il 2% della CPU disponibile.

Capitolo 6

Integrazione del sistema di controllo con il localizzatore

6.1 Integrazione del sistema con LSOFT

La fase conclusiva del progetto è l'integrazione del software che implementa gli algoritmi di localizzazione nell'architettura di controllo del robot.

Il procedimento di integrazione deve prevedere la progettazione di un modulo aggiuntivo atto a pilotare le funzioni del localizzatore e a coordinarne l'esecuzione con i restanti moduli del sistema.

6.1.1 Il progetto LSOFT

Il software di localizzazione su cui si è basata l'implementazione di questo sistema è stato sviluppato presso il Dipartimento di Ingegneria dell'Informazione dell'Università degli studi di Parma, e prende il nome di LSOFT [11, 21].

Si tratta di una libreria che fornisce le classi necessarie all'esecuzione di un filtro particellare, anche nella sua versione *real time* introdotta al paragrafo 2.2.3. Il software fornisce anche gli strumenti per visualizzare il risultato dell'elaborazione, in modo da rendere disponibile un *feedback* rapido sulle prestazioni in termini di accuratezza della localizzazione.

L'intero progetto è stato realizzato secondo i moderni paradigmi della programmazione a template, spingendo al massimo la riusabilità del codice; risulta relati-

vamente facile accedere al contenuto della libreria e realizzare qualche modifica, anche sostanziale. In particolare, viene ampiamente utilizzato il *pattern policy* [22].

L'idea fondamentale è quella di decomporre le classi che devono svolgere compiti complessi, potenzialmente in vari modi, in tante politiche (*policy*). Una politica definisce allora un'interfaccia che nelle sue varie istanze, le *policy class*, corrisponde a comportamenti differenti. Una classe derivata può acquisire quei comportamenti o ereditando dalle classi *policy* o trasformandole in parametri template. Ogni *policy*, che è di per sé un'interfaccia astratta, è contemporaneamente parametro template e classe base della classe risultante. L'esempio in figura 6.1 aiuta a comprendere meglio.

```
1 template<class T, template<class> class Policy>
2 class Derived : public Policy<T>
3 {
4     //...
5     void doSomething(){
6         //...
7         T t = operation();
8     }
9 };
10
11 template<T>
12 class PolicyImplA { T operation(); };
13
14 template<T>
15 class PolicyImplB { T operation(); };
```

Listato 6.1: Esempio di classe *policy*.

Si osservi che tramite una *policy* diventa possibile gestire facilmente sia la combinazione delle politiche sia l'adattamento al contesto. La classe `Derived` impone a `Policy` il metodo `operation()`, utilizzando il quale ne controlla la parametrizzazione secondo le proprie esigenze.

Questo metodo di implementare algoritmi soffre, purtroppo, di una limitazione che può risultare fastidiosa. Come si fa notare in [23], la programmazione generica in C++ introduce dei vincoli impliciti sull'interfaccia. Mentre con la usuale pro-

grammazione a oggetti l'interfaccia della classe è ben esposta, in questo caso non c'è modo di indicare al programmatore quali siano i metodi che una classe deve implementare per essere aderente a un determinato *pattern*; il controllo avviene, chiaramente, in fase di compilazione, tuttavia, non è sempre immediato risalire alla causa dell'errore e correggerlo.

6.1.1.1 Organizzazione della libreria

In questo paragrafo viene presentata la libreria *LSOFT* e la sua organizzazione complessiva. Durante la sua progettazione è stata fatta una suddivisione ben precisa, atta a distinguere in modo molto netto le varie parti che compongono il sistema. Queste parti, che riprendono necessariamente il meccanismo di funzionamento del *particle filter*, sono quattro e riguardano il modello dinamico, il modello sensoriale, la mappa e il localizzatore.

Il modello dinamico

Il modello dinamico rappresenta la regola che viene seguita al fine di effettuare una predizione dello stato del sistema.

Ai fini pratici, la classe che implementa questo modello è la `SystemModel`; essa ha quindi la responsabilità di tradurre stato e controlli in forma vettoriale, di impiegarli per aggiornare lo stato, di simulare la casualità del rumore e di riconvertire il vettore nel formato dello stato. La sua interfaccia è riportata nel listato [6.2](#):

Si noti il metodo `update()` che consente di aggiornare le informazioni dello stato sulla base del controllo u .

Il modello sensoriale

La classe `SensorModel` è analoga alla precedente, in quanto codifica le informazioni relative alla fase di correzione del filtro particellare. Per questo motivo, sono state adottate le stesse tecniche di codifica, producendo una classe dall'interfaccia seguente

Il metodo `updateWeight()` viene utilizzato durante la fase di correzione per ricalcolare il peso di ogni campione in modo corretto (listato [6.3](#)).

```
1 template
2 <
3   class State = boost::numeric::ublas::vector<double>,
4   class Control = boost::numeric::ublas::vector<double>,
5   class Model = WMRLinearModel
6 >
7 class SystemModel {
8   public:
9     SystemModel(Pdf *noise,
10                unsigned int stateDimension,
11                unsigned int controlDimension);
12
13   void update(State &x,const Control &u);
```

Listato 6.2: Interfaccia della classe SystemModel.

La mappa

La mappa è l'area di rappresentazione dei risultati, ma soprattutto, è un contenitore di dati che immagazzina le informazioni relative al nostro particolare ambiente. Gli aspetti rilevanti della mappa sono il tipo di dato a cui si accede e la modalità di accesso. Possiamo pensare alla mappa come ad una memoria associativa in cui la chiave è costituita da un vettore spaziale, mentre il valore può essere definito a seconda dell'uso che ne facciamo.

A costruttore devono essere specificate le dimensioni della mappa e la sua risoluzione, nonché il valore dell'elemento di default.

Il localizzatore

Visto che le classi presentate finora rispondono già in maniera adeguata a quanto è richiesto a una libreria di localizzazione, la classe `Localizer` si propone come un loro coordinatore. Di conseguenza, l'interfaccia della classe è piuttosto limitata in quanto espone esclusivamente i metodi per avere accesso alle funzioni che le altre classi implementano.

Il listato 6.4 mostra che la classe `Localizer` possiede diversi parametri template che ne parametrizzano il comportamento. In particolare, vengono definiti tutti gli

```
1  template
2  <
3    class State,
4    class SensorData = MultiMeasure,
5    template <class> class RV = RangeRandomVariable,
6    template <class> class Fusion = IndependenceFusion,
7    template <class> class StateConverter = Olonomic2DStateConverter
8  >
9  class SensorModel : public RV<SensorData>,
10                      private StateConverter<State>
11  {
12  public:
13    SensorModel(Pdf *pdf, MapSet &ms);
14
15    double updateWeight(const SensorData &z,
16                       State &x,
17                       Fusion<SensorData> &fusion,
18                       double w);
```

Listato 6.3: Interfaccia della classe SensorModel.

elementi che compongono il localizzatore; vediamo di esplicitare qual'è il loro tipo.

- **State:** Rappresenta lo stato del robot, ovvero, la sua posizione e il suo orientamento. Per semplicità e comodità di utilizzo, è stata usata nella maggior parte dei casi la classe `boost::numeric::ublas::vector` [24], sulla quale è possibile svolgere le basilari operazioni vettoriali.
- **SensorData:** Questa classe rappresenta i dati sensoriali che vengono acquisiti. Durante la fase di correzione, infatti, è necessario che il sistema ottenga una stima della osservazioni effettuate dal robot.
- **Control:** Rappresenta il controllo u_t utilizzato per muovere il robot, ovvero, la stima della differenza tra la posizione iniziale e quella ottenuta dopo l'applicazione del controllo. Similmente alla classe `State` è stato scelto di utilizzare la classe `ublas::vector`.

```
1 template
2 <
3   unsigned int D,
4   class State,
5   class SensorData,
6   class Control,
7   class SampleCounter,
8   template <unsigned int, class,class,class> class SampleManager,
9   template <class> class Fusion = IndependenceFusion
10 >
11 class Localizer : public SampleManager<D,State,Control,SampleCounter> {
12   public:
13     Localizer(Pdf &pdf,
14               unsigned int min_num_samples,
15               unsigned int max_num_samples,
16               boost::numeric::ublas::vector<double> &res);
17
18   template<class SystemT>
19   void update(Control &u, SystemT &sys);
20
21   template<class SensorT>
22   void update(const SensorData &z, SensorT &sense);
23 };
```

Listato 6.4: La classe Localizer.

- **SampleCounter:** Questa classe è stata pensata per sviluppi futuri della libreria. Attualmente, rappresenta il contenitore del numero dei campioni totali.
- **SampleManager:** Questa classe fornisce l'accesso ai campioni delle distribuzioni. In particolare, è la classe addetta alla fase di ricampionamento dei campioni.
- **Fusion:** Questa classe consente di determinare la politica di fusione delle informazioni sensoriali.

Possiamo cercare di astrarre, da questa breve analisi del software esistente, quali sono le informazioni utili al fine di integrare il software *LSOFT* con la nostra architettura di controllo.

6.1.2 Il modulo Localizer

Interfacciamento Nomad-LSOFT

Come si è visto, LSOFT non fornisce nessun localizzatore preconfezionato, ma rende disponibili le classi di base tramite le quali possiamo assemblarlo. Per ottenere una applicazioni funzionante si rende necessaria la creazione di un *system model*, di un *sensor model*, della mappa e, infine, della classe localizzatore che serve per unire il tutto.

Per il nostro scopo, risulterebbe utile avere una classe che contenga tutto il necessario, in modo da poterla utilizzare all'esterno senza occuparci dei dettagli implementativi. Per questo motivo, si è scelto di implementare una classe *LocalizerContainer* che raccoglie al suo interno l'insieme delle classi necessarie al corretto funzionamento del localizzatore. L'interfaccia di questa classe deve essere tale da consentire l'accesso agli elementi che compongono il localizzatore, ovvero, il *system model*, il *sensor model* e la classe *Localizer* (listato 6.5).

```
1 typedef LSOFT::SystemModel< ... > system_type;
2 typedef LSOFT::SensorModel< ... > sensor_type;
3 typedef LSOFT::Localizer< ... > localizer_type;
4
5 class LocalizerContainer {
6
7     public:
8     LocalizerContainer(std::string config_file);
9
10    system_type& getSystemModel();
11    sensor_type& getSensorModel();
12    localizer_type& getLocalizer();
13 };
```

Listato 6.5: Interfaccia della classe LocalizerContainer.

L'interfaccia della classe mostra in che modo il *LocalizerContainer* fornisce l'accesso agli elementi del localizzatore; l'utilizzatore della classe deve semplicemente istanziare un oggetto di questo tipo e chiamarne i tre metodi *getter*, i quali restituiscono una reference agli oggetti che compongono la libreria *LSOFT*, mante-

nendone il contenuto all'interno della classe. Chiaramente, questo oggetto mantiene anche le informazioni sulla mappa, ma le contiene tutte in modo privato, in quanto l'accesso dall'esterno non è richiesto.

A questo punto, possiamo raggruppare tutte le informazioni che abbiamo raccolto per tracciare una panoramica completa e interfacciare la libreria di localizzazione con il robot di riferimento. Come già discusso il software di localizzazione esegue il suo algoritmo alternando due fasi fondamentali: predizione, correzione. Ciascuna di queste non può essere svolta senza la presenza dei dati che riguardano il suo algoritmo. Nel caso della predizione, stiamo applicando il controllo u_t al modello dinamico, quindi, ciò che ci serve è una informazione odometrica riguardante il movimento del robot. Invece, la fase di correzione sfrutta le informazioni sensoriali z_t per migliorare la stima della posizione: questo non può essere fatto che con l'invio delle informazioni acquisite dallo scanner laser.

Predisponiamo, allora, una classe ulteriore al fine di facilitare la comunicazione tra il localizzatore e l'architettura di controllo. Questa classe è sicuramente fortemente legata alla struttura del Nomad, in quanto deve servire a interfacciare i dati provenienti dai sensori e attuatori del robot con la libreria di localizzazione. Il listato 6.6 mostra la semplice interfaccia di questa classe.

```
1 class NomadLocalizer {
2
3   public:
4   typedef double weight_type;
5   typedef std::pair<motion::Pose, weight_type> location_type;
6
7   NomadLocalizer(std::string config_file);
8
9   void updateSystem(const motion::Pose& pose, std::list<location_type>& list);
10  void updateSensor(const std::vector<double>& ranges);
11
12  private:
13   LocalizerContainer container;
14 };
```

Listato 6.6: Interfaccia della classe NomadLocalizer.

Come si vede, sono esposti unicamente due metodi che interfacciano i due ambienti software. Questi metodi vengono utilizzati per l'aggiornamento del modello di sistema in una delle due fasi principali del *particle filter*. Notare anche la relazione di contenimento tra il *NomadLocalizer* e il *LocalizerContainer*.

Il metodo `updateSystem()` viene usato per aggiornare le informazioni del sistema dinamico. Ricordando che l'attività di calcolo odometrico è indipendente dal resto del sistema e che produce una stima ogni T_{odom} secondi, passiamo in ingresso al metodo la stima del movimento effettuato, calcolabile come somma vettoriale di tutti gli spostamenti rilevati dal calcolo odometrico. Inoltre, al fine di avere in uscita una stima posizionale, questo metodo riempie una lista con tutte le possibili ipotesi di localizzazione.

Una cosa simile avviene per il metodo `updateSensor()`; in questo caso l'elemento da passare è costituito da un vettore contenente le scansioni del laser. Queste sono contenute dentro un `std::vector<double>` e, prima di poter essere utilizzate, devono essere convertite in un tipo consono al localizzatore: in questo caso dobbiamo usare la classe *ScanMeasure* di *LSOFT*. Oltre a presentare un contenitore per le distanze misurate, la *ScanMeasure* contiene anche alcune informazioni riguardanti la varianza dell'osservazione e la posizione relativa del sensore. Questi parametri vengono utilizzati per posizionare spazialmente la misurazione e per stimare quale può essere l'errore commesso. Nel nostro caso, il sensore laser è stato posizionato a 14cm dal centro del Nomad, mentre la varianza del sensore è stata stimata essere circa 1cm.

L'insieme delle due classi *NomadLocalizer* e *LocalizerContainer* ci dà un pieno accesso alla libreria di localizzazione. Sarebbe stato sufficiente implementare un'unica classe che incorporasse tutti gli elementi necessari al funzionamento ma, in questo modo, risulta relativamente facile riutilizzare la classe *LocalizerContainer* con la sola modifica dei metodi dell'oggetto *NomadLocalizer* e l'adattamento a uno specifico robot.

Progettazione del modulo

La figura 4.1 ha mostrato una visione di insieme del sistema che stiamo costruendo. Tralasciando, per il momento, il *Supervisor*, l'ultimo tassello che manca è il *modulo*

Localizer.

Possiamo impostare l'algoritmo di localizzazione su due *Activity* differenti, ognuna che si occupa di una singola fase. Si delineano, in questo modo, due attività distinte che, potenzialmente, possono concorrere tra di loro. Tuttavia, nella letteratura di settore [5, 6] queste due fasi vengono sempre descritte in modo che l'una segua sempre l'altra, ovvero, a seguito della fase di predizione avviene la correzione. La libreria di cui ci siamo occupati, tuttavia, non impone nessun vincolo di questo tipo; le chiamate ai metodi `update` e `()` della classe del localizzatore sono, infatti, completamente asincrone e indipendenti l'una dall'altra. Teoricamente, sarebbe possibile eseguirle in parallelo, anche se questo approccio potrebbe risultare in una instabilità del sistema. Tralasciando questa ultima ipotesi, si è pensato di non vincolare in nessun modo l'ordine temporale di esecuzione delle due attività, ma di lasciare un *hot-spot* configurabile che permetta di selezionare, in modo indipendente, quali sono i tempi di *update* delle due fasi.

In questo modo, a fianco del solito paradigma di esecuzione sequenziale delle due fasi, si può prevedere che il localizzatore esegua, ad esempio, due fasi di predizione per ogni fase di correzione.

La struttura del modulo si delinea, fin da subito, con la presenza di queste due *Activity*. È interessante notare che un modulo *YARA* gestisce l'esecuzione delle sue attività in modalità monitor, escludendo, a livello del *framework*, l'esecuzione contemporanea di due metodi non costanti; in questo modo, siamo svincolati dalla preoccupazione di sincronizzare le due attività in modo che non vengano eseguite in parallelo.

Oltre alle due attività, è necessario che il modulo si predisponga alla ricezione dei dati odometrici e sensoriali: come già fatto in precedenza, è richiesto, quindi, un *Receiver* per ogni dato.

La figura 6.1 mostra la struttura a classi del modulo. Il disaccoppiamento di cui si è parlato viene raggiunto tramite la diversa impostazione dei periodi delle attività. Il modulo, infatti, riceve in modo indipendente le misure sensoriali e odometriche e solo le attività possono decidere se utilizzare quelle informazioni, oppure se attendere le successive.

Si riportano, per completezza, anche alcuni dettagli implementativi della classe. Alle righe 4 e 5 viene dichiarato il periodo delle attività; sulla base di questo valore

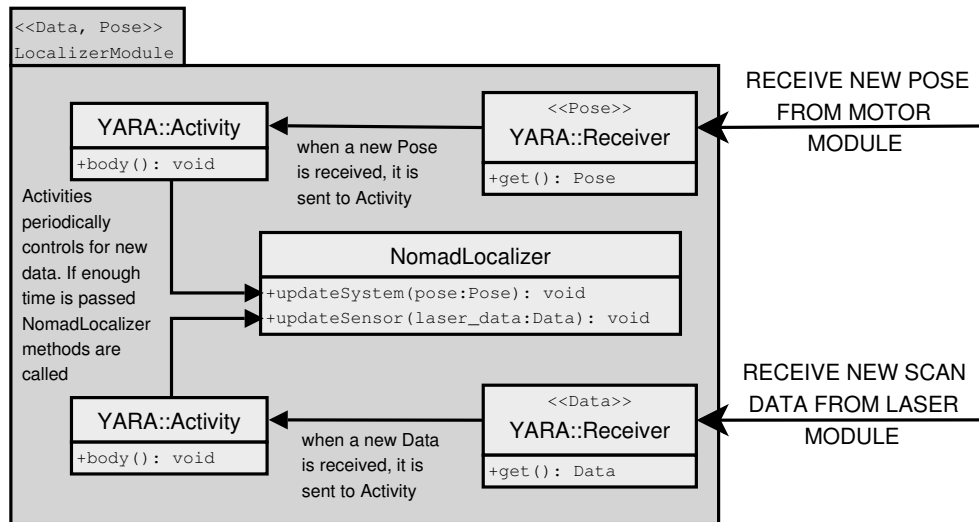


Figura 6.1: Schema a blocchi del modulo Localizer.

è possibile regolare in quale proporzione vengono eseguite le fasi di predizione e correzione.

6.1.3 Verifica delle prestazioni

Individuazione del problema

La sperimentazione condotta ha consentito di verificare che l'architettura è in grado di far funzionare il localizzatore in tempo reale. A differenza della libreria pre-esistente, la presenza dell'infrastruttura real time di controllo impone dei vincoli temporali che il localizzatore deve essere in grado di rispettare. Si è pertanto indagato quali fossero le prestazioni correnti della libreria di localizzazione, in modo da avere una stima dei tempi massimi di esecuzione.

Con qualche sorpresa, si è rilevato che i tempi di elaborazione risultano crescenti con il procedere della localizzazione, arrivando anche a occupare parecchi secondi; il funzionamento in tempo reale ne risulta evidentemente compresso.

Si è cercato allora di individuare quale fosse la causa di questo malfunzionamento. I dati di *profiling*, in particolare, hanno evidenziato come la fase più costosa sia quella della predizione. Il grafico in figura 6.2 mostra l'andamento dei tempi, espresso in millisecondi, delle due fasi del localizzatore.

```
1 class LocalizerModule : public YARA::Module {
2
3   public:
4     static const double CORRECTION_PERIOD;
5     static const double PREDICTION_PERIOD;
6
7     LocalizerModule(YARA::Unit& u, std::string config_file = "localizer.cfg");
8
9     void start();
10    void stop();
11
12   private:
13     // LOCALIZER
14     NomadLocalizer localizer;
15
16     // LASER MEMBERS
17     YARA::Receiver< std::vector<double> > laser_receiver;
18     YARA::Activity<void, void> laser_receiver_activity;
19
20     // MOTOR MEMBERS
21     YARA::Receiver< motion::Pose > motor_receiver;
22     YARA::Activity<void, void> motor_receiver_activity;
23 };
```

Listato 6.7: Interfaccia della classe LocalizerModule.

Il tempo di esecuzione della fase di predizione esplode dopo poche iterazioni, e si assesta solamente dopo il decimo ciclo. A questo punto, però, i tempi sono ormai diventati troppo elevati e il sistema è definitivamente compromesso.

La fase di predizione è costituita, praticamente, solo dall'algoritmo di ricampionamento [1](#), la cui realizzazione è riportata nel listato [6.8](#).

L'algoritmo riprende quanto visto in precedenza [\[25\]](#) e non presenta particolare difficoltà. Sostanzialmente, scorre tra tutti i campioni disponibili, ricalcola quante copie del campione corrente deve inserire nel set S_t e, infine, aggiunge il campione. L'unico punto critico sembra essere la fase di inserimento (riga 19). L'inserimento viene fatto tramite l'utilizzo di un *insert iterator* che viene passato come parametro. La chiamata effettiva viene ricondotta alla classe *ClusterContainer*.

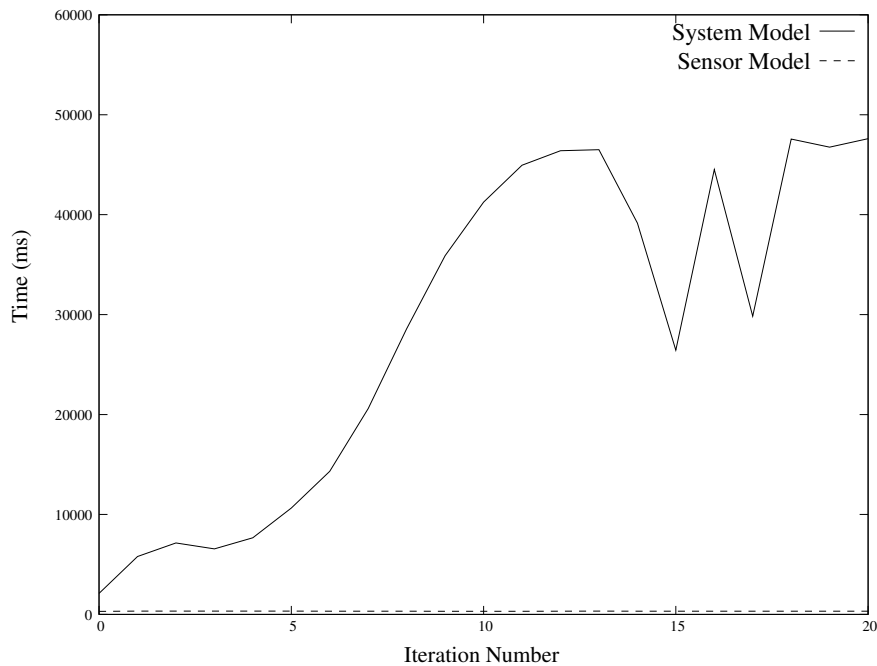


Figura 6.2: Tempi di esecuzione del system model e del sensor model.

Clustering

Per clustering si intende *aggregazione di elementi*. Il filtro particellare riesce nell'intento di selezionare quali sono i campioni più probabili a rappresentare la traiettoria del robot, tuttavia, ragionando in termini di migliaia di campioni, risulta necessario aggregare i campioni più vicini in modo da ottenere una stima unica della posizione.

Durante la fase di ricampionamento i campioni vengono inseriti nella distribuzione in modo indipendente l'uno dall'altro; se non teniamo traccia della loro vicinanza non possiamo in nessun modo ottenere l'informazione posizionale desiderata.

La classe `ClusterContainer` svolge la funzione di aggregare tra di loro i campioni più vicini. Al suo interno, i campioni sono mantenuti in un *KD-Tree container*. Ogni volta che viene chiamato il metodo `insert()` che provvede all'inserimento, viene avviato un algoritmo di ricerca nel *KD-Tree* per i vicini del campione inserito.

L'algoritmo di clustering utilizzato è presentato nell'algoritmo 3. Se il campione non trova alcun vicino, significa che è isolato e procede alla creazione di un nuovo

```
1 template<class SrcIterator,class InserterIterator>
2 inline void resample_rsr(SrcIterator begin,
3                         SrcIterator end,
4                         InserterIterator inserter,
5                         unsigned int n,
6                         double seed,
7                         double sum = 1.0)
8 {
9     double d = seed;
10    SrcIterator iter = begin;
11    typename SrcIterator::value_type sample;
12
13    for (; iter != end; ++iter){
14        // c is the number of copies of the current sample inserted in resampling set
15        int c = (int)floor( n * (iter->getWeight() / sum - d) ) + 1;
16        sample = *iter;
17        sample.setWeight(1.0);
18
19        for (int i = 0; i < c; ++i) inserter = sample;
20
21        d = d + (double)c / n - iter->getWeight() / sum;
22    }
23 }
```

Listato 6.8: Algoritmo di ricampionamento.

cluster di cui è l'unico membro. Se invece trova qualche vicino, allora, alla prima iterazione si unisce a un cluster già creato mentre, dalla seconda in poi, effettua la fusione tra i due cluster differenti. Il sistema delle etichette serve proprio a riconoscere a quale cluster appartengono i campioni; ogni volta che un campione viene inserito nella distribuzione esso viene etichettato con un numero che rappresenta il suo cluster. Poiché un cluster rappresenta, in ultima analisi, una stima posizionale, la libreria mantiene in memoria una mappa dove vengono memorizzati quali sono i cluster correnti.

Abbiamo effettuato una stima dei costi che l'algoritmo richiede. Si è visto che tutto il tempo richiesto dall'algoritmo è utilizzato nella fase di ricerca dei vicini (riga 2). Tracciando un grafico al variare del numero delle iterazioni (Figura 6.3) si

Algorithm 3 Algoritmo di clustering

Require: Un container per i campioni, una lista di cluster, campione s in ingresso

- 1: Inserisci il campione s nel container
 - 2: Ricerca i vicini del campione s entro un range di 1.5 celle
 - 3: **if** (s non ha vicini) **then**
 - 4: Crea un nuovo cluster con solo il campione s
 - 5: Aggiungi il cluster all'elenco dei cluster
 - 6: **else**
 - 7: **for all** *vicino* tra i vicini trovati **do**
 - 8: **if** s non è stato etichettato **then**
 - 9: Etichetta s con l'etichetta del vicino
 - 10: **else**
 - 11: Effettua il join tra i due cluster
 - 12: **end if**
 - 13: **end for**
 - 14: **end if**
-

ottiene un andamento praticamente uguale a quello visto per il *system model* (Figura 6.2).

Confrontando il grafico in figura 6.2 con quello in figura 6.3, si vede come quasi tutto il tempo richiesto dall'algoritmo di ricampionamento venga perso durante questa ricerca. Conteggiando il numero di accessi al container dei campioni per ogni iterazione, si rileva che questo numero esplose nel giro di qualche iterazione e si porta a quota 2 milioni. Questo risultato è motivato dal fatto che allo scorrere del tempo i campioni tendono ad ammassarsi attorno all'ipotesi più probabile, e quindi tendono a formare un solo cluster. Tuttavia, l'algoritmo 3 mostra che ogni volta che viene rilevata la presenza di qualche vicino, si deve effettuare una serie di controlli di etichette, al fine di stabilire a quale cluster aggiungere il campione. Se la distribuzione dei campioni è ammassata attorno a un'unica ipotesi, questo porta a svolgere un numero elevatissimo di confronti inutili.

Il problema risiede nella modalità con cui si effettuano i controlli di vicinanza; infatti, i controlli vengono ripetuti per ogni singolo campione inserito, anche se in quella posizione è già stato inserito in precedenza un altro campione.

A ulteriore riprova di quanto detto, se si grafica il numero di accessi al container dei campioni, si ottiene un grafico del tutto identico ai precedenti (Figura 6.4).

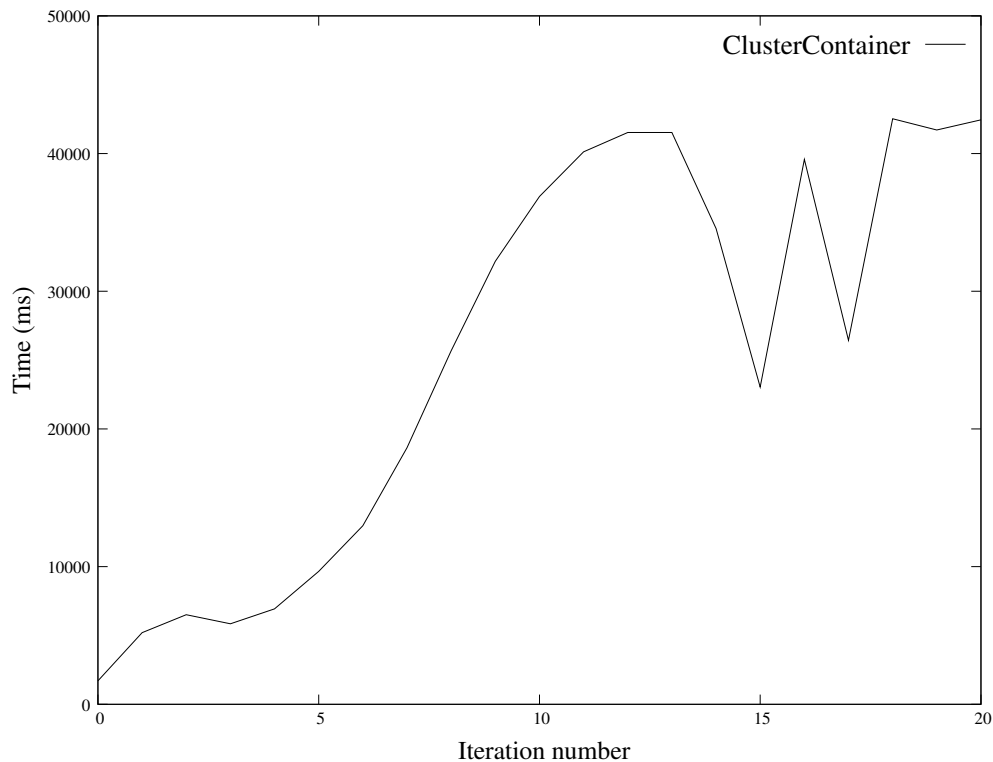


Figura 6.3: Tempi di esecuzione richiesti per la ricerca dei vicini.

6.1.4 Il ClusterGridContainer

Per permettere al sistema di funzionare in tempo reale è quindi necessario correggere i problemi descritti nei paragrafi precedenti.

La soluzione adottata è una diversa implementazione dell'algoritmo di ricerca dei vicini. Nella precedente versione si utilizzano i campioni come sorgente delle informazioni spaziali del cluster; non c'è un oggetto che tiene traccia globalmente di tutti i cluster costruiti, ma ogni volta si controllano tutti i campioni che vengono considerati vicini.

Come primo passo, dunque, occorre cambiare la rappresentazione dei cluster. Invece di etichettare i campioni, la soluzione adottata prevede di etichettare le celle da cui è formata la mappa. In questo modo, la mappa rappresenta un oggetto globale che tiene traccia dell'esistenza di tutti i cluster. Il guadagno che si ottiene da questa rappresentazione è che ogni volta che un campione viene aggiunto alla *set* esso deve controllare unicamente le griglie adiacenti alla sua; nel caso precedente, invece, il

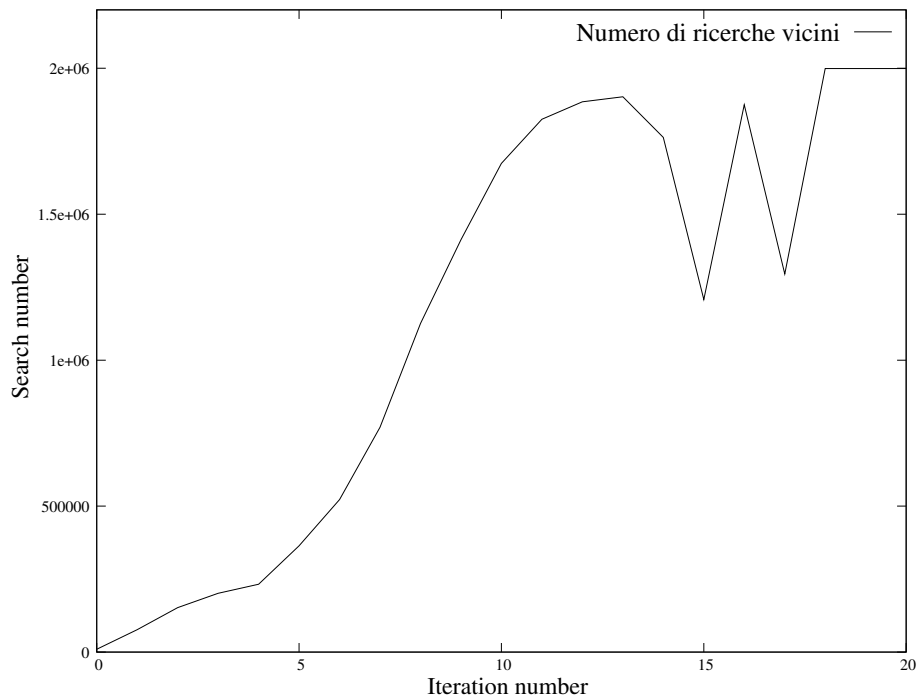


Figura 6.4: Numero di accessi al container dei campioni.

campione era costretto a controllare tutti i campioni adiacenti, e questo si è rivelato essere un numero grande, in conseguenza del loro raggruppamento.

Supponiamo di trovarci nella situazione (a) indicata in figura 6.5. All’aggiunta di un campione al centro della griglia, l’algoritmo della classe *ClusterContainer* effettua un controllo dei vicini per un totale di 13 volte. Al termine dei controlli assegna l’etichetta 1 al campione e segna nella mappa che il cluster 1 e 2 sono equivalenti. Il nuovo algoritmo, invece, non etichetta il campione ma la griglia. Quando il campione viene assegnato al centro della griglia, l’algoritmo controlla unicamente le 5 celle adiacenti che contengono almeno un campione. A questo punto, assegna alla griglia l’etichetta 1 e scrive sulla mappa che i cluster 1 e 2 sono uguali. In questo caso si ha un risparmio di ben 8 iterazioni.

Il nuovo algoritmo, in realtà, si può comportare ancora meglio. Consideriamo il caso (b); quando il campione viene assegnato al centro della griglia, esso si accorge che in quella posizione c’è già un altro campione, che per il punto (a) ha già unito i due cluster. Il nuovo campione arrivato non deve fare altro che aggiungersi all’elenco dei campioni, senza modificare i cluster. In questo caso il nuovo algoritmo

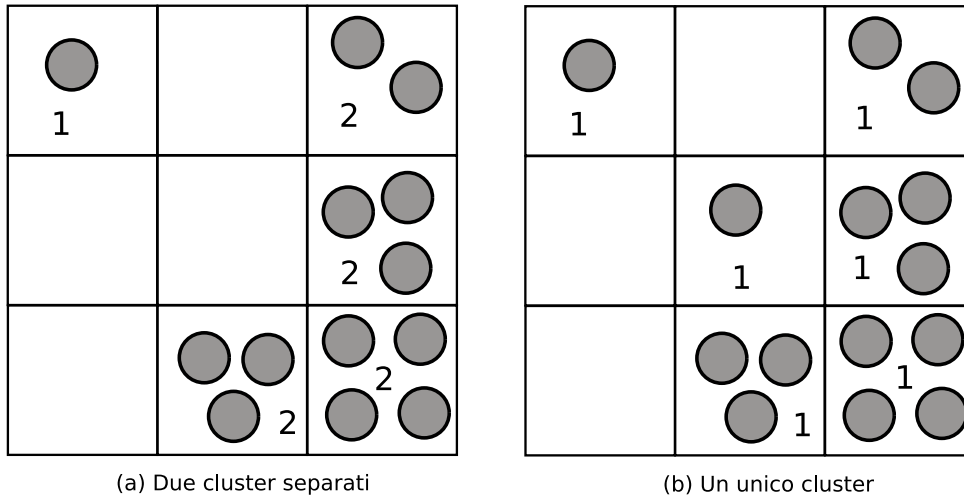


Figura 6.5: Esempi di clustering tra campioni.

di clustering compie 0 iterazioni, contro le 14 che avrebbe impiegato l'algoritmo precedente.

Infine, se stimiamo quale può essere il caso peggio del nuovo algoritmo, risulta che al massimo si dovranno controllare le 8 griglie adiacenti per ogni campione, quindi la velocità di esecuzione va come $O(8N)$. L'algoritmo precedente, invece, supponendo di avere tutte le N particelle aggregate in 9 griglie come quelle della figura 6.5, ha un andamento proporzionale a $O(\frac{N(N+1)}{2})$. L'algoritmo 4 presenta la traccia del nuovo algoritmo di clustering.

Lo studio effettuato ha portato alla creazione di una nuova classe chiamata *ClusterGridContainer*. Un aspetto importante che si è discusso ha riguardato il modo con cui implementare il contenitore delle etichette. La libreria *LSOFT* mette a disposizione due classi container: *ArrayStorage* e *KDTreeStorage*. La differenza risiede nel diverso modo di allocare gli elementi che vengono inseriti. La prima effettua allocazione statica di tutto il contenuto della griglia: in questo modo si spende una quantità considerevole di memoria, ma l'accesso casuale garantito dall'implementazione fornisce la massima velocità possibile. Il *KDTreeStorage*, invece, costituisce un compromesso tra la velocità e la memoria occupata, in quanto alloca dinamicamente ogni singolo elemento, mantenendo comunque un buon ordinamento.

Si è scelto di progettare il *ClusterGridContainer* lasciando libero il programmatore di decidere quale delle due implementazioni utilizzare. I test sono stati comunque effettuati su entrambi i container.

Algorithm 4 Nuovo algoritmo di clustering

Require: Una lista per i campioni, una lista di cluster, campione s in ingresso

Require: Una griglia per contenere le etichette $grid$

```
1: Inserisci il campione  $s$  nella lista dei campioni
2:  $pose = s.getPose()$ ; // preleva la pose del campione  $s$ 
3: if  $grid[pose]$  è già etichettata then
4:   Unisco il campione  $s$  al cluster presente in  $grid[pose]$ 
5:   return
6: end if
7: Prelevo le celle occupate adiacenti al campione  $s$  entro un range di 1.5 celle
8: for all  $grids$  tra le griglie trovate do
9:   if  $grid[pose]$  non è stato etichettato then
10:    Etichetta la griglia  $grid[pose]$  con l'etichetta della griglia vicina  $grids$ 
11:   else
12:    Effettua il join tra i due cluster  $grids$  e  $grid[pose]$ 
13:   end if
14: end for
15: if  $grid[pose]$  non è stata etichettata then
16:   Etichetta  $grid[pose]$  con una nuova etichetta
17:   Crea un nuovo cluster e aggiungilo alla lista dei cluster
18: end if
19: return
```

6.2 Progettazione dell'architettura di tracking

Presentiamo in questa sezione la progettazione dell'ultima parte del sistema, ovvero, il tracker posizionale. Avendo ora a disposizione tutti gli strumenti per effettuare navigazione e localizzazione, ci rimane da testare quale sia l'accuratezza del filtro particellare. Per fare questo, si è ampliata ulteriormente l'architettura del sistema, riconducendola in modo più preciso alla figura 4.1.

6.2.1 ARToolkit

ARToolkit è una applicazione utilizzata in visione artificiale la cui caratteristica fondamentale è l'abilità di riconoscere dei *marker* e di stimarne l'orientamento. Sotto ARToolkit è stata sviluppata un'ulteriore applicazione che consente di calcolare la posizione del robot in modo assoluto rispetto a una data mappa.

Il funzionamento prevede l'applicazione di una serie numerata di *marker* alle pareti, di cui è nota la posizione, e di un *marker* al centro della torretta del robot. Se si riesce a inquadrare con una videocamera il *marker* del robot e, contemporaneamente, almeno un *marker* dell'ambiente, allora il software è in grado di calcolarne l'orientamento relativo e, essendo nota la posizione del *marker* a muro, a stimare la posizione e l'orientamento del robot. Un volta che i dati posizionali sono calcolati, essi vengono inviati tramite socket *Windows* al client che li richiede, sotto forma di stringhe numeriche.

L'algoritmo che sta alla base del sistema è abbastanza leggero e richiede circa mezzo secondo per ogni iterazione. L'applicazione consente pertanto di ottenere una stima della posizione del robot campionata ogni mezzo secondo; ovviamente, riuscire ad inquadrare contemporaneamente almeno due *marker* non è così semplice, perciò può capitare di non riuscire ad avere la stima a ogni periodo.

L'utilità principale di questa applicazione è il fatto che essa fornisce una stima indipendente della posizione reale del robot in un ambiente di prova, senza dipendere dalle sensorialità *proprioceettiva* ed *eteroceettiva* utilizzate dal localizzatore.

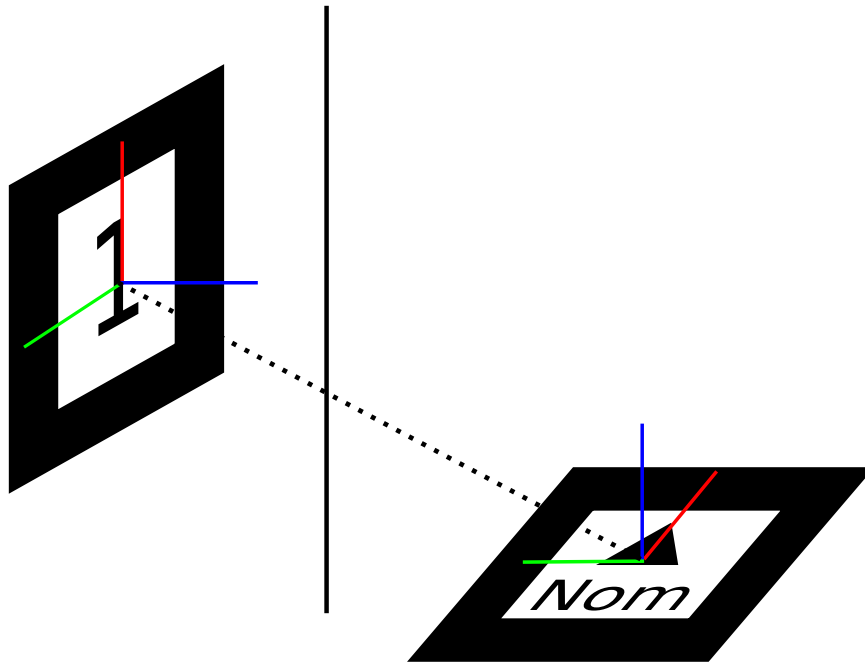


Figura 6.6: Esempio del funzionamento di Artoolkit.

6.2.2 Il NomadTracker

L'utilizzo di ARToolkit è una pratica ormai consolidata nel laboratorio dove è stata svolta questa tesi. Tuttavia, mancano completamente gli strumenti per interfacciarlo con il framework *YARA*.

Dal paragrafo precedente, si evince che il sistema ARToolkit invia su socket i dati calcolati; per massimizzare il riuso, in questo caso, possiamo pensare a un semplice modulo *YARA* che si interfaccia con la rete e che preleva i dati man mano che sono presenti. In futuro sarà relativamente facile riutilizzare questo modulo per prelevare i dati prodotti da ARToolkit e integrarlo in un'altra architettura.

Sorge tuttavia un problema: le primitive di comunicazione basate su socket sono bloccanti, ma per inserire un modulo all'interno di *YARA* è necessario che le sue attività terminino entro le loro *deadline*. Questo può non accadere per questo modulo, in quanto il sistema ARToolkit che invia i dati è subordinato all'intervento umano. Per risolvere questa situazione si è scelto di implementare un modulo che si mettesse in attesa selettiva sulla socket per il 50% del suo periodo (in termini di programmazione si è realizzata la funzionalità con la primitiva `select ()`); in questo

modo il modulo lascia libera la CPU per quasi tutto il suo periodo, consentendo agli altri moduli di eseguire senza problemi. Nel caso arrivino i dati prodotti da ARToolkit, il modulo si risveglia immediatamente e in pochi cicli di CPU li rende disponibili per poi sospendersi nuovamente.

Il modulo realizzato è denominato *ArtoolkitClient*; la struttura del modulo è riportata in figura 6.7.

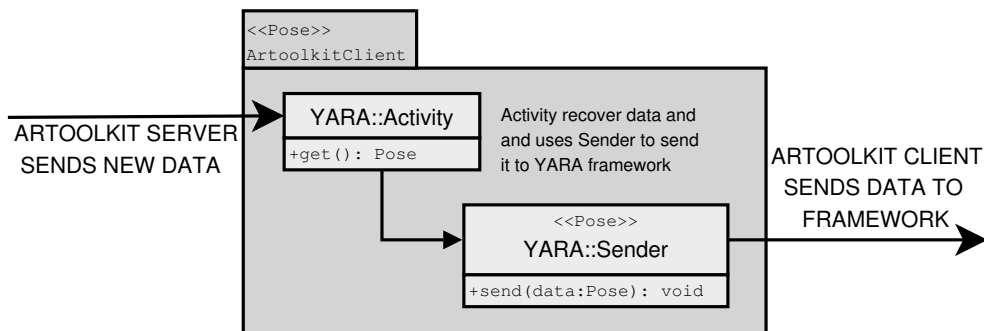


Figura 6.7: Schema del modulo ArtoolkitClient.

Essa tuttavia non è sufficiente da sola per lo scopo di confrontare i risultati prodotti dal localizzatore con quelli provenienti da Artoolkit. Il modulo *ArtoolkitClient* risulta utile in quanto è in grado di recuperare i dati prodotti dal server.

Per finalizzare il tracker è necessario un ulteriore modulo, che espleta le funzioni previste dal blocco *Supervisor* in figura 4.1. Il modulo è stato chiamato *NomadTracker* per riflettere lo scopo della sua funzione. In sostanza, il suo compito è quello di ricevere i dati. È naturale che in questo modo sono richieste anche delle modifiche al modulo *Localizer*, poichè ora deve prevedere la possibilità di inviare la stima posizionale al tracker.

È necessario anche prevedere in che misura verranno prodotti i risultati. Si è già detto che ARToolkit produce una stima posizionale ogni mezzo secondo; il localizzatore, invece, richiede tipicamente tempi più lunghi. Occorre, allora, sincronizzare l'arrivo dei due dati in modo che si riferiscano il più possibile allo stesso istante temporale: il *NomadTracker* calcola la differenza degli istanti di arrivo e si assicura che non sia troppo elevata. È da notare che è impossibile che i dati arrivino precisamente nello stesso istante, quindi, a seconda della velocità del robot e del periodo di elaborazione di ARToolkit, si introduce inevitabilmente un errore nella misura.

Riportiamo infine uno schema del NomadTracker (Figura 6.8) e uno schema del sistema complessivo (Figura 6.9).

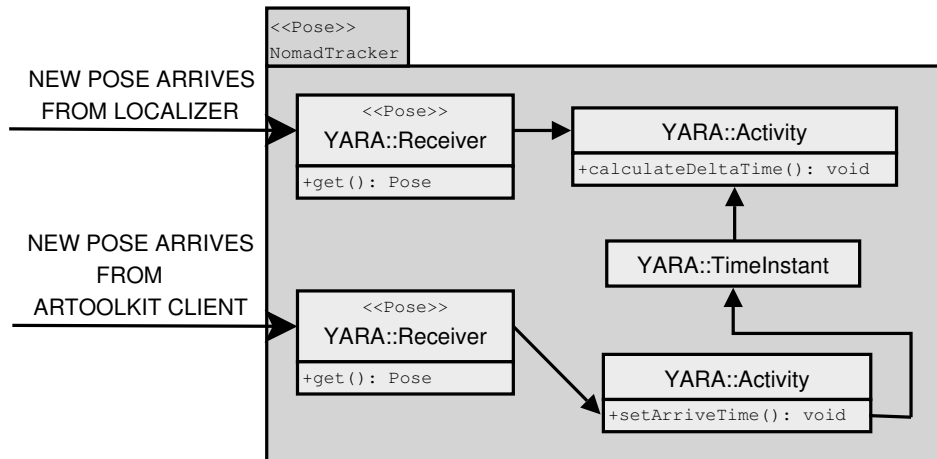


Figura 6.8: Schema del modulo NomadTracker.

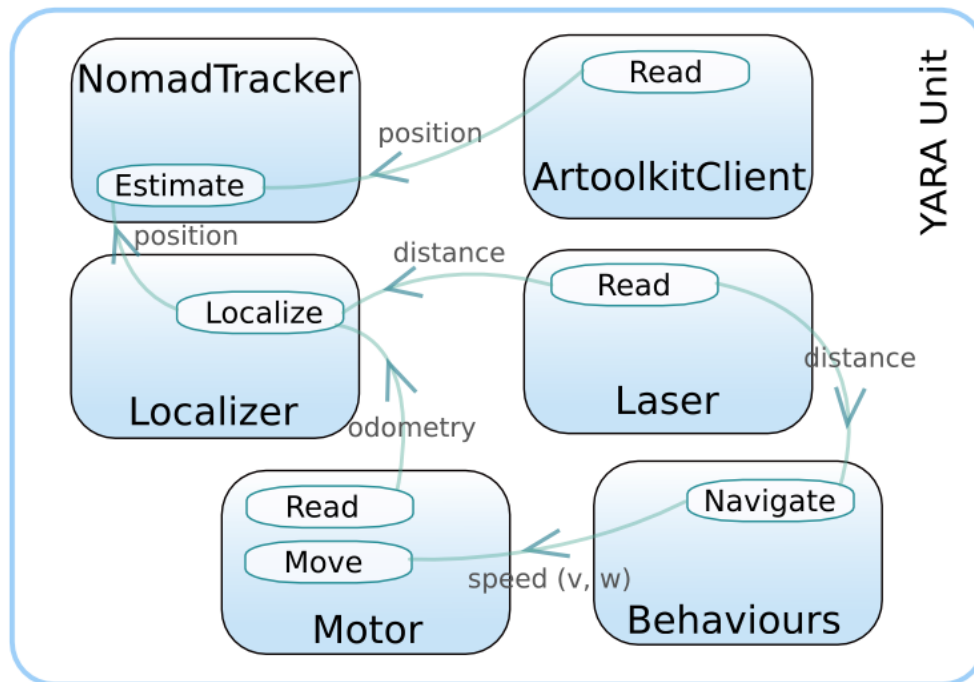


Figura 6.9: Schema a blocchi del sistema completo di controllo e localizzazione.

6.3 Risultati sperimentali

In questa sezione vengono presentati i risultati sperimentali ottenuti testando le modifiche effettuate al software *LSOFT*. In particolare, sono state eseguite due serie di test:

- Test sulle prestazioni, per stimare quale è il periodo nominale di localizzazione del robot assicurato dalla libreria.
- Test di accuratezza, per stimare qual'è l'errore commesso dal localizzatore.

6.3.1 Tempi di esecuzione

Test del GridContainer

La prima serie di test ha coinvolto, necessariamente, il nuovo container realizzato (vedi paragrafo 6.1.4). In questo modo si è potuto ricavare una stima del miglioramento apportato al software di localizzazione.

Sono state confrontate le prestazioni ottenute con entrambe le versioni del container, in modo da valutare la differenza tra la politica *ArrayStorage* e *KDTreeStorage*. I confronti tra i container sono stati effettuati prima in simulazione e, in seguito, durante l'esecuzione reale. Entrambi i casi hanno riportato gli stessi risultati. Il grafico in figura 6.10 mostra la quantità di tempo necessaria a eseguire la sola fase di resampling.

Dalla figura si nota solamente il grande divario tra il precedente container e le nuove implementazioni (paragrafo 6.1.4). Il grafico conferma l'andamento progressivamente crescente dei tempi di resampling dell'algoritmo del precedente container. Il problema fondamentale del *ClusterContainer* risiede nel fatto che esso vuole etichettare ogni campione, ma per fare questo è necessaria la ricerca dei vicini, un'operazione di complessità elevatissima.

La figura 6.11 mostra in dettaglio il confronto diretto tra le prestazioni del container che utilizza l'*ArrayStorage* e quello che utilizza il *KDTreeStorage*.

Il grafico in figura 6.11 riesce a mostrare la differenza tra le due implementazioni del nuovo container. L'allocazione statica della memoria e il conseguente accesso

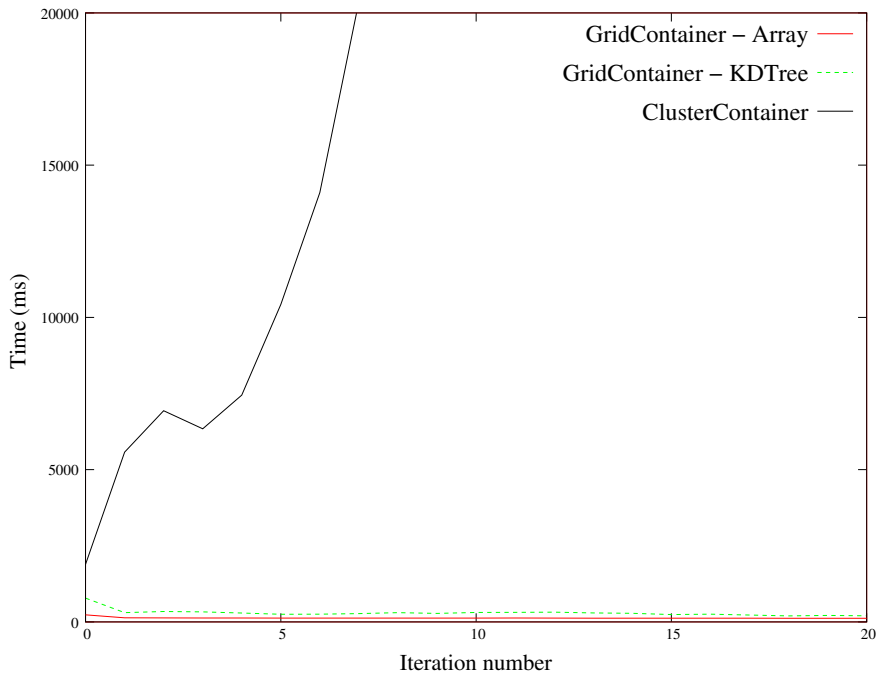


Figura 6.10: Confronto tra i tempi dei container nella fase di resample.

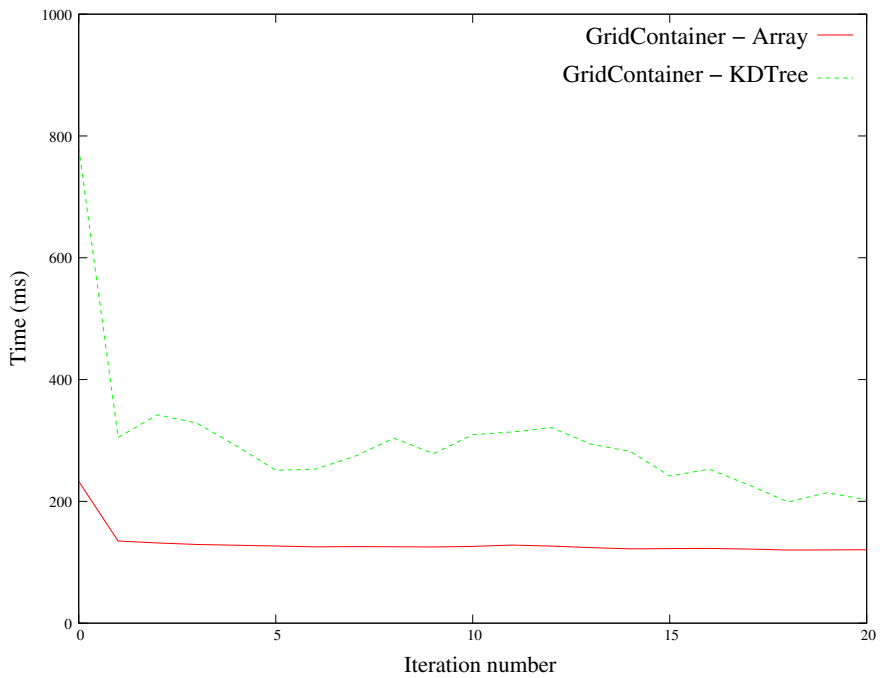


Figura 6.11: Confronto tra i tempi dei grid container nella fase di resample.

casuale di cui possiamo usufruire portano benefici rilevanti nei tempi di esecuzione: nel caso del *KDTreeStorage* i tempi sono circa due volte maggiori.

In figura 6.11 si nota che le prime iterazioni dell'algoritmo del *grid container* richiedono più tempo delle successive. Questo è dovuto al fatto che le particelle sono ancora molto sparse nella mappa, e ancora poche sono state aggregate. Nel momento in cui le particelle si aggregano e formano sempre più cluster, allora l'algoritmo consente di evitare un gran numero di iterazioni inutili alla ricerca dei vicini. A dimostrazione di questo fatto, si riporta l'andamento del numero degli accessi alla griglia fatti per ricercare i campioni vicini (grafico in figura 6.12).

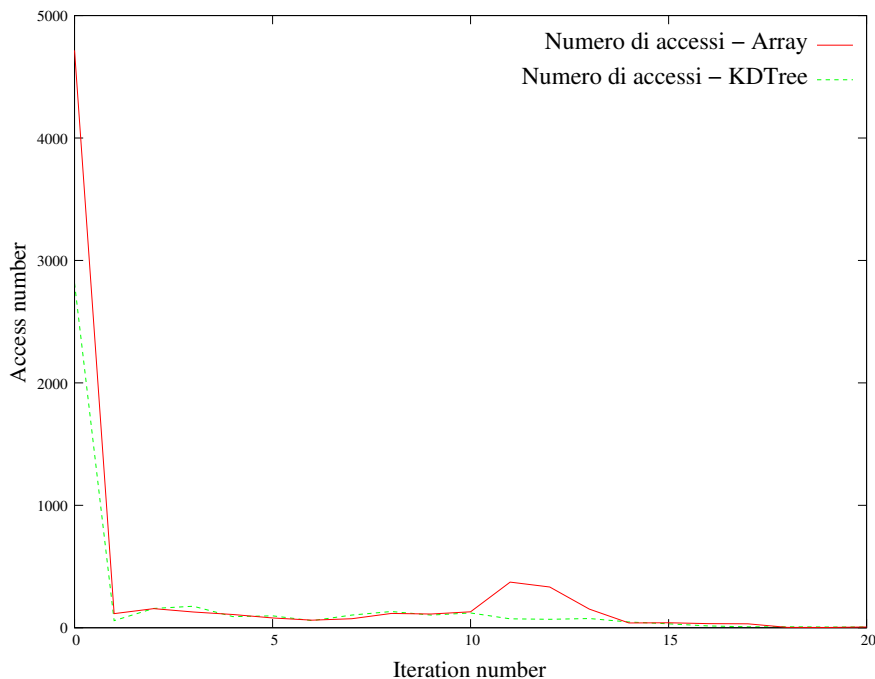


Figura 6.12: Numero di accessi alla griglia per la ricerca dei vicini.

L'approccio utilizzato per ridurre il numero dei controlli è piuttosto efficace; in particolare, quando le particelle si accumulano in un unico cluster non avviene praticamente nessun controllo, perchè i cluster si sono già formati e le particelle vengono aggiunte senza apportare nessuna modifica.

Test del localizzatore

In questa serie di prove si è voluto stimare quali siano i tempi di esecuzione del filtro particellare. Come è già emerso dai paragrafi precedenti, valutare la differenza tra il preesistente container e quello realizzato in questa sede non produce risultati degni di nota, perchè la differenza è talmente grande che non si possono apprezzare gli effetti delle altre scelte progettuali. Il localizzatore originale, pur impiegando pochi secondi nelle prime iterazioni, non mantiene costanti le sue prestazioni; il periodo del filtro diventa anche superiore ai 40 secondi per iterazione, con il risultato che è impossibile inserirlo all'interno di un modulo *YARA* con periodo costante.

Con ogni probabilità, l'enorme differenza di tempi che si è misurata valutando le prestazioni dell'algoritmo di clustering si presenta anche nel localizzatore. Per questo motivo, le verifiche sperimentali si sono incentrate sulla valutazione delle prestazioni solo nel caso del nuovo grid container.

Come si è detto in precedenza, la complessità algoritmica del filtro particellare dipende in modo particolare dal numero di campioni che viene utilizzato per approssimare la PDF. Per avere una stima migliore della qualità del container implementato, si è testato il minimo periodo necessario per poter inserire il modulo *Localizer* all'interno dell'architettura, senza influenzare negativamente il funzionamento del sistema. Questa tipologia di test è stata ripetuta sia per il *particle filter* sia per l'*RTPF*, variando il numero di campioni da 1000 a 8000 a passi di 1000. I test hanno riportato che l'aumento del numero dei campioni è minimamente influente sul periodo di localizzazione, il quale rimane comunque costante. Il motivo di questo risultato è da ricercarsi nei miglioramenti ottenuti grazie all'implementazione del nuovo grid container. Infatti, l'aumento del numero dei campioni non porta necessariamente a effettuare maggiori controlli, in quanto, quando l'ipotesi di localizzazione è già sufficientemente clusterizzata l'aggiunta di un campione richiede esclusivamente la sua aggiunta a una lista. Questo è un risultato molto positivo perchè ci consente di aumentare il numero dei campioni senza determinare un aumento di costi in termini computazionali.

Il grafico in figura 6.13 riporta i tempi richiesti per l'esecuzione del localizzatore su 20 iterazioni.

Lo step iniziale è sempre il caso peggiore, in quanto le particelle sono comple-

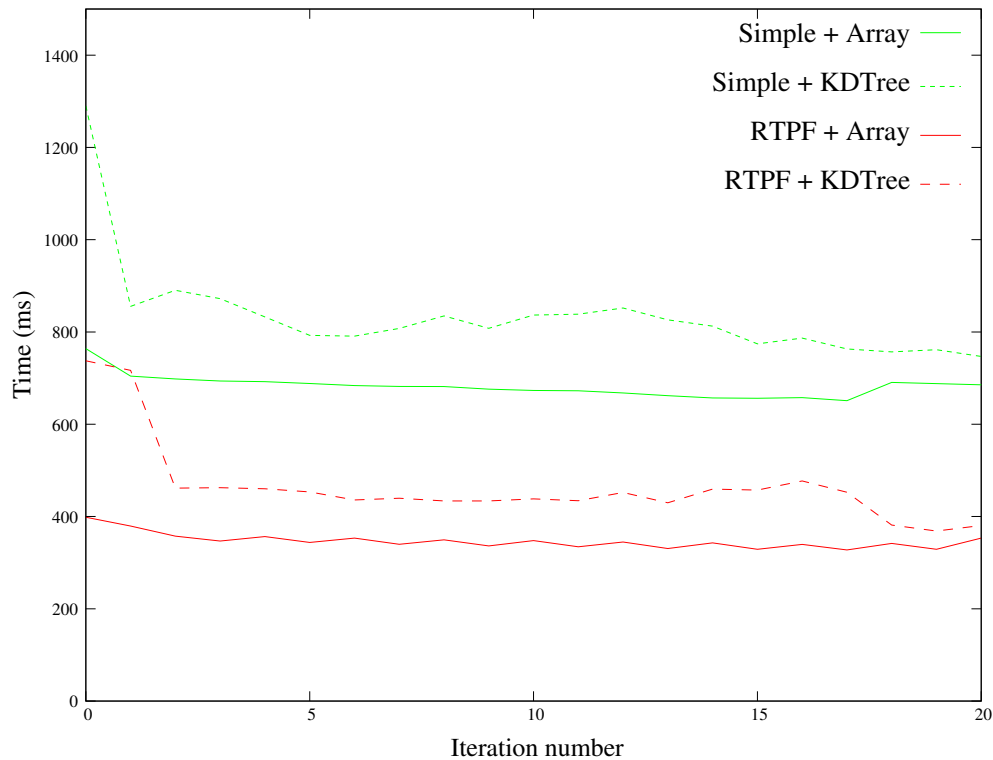


Figura 6.13: Tempi di esecuzione dei 4 localizzatori.

tamente sparse su tutta la mappa. Nelle iterazioni successive, i campioni si accumulano verso le ipotesi di localizzazione e, di conseguenza, i tempi si abbassano considerevolmente. Riportiamo in tabella i casi peggiori stimati, in modo da poter impostare un periodo valido in tutti i casi.

| Tipo di filtro | Tipo di container | Tempo richiesto (ms) |
|------------------------|--------------------------|----------------------|
| Simple Particle Filter | GridContainer con KDTree | 1290.01 |
| Simple Particle Filter | GridContainer con Array | 763.965 |
| RTPF | GridContainer con KDTree | 737.597 |
| RTPF | GridContainer con Array | 398.414 |

È da notare come l'utilizzo del container con KDTree faccia aumentare in modo considerevole i tempi totali richiesti, rispetto alla versione con Array. Ricordiamo che i tempi in grafico e in tabella sono in parte influenzati dalla presenza dell'architettura di controllo; nonostante il suo impatto non sia eccessivo, circa il 16%,

i tempi misurati in simulazione, senza l'architettura di controllo, sono risultati più contenuti.

6.3.2 Accuratezza nella localizzazione

I test di tracking sono stati svolti per valutare quale sia la precisione dell'algoritmo di localizzazione; l'ausilio dello strumento ARToolkit e dei moduli creati appositamente (vedi 6.2.2) ha consentito di ricavare i risultati esposti in questo paragrafo.

Mentre il robot è lasciato libero di navigare secondo il modello definito nel *behaviour wall following*, il server di ARToolkit preleva le informazioni provenienti dalla visione e le invia al modulo *NomadTracker* in esecuzione sul robot mobile. Contemporaneamente, il localizzatore produce la sua stima posizionale secondo l'algoritmo previsto. Le notevoli difficoltà manuali che ARToolkit prevede non hanno permesso di raccogliere dati in abbondanza; tuttavia è stato possibile prelevare una serie di posizioni stimate sia dal localizzatore che da ARToolkit. Abbiamo così ottenuto una stima della traiettoria compiuta, calcolata da entrambe le applicazioni.

Nelle figure seguenti è riportato l'output prodotto dal localizzatore. In questo caso, l'output è stato raccolto anche sotto forma testuale per permettere il confronto con i dati prodotti da ARToolkit.

Il robot ha percorso tutta la lunghezza del muro alla sua sinistra, producendo in questa lunghezza 15 stime posizionali. La difficoltà di manovrare la telecamera in modo da inquadrare simultaneamente il robot e i marker ha portato ARToolkit a produrre solamente 8 stime posizionali sincronizzate con il Nomad. Per poter tracciare una traiettoria sovrapponibile con le ipotesi di localizzazione si sono interpolati i punti mancanti, ottenendo il risultato in figura 6.15.

La traiettoria continua è quella ricostruita a partire dai punti prodotti da ARToolkit, quindi, è quella che deve essere considerata come la traiettoria reale del robot. I cerchi rappresentano i punti dove la posizione calcolata da ARToolkit. Infine, le croci rappresentano tutte le stime posizionali prodotte da LSOFT.

Riportiamo in una tabella i dati relativi alle posizioni realmente calcolate da ARToolkit.



Figura 6.14: Localizzazione del behaviour di *wall following*.

| X Localizer | Y Localizer | X ARToolkit | Y ARToolkit | Errore |
|-------------|-------------|-------------|-------------|--------|
| 0.7262 | 0.1979 | 0.4480 | 0.7520 | 0.6201 |
| 0.4156 | 0.3855 | 0.4374 | 1.0870 | 0.7018 |
| 0.3321 | 0.7231 | 0.4203 | 1.4643 | 0.7464 |
| 0.3280 | 1.0979 | 0.5911 | 1.9147 | 0.8581 |
| 0.2841 | 2.6675 | 0.4464 | 3.4951 | 0.8434 |
| 0.2994 | 3.0478 | 0.5422 | 3.8870 | 0.8736 |
| 0.3131 | 3.5076 | 0.2624 | 4.3628 | 0.8567 |
| 0.2962 | 3.9307 | 0.4209 | 4.6861 | 0.7656 |

Tabella 6.1: Tabella delle traiettorie misurate da ARToolkit e da LSOFT (grandezze espresse in metri).

Dalla tabella si ricava che l'errore medio commesso è di 0.778m, un valore tutto sommato accettabile, mentre la deviazione standard è di 0.045m.

A ulteriore merito del localizzatore va il fatto che i marker lungo la zona percorsa erano molto scarsi. Questo fatto ha costretto a prelevare i dati anche da marker molto distanti, riflettendosi in una distorsione dell'immagine passata a ARToolkit e, conseguentemente, sull'errore totale.

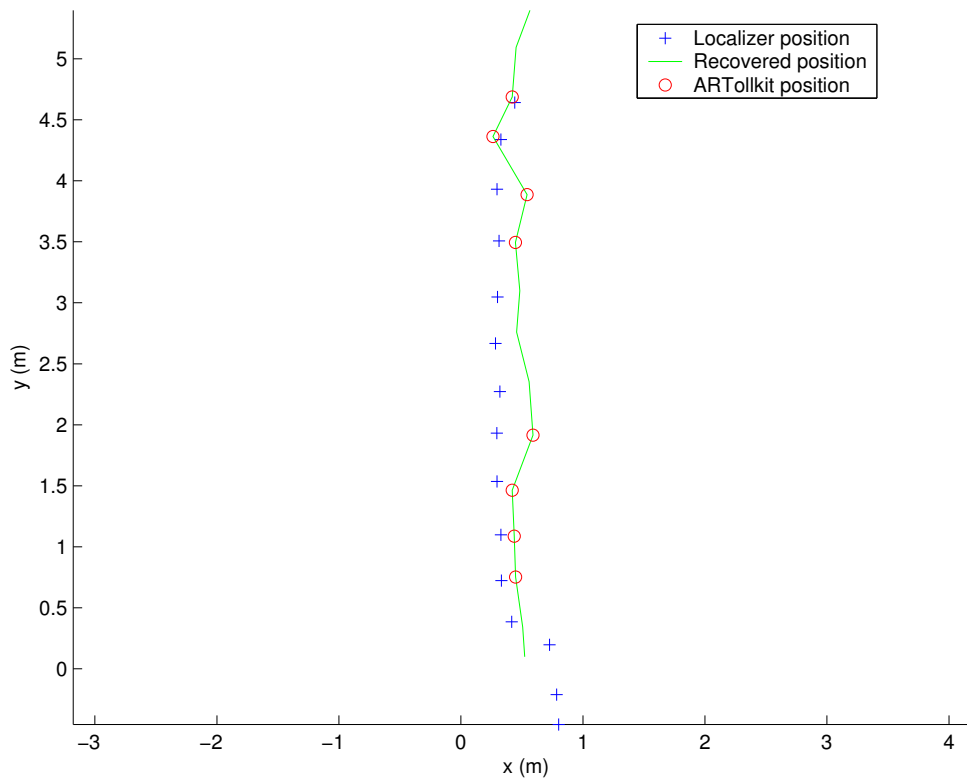


Figura 6.15: Sovrapposizione delle traiettorie generate da ARToolkit e dal localizzatore.

Qualitativamente, sembra che la precisione reale sia maggiore; osservando le immagini prodotte dal localizzatore in punti ben noti, l'errore sembra essere davvero minimo. In un esperimento è stato fatto percorrere al robot un corridoio fino a incontrare un vicolo cieco, in modo da farlo tornare indietro. È stata imposta una distanza dal muro di 50cm. Le immagini in figura 6.16 mostrano come il localizzatore riesca a inseguire perfettamente la traiettoria percorsa.

Il corridoio riportato in figura 6.16 è largo 1.80m; se l'errore fosse davvero di 80cm, quando il robot è al centro del corridoio in realtà l'errore lo porterebbe a essere ancora all'inizio, oppure alla fine. È da notare che, in figura, le ipotesi predominanti sono due, ma quella più avanti è preponderante tanto che nell'ultima immagine l'ipotesi meno valida viene scartata.

Uno studio più approfondito sull'applicazione di tracking potrebbe condurre a risultati migliori per quanto riguarda la stima dell'errore reale di localizzazione.

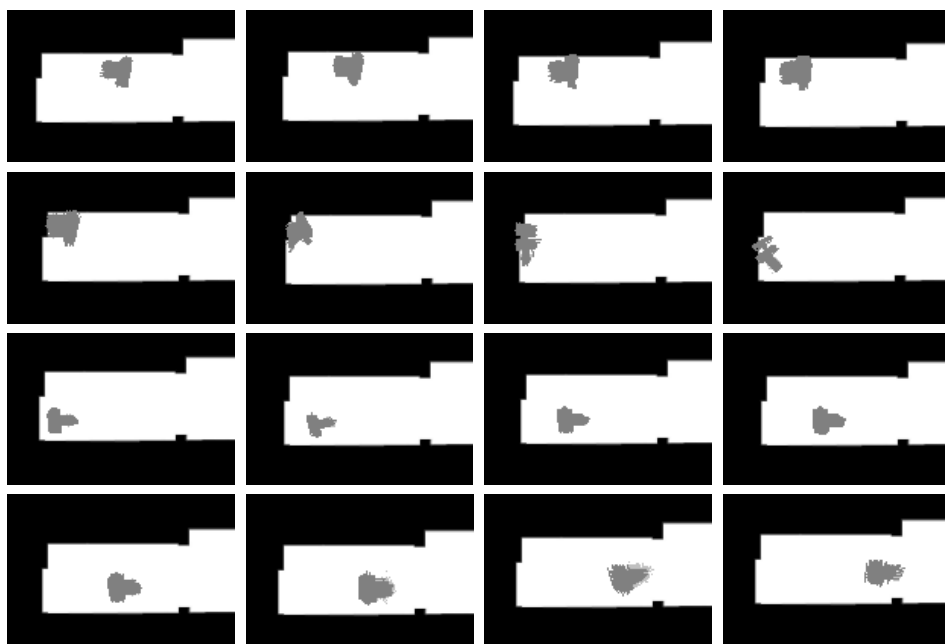


Figura 6.16: Seconda localizzazione del behaviour di *wall following*.

Capitolo 7

Conclusioni

In questa tesi è stato affrontato il problema della realizzazione di un'architettura robotica software che metta a disposizione un insieme di funzionalità per consentire la navigazione e la localizzazione di robot mobili. Benchè il progetto sia stato sviluppato in modo specifico per il robot Nomad 200, la struttura altamente modulare del sistema ne consente una agevole integrazione verso qualunque piattaforma robotica permettendo, allo stesso tempo, di estendere quanto progettato con nuove funzionalità.

Il progetto è stato realizzato utilizzando un *framework* software appositamente studiato per il controllo di sistemi robotici e questo ha favorito i buoni risultati che sono stati raggiunti. Il *framework*, infatti, ha reso disponibili gli strumenti per implementare sistemi di questo tipo potendo porre vincoli di tipo *real time*.

La prima fase del lavoro è consistita nella progettazione di una infrastruttura di base per il controllo del robot in grado di dotarlo, principalmente, di sensorialità e movimento. Al fine di collaudare questo primo livello di controllo sono stati implementati alcuni moduli comportamentali che hanno permesso di verificare le prestazioni dell'architettura di base. Rispetto all'architettura precedente, implementata con un *framework* non *real time*, le prestazioni del sistema si sono rivelate fin da subito ottime, soprattutto dal punto di vista della affidabilità.

Il contributo principale della tesi è stato il lavoro che ha consentito di integrare all'interno dell'architettura di base un sistema di localizzazione globale basato su filtri particellari. L'analisi del software di localizzazione ha permesso di effettuare

alcune modifiche sostanziali al filtro, permettendo di migliorarne le prestazioni in modo significativo. Quest'ultimo passo ha consentito l'integrazione delle due funzionalità in un unico sistema, cosa per altro impossibile in precedenza, ottenendo un robot mobile in grado di navigare all'interno di un ambiente e di localizzarsi in tempo reale previa conoscenza della sua mappa.

L'ultima fase del lavoro di tesi ha riguardato l'esecuzione di una molteplicità di test atti a verificare le prestazioni del sistema complessivo. Oltre alle misurazioni relative ai tempi di esecuzione dell'attività di localizzazione, è stato possibile effettuare alcune stime dell'errore posizionale commesso dal filtro partellare. La misura di questo errore è avvenuta grazie all'integrazione del sistema con una applicazione di *tracking* esistente, basata su visione artificiale.

Le probabili estensioni di questa tesi sono le seguenti:

- Coordinazione dei moduli comportamentali per raggiungere la capacità di esibire comportamenti più sofisticati. Altri *behaviours* possono essere implementati e la loro interazione può portare a una navigazione che consenta di muovere il robot in ambienti complessi.
- Integrazione della localizzazione nella navigazione. Attualmente la localizzazione è un *task* slegato dai moduli comportamentali, ma è possibile concepire nuovi comportamenti in cui intervenga anche l'informazione di localizzazione.
- Integrazione del sistema motorio e di navigazione con la parte di manipolazione degli oggetti. Il Nomad è dotato di un manipolatore meccanico che consente di spostare oggetti. L'architettura può essere coordinata da un modulo di livello superiore, che gestisce navigazione e manipolatore, per compiere *task* di complessità più elevata come lo spostamento del robot al fine di muovere un oggetto.
- La costruzione di un insieme completo di moduli che forniscano l'accesso a altre eventuali periferiche sensoriali, dato che attualmente è disponibile un'interfaccia solo per il sensore laser.

Infine, un ulteriore possibile sviluppo riguarda la realizzazione di un sistema di localizzazione e mappatura simultanei, che consenta di prescindere dalla disponibilità di una mappa accurata dell'ambiente in cui opera il robot.

Appendice A

Realizzazione dei behaviours per il framework Smartsoft

In questa appendice vengono presentati i behaviours che sono stati implementati per il *framework Smartsoft*. Tralasciando i principi algoritmici, identici a quelli visti al paragrafo 4.2.3, diamo una delucidazione sul modo in cui sono stati implementati.

Trattiamo, per prima cosa, il server relativo al laser Sick. L'insieme delle classi disponibili per Smartsoft non comprendeva un oggetto che si occupasse della lettura dei dati prodotti dal laser. Si è proceduto, allora, alla sua implementazione.

Le comunicazioni del server vengono gestite da un *pattern* di tipo *PUSH*, mentre a livello inferiore lo scambio dati con la periferica è stato incapsulato nella classe *SickLMS200* utilizzata anche nel sistema realizzato con *YARA*. Il listato A.1 presenta quanto è stato fatto.

Il server del laser è poi stato inserito all'interno di un più generico *NomadServer* che si occupa di instanziare un oggetto per ogni periferica che mette a disposizione. Quello che si è fatto, è stato aggiungere l'oggetto relativo al server laser.

L'ultima fase è stata la creazione del behaviour. La differenza principale con *YARA* risiede nel fatto che il behaviour viene creato come una *thread* indipendente, mentre in *YARA* il modulo è sottoposto comunque al controllo della sua *Unit*. La classe che implementa il comportamento deve registrarsi per accedere alle letture del laser e deve predisporre all'invio dei comandi al motor. La funzione di controllo, chiamata `controller()` implementa l'algoritmo visto al paragrafo 4.2.3.

```
1 class DistanceSickPushTimedHandler
2     : public CHS::PushTimedHandler<Smart::CommLaserScan> {
3
4     public:
5     DistanceSickPushTimedHandler();
6     virtual ~DistanceSickPushTimedHandler();
7
8     void handlePushTimer(CHS::PushTimedServer<Smart::CommLaserScan>
9         &server) throw();
10
11    private:
12    laser::SickLMS200 laser_driver;
13    std::vector<double> myData;
14    Smart::CommLaserScan _scan;
15 };
```

Listato A.1: Interfaccia del server del laser Sick LMS 200.

```
1 DistanceSickPushTimedHandler distancesickPushTimedHandler;
2 CHS::ThreadQueuePushTimedHandler<Smart::CommLaserScan>
3     threadDistanceSickPushTimedHandler(distancesickPushTimedHandler);
4 CHS::PushTimedServer<Smart::CommLaserScan>
5     distanceSickPushTimedServer(&component, "distancesick",
6         threadDistanceSickPushTimedHandler,
7         0.001*param_baseSickPushTimedInterval);
```

Listato A.2: Instanziamento del Sick server nel Nomad Server.

```
1 class WallClientThread : public CHS::SmartTask
2 {
3 public:
4   WallClientThread(CHS::PushTimedClient<Smart::CommLaserScan> &laser,
5     CHS::SendClient<Smart::CommNavigationVelocity> &motor)
6     : _laser(laser), _motor(motor) {}
7
8   int svc();
9 private:
10
11   CHS::PushTimedClient<Smart::CommLaserScan> &_laser;
12   CHS::SendClient<Smart::CommNavigationVelocity> &_motor;
13
14
15   double controller(double lambda, double c, double theta,
16     double right_distance, double left_distance,
17     double right_wall_normal, double left_wall_normal);
18
19   Smart::CommLaserScan::polar_point_type getMinWallDistance(
20     Smart::CommLaserScan::const_polar_iterator& data, int start, int end);
21
22 };
```

Listato A.3: Interfaccia del behaviour di wall following.

Appendice B

Interfacciamento remoto del robot

Nella discussione della tesi è stato presentato un insieme di moduli che tendono a essere autonomi. Il sistema non richiede mai l'intervento dell'uomo, ma si comporta unicamente secondo modelli prestabiliti. È risultata piuttosto utile la creazione di un modulo di interfacciamento con il mondo esterno. Si è pensato a un modo per pilotare il Nomad da remoto e si è scelto di utilizzare una connessione basata su Socket UNIX e sul protocollo UDP. È stato quindi realizzato un modulo *Proxy* che consente di fornire l'accesso a utenti umani.

Le primitive di comunicazione basate su socket sono bloccanti, mentre per inserire un modulo all'interno di *YARA* è necessario che le sue attività terminino entro le loro *deadline*. Un server in ascolto di connessioni non può sapere quando avverrà l'intervento umano, quindi, non è possibile determinare il periodo della sua attività. Si è risolta la situazione in modo analogo a quanto fatto per il *NomadTracker* (vedi 6.2.2), ovvero implementando un modulo che si mettesse in attesa selettiva su una serie di *file descriptor* per il 70% del suo periodo (in termini di programmazione si utilizzava la primitiva `select()`). Nel caso arrivi un comando da remoto, il modulo si risveglia immediatamente e in pochi cicli di CPU invia il comando al motore, per poi sospendersi nuovamente.

Da lato client, è stata realizzata una interfaccia per permettere l'accesso da qualunque punto della rete: il Nomad, infatti, viene localizzato grazie alla URL fornita dall'interfaccia.

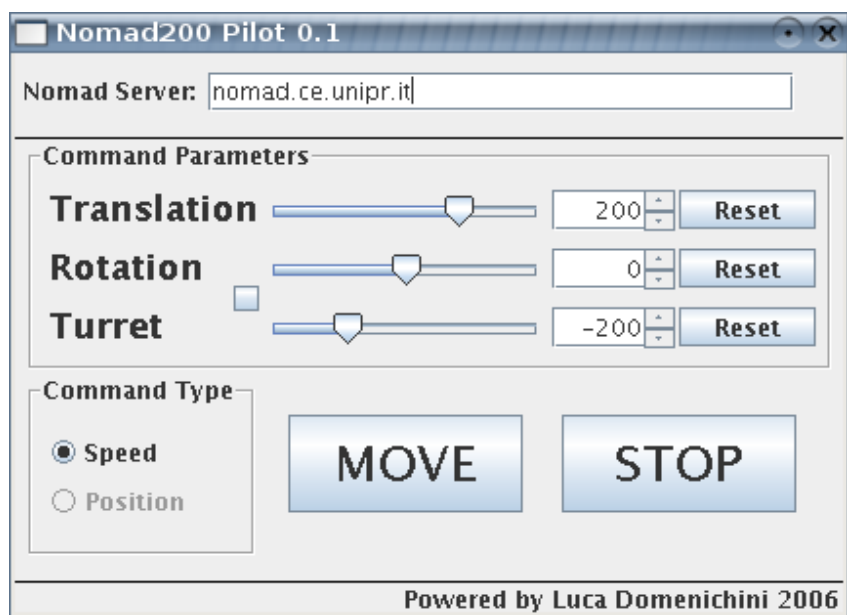


Figura B.1: Screenshot dell'interfaccia client.

Bibliografia

- [1] R. Siegwart and I.R. Nourbakhsh. *Introduction to Autonomous Mobile Robots*. MIT Press, 2004.
- [2] J. Borenstein, H.R. Everett, and L. Feng. “Where am I?”, *Sensors and Methods for Mobile Robot Positioning*. 1996.
- [3] A. Elfes. Using Occupancy Grid for Mobile Robot Perception and Navigation. *Computer*, June 1989.
- [4] A. Moravec. Using Occupancy Grid for Mobile Robot Perception and Navigation. *AI Magazine*, June 1988.
- [5] D. Fox. Adapting the Sample Size in Particle Filters through KLD-Sampling. *International Journal of Robotics Research*, 22(12), 2003.
- [6] D. Fox. Real-Time Particle Filters. *Proceedings of the IEEE*, 92(2), 2004.
- [7] Fox Thrun, Burgard. *Probabilistic Robotics*. MIT Press, 2005.
- [8] A. Doucet, M. De Freitas, and N. Gordon. *Sequential Monte Carlo methods in practice*. Springer, 2001.
- [9] Miodrag Bolic, Petar M. Djuric, and Sangjin Hong. Resampling algorithms for particle filters: A computational complexity perspective. *EURASIP Journal of Applied Signal Processing*, 2004.
- [10] T.M. Cover and J.A. Thomas. *Elements of Information Theory*. Wiley, 1991.
- [11] D. Lodi Rizzini. Progettazione di una libreria per la localizzazione e fusione sensoriale basata su filtri particellari. Tesi di Laurea Magistrale in Ingegneria Informatica, Università degli Studi di Parma, 2005.
- [12] Nomadic Technologies Inc. *The Nomad 200 User Guide*, December 1993.
- [13] SOYO Computer Inc. SY-7VEM Motherboard.
<http://www.soyousa.com/products>.
- [14] Galil Motion Control Inc. Multi Axis Motion Controllers. <http://www.galilmc.com>.
- [15] C. Schlegel. Communications Patterns for OROCOS. Hints, Remarks, Specifications. Technical report, Research Institute for Applied Knowledge Processing (FAW), February 2002.
- [16] F. Monica. Progettazione di un'architettura modulare, aperta ed in tempo reale per un robot mobile. Tesi di Laurea in Ingegneria Informatica, Università degli Studi di Parma, 2003.

-
- [17] D. Pallastrelli. Studio e Realizzazione di un Framework Orientato agli Oggetti per Applicazioni Real-time. Tesi di Laurea in Ingegneria Informatica, Università degli Studi di Parma, 2002.
- [18] R.A. Brooks. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, March 1986.
- [19] H. R. Everett. *Sensors for mobile Robots, theory and application*. 1995.
- [20] P. Althaus and H. Christensen. Behaviour coordination in structured environments. *Advanced Robotics*, 17(7):657–674, 2003.
- [21] F. Pedrielli. Realizzazione di un Sistema di Localizzazione Probabilistico basato su Filtro Particellare per Robot Mobili. Tesi di Laurea in Ingegneria Informatica, Università degli Studi di Parma, 2004.
- [22] A. Alexandrescu. *Modern C++ Design, Generic Programming and Design Pattern Applied*. 2001.
- [23] C. Pescio. Programmazione ad oggetti e programmazione generica. *Computer Programming*, (62). <http://www.eptacom.net>.
- [24] Boost. <http://www.boost.org>.
- [25] M. Bolic, P. Djuric, and S. Hong. Resampling Algorithms for Particle Filters: a Computational Complexity Perspective. *Department of Electrical and Computer Engineering, Stony Brook University*.