UNIVERSITÀ DI BRESCIA
FACOLTÀ DI INGEGNERIA
Dipartimento di Elettronica per l'Automazione

Laboratorio di Robotica Avanzata
Advanced Robotics Laboratory

Corso di Robotica
(Prof. Riccardo Cassinis)

Software per il parcheggio del robot Speedy

Elaborato di esame di: **Alessandro Ricchetti**

Consegnato il: **7 Luglio 2006**

Sommario

All'interno del Laboratorio di Robotica Avanzata della facoltà è stato ideato un sistema per la ricarica autonoma del robot "Speedy". Con questo sistema il robot è in grado di localizzare e raggiungere, in modo corretto, la propria stazione di ricarica a partire da qualunque posizione all'interno del laboratorio. Inizialmente, il robot viene fatto muovere con un comportamento di tipo "wander" alla ricerca di un landmark artificiale passivo rappresentato da un rettangolo di nastro adesivo verde fissato sul pavimento. A partire da questo punto, il robot segue un secondo landmark costituito da una striscia di nastro adesivo giallo, anch'esso fissato al pavimento, che lo conduce sino alla docking station. La discriminazione dei due landmark è ottenuta elaborando le immagini catturate da una webcam posta sulla parte frontale del robot. Questo rapporto tecnico descrive un'implementazione software per gestire la navigazione del robot nel contesto descritto. La soluzione adottata utilizza i dati forniti dal software di gestione della visione sviluppato dal gruppo composto dagli studenti: Bianchi, Cucchi, Forino, Lombardo e Sacco.

1. Introduzione

Inizialmente viene presentato il robot Speedy attraverso una breve descrizione delle sue caratteristiche peculiari, e delle modifiche apportate per poter essere inserito nel contesto presentato. Segue una breve spiegazione del meccanismo delle azioni e dell'architettura del pacchetto ARIA in generale.

1.1. Il robot "Speedy"

Il robot denominato "Speedy" è il modello Pioneer 1 commercializzato da ActivMedia Robotics. Si tratta di un robot con architettura differential drive, con due ruote motrici fisse ed indipendenti ed una ruota folle e pivottante. Sulla parte superiore del robot è appoggiato un calcolatore portatile sul quale è installato il pacchetto dei software utilizzati per programmare la movimentazione del robot. Calcolatore e robot comunicano attraverso un cavo seriale con protocollo RS232. Questo modello dispone di sette sonar disposti sulla parte anteriore che permettono di rilevare la presenza di ostacoli, fino ad una distanza massima di circa 4 m, su un intervallo poco più ampio da -90° a 90° secondo la convenzione schematizzata in figura 2. Sulla parte frontale del robot è montata una webcam inclinata verso il pavimento, utilizzata per la discriminazione dei landmark.

(a) (b)

Figura 1. (a) Il robot Speedy. (b) Particolare della webcam e dei sonar.

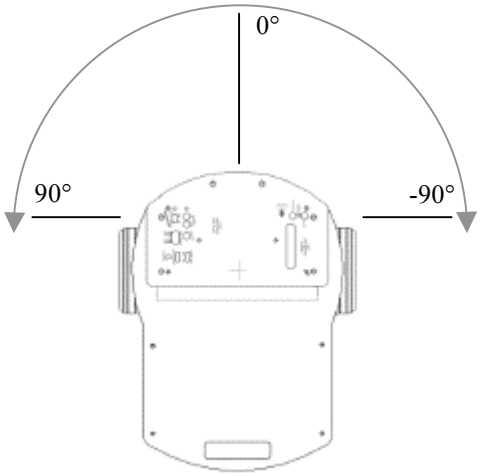


Figura 2. Convenzione angolare.

1.2. Il pacchetto ARIA

Il pacchetto ARIA costituisce un'interfaccia, orientata agli oggetti, per la programmazione di applicazioni per il controllo di robot appartenenti alla linea commerciale di ActivMedia Robotics, infatti, la sigla ARIA è l'acronimo di ActivMedia Robotics Interface for Application. Costituita da una libreria di classi scritte in C++, ARIA supporta nativamente l'architettura multi-threading. La tipologia di architettura scelta per interfacciare l'applicativo di movimentazione al robot fisico è di tipo client-server. Il programma dell'utente, grazie all'interfaccia ARIA, costituisce la parte client mentre il robot costituisce la parte server. Il lato server è implementato usando una struttura, denominata *State Reflector*, costituita da un insieme di variabili contenenti valori quali, le velocità dei motori, le letture dei sonar e le altre grandezze che concorrono a definire l'attuale stato del robot. Il programma client ottiene le informazioni sullo stato del robot, ed inoltra le proprie richieste, a partire dal contenuto di queste variabili. Ciclicamente il microcontrollore incorporato nel robot esegue il polling di queste variabili, tipicamente dieci volte al secondo, aggiornandone il valore ed eseguendo le richieste del programma client.

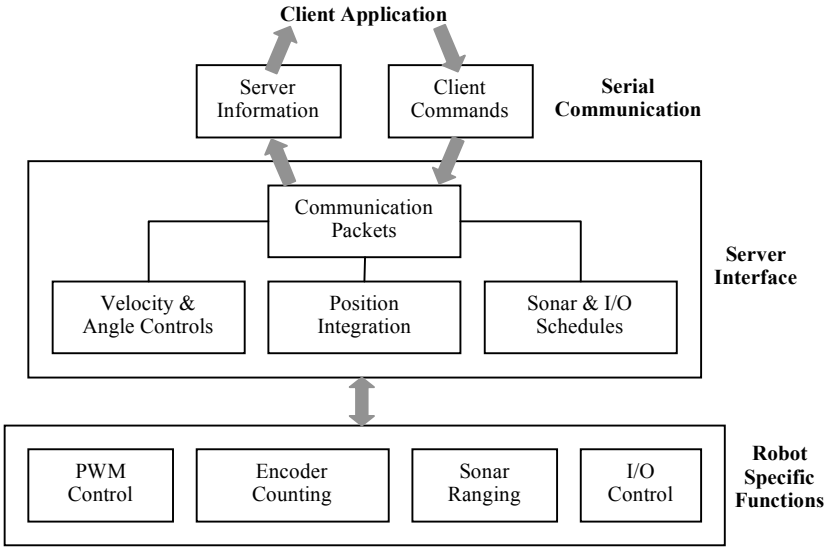


Figura 3. Architettura client-server di ARIA.

Per movimentare il robot sono disponibili, principalmente, due modalità. Si possono usare comandi diretti all'interno del programma principale, oppure può essere usato il meccanismo delle azioni. La classe *ArAction* mette a disposizione, da libreria, numerose utili azioni già pronte per essere utilizzate. Ad esempio, il software di navigazione che è stato sviluppato utilizza l'azione per gestire un'eventuale situazione di stallo del robot, e l'azione per evitare ed aggirare gli ostacoli. Grazie, invece, alla classe *ArActionDesired* è possibile definire azioni con il comportamento preferito dall'utente. La parte principale nella definizione di un'azione è la funzione denominata *fire*, che contiene l'algoritmo del comportamento dell'azione. Per comprendere più chiaramente quanto accennato, può essere utile esaminare il listato del codice riportato in appendice. Per poter essere eseguite, le azioni devono essere inserite in una lista tramite l'istruzione *addAction*. Una volta inserita, un'azione può venire rimossa o disattivata usando, rispettivamente, le istruzioni *remAction* e *deactivate*. Queste due operazioni sono differenti, mentre la prima rimuove completamente l'azione dalla lista, la seconda inibisce i comandi impartiti dall'azione, la quale non viene rimossa dalla lista. Un'azione disattivata rimane tale fino a quando non viene invocato, per essa, il comando *activate*. Ad ogni azione viene associato un valore, compreso tra 0 e 100, indicato come priorità. Questo parametro è usato dal *Resolver* per stabilre l'ordine di esecuzione delle azioni inserite nella lista. Viene eseguita per prima l'azione con priorità più elevata, seguita da tutte le altre fino a quella con la priorità più bassa.

2. Il problema affrontato

Come accennato nel sommario, nel Laboratorio di Robotica Avanzata della facoltà è stato ideato un sistema per la ricarica autonoma del robot Speedy. Il sistema è composto dalla stazione di ricarica e un metodo di localizzazione, per raggiungere tale stazione, basato sul riconoscimento di landmark artificiali passivi. Il modello semplificato di questa architettura è rappresentato in figura 4.

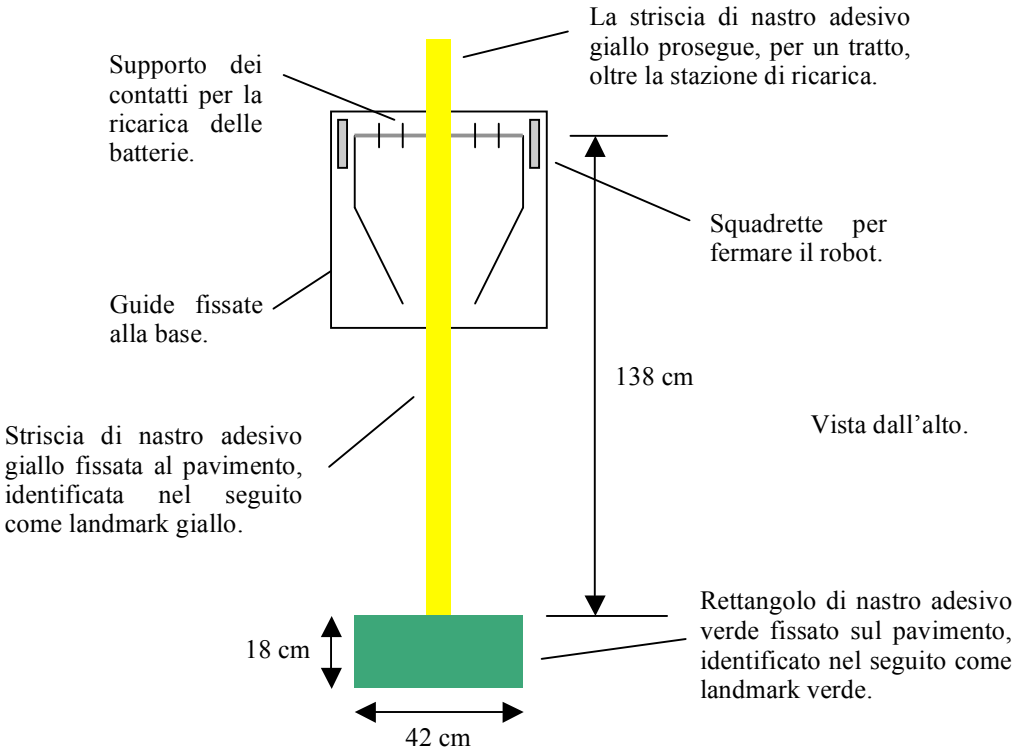


Figura 4. Modellizzazione della struttura ideata. Comprende un sistema di localizzazione basato su landmark artificiali e la stazione di ricarica.

La base di partenza è rappresentata da un landmark costituito da un rettangolo di nastro adesivo verde incollato al pavimento del laboratorio. A partire da un qualunque punto del laboratorio il robot viene fatto muovere con comportamento "wander", ossia, avanzando in linea retta a meno di ostacoli che

vengono evitati con azioni di rotazione. Questo comportamento viene mantenuto fintanto che non viene individuato il landmark verde da parte del sistema di visione, tramite la webcam montata su Speedy. A questo punto il robot si dirige verso il landmark verde basandosi, sempre, sui dati forniti dal software di elaborazione delle immagini catturate dalla telecamera. Giunto in prossimità del landmark verde, il robot cerca un secondo landmark, costituito nella realtà, da una striscia di nastro adesivo giallo. Come mostrato nel modello di figura 4, questa striscia di nastro adesivo congiunge il landmark verde con la stazione di ricarica. Dopo essersi allineato rispetto al landmark giallo, il robot inizia ad avanzare cercando di mantenere l'allineamento. Speedy segue il landmark giallo basandosi, ovviamente, sui dati forniti dal software di visione. Continuando a seguire il landmark giallo il robot raggiunge la stazione di ricarica. La disposizione delle guide fissate sulla base permette al robot di arrivare alla stazione con un sufficiente margine di errore, conducendolo fino al corretto posizionamento finale. La stazione di ricarica è completata dalla struttura dei contatti per la ricarica delle batterie e da due squadrette di fermo corsa. Tali squadrette, utilizzate per fermare il robot nella posizione desiderata, vengono sfruttate dall'algoritmo di risoluzione per determinare la condizione di terminazione del programma, una volta rilevata la contemporanea situazione di stallo di entrambe le ruote contro di esse.

(a) (b)

Figura 5. (a) La stazione di ricarica di Speedy. (b) Particolare dei contatti.

3. La soluzione adottata

3.1. Navigazione e visione

Nel contesto descritto sono individuabili due principali componenti. La prima legata alla movimentazione del robot, e una seconda legata all'elaborazione del flusso video acquisito dalla webcam. Sfruttando questa distinzione, il progetto è stato strutturato in due programmi separati che comunicano tra loro, con un programma che gestisce la navigazione e l'altro la visione. Per la comunicazione tra i due software è stato scelto il meccanismo dei *socket*, implementando in questo modo un'architettura client-server. Il programma di navigazione costituisce la parte client, mentre quello di visione costituisce la parte server, come schematizzato in figura 6.

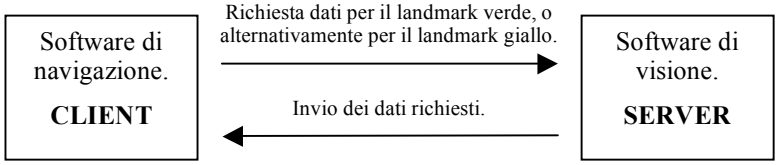


Figura 6. Architettura di comunicazione tra i software di navigazione e visione.

Secondo le esigenze, il client invia una richiesta per i dati del landmark verde o per quello giallo usando, rispettivamente, le istruzioni:

```
write(socket_ID, REQUEST_GREEN, sizeof(REQUEST_GREEN));
write(socket_ID, REQUEST_YELLOW, sizeof(REQUEST_YELLOW));
```

In queste istruzioni `socket_ID` è l'identificatore del socket creato. I dati relativi al landmark verde sono costituiti dalle coordinate, rispetto all'angolo in alto a sinistra dell'immagine, del baricentro della porzione di landmark rilevato. L'intervallo dei valori va da 0 a 320 per l'ascissa, e da 0 a 240 per l'ordinata. Le coordinate relative al landmark giallo si riferiscono, invece, al punto medio della porzione di landmark giallo all'interno di una fascia a ridosso della base dell'immagine. L'intervallo dei valori va da -50 a 50 per l'ascissa, per cui a 0 corrisponde il landmark giallo posizionato a centro immagine. In questo caso, invece, il valore dell'ordinata, fisso e costante, non è significativo.

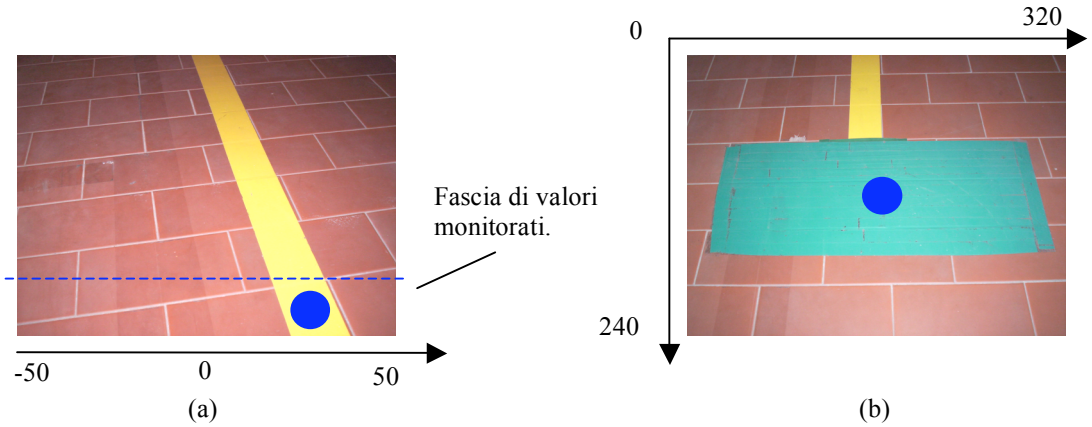


Figura 7. Baricentro calcolato e sue coordinate. (a) Landmark giallo. (b) Landmark verde.

Nel caso in cui non venga rilevata nessuna porzione di landmark, per entrambe le coordinate viene fornito il valore di default 999. La comunicazione di questi valori viene effettuata trasmettendo una stringa di messaggio con il seguente formato:

"ascissa<spazio>ordinata"

In ricezione tale stringa viene memorizzata in un buffer di caratteri denominato `values_BUFFER`. I valori numerici delle coordinate vengono, quindi, letti da questo buffer con un'istruzione `sscanf`.

3.2. L'algoritmo di navigazione

- **Per la descrizione dell'algoritmo di risoluzione ideato è stata utilizzata una struttura a stati. Tale architettura ricalca quella del software sviluppato, anche se nel codice non è stata utilizzata una variabile di stato globale con le etichette definite nel seguito.**

Per ottenere l'obiettivo richiesto, la movimentazione del robot è stata suddivisa in quattro stati o fasi principali. La successione cronologica di questi quattro stati, conduce il robot dall'iniziale ricerca del landmark verde sino al raggiungimento della propria stazione di ricarica. La prima fase è stata identificata con la ricerca del landmark verde, utilizzando per la navigazione del robot, durante questa operazione, il comportamento "wander". Nella seconda fase è stato confinato il compito di raggiungere il landmark verde, utilizzando le informazioni fornite dal sistema di visione. La terza fase rappresenta l'operazione di allineamento iniziale del robot rispetto al landmark giallo. La quarta, ed ultima, fase racchiude l'avanzamento del robot lungo il landmark giallo sino al raggiungimento della stazione. In figura 8 è mostrata l'architettura di questi stati, ordinati cronologicamente.

La soluzione adottata nell'implementazione del software di navigazione è basata sul meccanismo delle azioni del pacchetto ARIA. Le quattro azioni principali, che sono state sviluppate, sono:

- "WanderForGreenLandmark": gestisce la movimentazione del robot durante la ricerca del landmark rettangolare di colore verde.
- "GoToGreenLandmark": contiene l'algoritmo necessario affinché il robot raggiunga il landmark verde, individuato dal sistema di visione elaborando le immagini acquisite dalla webcam.
- "SearchYellowLandmark": questa azione si occupa del necessario allineamento iniziale del robot rispetto al landmark rappresentato dalla striscia di nastro adesivo giallo.
- "KeepYellowLandmark": mantenendo allineato il robot al landmark giallo, lo conduce sino al conclusivo posizionamento dentro la stazione di ricarica..

Della classe *ArAction* sono state usate le azioni:

- ArActionStallRecover: utilizzata per gestire un'eventuale situazione di stallo del robot. Il nome utilizzato nel programma per questa azione è "StallRecover".
- ArActionAvoidFront: utilizzata per evitare ed aggirare eventuali ostacoli incontrati lungo la traiettoria seguita. Il nome di questa azione all'interno del programma è "AvoidFront". Per questa tipologia di azione, oltre al nome, sono configurabili altri quattro parametri. La distanza a cui rilevare gli ostacoli è stata impostata a 400 mm, la velocità del robot durante l'aggiramento di un ostacolo è 60 mm/sec, l'intensità di rotazione è di 15°, mentre con **`false`** viene indicata l'indisponibilità di sensori IR. La corrispondente riga di codice è riportata di seguito:

```
ArActionAvoidFront avoidFront("AvoidFront", 400, 60, 15, false);
```

Figura 8. Schematizzazione dell'architettura a stati.

In figura 9 è riportata la lista delle azioni utilizzate, ordinate in base ai valori di priorità assegnati. A seconda dello stato in cui ci si trova, determinate azioni sono attive mentre le restanti risultano disattivate. È stato preferito l'approccio della disattivazione, probabilmente meno oneroso, rispetto a quello della rimozione e successiva riaggiunta alla lista delle azioni. In questo modo le azioni vengono inibite con l'istruzione *deactivate*, e risvegliate con l'istruzione *activate*, quando necessario. Le quattro azioni principali presentate in precedenza sono mutuamente esclusive, ossia quando una di tali azioni è attiva le altre tre sono disattive. Le operazioni di attivazione e disattivazione sono controllate direttamente dall'interno della struttura *fire* di queste quattro azioni. Ad ogni stato corrisponde un'azione tra quelle definite appositamente per il programma. Questa corrispondenza, tra stati e azioni, è mostrata in figura 10.

Figura 9. Lista delle azioni inserite.

STATO “Ricerca landmark verde”	È attiva l’azione WanderForGreenLandmark.
STATO “Raggiungere landmark verde”	È attiva l’azione GoToGreenLandmark.
STATO “Allineamento”	È attiva l’azione SearchYellowLandmark.
STATO “Seguire landmark giallo”	È attiva l’azione KeepYellowLandmark.

Figura 10. Corrispondenza tra gli stati definiti e le azioni del programma.

3.3. Lo stato “Ricerca landmark verde”

Si tratta dello stato iniziale, durante il quale il robot si muove all’interno dell’ambiente, evitando gli ostacoli, alla ricerca del landmark verde. Come mostrato dal diagramma in figura 11 la condizione di uscita da questo stato è l’identificazione del landmark verde. Individuato tale riferimento si passa nello stato”Raggiungere landmark verde”.

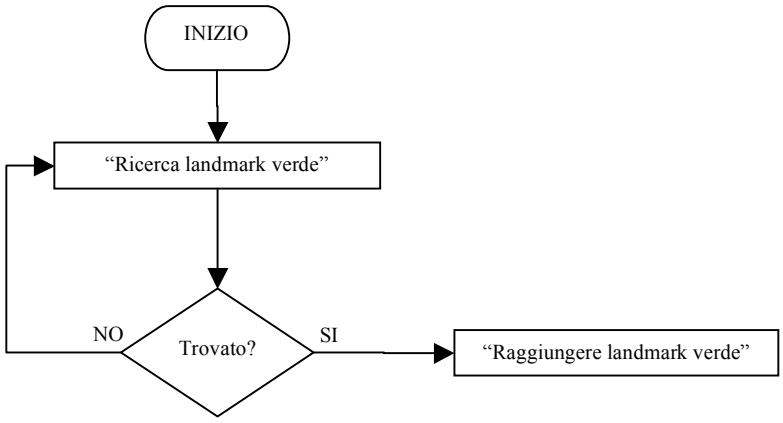


Figura 11. Diagramma di flusso per la fase “Ricerca landmark verde”.

Oltre all'azione WanderForGreenLandmark in questa fase sono attive le azioni StallRecover a cui è affidata la massima priorità, e l'azione AvoidFront con priorità intermedia ai valori delle due azioni precedenti. In figura 12 è stato schematizzato il campo di visione della webcam montata su Speedy. Il raggio massimo è stato misurato e vale approssimativamente 140 cm. Questo significa che per poter individuare il landmark verde, il robot deve giungere a una distanza inferiore a 140 cm dal landmark stesso. Considerando le dimensioni dello spazio, all'interno del laboratorio, in cui il robot può muoversi, la ricerca del landmark verde può divenire un'operazione difficoltosa necessitando di intervalli di tempo piuttosto lunghi.

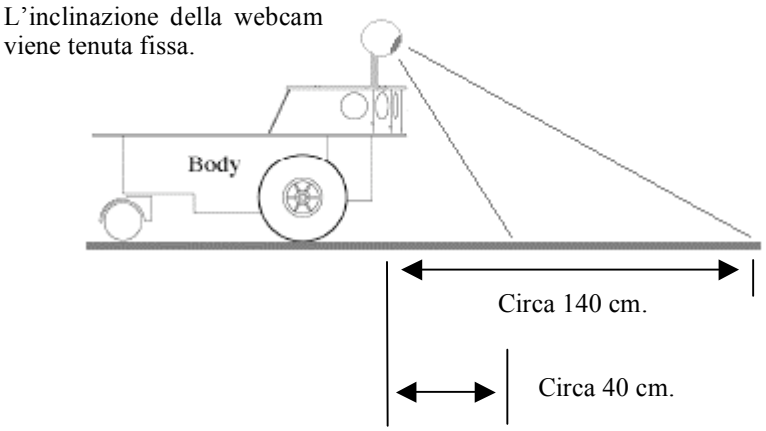


Figura 12. Campo di visione della webcam.

In giallo vengono indicate le azioni disattivate.

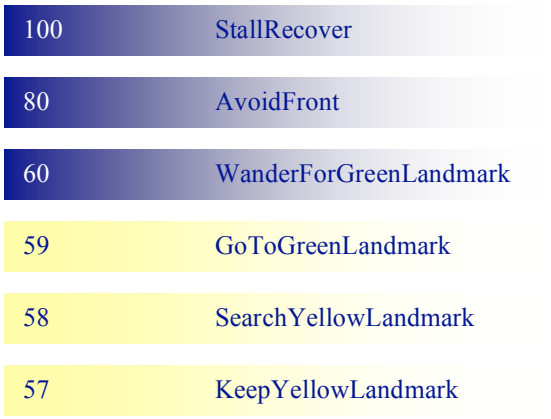


Figura 13. Le azioni attive durante la fase "Ricerca landmark verde".

3.3.1. L'azione "WanderForGreenLandmark"

Come suggerito dal nome di questa azione, il robot tiene un comportamento tipo "wander" durante la ricerca del landmark verde. Una volta identificato tale riferimento questa azione si disattiva lasciando il controllo all'azione GoToGreenLandmark. La velocità del robot durante la fase "wander" è stata fissata al valore di 120 mm/sec. Fintanto che non ci sono ostacoli in un raggio di 400 mm il robot viaggia in linea retta con la velocità dichiarata. Al rilevamento di un ostacolo l'azione AvoidFront gestisce la situazione secondo i parametri impostati, imponendosi sull'azione WanderForGreenLandmark grazie al valore di priorità più elevato. Nel caso in cui l'azione WanderForGreenLandmark sia stata riattivata dalle azioniGoToGreenLandmark e KeepYellowLandamrk, in seguito alla perdita dei landmark da parte del sistema di visione, il comportamento "wander" viene fatto precedere da una panoramica del robot.

Questa modalità è stata scelta considerando le dimensioni del sistema di localizzazione in rapporto al campo di visione della webcam. La probabilità che compiendo una panoramica dal punto in cui è stato perso il landmark, sia possibile individuare quello verde, è elevata. Questo principio viene illustrato in figura 14. Ovviamente, se dopo aver terminato l'azione di rotazione della panoramica il landmark verde non è stato individuato, viene ripristinato il tradizionale comportamento "wander". Con riferimento al listato del codice, riportato in appendice, quando la variabile enumerativa `WanderForGreenLandmark_Flag` contiene il valore `WANDER` l'azione comanda il comportamento "wander". Viceversa, il valore `PAN_ROTATION` comanda l'azione di rotazione per la panoramica.

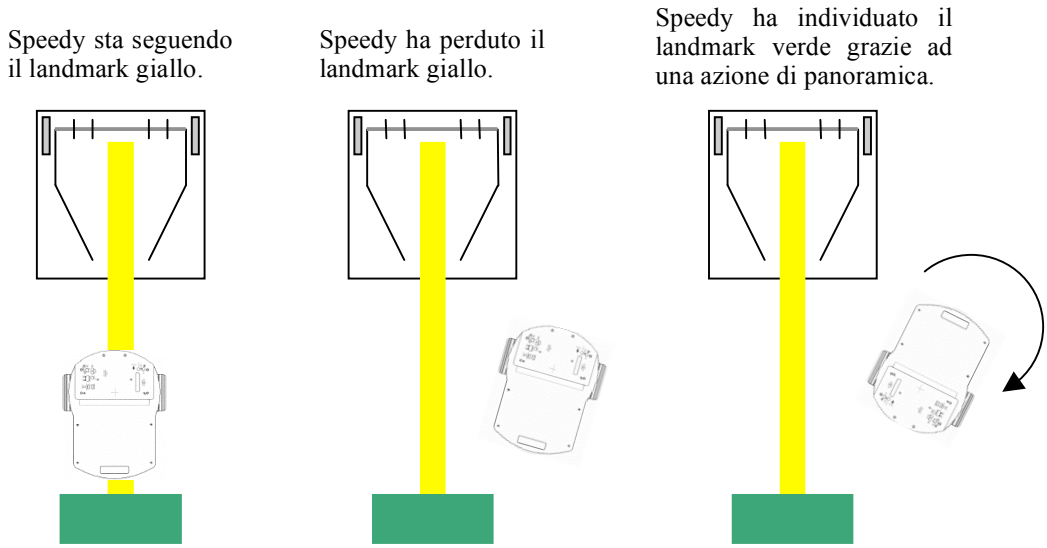


Figura 14. Sequenza che illustra il recupero del landmark verde.

3.4. Lo stato "Raggiungere landmark verde"

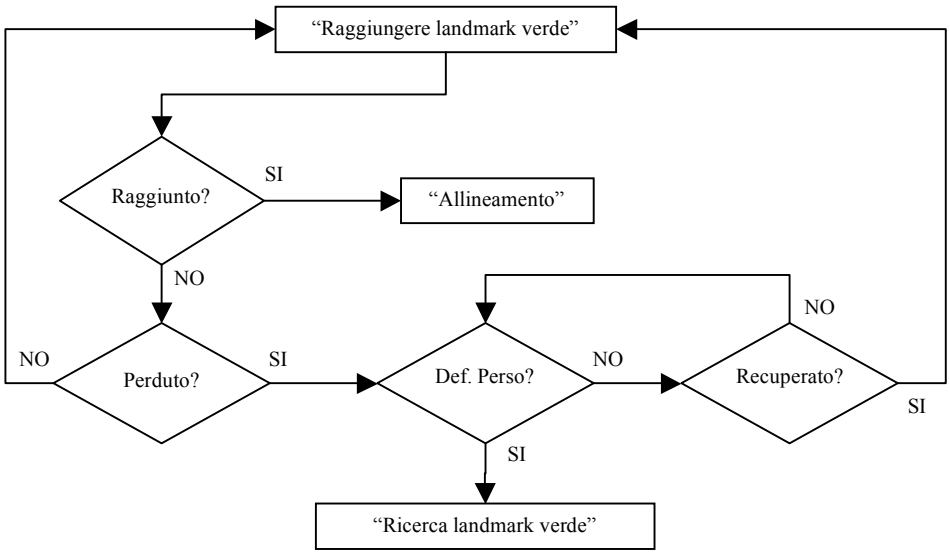


Figura 15. Diagramma di flusso per la fase "Raggiungere landmark verde".

A questo punto il landmark verde è stato individuato dal sistema di visione e il robot lo deve raggiungere. Ottenuto questo obiettivo, c'è il passaggio nello stato "Allineamento". Assieme all'azione GoToGreenLandmark restano attive l'azione di recupero da un'eventuale situazione di stallo e l'azione

di aggiramento degli ostacoli. Durante questa fase l'algoritmo di navigazione deve saper gestire l'eventualità di una temporanea perdita del landmark da parte del sistema di visione. Questa situazione può essere causata da un ostacolo in movimento che si pone sulla traiettoria seguita dal robot, ma sopratutto dalla fragilità del sistema con cui vengono acquisite le immagini.

In giallo vengono indicate le azioni disattivate.

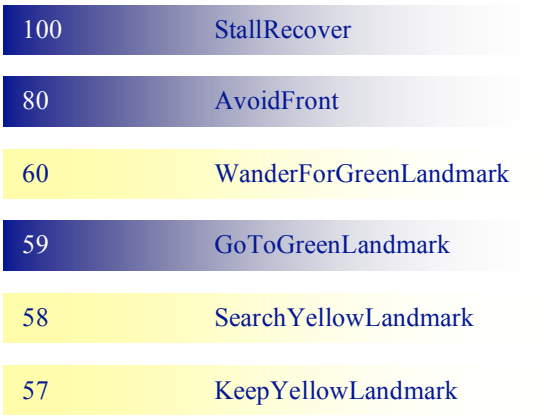


Figura 16. Le azioni attive durante la fase "Raggiungere landmark verde".

3.4.1. L'azione GoToGreenLandmark

L'azione GoToGreenLandmark conduce il robot sino al landmark verde utilizzando le coordinate inviate dal software di visione. Le istruzioni che movimentano il robot mentre segue il landmark verde sono:

```
myDesired.setVel(GO_VELOCITY); #define GO_VELOCITY 100
myDesired.setdeltaHeading((X_OFFSET-green_X)/GO_FACTOR); #define X_OFFSET 160
```

La prima istruzione impone al robot una velocità traslazionale pari a 100 mm/sec. Nella seconda la variabile `green_X` contiene l'ascissa del baricentro del landmark, che può assumere un valore compreso tra 0 e 320, mentre la costante `GO_FACTOR` è stata determinata sperimentalmente. La seconda istruzione è usata per dirigere il robot verso il landmark verde. Viene impostata una rotazione la cui intensità è proporzionale allo scostamento del baricentro, della porzione di landmark acquisita dalla webcam, rispetto al centro dell'immagine. Il verso di rotazione è tale da mantenere il baricentro a centro immagine. Considerato il sistema delle coordinate, mentre il robot si avvicina al landmark il valore dell'ordinata del baricentro aumenta, tale valore è contenuto nella variabile `green_Y`. Quando questa variabile supera il valore `Y_THRESHOLD` considero il robot arrivato in prossimità del landmark. A questo punto, il robot viene fatto avanzare in linea retta, a velocità ridotta e per un breve periodo di tempo, prima di passare il controllo all'azione SearchYellowLandmark. Nel caso in cui il landmark verde risulti temporaneamente perso, ad ogni ciclo viene decrementata una variabile contatore locale, denominata `lost_counter` ed inizialmente posta al valore `GREEN_LOST`, mentre la velocità del robot è gradualmente diminuita con l'istruzione:

```
myDesired.setVel((lost_counter/GREEN_LOST)*GO_VELOCITY);
```

Se il landmark verde è recuperato prima dell'azzeramento del contatore, viene ripristinato il comportamento descritto in precedenza. Viceversa, prima di disattivare questa azione, viene riattivata l'azione WanderForGreenLandmark. Mettendo al valore **`true`** la variabile `rotation_for_recover` viene comunicato di effettuare una panoramica iniziale prima di passare al comportamento "wander". Con riferimento al listato del codice, riportato in appendice, quando la variabile locale enumerativa `GoToGreenLandmark_Flag` contiene il valore `GO` il landmark verde risulta agganciato dal sistema di visione, e il robot viene comandato di conseguenza. Il valore `LOST_TEMP` indica che il landmark verde è stato temporaneamente perso, e il robot riceve le opportune istruzioni per diminuire gradualmente la velocità. Infine, il caso etichettato con `NEAR` gestisce la fase conclusiva di avvicinamento al landmark.

3.5. Lo stato “Allineamento”

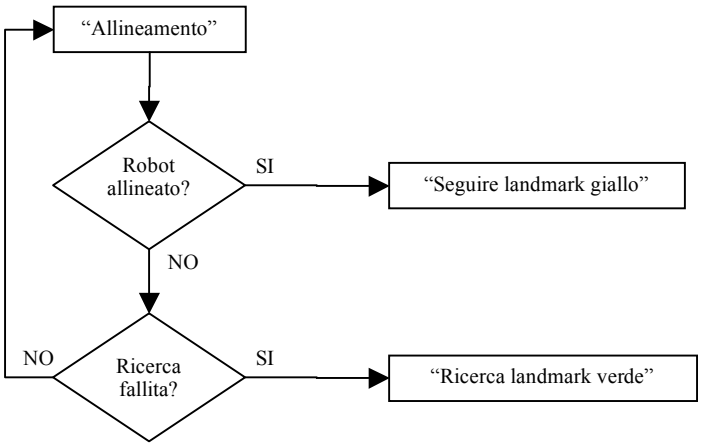


Figura 17. Diagramma di flusso per la fase “Allineamento”.

Il robot può raggiungere il landmark verde da diverse direzioni. Diviene quindi necessario prevedere una fase durante la quale il robot corregga il proprio allineamento rispetto al landmark giallo. Questa operazione viene effettuata, essenzialmente, facendo ruotare il robot a velocità traslazionale nulla. La verifica dell’allineamento viene eseguita controllando ad ogni ciclo il valore del baricentro del landmark giallo. È ovviamente prevista una condizione di fallimento, nel qual caso si ritorna a cercare il landmark verde. Ottenuto il corretto allineamento, il controllo passa nello stato “Seguire landmark giallo”.

In giallo vengono indicate le azioni disattivate.

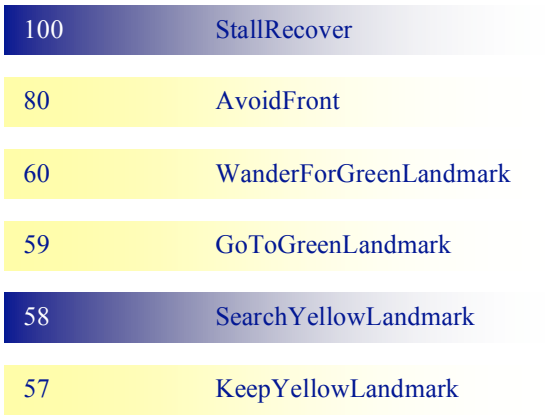


Figura 18. Le azioni attive durante la fase “Allineamento”.

3.5.1. L’azione SearchYellowLandmark

Dopo aver raggiunto il landmark verde, Speedy effettua l’allineamento con il landmark giallo tramite questa azione. La velocità traslazionale del robot è mantenuta nulla, mentre ad ogni ciclo viene impostata una rotazione di 10° verso sinistra col comando:

```
myDesired.setDeltaHeading(SEARCH_ANGLE); #define SEARCH_ANGLE -10
```

L’azione di rotazione del robot viene interrotta non appena il valore assoluto della variabile `yellow_x` diviene inferiore alla tolleranza fissata, identificata nel codice come `ALIGNMENT_TOLERANCE`. A questo punto il robot viene fermato per qualche ciclo prima di passare il controllo all’azione KeepYellowLandmark. Durante l’azione di rotazione, ad ogni ciclo, viene incrementata una variabile contatore locale, denominata `counter`, posta inizialmente a zero. Se tale variabile raggiunge il limite

`SEARCH_LIMIT` e il landmark giallo non è ancora stato trovato, l'allineamento con quest'ultimo viene considerato fallito. Viene, allora, riattivata l'azione WanderForGreenLandmark e disattivata SearchYellowLandmark. In questo caso, la variabile `rotation for recover` viene lasciata al valore **`false`**, in modo tale da non far precedere al comportamento tipo wander una panoramica completa del robot. Considerando, infatti, che l'allineamento appena fallito è una azione di rotazione, un'ulteriore panoramica risulta ridondante. Con riferimento al listato del codice riportato in appendice, quando la variabile enumerativa locale `SearchYellowLandmark_Flag` contiene il valore `ROTATION` l'azione impartisce i comandi relativi alla rotazione per allinearsi col landmark giallo. Nel caso etichettato come `STOP_BEFORE_KEEP`, il robot ha ormai compiuto l'allineamento, e viene fermato per un breve periodo di tempo prima di attivare l'azione KeepYellowLandmark.

3.6. Lo stato "Seguire landmark giallo"

Si tratta dello stato conclusivo, durante il quale il robot seguendo il landmark giallo raggiunge la propria stazione di ricarica. Anche in questa fase, deve essere prevista la gestione di una temporanea perdita del landmark giallo da parte del sistema di visione, per le medesime ragioni citate in precedenza. In questo caso, a rendere difficoltosa ed intermittente la rilevazione del landmark giallo, in particolar modo sono le condizioni di luminosità dell'ambiente circostante. Nel caso in cui il landmark giallo venisse perso in modo definitivo, si ritorna a cercare il landmark verde comandando un'iniziale panoramica da parte del robot. Se l'intero sistema ha funzionato in maniera corretta, questo stato si conclude col posizionamento di Speedy nella sua stazione di ricarica.

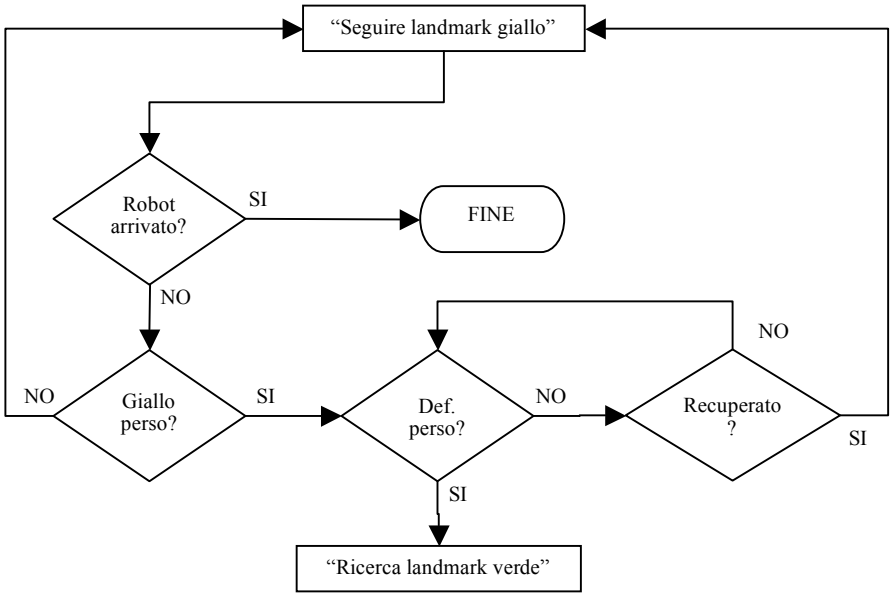


Figura 19. Diagramma di flusso per la fase "Seguire landmark giallo".

Come si nota dalla figura 20, durante la fase "Seguire landmark giallo", l'azione per gestire un'eventuale situazione di stallo del robot è disattivata. Questa operazione è resa necessaria dall'adozione di un metodo basato sulla rilevazione dello stallo dei motori per determinare quando il robot si è posizionato correttamente dentro la propria stazione di ricarica. Come l'azione StallRecover, anche l'azione AvoidFront resta disattivata durante questa fase, ovviamente per permettere al robot il corretto raggungimento della stazione.

In giallo vengono indicate le azioni disattivate.

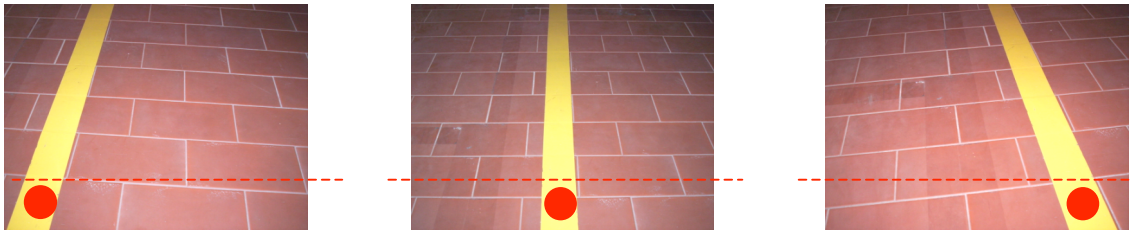
100	StallRecover
80	AvoidFront
60	WanderForGreenLandmark
59	GoToGreenLandmark
58	SearchYellowLandmark
57	KeepYellowLandmark

Figura 20. Le azioni attive durante la fase “Seguire landmark giallo”.

3.6.1. L’azione KeepYellowLandmark

L’azione KeepYellowLandmark conduce Speedy fino al suo corretto posizionamento all’interno della propria stazione di ricarica. Negli intervalli in cui il landmark giallo è constantemente rilevato la velocità del robot viene impostata a 80 mm/sec, mentre il corretto allineamento viene mantenuto con il seguente comando di rotazione, in cui il valore di `KEEP_FACTOR` è stato determinato tramite prove sperimentali:

```
myDesired.setDeltaHeading(-yellow_X/KEEP_FACTOR);
```



(a) (b) (c)

Figura 21. Le tre situazioni più significative relative all’allineamento.

Nella situazione di figura 21 (b) la variabile `yellow_X` contiene 0 e la rotazione impartita è nulla. Nei casi (a) e (c), la variabile `yellow_X` vale rispettivamente -50 e 50 in corrispondenza dei quali si ha il massimo valore di rotazione a destra e sinistra. Negli intervalli in cui la rilevazione del landmark giallo fallisce, ad ogni ciclo viene decrementata una variabile locale denominata `lost_counter`, inizialmente posta al valore `YELLOW_LOST`. Mentre `lost_counter` resta positiva, il landmark giallo viene considerato solo temporaneamente perso e la velocità del robot viene gradualmente diminuita tramite l’istruzione:

```
myDesired.setVel((lost_counter/YELLOW_LOST)*KEEP_VELOCITY);
```

Se la rilevazione del landmark giallo continua a fallire, l’azzeramento di tale contatore determina la disattivazione di KeepYellowLandmark e la riattivazione dell’azione WanderForGreenLandmark. Impostando al valore **`true`** la variable `rotation_for_recover`, l’azione per la ricerca del landmark verde fa precedere al comportamento tipo “wander” una panoramica completa del robot. Nel caso in cui il landmark giallo venisse recuperato, sono reimpostati i valori di velocità e rotazione, per la correzione dell’errore di allineamento, descritti precedentemente. La determinazione dell’avvenuto posizionamento del robot all’interno della stazione di ricarica è ottenuto controllando, in maniera indipendente l’uno dall’altro, lo stato di stallo dei due motori. Per monitorare la condizione dei due motori sono state usate due funzioni della classe ArRobot:

- bool isLeftMotorStalled(void)
- bool isRightMotorStalled(void)

quando entrambe queste funzioni booleane ritornano il valore **`true`** viene determinata la terminazione del programma. Questa condizione corrisponde, infatti, alla situazione in cui entrambe le ruote di Speedy sono arrivate contro le due squadrette di fermo corsa. A questo punto con i comandi *stop* e *disconnect* viene rispettivamente interrotto il ciclo delle azioni e richiesta la sconnessione dal robot. Con riferimento al listato del codice riportato in appendice, quando la variabile enumerativa locale `KeepYellowLandmark_Flag` contiene il valore `GO`, il robot sta seguendo il landmark giallo e l'azione impartisce gli appropriati comandi di avanzamento e correzione dell'allineamento. Nel caso etichettato come `LOST_TEMP` viene, invece, gestita la temporanea perdita del landmark giallo.

Figura 22. Il sistema ha funzionato correttamente e Speedy si è posizinato nella sua stazione di ricarica.

4. Modalità operative

I paragrafi che seguono descrivono gli elementi necessari e le modalità operative per eseguire in maniera corretta i software di navigazione e visione che sono stati creati.

4.1. Componenti necessari

Il software di navigazione descritto è stato sviluppato in ambiente Linux. Tale programma deve quindi essere preso in considerazione su un calcolatore con questo sistema operativo. Risulta necessaria l'installazione delle librerie ARIA, che di default vengono posizionate in */usr/local/Aria*. Per quanto riguarda il software di visione, inoltre, è necessaria l'installazione delle librerie *vislib* aggiornate alla versione 1.8. I programmi sviluppati sono, ovviamente, specifici al sistema di localizzazione e ricarica descritto, e in particolare al robot Speedy. Di conseguenza, è auspicabile il loro utilizzo all'interno del laboratorio assieme al robot citato. Nel seguito sono riportate le versioni dei tool software utilizzati durante lo sviluppo del progetto:

- Compilatore gcc 4.0.3.
- Pacchetto ARIA 2.4-1 e pacchetto vislib 1.8-v4l, reperibili sul sito web di ActivMedia Robotics.
- Simulatore MobileSim 0.3.0-0, reperibile sul sito web di ActivMedia Robotics.

> ➢ **Non utilizzare la versione 4.1 del compilatore gcc. Tentando di compilare i file sorgenti con questa versione si presentano errori che bloccano la procedura di compilazione, impedendo di ottenere il file binario desiderato.**

4.2. Modalità di installazione

In una directory denominata *Robotica* vengono forniti i cinque file sorgente, con estensione .cpp, assieme al makefile, appositamente creato, e un file di testo README. Dopo aver copiato tale directory nella posizione che si desidera, aprire una finestra di terminale ed entrare nella directory creata utilizzando il comando *cd*. Digitare il comado *make Robotica* e premere invio. A questo punto sono stati creati il file binario denominato Robotica e un file oggetto omonimo. Supponendo, ad esempio, di aver copiato la directory *Robotica* nella posizione */usr/local* i comandi da impartire sono i seguenti:

```
[user@generic]$ cd /usr/local/Robotica
[user@generic]$ make Robotica
```

Il "prompt dei comandi" mostrato è puramente a scopo indicativo. Per quanto riguarda il programma di visione, se non si ha già ha disposizione il file binario, basta seguire una procedura simile a quanto descritto precedentemente per effettuare la compilazione del sorgente. Dopo aver aperto una finestra di terminale, entrare nella directory, originariamente chiamata *visione*, contenente il file *visione.c* e il makefile appropriato, a questo punto basta digitare il comando *make* e premere invio. I due programmi devono essere mandati in esecuzione da due finestre di terminale separate, usando rispettivamente i comandi *./visione* e *./Robotica*. Il primo programma da eseguire è *visione*, il quale si pone in attesa della richiesta dei dati, a questo punto è possibile eseguire *Robotica*.

- **I software di navigazione e visione non possono essere utilizzati separatamente. Aprendo due finestre di terminale indipendenti, deve essere eseguito per primo il programma di visione, successivamente quello di navigazione.**

4.3. Modalità di taratura

Non ci sono particolari operazioni di taratura da effettuare, se non variare le condizioni di luminosità dell'ambiente, nel caso il comportamento del robot presenti evidenti anomalie, dovute a eccessive interferenze nel sistema di visione. Questa situazione è descritta, in modo più approfondito, nel paragrafo successivo.

4.4. Avvertenze

Le avvertenze più importanti sono relative alle condizioni di luminosità all'interno del laboratorio. Il metodo con cui vengono elaborate le immagini risente in maniera significativa di variazioni, anche modeste, del valore di luminosità dell'ambiente. Se fonti luminose provocano delle ombre che scuriscono l'immagine acquisita dalla webcam, l'identificazione del landmark giallo, in particolare, diviene molto più difficoltosa. Per quanto riguarda l'identificazione dei landmark da parte del sistema di visione, i migliori risultati sono stati ottenuti mantenendo una buona illuminazione di fronte al robot, in modo da illuminare sufficientemente i landmark senza creare fastidiosi effetti d'ombra che possano interferire con l'acquisizione delle immagini.. Ad esempio, durante le prove, è stato tenuto acceso un faretto, rivolto verso l'alto, che si trova alle spalle della stazione di ricarica.

5. Conclusioni e sviluppi futuri

I risultati ottenuti dalle prove che sono state effettuate possono essere considerati sufficientemente positivi. L'utilizzo di un comportamento puramente "wander" costringe, facilmente, il robot a vagare a lungo all'interno del laboratorio prima che sia individuato il landmark verde. Sono state provate diverse soluzioni per cercare di affinare il comportamento del robot durante questa fase. Ad esempio, si è provato a sostituire il semplice avanzamento in linea retta, orientando il robot verso la direzione lungo la quale i sonar indicano il raggio di libertà più ampio. In un approccio differente, si è provato ad aggiungere al movimento traslazionale una rotazione con verso e intensità pseudo-casuali, utilizzando la funzione `rand`. I risultati, tuttavia, non hanno mostrato vantaggi significativi facendo preferire il più semplice comportamento "wander". Probabilmente, l'unico modo per raggiungere efficacemente la regione in cui si trova il landmark verde è di utilizzare una navigazione con la mappa del laboratorio. Per

quanto riguarda il sistema di visione, oltre ai già citati problemi di sensibilità, durante le prove i landmark sono stati sporadicamente identificati in maniera errata e confusi con oggetti diversi. Nuove soluzioni posoono essere implementate adottando algoritmi flessibili che si adattano dinamicamente alle condizioni di luminosità dell'ambiente. Infine, considerando il legame inscindibile tra la parte di navigazione e quella di visione, è ipotizzabile una futura integrazione tra i due software comprendente numerose ottimizzazioni rispetto a questa prima versione.

6. Appendice A

Di seguito è riportato il listato del codice del software di navigazione sviluppato. Dal file Robotica.cpp, si può notare, come è stato scelto di mantenere separate le operazioni di richiesta e ricezione dei dati al software di visione, tramite socket, dal ciclo delle azioni. Infatti, sfruttando l'architettura multi-thread di ARIA, parallelamente al ciclo delle azioni viene eseguito, in maniera indipendente, un ciclo while che si occupa della richiesta e ricezione dei dati. Il periodo che determina la frequenza con cui viene scandita la lista delle azioni viene impostato all'inizio del programma tramite l'istruzione *setCycleTime*. La frequenza con cui vengono richiesti i dati al software di visione viene determinata, invece, con un'istruzione *sleep* posta all'interno del ciclo while. In questo modo, è possibile variare questi due parametri a piacimento, ottenendo un'ampia flessibilità.

6.1. Robotica.cpp

```
// File: Robotica.cpp

#define BUFFER_LENGTH 256
#define LANDMARK_LOST 999
#define REQUEST_GREEN "0"
#define REQUEST_YELLOW "1"
#define SEND_FINISH "2"
#define PORT 20000
#define VISIO_SERVER_IP "127.0.0.1"

#include "Aria.h"

enum Request_TYPE {GREEN, YELLOW} type_of_request;
char values_BUFFER[BUFFER_LENGTH];
bool Robotica_END = false;
bool rotation_for_recover = false;

ArAction *StallRecover_POINTER;
ArAction *AvoidFront_POINTER;
ArAction *WanderForGreenLandmark_POINTER;
ArAction *GoToGreenLandmark_POINTER;
ArAction *SearchYellowLandmark_POINTER;
ArAction *KeepYellowLandmark_POINTER;

#include "WanderForGreenLandmark.cpp"
#include "GoToGreenLandmark.cpp"
#include "SearchYellowLandmark.cpp"
#include "KeepYellowLandmark.cpp"

#include <netinet/in.h>
#include <sys/socket.h>
#include <arpa/inet.h>

int main(int argc, char **argv)
{
  int socket_ID;
  struct sockaddr_in visio_server;

  // Definizione di robot come istanza della classe ArRobot.
  ArRobot robot;

  // Definizione di sonar come istanza della classe ArSonarDevice.
  ArSonarDevice sonar;
  ArKeyHandler keyHandler;
```

```
// Definizione delle azioni da aggiungere.
ArActionStallRecover recover("StallRecover");
ArActionAvoidFront avoidFront("AvoidFront", 400, 60, 15, false);
ActionWanderForGreenLandmark wanderForGreenLandmark;
ActionGoToGreenLandmark goToGreenLandmark;
ActionSearchYellowLandmark searchYellowLandmark;
ActionKeepYellowLandmark keepYellowLandmark;

// Creazione del socket.
socket_ID = socket(AF_INET, SOCK_STREAM, 0);
if(socket_ID < 0)
{
  perror("Client: socket error");
  exit(0);
}
visio_server.sin_family = AF_INET;
visio_server.sin_addr.s_addr = inet_addr(VISIO_SERVER_IP);
visio_server.sin_port = htons(PORT);

// Richiesta di connessione al server.
if(connect(socket_ID, (struct sockaddr *)&visio_server,
   sizeof(visio_server)) < 0)
{
  perror("Client: connection failed");
  close(socket_ID);
  exit(0);
}

// Segmento relativo alle fasi di inizializzazione e connessione al robot.
Aria::init();
ArSimpleConnector connector(&argc, argv);
if(!connector.parseArgs() || argc > 1)
{
  connector.logOptions();
  exit(1);
}
Aria::setKeyHandler(&keyHandler);
robot.addRangeDevice(&sonar);
robot.attachKeyHandler(&keyHandler);
if(!connector.connectRobot(&robot))
{
  printf("Could not connect to robot... exiting\n");
  Aria::shutdown();
  return 1;
}

// Aggiunta delle azioni, che governano il comportamento del robot.
robot.addAction(&recover, 100);
robot.addAction(&avoidFront, 80);
robot.addAction(&wanderForGreenLandmark, 60);
robot.addAction(&goToGreenLandmark, 59);
robot.addAction(&searchYellowLandmark, 58);
robot.addAction(&keepYellowLandmark, 57);

// Inizializzazione dei puntatori alle azioni.
StallRecover_POINTER = robot.findAction("StallRecover");
AvoidFront_POINTER = robot.findAction("AvoidFront");
WanderForGreenLandmark_POINTER = robot.findAction("WanderForGreenLandmark");
GoToGreenLandmark_POINTER = robot.findAction("GoToGreenLandmark");
SearchYellowLandmark_POINTER = robot.findAction("SearchYellowLandmark");
KeepYellowLandmark_POINTER = robot.findAction("KeepYellowLandmark");

// Impostazione del periodo di ciclo per l'esecuzione delle azioni.
// Tale periodo è impostato al valore di 300 msec.
robot.setCycleTime(300);

// Massima velocità di rotazione in valore assoluto.
robot.setAbsoluteMaxRotVel(20);

// Richiesta dei primi dati al programma di visione.
type_of_request = GREEN;
write(socket_ID, REQUEST_GREEN, sizeof(REQUEST_GREEN));
```

```
  read(socket_ID, values_BUFFER, sizeof(values_BUFFER));
  ArUtil::sleep(100);

  // Comando di partenza del robot.
  robot.runAsync(true);

  // Ciclo per la richiesta dei dati al programma di visione.
  while(Robotica_END == false)
  {
    switch(type_of_request)
    {
    case GREEN:
      write(socket_ID, REQUEST_GREEN, sizeof(REQUEST_GREEN));
      break;
    case YELLOW:
      write(socket_ID, REQUEST_YELLOW, sizeof(REQUEST_YELLOW));
    }
    read(socket_ID, values_BUFFER, sizeof(values_BUFFER));
    ArUtil::sleep(100);
  }

  // Disattivazione di tutte le azioni.
  robot.lock();
  robot.deactivateActions();
  robot.unlock();

  // Invio del comando di terminazione al programma di visione.
  write(socket_ID, SEND_FINISH, sizeof(SEND_FINISH));

  // Chiusura della connessione al socket.
  if(close(socket_ID) < 0)
    exit(-1);

  Aria::shutdown();

  return 0;
}
```

6.2. WanderForGreenLandmark.cpp

```
// File: WanderForGreenLandmark.cpp

#define WANDER_VELOCITY 120
#define STOP_ROTATION 80
#define WANDER_ANGLE 10

// ****************************************
// Action WanderForGreenLandmark: Prototype.
// ****************************************
class ActionWanderForGreenLandmark : public ArAction
{
public:
  ActionWanderForGreenLandmark(void); // Constructor.
  virtual ~ActionWanderForGreenLandmark(void) {}; // Destructor.
  virtual ArActionDesired *fire(ArActionDesired currentDesired); // Fire.
  virtual void setRobot(ArRobot *robot);
protected:
  ArActionDesired myDesired;
  int green_X, green_Y, counter;
  enum {PAN_ROTATION, WANDER} WanderForGreenLandmark_Flag;
};

// ****************************************
// Action WanderForGreenLandmark: Constructor.
// ****************************************
ActionWanderForGreenLandmark::ActionWanderForGreenLandmark(void) :
ArAction("WanderForGreenLandmark")
{
  WanderForGreenLandmark_Flag = WANDER;
  counter = 0;
}
```

```
// ***************************************
// Action WanderForGreenLandmark: setRobot.
// ***************************************
void ActionWanderForGreenLandmark::setRobot(ArRobot *robot)
{
  myRobot = robot;
}

// *********************************
// Action WanderForGreenLandmark: FIRE.
// *********************************
ArActionDesired *ActionWanderForGreenLandmark::fire(ArActionDesired
currentDesired)
{
  // Se l'azione StallRecover non è attiva viene attivata.
  if(!StallRecover_POINTER->isActive())
    StallRecover_POINTER->activate();

  // Se l'azione AvoidFront non è attiva viene attivata.
  if(!AvoidFront_POINTER->isActive())
    AvoidFront_POINTER->activate();

  myDesired.reset();

  sscanf(values_BUFFER, "%d %d\r\n", &green_X, &green_Y);
  printf("**********************\nWanderForGreenLamdmark:\n");
  printf("Coordinate ricevute %d %d\n", green_X, green_Y);

  if(rotation_for_recover)
    WanderForGreenLandmark_Flag = PAN_ROTATION;

  switch(WanderForGreenLandmark_Flag)
  {
  // --------------------------------------
  // Il robot sta effettuando la rotazione.
  // --------------------------------------
  case PAN_ROTATION:
    rotation_for_recover = false;
    // Durante la rotazione viene individuato il landmark verde.
    if(green_X != LANDMARK_LOST || green_Y != LANDMARK_LOST)
    {
      printf("Landmark verde individuato.\n");
      type_of_request = GREEN;
      counter = 0;
      WanderForGreenLandmark_Flag = WANDER;
      GoToGreenLandmark_POINTER->activate();
      deactivate();
      return NULL;
    }
   // Il robot ha completato la rotazione.
   if(++counter == STOP_ROTATION)
   {
     counter = 0;
     WanderForGreenLandmark_Flag = WANDER;
     return NULL;
   }
   printf("Azione di rotazione del robot.\n");
   myDesired.setVel(0);
   myDesired.setDeltaHeading(WANDER_ANGLE);
   break;
 // ---------------------------------------------
 // Il robot si muove con un comportamento wander.
 // ---------------------------------------------
 case WANDER:
   // Durante il comportamento wander viene individuato il landmark verde.
   if(green_X != LANDMARK_LOST || green_Y != LANDMARK_LOST)
   {
     printf("Landmark verde individuato.\n");
     counter = 0;
     type_of_request = GREEN;
     WanderForGreenLandmark_Flag = WANDER;
     GoToGreenLandmark_POINTER->activate();
     deactivate();
```

```
      return NULL;
    }
    printf("Comportamento wander.\n");
    myDesired.setVel(WANDER_VELOCITY);
    myDesired.setDeltaHeading(0);
  }
  printf("************************\n\n");
  type_of_request = GREEN;
  return &myDesired;
}
```

6.3. GoToGreenLandmark.cpp

```
// File: GoToGreenLandmark.cpp

#define Y_THRESHOLD 220
#define NEAR_STOP 30
#define GO_FACTOR 10
#define GO_VELOCITY 100
#define NEAR_VELOCITY 80
#define GREEN_LOST 20
#define X_OFFSET 160

// ***********************************
// Action GoToGreenLandmark: Prototype.
// ***********************************
class ActionGoToGreenLandmark : public ArAction
{
public:
  ActionGoToGreenLandmark(void); // Constructor.
  virtual ~ActionGoToGreenLandmark(void) {}; // Destructor.
  virtual ArActionDesired *fire(ArActionDesired currentDesired); // Fire.
  virtual void setRobot(ArRobot *robot);

protected:
  ArActionDesired myDesired;
  int green_X, green_Y, lost_counter, counter;
  enum {GO, LOST_TEMP, NEAR} GoToGreenLandmark_Flag;
};

// ************************************
// Action GoToGreenLandmark: Constructor.
// ************************************
ActionGoToGreenLandmark::ActionGoToGreenLandmark(void) :
ArAction("GoToGreenLandmark")
{
  GoToGreenLandmark_Flag = GO;
  lost_counter = GREEN_LOST;
  counter = 0;
}

// ***********************************
// Action GoToGreenLandmark: setRobot.
// ***********************************
void ActionGoToGreenLandmark::setRobot(ArRobot *robot)
{
  myRobot = robot;
  // Inizialmente l'azione GoToGreenLandmark risulta disattivata.
  deactivate();
}

// ******************************
// Action GoToGreenLandmark: FIRE.
// ******************************
ArActionDesired *ActionGoToGreenLandmark::fire(ArActionDesired currentDesired)
{
  myDesired.reset();

  sscanf(values_BUFFER, "%d %d\r\n", &green_X, &green_Y);
  printf("******************\nGoToGreenLandmark:\n");
  printf("Coordinate ricevute %d %d\n", green_X, green_Y);
```

```
switch(GoToGreenLandmark_Flag)
{
// ------------------------------------------
// Il robot si dirige verso il landmark verde.
// ------------------------------------------
case GO:
  // Il robot ha perduto il landmark per la prima volta.
  if(green_X == LANDMARK_LOST && green_Y == LANDMARK_LOST)
  {
    printf("Landmark verde temporaneamente perso.\n");
    GoToGreenLandmark_Flag = LOST_TEMP;
    myDesired.setVel((--lost_counter/GREEN_LOST)*GO_VELOCITY);
  } else
  // Il robot è arrivato in prossimità del landmark.
  if(green_Y > Y_THRESHOLD)
  {
    printf("Speedy è arrivato al landmark verde.\n");
    GoToGreenLandmark_Flag = NEAR;
    myDesired.setVel(NEAR_VELOCITY);
    myDesired.setDeltaHeading(0);
  } else
  // Il robot sta andando verso il landmark verde.
  if(green_Y < Y_THRESHOLD)
  {
    printf("Speedy sta seguendo il landmark verde.\n");
    myDesired.setVel(GO_VELOCITY);
    myDesired.setDeltaHeading((X_OFFSET-green_X)/GO_FACTOR);
  }
  break;
// ------------------------------------------------
// Il robot ha temporaneamente perduto il landmark.
// ------------------------------------------------
case LOST_TEMP:
  // Il landmark verde risulta ancora perso.
  if(green_X == LANDMARK_LOST && green_Y == LANDMARK_LOST)
  {
    // Il riferimento verde è considerato perso in maniera definitiva.
    if(--lost_counter == 0)
    {
      printf("Landmark verde perduto in maniera definitiva.\n");
      lost_counter = GREEN_LOST;
      type_of_request = GREEN;
      WanderForGreenLandmark_POINTER->activate();
      deactivate();
      return NULL;
    }
    printf("Landmark verde temporaneamente perso.\n");
    myDesired.setVel((lost_counter/GREEN_LOST)*GO_VELOCITY);
  } else
  // Il robot ha recuperato il riferimento verde.
  {
    printf("Landmark verde recuperato.\n");
    lost_counter = GREEN_LOST;
    GoToGreenLandmark_Flag = GO;
    myDesired.setVel(GO_VELOCITY);
    myDesired.setDeltaHeading((X_OFFSET-green_X)/GO_FACTOR);
  }
// ------------------------------------------
// Fase conclusiva di avvicinamento al landmark.
// ------------------------------------------
case NEAR:
  printf("Avvicinamento al landmark verde.\n");
  if(++counter == NEAR_STOP)
  {
    counter = 0;
    GoToGreenLandmark_Flag = GO;
    type_of_request = YELLOW;
    SearchYellowLandmark_POINTER->activate();
    deactivate();
    return NULL;
  }
  myDesired.setVel(NEAR_VELOCITY);
  myDesired.setDeltaHeading(0);
```

```
  }
  printf("******************\n\n");
  type_of_request = GREEN;
  return &myDesired;
}
```

6.4. SearchYellowLandmark.cpp

```
// File: SearchYellowLandmark.cpp

#define ALIGNMENT_TOLERANCE 20
#define SEARCH_ANGLE -10
#define SEARCH_LIMIT 100
#define YELLOW_MAX 50
#define STOP_DURATION 5

// ************************************
// Action SearchYellowLandmark: prototype.
// ************************************
class ActionSearchYellowLandmark : public ArAction
{
public:
  ActionSearchYellowLandmark(void); // Constructor.
  virtual ~ActionSearchYellowLandmark(void) {}; // Destructor.
  virtual ArActionDesired *fire(ArActionDesired currentDesired); // Fire.
  virtual void setRobot(ArRobot *robot);

protected:
  ArActionDesired myDesired;
  int yellow_X, yellow_Y, counter;
  enum {ROTATION, STOP_BEFORE_KEEP} SearchYellowLandmark_Flag;
};

// ****************************************
// Action SearchYellowLandmark: Constructor.
// ****************************************
ActionSearchYellowLandmark::ActionSearchYellowLandmark(void) :
ArAction("SearchYellowLandmark")
{
  counter = 0;
  SearchYellowLandmark_Flag = ROTATION;
}

// ************************************
// Action SearchYellowLandmark: setRobot.
// ************************************
void ActionSearchYellowLandmark::setRobot(ArRobot *robot)
{
  myRobot = robot;
  // Inizialmente l'azione SearchYellowLandmark risulta disattivata.
  deactivate();
}

// *********************************
// Action SearchYellowLandmark: FIRE.
// *********************************
ArActionDesired *ActionSearchYellowLandmark::fire(ArActionDesired currentDesired)
{
  // Se l'azione AvoidFront è attiva viene disattivata.
  if(AvoidFront_POINTER->isActive())
    AvoidFront_POINTER->deactivate();

  myDesired.reset();

  sscanf(values_BUFFER, "%d %d\r\n", &yellow_X, &yellow_Y);
  printf("********************\nSearchYellowLandmark:\n");
  printf("Coordinate ricevute %d %d\n", yellow_X, yellow_Y);

  if(yellow_X == LANDMARK_LOST)
    yellow_X = YELLOW_MAX;

  switch(SearchYellowLandmark_Flag)
```

```
  {
  // ------------------------------------------------------------
  // Il robot sta ruotando per allinearsi con la striscia gialla.
  // ------------------------------------------------------------
  case ROTATION:
    // Il landmark giallo non è stato individuato.
    if(++counter == SEARCH_LIMIT)
    {
      printf("Il rilevamento del landmark giallo è fallito.\n");
      counter = 0;
      SearchYellowLandmark_Flag = ROTATION;
      type_of_request = GREEN;
      WanderForGreenLandmark_POINTER->activate();
      deactivate();
      return NULL;
    }
    // Il robot è allineato, entro la tolleranza, con la linea gialla.
    if(abs(yellow_X) < ALIGNMENT_TOLERANCE)
    {
      printf("Speedy è allineato con il riferimento giallo.\n");
      SearchYellowLandmark_Flag = STOP_BEFORE_KEEP;
      type_of_request = YELLOW;
      counter = 0;
      myDesired.setVel(0);
      myDesired.setDeltaHeading(0);
      break;
    }
    printf("Rotazione per allinearsi con il landmark giallo.\n");
    myDesired.setVel(0);
    myDesired.setDeltaHeading(SEARCH_ANGLE);
    break;
  // ------------------------------------------------------------------
  // Fermo il robot prima di passare all'azione KeepYellowLandmark.
  // ------------------------------------------------------------------
  case STOP_BEFORE_KEEP:
    printf("Robot fermo.\n");
    if(++counter == STOP_DURATION)
    {
      counter = 0;
      type_of_request = YELLOW;
      KeepYellowLandmark_POINTER->activate();
      deactivate();
      return NULL;
    }
    myDesired.setVel(0);
    myDesired.setDeltaHeading(0);
  }
  printf("**********************\n\n");
  type_of_request = YELLOW;
  return &myDesired;
}
```

6.5. KeepYellowLandmark.cpp

```
// File: KeepYellowLandmark.cpp

#define KEEP_FACTOR 5
#define YELLOW_LOST 20
#define KEEP_VELOCITY 80

// ************************************
// Action KeepYellowLandmark: Prototype.
// ************************************
class ActionKeepYellowLandmark : public ArAction
{
public:
  ActionKeepYellowLandmark(void); // Constructor.
  virtual ~ActionKeepYellowLandmark(void) {}; // Destructor.
  virtual ArActionDesired *fire(ArActionDesired currentDesired); // Fire.
  virtual void setRobot(ArRobot *robot);

protected:
```

```
  ArActionDesired myDesired;
  int yellow_X, yellow_Y, lost_counter;
  enum {GO, LOST_TEMP} KeepYellowLandmark_Flag;
};

// ************************************
// Action KeepYellowLandmark: Constructor.
// ************************************
ActionKeepYellowLandmark::ActionKeepYellowLandmark(void) :
ArAction("KeepYellowLandmark")
{
  KeepYellowLandmark_Flag = GO;
  lost_counter = YELLOW_LOST;
}

// ***********************************
// Action KeepYellowLandmark: setRobot.
// ***********************************
void ActionKeepYellowLandmark::setRobot(ArRobot *robot)
{
  myRobot = robot;
  // Inizialmente l'azione KeepYellowLandmark risulta disattivata.
  deactivate();
}

// *******************************
// Action KeepYellowLandmark: FIRE.
// *******************************
ArActionDesired *ActionKeepYellowLandmark::fire(ArActionDesired currentDesired)
{
  // Se l'azione StallRecover è attiva viene disattivata.
  if(StallRecover_POINTER->isActive())
    StallRecover_POINTER->deactivate();

  myDesired.reset();

  sscanf(values_BUFFER, "%d %d\r\n", &yellow_X, &yellow_Y);
  printf("********************\nKeepYellowLandmark:\n");
  printf("Coordinate ricevute %d %d\n", yellow_X, yellow_Y);

  switch(KeepYellowLandmark_Flag)
  {
  // ----------------------------------------
  // Il robot sta seguendo il landmark giallo.
  // ----------------------------------------
  case GO:
    // Il robot è arrivato a destinazione.
    if(myRobot->isLeftMotorStalled() && myRobot->isRightMotorStalled())
    {
      printf("Il robot è arrivato correttamente all'interno ");
      printf("della docking station.\n");
      printf("**** FINE PROGRAMMA ****\n");
      Robotica_END = true;
      myRobot->stop();
      myRobot->disconnect();
      deactivate();
      return NULL;
    }
    // Il robot sta seguendo il landmark giallo.
    if(yellow_X != LANDMARK_LOST)
    {
      printf("Speedy sta seguendo il landmark giallo.\n");
      myDesired.setVel(KEEP_VELOCITY);
      myDesired.setDeltaHeading(-yellow_X/KEEP_FACTOR);
    } else
    // Viene perso il riferimento per la prima volta.
    {
      printf("Landmark giallo temporaneamente perso.\n");
      KeepYellowLandmark_Flag = LOST_TEMP;
      myDesired.setVel((--lost_counter/YELLOW_LOST)*KEEP_VELOCITY);
    }
    break;
  // ----------------------------------------------------
```

```
  // Il robot ha temporaneamente perso il landmark giallo.
  // -----------------------------------------------------
  case LOST_TEMP:
    // Il riferimento risulta ancora perso.
    if(yellow_X == LANDMARK_LOST)
    {
      // Il landmark giallo è considerato definitivamente perso.
      if(--lost_counter == 0)
      {
        printf("Landmark giallo perduto definitivamente.\n");
        lost_counter = YELLOW_LOST;
        KeepYellowLandmark_Flag = GO;
        type_of_request = GREEN;
        rotation_for_recover = true;
        WanderForGreenLandmark_POINTER->activate();
        deactivate();
        return NULL;
      }
    printf("Landmark giallo temporaneamente perso.\n");
    myDesired.setVel((lost_counter/YELLOW_LOST)*KEEP_VELOCITY);
    } else
    // Il riferimento giallo è stato recuperato.
    {
      printf("Landmark giallo recuperato.\n");
      lost_counter = YELLOW_LOST;
      KeepYellowLandmark_Flag = GO;
      myDesired.setVel(KEEP_VELOCITY);
      myDesired.setDeltaHeading(-yellow_X/KEEP_FACTOR);
    }
  }
  printf("******************\n\n");
  type_of_request = YELLOW;
  return &myDesired;
}
```

Bibliografia

[1] ActivMedia Incorporated: “Pioneer Mobile Robots Operation Manual”, 2nd Edition, Gennaio 1998.

[2] Cassinis, R., Tampalini, F., Bartolini, P., Fedrigotti, R.: “Docking and Charging System for Autonomous Mobile Robots”, DEA-Unibs, 2005.

[3] Heesch, D.: “ARIA overview 2.4.1”, *html document generated by Doxygen*, Settembre 2005.

[4] ActivMedia Robotics, LLC: “CMPE 300 ARIA LAB MANUAL”, 2002.

[5] Borenstein, J. Everett, H. R., Feng, L.: “Where am I? Sensors and Methods for Mobile Robot Positioning”, The University of Michigan, Aprile 1996.

Indice

SOMMARIO **1**

1. INTRODUZIONE **1**
1.1. Il robot "Speedy" **1**
1.2. Il pacchetto ARIA **2**

2. IL PROBLEMA AFFRONTATO **3**

3. LA SOLUZIONE ADOTTATA **4**
3.1. Navigazione e visione **4**
3.2. L'algoritmo di navigazione **5**
3.3. Lo stato "Ricerca landmark verde" **7**
3.3.1. L'azione "WanderForGreenLandmark" 8
3.4. Lo stato "Raggiungere landmark verde" **9**
3.4.1. L'azione GoToGreenLandmark 10
3.5. Lo stato "Allineamento" **11**
3.5.1. L'azione SearchYellowLandmark 11
3.6. Lo stato "Seguire landmark giallo" **12**
3.6.1. L'azione KeepYellowLandmark 13

4. MODALITÀ OPERATIVE **14**
4.1. Componenti necessari **14**
4.2. Modalità di installazione **15**
4.3. Modalità di taratura **15**
4.4. Avvertenze **15**

5. CONCLUSIONI E SVILUPPI FUTURI **15**

6. APPENDICE A **16**
6.1. Robotica.cpp **16**
6.2. WanderForGreenLandmark.cpp **18**
6.3. GoToGreenLandmark.cpp **20**
6.4. SearchYellowLandmark.cpp **22**
6.5. KeepYellowLandmark.cpp **23**

BIBLIOGRAFIA **25**

INDICE **26**