Applicazione della logica fuzzy a problemi di robotica su piattaforma LEGO® Mindstorms NXT e sistema operativo LejOS

Marcello Battain

14 agosto 2010

Sommario

Con la seguente relazione si è inteso, tramite l'utilizzo del robot LEGO® Mindstorms NXT, su cui è stato opportunamente caricato il firmware LejOS, evidenziare la differenza tra due approcci di programmazione applicati a problemi di robotica.
Nel caso specifico si è approfondita la possibile applicazione della logica Fuzzy, o logica sfumata, confrontandola invece con l'approccio deterministico della programmazione a Behavior.
Nei primi capitoli sarà brevemente introdotta la logica Fuzzy, poiché la trattazione in maniera approfondita va oltre lo scopo di questa tesi, e spiegata l'applicazione e i metodi usati per la programmazione Fuzzy nel linguaggio Java.
In quelli successivi saranno illustrati nel dettaglio i due esempi scelti per il confronto tra i due metodi di programmazione, il primo sarà un "line follower", cioè un mezzo bimotore il cui compito è quello di seguire una linea, mentre il secondo un braccio meccanico che dovrà eseguire dei semplici movimenti per spostare una pallina.

Indice

1. Introduzione alla logica fuzzy..... 7
1.1 Introduzione storica..... 7
1.2 Teoria delle logica fuzzy..... 9
1.2.1 Insiemi fuzzy..... 9
1.2.2 Funzioni di appartenenza..... 11
1.2.3 Controllore fuzzy..... 16
1.2.4 Rule Base..... 17
1.2.5 Inferenza fuzzy..... 18
1.2.6 Defuzzificazione..... 20
1.3 Esempio di applicazione: il pendolo inverso..... 23

2. LejOS e la programmazione a behavior..... 30
2.1 Introduzione a LejOS..... 30
2.2 La programmazione a behavior..... 30
2.2.1 Introduzione..... 30
2.2.2 L'interfaccia di programmazione a behavior..... 31
2.2.3 Esempio di programmazione a behavior..... 33

3. Applicazione della logica fuzzy a Java..... 39
3.1 Introduzione..... 39
3.2 Costruttore..... 39
3.3 Fuzzificazione..... 40
3.4 Inferenza..... 43
3.5 Defuzzificazione..... 47

4. Il line follower..... 52
4.1 Introduzione..... 52
4.2 Il line follower implementato a behavior..... 55
4.3 Il line follower fuzzy..... 64
4.3.1 Progettazione..... 64
4.3.2 Realizzazione..... 67
4.4 Confronto..... 73

5. Il braccio meccanico..... 75
5.1 Introduzione..... 75
5.2 Il braccio meccanico implementato a behavior..... 77
5.3 Il braccio meccanico fuzzy..... 90
5.3.1 Progettazione..... 90
5.3.2 Realizzazione..... 103
5.4 Confronto..... 118

Bibliografia..... 119

Elenco delle figure

1.1 Funzione Sigma Right-open..... 11
1.2 Funzione Sigma Left-open..... 12
1.3 Funzione Triangolare..... 13
1.4 Funzione Trapezoidale..... 13
1.5 Funzione S-shape..... 14
1.6 Funzione Bell-shape..... 15
1.7 Funzione Gaussiana 15
1.8 Struttura di un controllore fuzzy..... 16
1.9 Pendolo inverso..... 23
1.10 Funzioni di appartenenza variabile Errore..... 24
1.11 Funzioni di appartenenza variabile Velocità angolare..... 24
1.12 Funzioni di appartenenza variabile Coppia applicata..... 25
1.13 Gradi di apartenenza Errore..... 26
1.14 Gradi di appartenenza Velocità angolare..... 26
1.15 Fuzzy set di output Coppia applicata..... 28
2.1 Schema esempio di programmazione strutturata..... 31
2.2 Il behavior a priorità alta sopprime quello a priorità Bassa..... 33
3.1 Punti necessari alla rappresentazione delle principali funzioni di appartenenza..... 40
3.2 Implicazione di Mamdani..... 43
3.3 Aree comprese tra min1 e max1..... 48
3.4 Aree comprese tra max1 e max2..... 49
3.5 Aree comprese tra max2 e min2..... 49
4.1 Robot Line Follower..... 52
4.2 Brick NXT..... 53
4.3 Sensore di luminosità..... 53
4.4 Motore NXT..... 54
4.5 Funzioni di appartenenza variabile Luce..... 66
4.6 Funzioni di appartenenza variabili MotorB e MotorC..... 66
4.7 Percorso nero continuo..... 73
5.1 Robot Arm..... 75
5.2 Sensore di campo magnetico terrestre..... 76
5.3 Sensore di distanza ad ultrasuoni..... 76
5.4 Funzioni di appartenenza variabile DegreeL (parte sinistra)..... 93

5.5 Funzioni di appartenenza variabile DegreeR (parte destra)..... 93
5.6 Funzioni di appartenenza variabile Time..... 94
5.7 Funzioni di appartenenza variabile MotorC..... 94
5.8 Funzioni di appartenenza variabile Height..... 96
5.9 Funzioni di appartenenza variabile Time..... 97

5.10 Funzioni di appartenenza variabile Height (elevamento)..... 99
5.11 Funzione di appartenenza insieme fuzzy slowUp..... 99
5.12 Funzioni di appartenenza variabile Degree (palla rossa)..... 101
5.13 Funzioni di appartenenza variabile Degree (palla blu).... 102

Capitolo 1

Introduzione alla logica fuzzy

1.1 Introduzione storica

Il concetto di logica multi-valore si può far risalire all'anno 1920, molto prima quindi della logica fuzzy, ad opera di Jan Lukasiewicz.
La nascita della logica fuzzy invece avviene in maniera ufficiale con la pubblicazione, nel 1965, da parte di Lofti A. Zadeh dell'articolo *Fuzzy Sets* sulla rivista *Information and Control.*
In questo articolo il professor Zadeh nell'introdurre il concetto di *insieme fuzzy* scrisse: *"Più spesso del contrario, le classi di oggetti incontrati nel mondo fisico reale non hanno criteri di appartenenza ben definiti. Per esempio la classe degli animali chiaramente include cani, cavalli, uccelli etc. e chiaramente esclude oggetti come rocce, fluidi, piante etc. Comunque, oggetti come stelle marine, batteri etc. hanno uno stato ambiguo rispetto alla classe degli animali. Lo stesso tipo di ambiguità sorge nel caso di un numero, come 10, in relazione alla "classe" di tutti i numeri reali molto più grandi di 1.*
Chiaramente, "la classe di tutti i numeri reali molto più grandi di 1", o "la classe delle donne belle", o "la classe degli uomini alti", non costituiscono classi o insiemi nel senso matematico classico di questi termini. Però, rimane il fatto che tali impropriamente definite "classi" giocano un ruolo importante nel pensiero umano in particolare nel riconoscimento di modelli, nella comunicazione di informazioni, nell'astrazione.
Lo scopo di questa nota è di esplorare in maniera preliminare alcuni aspetti basilari e implicazioni di un concetto che può essere usato riguardo alle "classi" del tipo citato sopra. Il concetto in questione è quello di un "insieme fuzzy", che è una classe con un continuo di gradi di appartenenza. Come si vedrà più avanti, la nozione di insieme fuzzy costituisce un conveniente punto di partenza per la costruzione di una struttura concettuale che in molti aspetti è parallela alla struttura utilizzata nel caso degli insiemi ordinari, ma è molto più generale di quest'ultima e, potenzialmente, può provare di avere un campo di applicazione più ampio, nella classificazione di modelli e nell'elaborazione di informazioni. Essenzialmente, come uno schema fornisce una maniera naturale di trattare con problemi in cui la fonte di imprecisione è la

mancanza di criteri definiti nettamente della classe di appartenenza piuttosto che la presenza di variabili casuali…"
Quindi il concetto di insieme fuzzy pone le basi per un approccio nell'affrontare alcune classi di problemi sfruttando la nozione di informazione non precisa, cosa che rispecchia maggiormente il ragionamento umano nella risoluzione dei problemi.
Mentre alcuni matematici accolsero con entusiasmo le nuove idee, la maggioranza delle reazioni furono tra lo scetticismo, se non addirittura l'aperta ostilità nei confronti di questa teoria, in particolare nella comunità accademica.
In Giappone le ricerche sulla logica sfumata cominciarono verso la fine degli anni 70', da parte di due piccoli gruppi universitari, guidati rispettivamente da T. Terano e H. Shibata il primo, da K. Tanaka e K. Asai il secondo. Anche questi, nei primi tempi, si scontrarono con un ambiente fortemente ostile alla logica fuzzy, ma i loro sforzi avrebbero prodotto ottimi risultati già dopo un decennio, fornendo molti importanti contributi sia alla teoria della logica fuzzy che alla sua applicazione.
Le polemiche rimasero sempre molto aspre soprattutto nei primi anni di vita della logica fuzzy, quando i suoi sostenitori non erano ancora in grado di mostrarne alcuna applicazione.
La prima venne riconosciuta con la progettazione, nel 1974 in Gran Bretagna, del primo sistema di controllo di un generatore a vapore basato sulla logica fuzzy da parte di Ebrahim H. Mamdani e Seto Assilan; Mamdani lavorò anche sul controllo di una fornace per la produzione di cemento, che venne resa operativa nel 1982.
Nel corso degli anni '80 furono lanciate con successo diverse applicazioni industriali della logica fuzzy e, dopo anni di ricerche e sviluppo, nel 1987 venne messo appunto dalla Hitachi un sistema automatizzato per il controllo operativo dei treni metropolitani della città di Sendai. Da qui in poi, questa ed altre applicazioni spinsero molti ingegneri giapponesi ad approfondire una vasta gamma di applicazioni inedite, che portò ad un vero e proprio boom della logica fuzzy.
Nel 1987 furono decisi due progetti di ricerca nazionali su larga scala da agenzie governative giapponesi, il più noto dei quali sarebbe stato il *Laboratory for International Fuzzy Engineering Research* (LIFE). Alla fine del gennaio 1990, la Matsushita Electric Industrial Co., alias Panasonic, lanciò il primo elettrodomestico "fuzzy", la lavatrice a controllo automatico *Asai-go Day Fuzzy.*
In quegli anni il termine "fuzzy" fu introdotto nella lingua giapponese con un nuovo significato, intelligente. Molte altre aziende seguirono negli anni successivi le orme della Panasonic e

ciò ebbe come risultato l'esplodere di una vera e propria mania per tutto ciò che era etichettato come fuzzy. I successi giapponesi suscitarono l'interesse per questa tecnologia in Corea, Europa e, in maniera minore, negli Stati Uniti.
La logica fuzzy ha trovato numerose applicazioni anche in ambito finanziario. Il primo sistema per le trattazioni finanziarie ad usare la logica sfumata è stato lo Yamaichi Fuzzy Fund: usato in 65 aziende, tratta la maggioranza dei titoli quotati dell'indice Nikkei Dow e consiste in circa 800 regole. Il sistema è stato collaudato per due anni e le prestazioni in termini di rendimento hanno superato l'indice Nikkei Average di oltre il 20%. Durante il periodo di prova il sistema consigliò di vendere ben 18 giorni prima del 19 ottobre 1987 (Lunedì Nero). E' divenuto operativo nel 1988.
Il primo chip VLSI (Very Large Scale Integration) dedicato all'elaborazione di inferenze fuzzy fu sviluppato da M. Togai e H. Watanabe nel 1986. Diverse imprese sono state costituite per commercializzare strumenti hardware e software per lo sviluppo di sistemi a logica sfumata. Allo stesso tempo anche i produttori di software cominciarono a introdurre pacchetti di progettazioni di sistemi fuzzy come il *Fuzzy Logic Toolbox* per MATLAB (1994).

1.2 Teoria della logica fuzzy

1.2.1 Insiemi Fuzzy

Secondo la logica classica la definizione di insieme è:
Qualunque collezione di oggetti per il quale sia sempre possibile decidere se un oggetto appartiene o no alla collezione stessa.
Il punto centrale della teoria degli insiemi è la nozione di appartenenza ad un insieme. Per definire, quindi, un insieme A si ricorre alla sua funzione caratteristica o funzione di appartenenza, che permette di determinare, per ogni oggetto x, se tale oggetto appartiene all'insieme:

$$\mu_A(x) = \begin{cases} 1 \; se \; x \in A \\ 0 \; altrimenti \end{cases}$$

La funzione di appartenenza di un insieme è, quindi, una funzione booleana che assume il valore 1 se l'oggetto appartiene all'insieme e 0 in caso contrario; un insieme di questo tipo è detto "*Crisp*".

L'uso di insiemi crisp è però limitativo se si intende usarli per rappresentare concetti sfumati, come per esempio il fatto che un oggetto sia vicino o meno. Si potrebbe per esempio definire la funzione di appartenenza dell'insieme crisp V degli oggetti vicini ad un dato punto P, in questo modo:

$$\mu_V(x) = \begin{cases} 1 \; se \; d(x,P) < 100 \; cm \\ 0 \; altrimenti \end{cases}$$

dove con d(x,P) si intende la distanza in linea retta tra l'oggetto x e il punto P.

Come si può facilmente intuire, tale definizione può risultare fortemente limitativa se si pensa che un oggetto posto a distanza di 99 centimetri risulterebbe vicino, mentre uno a distanza di un metro no!
E' quindi necessario riformulare il concetto di insieme in maniera meno rigida, introducendo il concetto di *grado di appartenenza*, in questo modo è possibile definire il concetto di oggetti più o meno vicini all'interno dell'insieme.
Dato, quindi, un grado di appartenenza di un insieme A, $\mu_A(x)$:

- Per $\mu_A(x) = 1$, x appartiene sicuramente all'insieme A
- Per $\mu_A(x) = 0$, x non appartiene all'insieme A
- Per $0 < \mu_A(x) < 1$, x appartiene non in forma certa all'insieme A, con un grado di appartenenza indicato da $\mu_A(x)$

Possiamo così definire un *insieme fuzzy* come un insieme di coppie ordinate formate ciascuna da un elemento e dal suo grado di appartenenza all'insieme.

$$A = \{(x, \mu_A(x)) : x \in X\}$$

Quando il dominio di X è continuo, l'insieme A può essere composto da un numero infinito di elementi, quando invece è discreto, è possibile esplicitare tutti gli elementi dell'insieme A.

1.2.2 Funzioni di appartenenza

La funzione di appartenenza di un insieme fuzzy *A* in *X*, è una funzione $f_A(x)$ che associa a ciascun punto in *X* un numero reale nell'intervallo [0,1], dove il valore di $f_A(x)$ rappresenta il grado di appartenenza di *x* in *A.* Se l'insieme è crisp allora la funzione di appartenenza può assumere solamente i valori 0 e 1.

Tra le funzioni di appartenenza più utilizzate nei sistemi fuzzy, citiamo:

- **Singleton**, $\mu_A(x)$ è uguale a 1 su un solo elemento e 0 altrove.

- **Sigma Right-open**

$$\mu(x) = \max\{\min\{\frac{(x-a)}{(b-a)}, 1\}, 0\}$$

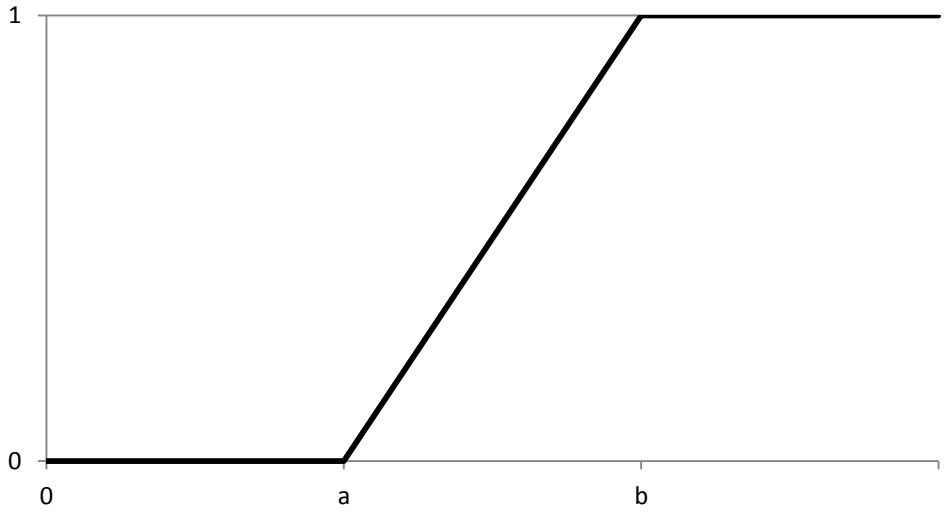

Figura 1.1: Funzione Sigma Right-open

- **Sigma Left-open**

$$\mu(x) = \max\{\min\{\frac{(b-x)}{(b-a)}, 1\}, 0\}$$

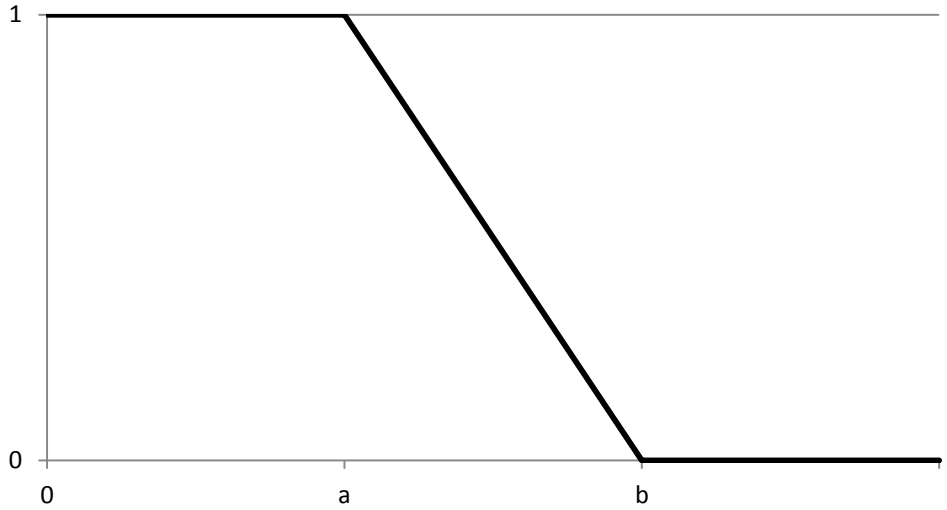

Figura 1.2: Funzione Sigma Left-open

- **Triangolare**

$$\mu(x) = \begin{cases} \frac{(x-a)}{(b-a)} & per\ a < x < b \\ \frac{(c-x)}{(c-b)} & per\ b \leq x < c \\ 0 & altrove \end{cases}$$

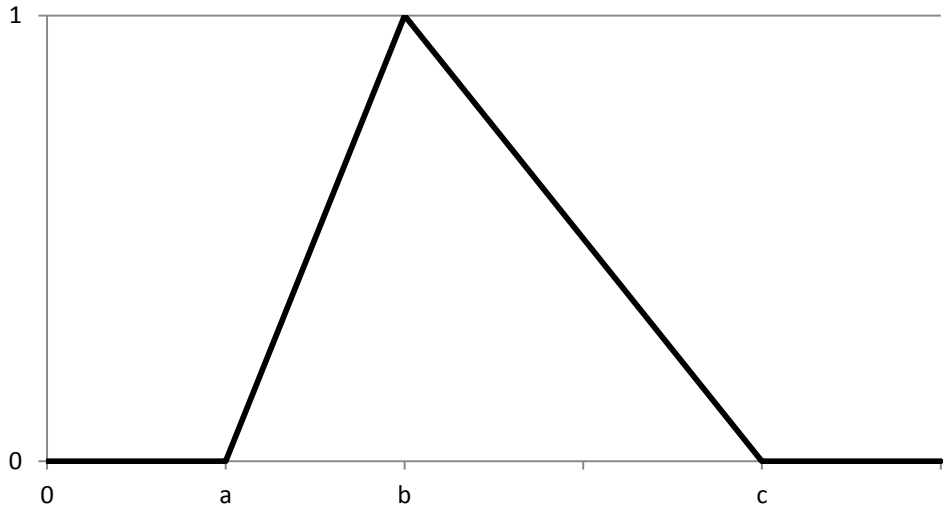

Figura 1.3: Funzione Triangolare

- **Trapezoidale**

$$\mu(x) = \begin{cases} \dfrac{(x-a)}{(b-a)} \ \ per\ a < x < b \\ \dfrac{(d-x)}{(d-c)} \ \ per\ c < x < d \\ 1\ per\ b \ \leq x \ \leq c \\ 0 \ \ altrove \end{cases}$$

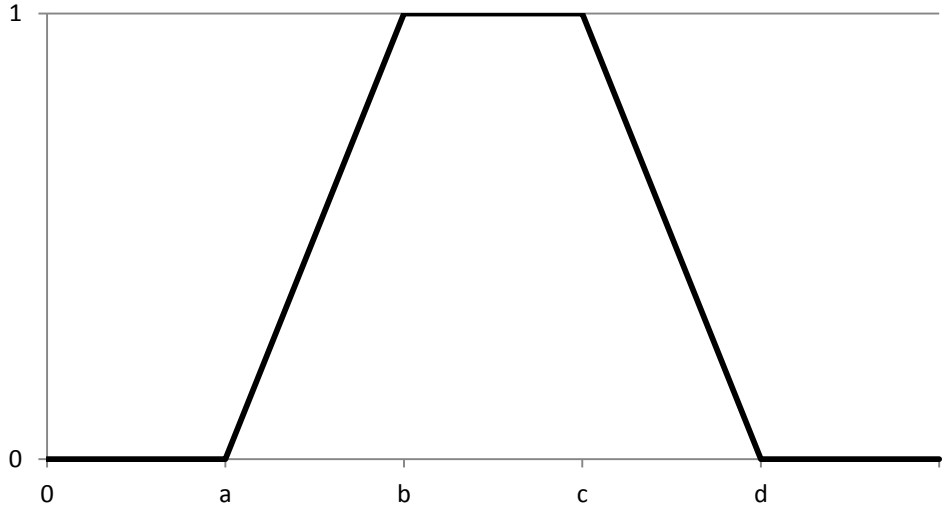

Figura 1.4: Funzione Trapezoidale

- **S-Shape**

$$\mu(x) = \begin{cases} 0 \ per \ x < a \\ 2\left(\frac{x-a}{c-a}\right)^2 \ per \ a \leq x \leq b \\ 1 - 2\left(\frac{c-x}{c-a}\right)^2 \ per \ b < x \leq c \\ 1 \ per \ x > c \end{cases}$$

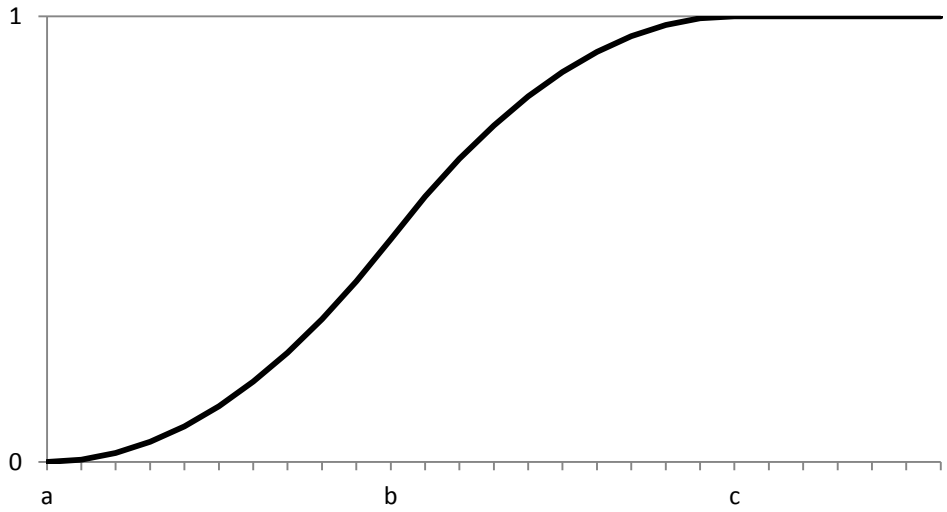

Figura 1.5: Funzione S-shape

- **Bell-Shape**

$$\mu(x) = \begin{cases} 2\left(\frac{x-a}{c-a}\right)^2 \ per \ a \leq x \leq b \\ 1 - 2\left(\frac{c-x}{c-a}\right)^2 \ per \ b < x \leq c \\ 1 - 2\left(\frac{x-c}{e-c}\right)^2 \ per \ c \leq x \leq d \\ 2\left(\frac{e-x}{e-c}\right)^2 \ per \ d < x \leq e \\ 0 \ altrove \end{cases}$$

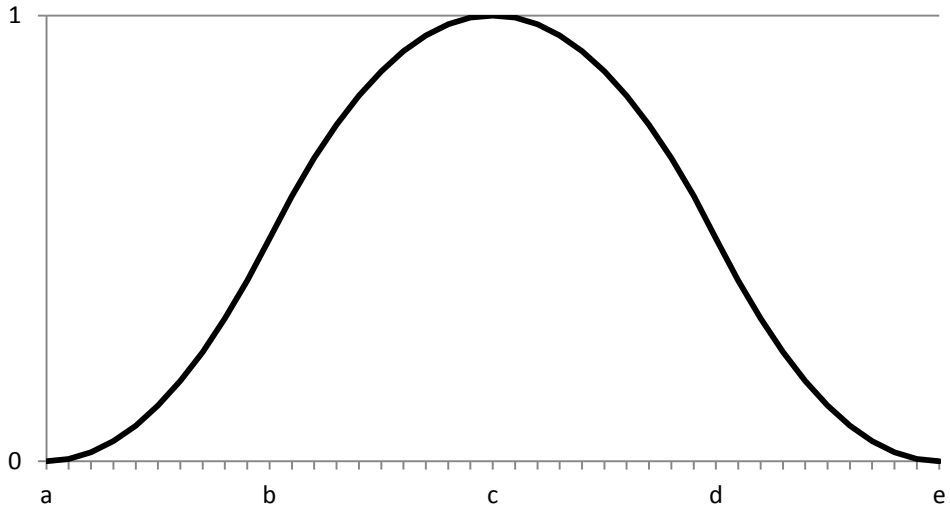

Figura 1.6: Funzione Bell-shape

- **Gaussiana**

$$\mu(x) = e^{-\frac{(x-a)^2}{2b^2}}$$

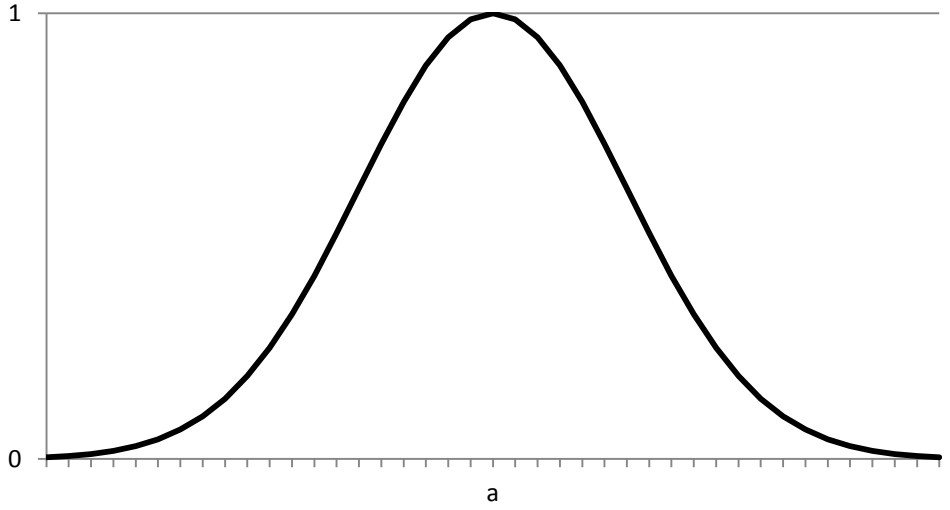

Figura 1.7: Funzione Gaussiana

1.2.3 Controllore fuzzy

Il controllore fuzzy costituisce la parte centrale di un sistema fuzzy ed è un dispositivo basato sulla logica fuzzy che può essere usato come componente di controllo in un sistema a catena chiusa.

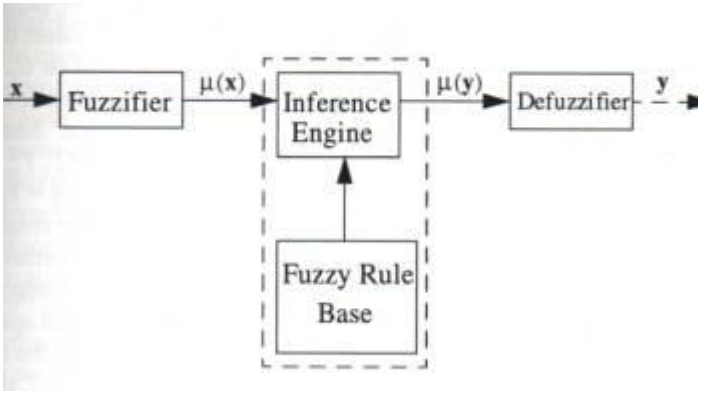

Figura 1.8: Struttura di un controllore fuzzy

La struttura del controllore può essere suddivisa in vari blocchi, come è visibile in figura 1.8:

- Il *fuzzifier* ha il compito di leggere le variabili di ingresso (*x* in figura 1.8) e di convertire i valori numerici in opportuni valori linguistici, ai quali corrispondono insiemi fuzzy; in pratica, dato il valore *x* della variabile, ricava il valore *μ(x)* del grado di appartenenza usando l'equazione della relativa funzione di appartenenza.

- La *Rule Base è* basata su un insieme di regole che presentano una serie di variabili e i corrispondenti termini linguistici, in cui una parte è chiamata *"antecedente"* e l'altra *"conseguente".*

Es.

IF (Tempertura is "fredda") THEN (Calorifero is "alto")

- Il *Motore di inferenza*, in base agli ingressi nell'istante di tempo considerato, elabora, attraverso la procedura di inferenza, una funzione di appartenenza per il segnale di uscita.

- L'interfaccia di *Defuzzifier* riceve in ingresso le funzioni di appartenenza degli output e le converte in valori numerici utilizzabili per la soluzione del problema fuzzy.

1.2.4 Rule Base

In questa trattazione non approfondiremo le varie tipologie di regole fuzzy, ma solamente quella più comunemente utilizzata, ovvero la forma *If-Then.*
Queste regole sono utilizzate per formulare dichiarazioni condizionali che contengono la logica fuzzy. La forma generale per una regola *If-Then* è la seguente:

If X is A then Y is B

in cui A e B sono i valori linguistici a cui sono associati gli insiemi fuzzy e le relative funzioni di appartenenza, definiti rispettivamente sugli intervalli X e Y.
La parte *X is A* è denominata *antecedente* o *premessa*, mentre la parte *Y is B* è detta *conseguente* o *conclusione*.
Generalmente l'input per una regola *If-Then* è il valore corrente per la variabile di input mentre l'output è un intero insieme fuzzy, che verrà poi 'defuzzificato' in un valore numerico nella fase di 'defuzzificazione'.

L'interpretazione delle regole *If-Then* è un processo costituito da tre parti:

1. '*Fuzzificazione' degli input*: avviene nel fuzzificatore e associa a tutte le affermazioni fuzzy nell'antecedente un grado di appartenenza compreso tra 0 e 1.

2. *Applicazione degli operatori della logica fuzzy ad antecedenti con parti multiple*: se l'antecedente è composto da parti multiple si applicano gli operatori fuzzy e si associa all'antecedente un singolo numero compreso tra 0 e 1, detto *grado di supporto* per la regola.
   Al riguardo, in questa tesi verranno utilizzati i due soli connettori *and* e *or* con le seguenti regole:

   - *If X is A and Y is B then…*
     $\mu_{Antecedente} = \min\{\mu_A(x), \mu_B(x)\}$

- *If X is A or Y is B then…*
  $\mu_{Antecedente} = \max\{\mu_A(x), \mu_B(x)\}$

3. *Applicazione del metodo di implicazione*: si usa il grado di supporto dell'intera regola per modellare l'insieme fuzzy di output. Quest'ultimo sarà troncato secondo il metodo di implicazione scelto, argomento del prossimo paragrafo.

1.2.5 Inferenza fuzzy

Nella logica del primo ordine, il conseguente si ottiene dalle premesse attraverso il *modus ponens*, che si può generalizzare così:

- Premessa: x è A
- Implicazione: *se x è A allora y è B*
- Conseguenza: *y è B*

Come si può intuire, il modus ponens può essere applicato solamente se la premessa è interamente soddisfatta, ovvero se $\mu_A(x) = 1$.
Al contrario, nella logica fuzzy, il valore del grado di verità della premessa può assumere un qualsiasi valore compreso tra 0 e 1.
Per questo è opportuno ricorrere al *modus ponens generalizzato* in cui la premessa A' e la conseguenza B' non corrispondono ad A e B, ma in qualche modo sono in relazione con esse:

- Premessa: *x è A'*
- Implicazione: *se x è A allora y è B*
- Conseguenza: *y è B'*

In pratica A' e B' sono due insiemi fuzzy, e per elaborare B' si guarda all'implicazione come ad una relazione fuzzy R, detta *funzione di implicazione fuzzy*; dopo di che si effettua la composizione dei due insiemi A' e R secondo la regola del sup-min, così definita:

$$\mu_{B'}(y) = \sup_{x \in X} \min\{\mu_{A'}(x), \mu_R(x,y)\}$$

La costruzione dell'implicazione fuzzy offre un certo grado di libertà e sta al progettista del controllore scegliere il metodo di implicazione più adeguato a seconda della situazione.
Alcune delle funzioni di implicazione più usate sono:

- Regola del minimo (Mamdani):

$$\mu_R(x,y) = \min\{\mu_A(x), \mu_B(y)\}$$

- Regola del prodotto (Larsen):

$$\mu_R(x,y) = \mu_A(x) \cdot \mu_B(y)$$

- Regola del max-min (Zadeh):

$$\mu_R(x,y) = \max\{\min\{\mu_A(x), \mu_B(y)\}, 1 - \mu_A(y)\}$$

- Sequenza standard (Rescher-Gainess):

$$\mu_R(x,y) = \begin{cases} 1 \; se \; \mu_A(x) \leq \mu_B(y) \\ 0 \; se \; \mu_A(x) > \mu_B(y) \end{cases}$$

- Kleene-Dienes:

$$\mu_R(x,y) = \max\{1 - \mu_A(x), \mu_B(y)\}$$

- Gödel:

$$\mu_R(x,y) = \begin{cases} 1 \; se \; \mu_A(x) \leq \mu_B(y) \\ \mu_B(y) \; se \; \mu_A(x) > \mu_B(y) \end{cases}$$

- Lucasiewicz:

$$\mu_R(x,y) = \max\{1 - \mu_A(x) + \mu_B(y), 1\}$$

- Gougen:

$$\mu_R(x,y) = \begin{cases} 1 \; se \; \mu_A(x) \leq \mu_B(y) \\ \dfrac{\mu_B(y)}{\mu_A(x)} \; se \; \mu_A(x) > \mu_B(y) \end{cases}$$

1.2.6 Defuzzificazione

Come era stato già accennato in precedenza, il processo di defuzzificazione consiste nella trasformazione degli insiemi fuzzy degli output in valori numerici.
Come per la scelta del metodo di inferenza, anche qui esistono varie tecniche di defuzzificazione, alcune basate sulle sugli insiemi fuzzy risultanti dalle singole regole prese separatamente (Implied Fuzzy Sets) e altre basate sull'insieme fuzzy globale che rappresenta la soluzione raggiunta considerando tutte le regole (Overall Implied Fuzzy Sets).
Verranno ora presentate le tecniche di defuzzificazione più comunemente usate.

Implied Fuzzy Sets

- *Center of gravity (COG)*: il valore di uscita $\boldsymbol{y_q^{crisp}}$ è scelto usando il centro dell'area e l'area stessa sottesa dalla funzione di appartenenza di ogni insieme fuzzy di uscita ed è dato dalla formula:

$$y_q^{crisp} = \frac{\sum_{i=1}^{R} b_i^q \int_{y_q} \mu_{\hat{B}_q^i}(y_q) dy_q}{\sum_{i=1}^{R} \int_{y_q} \mu_{\hat{B}_q^i}(y_q) dy_q}$$

  dove R è il numero delle regole, $\boldsymbol{b_i^q}$ è il centro dell'area della funzione di appartenenza associata all'insieme fuzzy dell'output

per l'i-esima regola e $\int_{y_q} \boldsymbol{\mu}_{\hat{\boldsymbol{B}}_q^i}(\boldsymbol{y_q})\boldsymbol{dy_q}$ denota l'area sottointesa da $\boldsymbol{\mu}_{\hat{\boldsymbol{B}}_q^i}(\boldsymbol{y_q})$.
Il COG può essere calcolato facilmente in quanto spesso è semplice trovare delle formule per calcolare le aree sottese dalle funzioni di appartenenza.

- *Center-average*: il valore $\boldsymbol{y_q^{crisp}}$ dell'uscita è scelto usando il centro di ogni funzione di appartenenza e il massimo della funzione stessa, secondo la formula:

$$y_q^{crisp} = \frac{\sum_{i=1}^{R} b_i^q \, sup_{y_q}\{\mu_{\hat{B}_q^i}(y_q)\}}{\sum_{i=1}^{R} sup_{y_q}\{\mu_{\hat{B}_q^i}(y_q)\}}$$

dove $\boldsymbol{b_i^q}$ è il centro dell'area della funzione di appartenenza dell'output associato alla i-esima regola e $\boldsymbol{sup}$ indica l'estremo superiore.
Da notare che $\boldsymbol{sup_x}\{\boldsymbol{\mu}(\boldsymbol{x})\}$ può essere spesso semplicemente pensato come il valore più alto di $\boldsymbol{\mu}(\boldsymbol{x})$.

Overall Implied Fuzzy Sets

- *Max criterion:* il valore di uscita $\boldsymbol{y_q^{crisp}}$ è scelto come il punto dell'universo del discorso Y_q per il quale il l'insieme fuzzy $\hat{\boldsymbol{B}}_{\boldsymbol{q}}$ raggiunge un massimo, che è:

$$y_q^{crisp} \in \{\arg\sup_{Y_q}\{\mu_{\hat{B}_q}(y_q)\}\}$$

A volte l'estremo superiore può essere raggiunto in più di un punto in Y_q, in questo caso bisogna specificare anche una strategia per scegliere solo un punto per $\boldsymbol{y_q^{crisp}}$.
Spesso questa strategia di defuzzificazione è evitata appunto per questa ambiguità nella scelta finale.

- *Mean of maximum*: il valore di uscita $\boldsymbol{y_q^{crisp}}$ è scelto per rappresentare la media di tutti gli elementi la cui funzione ha un

massimo in $\widehat{\boldsymbol{B}}_{\boldsymbol{q}}$. Definiamo $\widehat{\boldsymbol{b}}_{\boldsymbol{q}}^{\boldsymbol{max}}$ come l'estremo superiore della funzione di appartenenza dell'insieme fuzzy $\widehat{\boldsymbol{B}}_{\boldsymbol{q}}$.
Definiamo inoltre un insieme fuzzy $\widehat{\boldsymbol{B}}_{\boldsymbol{q}}^{*} \in \boldsymbol{Y}_{\boldsymbol{q}}$ con una funzione di appartenenza definita come:

$$\mu_{\widehat{\boldsymbol{B}}_{\boldsymbol{q}}^{*}}(y_q) = \begin{cases} 1 \; se \; \mu_{\widehat{\boldsymbol{B}}_{\boldsymbol{q}}} = \widehat{\boldsymbol{b}}_{\boldsymbol{q}}^{\boldsymbol{max}} \\ 0 \; altrove \end{cases}$$

Allora l'output è definito, usando il mean of maximum, come:

$$y_q^{crisp} = \frac{\int_{Y_q} y_q \mu_{\widehat{\boldsymbol{B}}_{\boldsymbol{q}}^{*}}(y_q) dy_q}{\int_{Y_q} \mu_{\widehat{\boldsymbol{B}}_{\boldsymbol{q}}^{*}}(y_q) dy_q}$$

- *Center of area (COA)*: il valore di uscita $\boldsymbol{y}_{\boldsymbol{q}}^{\boldsymbol{crisp}}$ come il centro dell'area della funzione di appartenenza del fuzzy set globale $\widehat{\boldsymbol{B}}_{\boldsymbol{q}}$.
  Per un universo del discorso Y_q continuo, il centro dell'area si ottiene dalla formula:

$$\boldsymbol{y}_{\boldsymbol{q}}^{\boldsymbol{crisp}} = \frac{\int_{Y_q} y_q \mu_{\widehat{\boldsymbol{B}}_{\boldsymbol{q}}}(y_q) dy_q}{\int_{Y_q} \mu_{\widehat{\boldsymbol{B}}_{\boldsymbol{q}}}(y_q) dy_q}$$

  Sia questo che il mean of maxima possono essere estremamente dispendiosi in termini di calcoli.

Alla fine, vediamo che usare un Overall Implified Fuzzy Set è spesso poco raccomandabile per due ragioni:

(1) Il fuzzy set complessivo $\widehat{\boldsymbol{B}}_{\boldsymbol{q}}$ è generalmente difficile da calcolare

(2) Le tecniche di defuzzificazione basate su un meccanismo di inferenza che calcola $\widehat{\boldsymbol{B}}_{\boldsymbol{q}}$ sono anche esse difficili da calcolare

E' per questi motivi che la maggio parte di sistemi di controllo fuzzy esistenti usano tecniche di defuzzificazione basate sugli insiemi fuzzy separati, in particolare appunto, il metodo *COG* e il *Center-average*.

1.3 Esempio di applicazione: il pendolo inverso

Per chiarire i concetti esposti nei paragrafi precedenti di questo capitolo verrà presentato ora un esempio abbastanza comune di applicazione della logica fuzzy, cioè il pendolo inverso.
Il problema consiste nel mantenere in equilibrio un pendolo ruotato di 180°.

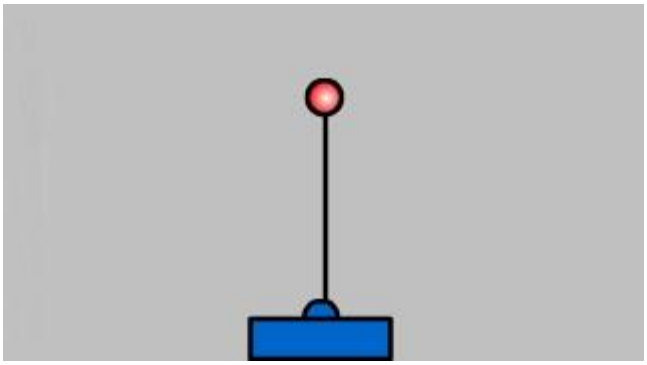

Figura 1.9: Pendolo inverso

Le variabili di input per il funzionamento del motore fuzzy sono:

- L'*errore* del pendolo (o orientazione) che viene misurato in *gradi*, va da -90° a 90° ed è pari a 0 quando il pendolo è in equilibrio
- La *velocità angolare* che viene misurata in *rad/s*, consideriamo positiva quando il pendolo ruota in senso e va da -26 *rad/s* a 26 *rad/s*

La variabile di output è invece:

- La *coppia applicata* al motore del pendolo che imprime la rotazione necessaria a mantenere in equilibrio lo stesso, viene misurata in *Nm,* la consideriamo positiva quando deve far ruotare il pendolo in senso orario e per la quale non specifichiamo qui un intervallo preciso in quanto esso dipende anche dalla lunghezza del pendolo, va quindi da –MAX a MAX

Gli insiemi fuzzy in cui viene suddiviso l'intervallo dell'*errore* e le relative sigle con cui saranno rappresentati sono i seguenti:

1. Errore negativo medio (NM)
2. Errore negativo piccolo (NP)
3. Errore zero (ZE)
4. Errore positivo piccolo (PP)
5. Errore positivo medio (PM)

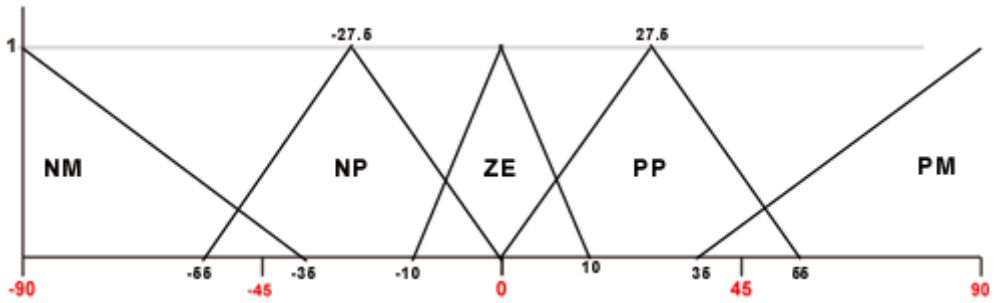

Figura 1.10: Funzioni di appartenenza variabile Errore

Gli insiemi fuzzy in cui viene suddiviso l'intervallo della velocità angolare con le relative abbreviazioni sono:

1. Velocità angolare negativa media (NM)
2. Velocità angolare negativa piccola (NP)
3. Velocità angolare zero (ZE)
4. Velocità angolare positiva piccola (PP)
5. Velocità angolare positiva media (PM)

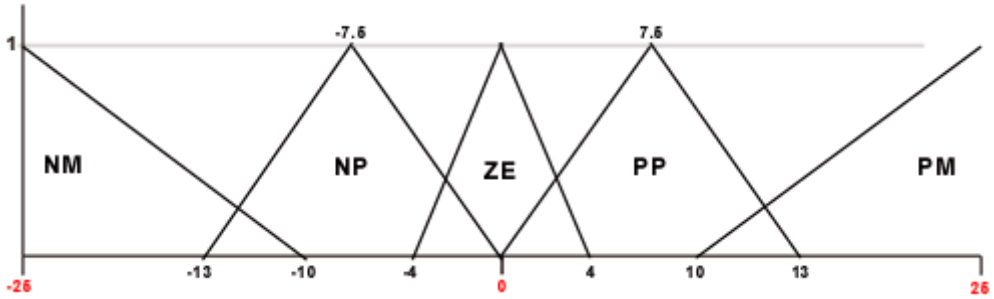

Figura 1.11: Funzioni di appartenenza variabile Velocità angolare

Gli insiemi fuzzy in cui viene suddiviso l'intervallo della coppia applicata al motore con le relative abbreviazioni sono:

1. Coppia applicata negativa media (NM)
2. Coppia applicata negativa piccola (NP)
3. Coppia applicata zero (ZE)
4. Coppia applicata positiva piccola (PP)
5. Coppia applicata positiva media (PM)

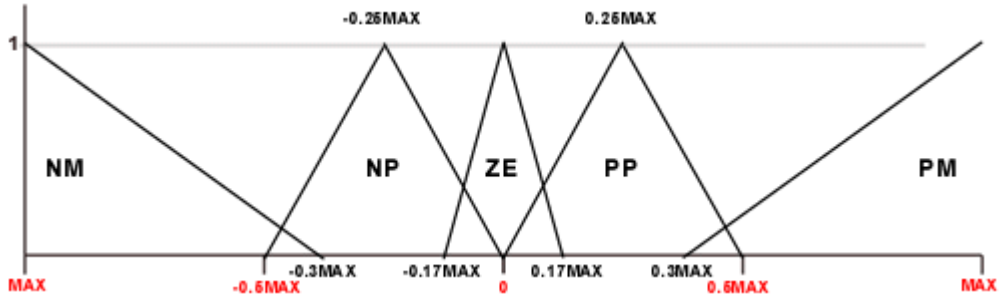

Figura 1.12: Funzioni di appartenenza variabile Coppia applicata

Viene presentata ora la matrice che riassume le regole fuzzy su cui si basa il funzionamento di questo controllore.

	ERRORE					
VELOCITA' ANGOLARE	and	NM	NP	ZE	PP	PM
	NM	PM	PP	PM	ZE	NM
	NP	PM	PP	PP	ZE	NM
	ZE	PM	PP	ZE	NP	NM
	PP	PM	ZE	NP	NP	NM
	PM	PM	ZE	NM	NP	NM

La tabella è usata per semplificare la lettura delle regole, per esempio la casella che incrocia la colonna *Errore–NP* e la riga *Velocità-PP* rappresenta la regola:

- $if\ \boldsymbol{Errore}\ is\ \boldsymbol{Negativo\ Piccolo}\ and\ \boldsymbol{Velocità\ angolare}\ is\ \boldsymbol{Positiva\ Piccola}\ then\ \boldsymbol{Coppia}\ is\ \boldsymbol{Zero}$

Procediamo ora con l'applicazione del processo eseguito dal controllore fuzzy con un esempio di valori di input rilevati in un istante, ponendo che essi siano:

- $Errore = -41°$
- $Velocità = 0{,}7\ rad/s$

La prima parte consiste nella '*fuzzificazione*' dei valori rilevati per ottenere i gradi di appartenenza agli insiemi fuzzy che vengono interessati.
Come è facile intuire guardando la figura, per quanto riguarda *l'errore,* vengono interessati i fuzzy set NM e NP. I due gradi di appartenenza possono essere calcolati applicando le formule esposte nel paragrafo 1.2.2:

- $\mu_{NM} = \frac{(-35-x)}{[-35-(-90)]} = \frac{(-35+41)}{(-35+90)} = 0{,}11$
- $\mu_{NP} = \frac{[x-(-55)]}{[-27{,}5-(-55)]} = \frac{(-41+55)}{(-27{,}5+55)} = 0{,}51$

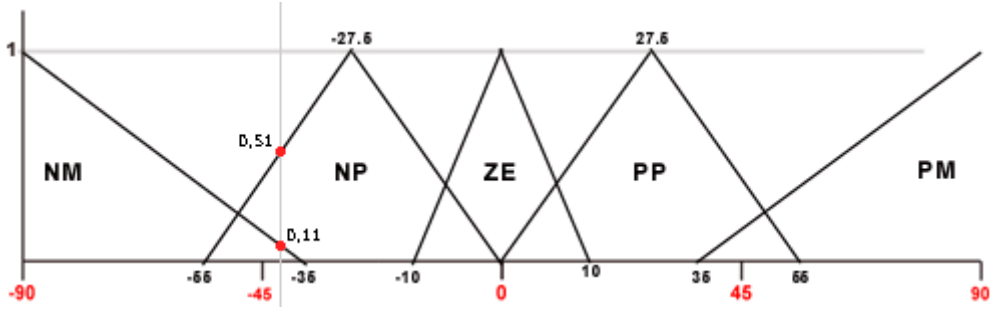

Figura 1.13: Gradi di appartenenza Errore

Per quanto riguarda invece la velocità angolare, il valore di input interessa i due fuzzy set ZE e PP, con gradi di appartenenza:

- $\mu_{ZE} = \frac{(4-x)}{(4-0)} = \frac{(4-0{,}7)}{(4)} = 0{,}83$
- $\mu_{PP} = \frac{(X-0)}{(7{,}5-0)} = \frac{(0{,}7)}{(7{,}5)} = 0{,}09$

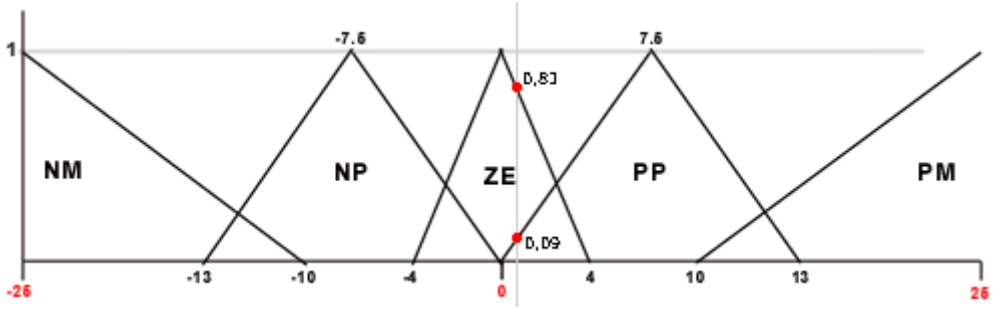

Figura 1.14: Gradi di appartenenza Velocità angolare

Passiamo ora alla fase di *valutazione delle regole*, calcolando il grado di verità per ogni regola: risulta che le regole che vengono attivate dai valori di input (cioè quelle per cui il grado di verità è diverso da 0) sono quattro.

	ERRORE					
VELOCITA' ANGOLARE	and	NM	NP	ZE	PP	PM
	NM	PM	PP	PM	ZE	NM
	NP	PM	PP	PP	ZE	NM
	ZE	PM	PP	ZE	NP	NM
	PP	PM	ZE	NP	NP	NM
	PM	PM	ZE	NM	NP	NM

Applicando l'operatore *min* per ottenere i vari gradi di verità delle quattro regole risulta:

1. $grado_{PM} = \min(0{,}11; 0{,}83) = 0{,}11$
2. $grado_{PM} = \min(0{,}11; 0{,}09) = 0{,}09$
3. $grado_{PP} = \min(0{,}51; 0{,}83) = 0{,}51$
4. $grado_{ZE} = \min(0{,}51; 0{,}09) = 0{,}09$

Considerando ora che le regole 1 e 2 forniscono un valore di verità per lo stesso insieme fuzzy PM, si opera applicando l'operatore *max* tra i due valori, visto che gli antecedenti delle due regole potrebbero essere resi uno unico connettendoli con un *or* senza variare il risultato del processo. I gradi di appartenenza delle funzioni di output risultano quindi essere:

- $\mu_{PM} = 0{,}11$
- $\mu_{PP} = 0{,}51$
- $\mu_{ZE} = 0{,}09$

Per la fase di *inferenza* utilizziamo l'implicazione di Mamdani, il cui effetto è quello di troncare le funzioni di appartenenza degli insiemi fuzzy degli output interessati all'altezza indicata dai rispettivi gradi di appartenenza, risultati dalla valutazione delle regole.
Il risultato si può vedere in figura 1.15:

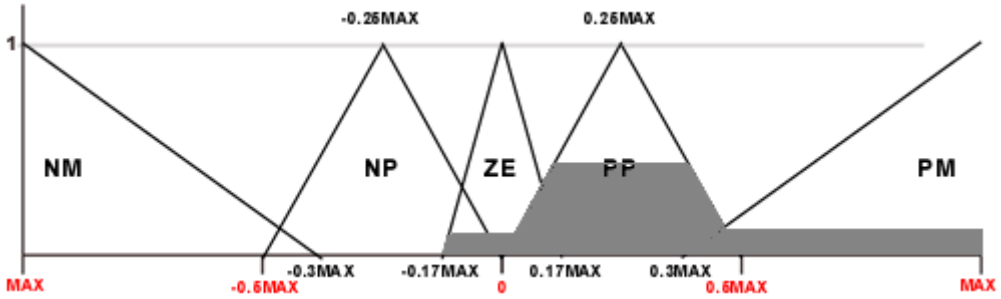

Figura 1.15: Fuzzy set di output Coppia applicata

La funzione di appartenenza dell'insieme fuzzy ZE è ora trapezoidale, quindi dobbiamo calcolare le ascisse dei due estremi della base minore usando e formule inverse a quelle da cui si ricava il grado di appartenenza, per poterne successivamente calcolare l'area:

- $a_{bm_{ZE}} = -0{,}17MAX + \mu_{ZE} * [0 - (-0{,}17MAX)] = -0{,}17MAX + \mu_{ZE} * 0{,}17MAX = -0{,}155MAX$
- $b_{bm_{ZE}} = 0{,}17MAX - \mu_{ZE} * (0{,}17MAX - 0) = 0{,}17MAX - \mu_{ZE} * 0{,}17MAX = 0{,}155MAX$

Lo stesso discorso vale per la nuova funzione di appartenenza dell'insieme fuzzy PP, gli estremi della sua base minore sono:

- $a_{bm_{PP}} = 0 + \mu_{PP} * (0{,}25MAX - 0) = \mu_{PP} * 0{,}25MAX = 0{,}128MAX$
- $b_{bm_{PP}} = 0{,}5MAX - \mu_{PP} * (0{,}5MAX - 0{,}25MAX) = 0{,}5MAX - \mu_{PP} * 0{,}25MAX = 0{,}372MAX$

Infine per quanto riguarda la funzione di appartenenza di output dell'insieme fuzzy PM, dobbiamo calcolare l'ascissa di un solo punto, in quanto quella dell'altro estremo della base minore rimane invariato:

- $a_{bm_{PM}} = 0{,}3MAX + \mu_{PM} * (MAX - 0{,}3MAX) = 0{,}3MAX + \mu_{PM} * 0{,}7MAX = 0{,}377MAX$

L'ultima fase è quella di '*defuzzificazione*', necessaria per ottenere il valore di output della coppia da applicare al motore del pendolo inverso.
Utilizziamo il metodo *Center of gravity (COG)*, spiegato al paragrafo 1.2.6, per il quale ci occorre sapere innanzitutto le aree delle funzioni di appartenenza degli insiemi fuzzy di uscita e i centri delle aree stesse:

1. Funzione di appartenenza dell'insieme di uscita ZE
   a. $centro_{ZE} = -0{,}17MAX + \frac{(0{,}17MAX + 0{,}17MAX)}{2} = 0$
   b. $area_{ZE} = \frac{[(0{,}17MAX + 0{,}17MAX) + (0{,}155MAX + 0{,}155MAX)] * 0{,}09}{2} = 0{,}029MAX$
2. Funzione di appartenenza dell'insieme di uscita PP
   a. $centro_{PP} = 0 + \frac{(0{,}5MAX - 0)}{2} = 0{,}25MAX$
   b. $area_{PP} = \frac{[(0{,}5MAX + 0) + (0{,}372MAX - 0{,}128MAX)] * 0{,}51}{2} = 0{,}189MAX$
3. Funzione di appartenenza dell'insieme di uscita PM
   a. $centro_{PM} = 0{,}3MAX + \frac{(MAX - 0{,}3MAX)}{2} = 0{,}65MAX$
   b. $area_{PM} = \frac{[(MAX - 0{,}3MAX) + (MAX - 0{,}377MAX)] * 0{,}11}{2} = 0{,}073MAX$

Inseriamo ora i risultati ottenuti nell'equazione del *Center of Gravity Method*:

$$y_q^{crisp} = \frac{\sum_{i=1}^{R} b_i^q \int_{y_q} \mu_{\hat{B}_q^i}(y_q) dy_q}{\sum_{i=1}^{R} \int_{y_q} \mu_{\hat{B}_q^i}(y_q) dy_q} =$$

$$= \frac{(0 * 0{,}029MAX) + (0{,}25MAX * 0{,}189MAX) + (0{,}65MAX * 0{,}073MAX)}{0{,}029MAX + 0{,}189MAX + 0{,}073MAX}$$

$$= \frac{0 + 0{,}047MAX^2 + 0{,}047MAX^2}{0{,}291MAX} = \frac{0{,}094MAX^2}{0{,}291MAX} = 0{,}323MAX$$

Quindi la coppia applicata al motore deve essere 0,323MAX Nm.

Capitolo 2

LejOS e la programmazione a Behavior

2.1 Introduzione a LejOS

LejOS è un ambiente di programmazione Java per LEGO MINDSTORMS NXT® che permette di programmare i robot LEGO® in Java.

Esso consiste in:
- Un firmware di rimpiazzo per il brick NXT che include una Java Virtual Machine
- Un libreria di classi Java (Classes.jar) the implementa l'interfaccia di programmazione (API) di lejOS NXJ
- Un linker per connettere le classi utente Java con classes.jar tramite un file binario che può essere caricato ed eseguito nel NXT
- Una PC API per scrivere programmi per PC che comunichino con i programmi lejOS NXJ usando flussi Java attraverso Bluetooth o USB, o usando il protocollo di comunicazione LEGO (LCP)

La versione di LejOS utilizzata per lo svolgimento degli esempi trattati in questa tesi, è l'ultima versione disponibile al momento della redazione, ovvero la 0.85.

2.2 La programmazione a Behavior

2.2.1 Introduzione

La maggioranza delle persone che cominciano a programmare un robot pensano al flusso del programma come ad una serie di If-then, che è una influenza della programmazione strutturata (figura 2.1). Questo tipo di programmazione non richiede alcun tipo di studio preventivo; il problema è che il codice finisce per essere aggrovigliato e difficile da espandere.
Al contrario, il modello di programmazione a behavior richiede un po' di pianificazione prima di cominciare a scrivere il codice, ma il vantaggio è che il codice sarà molto più incapsulato dentro una struttura facile da comprendere.

Questo teoricamente rende il codice più comprensibile a chi è familiare con il modello di controllo a behavior, ma soprattutto diventa molto più semplice aggiungere o rimuovere behavior dalla struttura complessiva senza ripercussioni negative sul resto del codice.

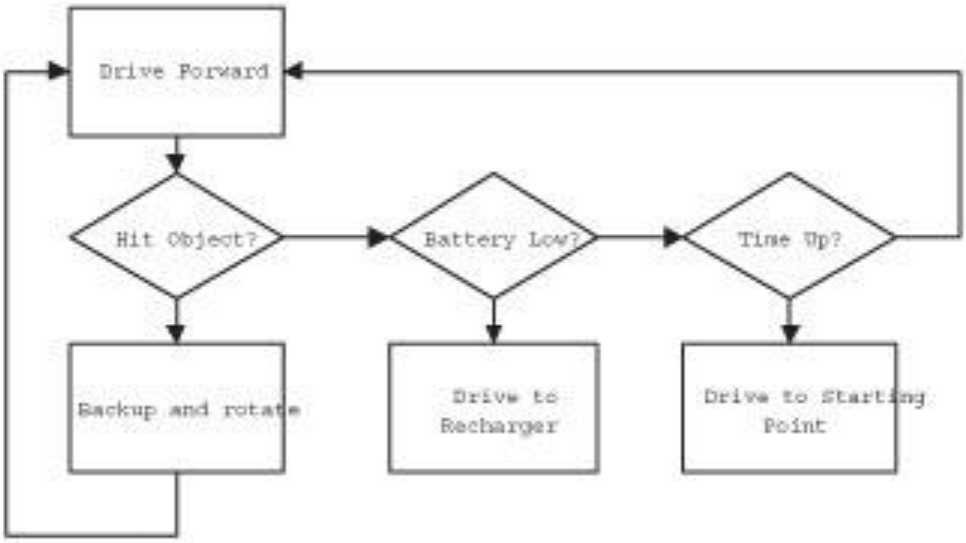

Figura 2.1: Schema esempio di programmazione strutturata

2.2.2 L'interfaccia di programmazione a behavior

L'interfaccia di programmazione a behavior è molto semplice, ed è composta solamente da un'interfaccia e una classe. L'interfaccia *Behavior* è usata per definire i behavior stessi; essa è molto generale, sicché funziona abbastanza bene perché le singole implementazioni dei behavior variano ampiamente.
Una volta che sono stati definiti tutti i behavior, essi vengono affidati ad un *Arbitrator* per regolare quali di essi dovrebbero essere attivati. Tutte le classi di libreria e le interfacce per il controllo a behavior sono collocate nel package *lejos.subsumption.*

Lejos.subsumption.Behavior:

- *boolean takeControl()*
  Restituisce un valore booleano che indica se il behavior dovrebbe diventare attivo

- *void action()*

il codice contenuto in questo metodo viene eseguito quando il behavior diventa attivo.

- *void suppress()*
  Il codice nel metodo *suppress* dovrebbe terminare immediatamente il codice attivo nel metodo *action*. Il metodo *suppress* può essere anche utilizzato per aggiornare qualsiasi dato prima del completamento del behavior.

Lejos.subsumption.Arbitrator:

- *public Arbitrator(Behavior[] behaviors)*
  questo è il costruttore della classe che crea un oggetto Arbitrator che regola quando ciascuno dei behavior deve diventare attivo; maggiore è il numero nell'indice dell'array argomento, maggiore sarà la priorità del corrispondente behavior.

- *public void start()*
  Avvia il sistema Arbitrator

Quando viene creata un'istanza di Arbitrator, gli viene fornito un array di oggetti Behavior; una volta fatto questo, il metodo *start* viene chiamato.
L'Arbitrator chiama il metodo *takeControl* di ciascun oggetto Behavior, cominciando dall'oggetto con indice maggiore nell'array, continua in questa maniera finché non incontra un behavior che vuole prendere il controllo. Quando ne incontra uno, esso esegue il metodo *action* di quel behavior una ed una volta sola; se due behavior vogliono prendere il controllo, verrà dato il permesso solamente a quello con il livello di priorità maggiore (figura 2.2).

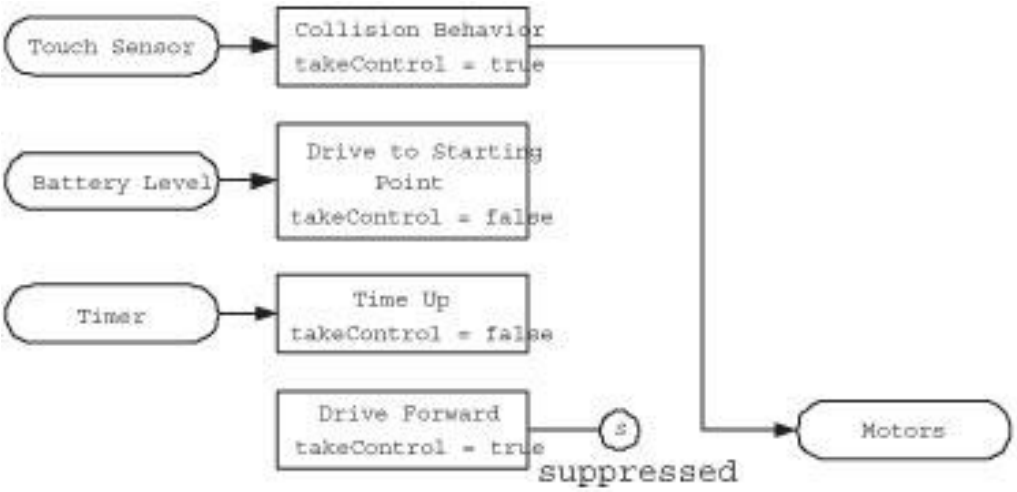

Figura 2.2: Il Behavior a priorità alta sopprime quello a priorità bassa

2.2.3 Esempio di programmazione a Behavior

In questo esempio verrà mostrata la programmazione di un semplice robot con sterzo differenziale. Questo robot avanza in modo rettilineo come behavior low-level principale; quando incontra un oggetto, un behavior a più alto livello di priorità diventa attivo e fa retrocedere il robot e quindi ruotare di 90 gradi. C'è infine un terzo behavior che si attiva quando gli altri due sono stati completati.

Come visto nell'interfaccia *Behavior*, dobbiamo implementare i metodi *action*, *suppress* e *takeControl.*
Il metodo *action* del behavior per andare dritto è molto semplice, deve far ruotare i motori A e C in avanti:

```
public void action() {
  Motor.A.forward();
  Motor.C.forward();
}
```

Ora il metodo *suppress* che deve fermare l'azione precedente quando questo viene invocato:

```
public void suppress() {
  Motor.A.stop();
  Motor.C.stop();
}
```

Ora dobbiamo implementare un metodo che dica all'Arbitrator quando questo behavior dovrebbe diventare attivo. Come è stato detto in precedenza, il robot dovrebbe andare sempre dritto, a meno che qualcosa altro non sopprima questo behavior, sicché questo ultimo deve sempre voler prendere il controllo, quindi il metodo *takeControl* restituirà sempre il valore *true*:

```
public boolean takeControl() {
  return true;
}
```

Il codice completo di questa prima classe risulta così:

```
import lejos.subsumption.*;
import lejos.nxt.*;

public class DriveForward implements Behavior {
   public boolean takeControl() {
      return true;
   }

   public void suppress() {
      Motor.A.stop();
      Motor.C.stop();
   }

   public void action() {
      Motor.A.forward();
      Motor.C.forward();
   }
}
```

Il secondo behavior è leggermente più complicato, ma ancora molto simile al primo: la sua azione principale è di indietreggiare e ruotare di 90 gradi quando il robot incontra un oggetto.
In questo esempio vogliamo che questo behavior prenda il controllo solo quando il sensore di contatto viene premuto:

```
public boolean takeControl() {
  return touch.isPressed();
}
```

Questo presuppone che sia stato precedentemente creata un'istanza di un oggetto di tipo *TouchSensor* chiamata *touch*.
Il metodo *action* risulta così:

```
public void action() {
    // Back up:
    Motor.A.backward();
    Motor.C.backward();
    try{Thread.sleep(1000);}catch(Exception e) {}

    // Rotate by causing one wheel to stop:
    Motor.A.stop();
    try{Thread.sleep(300);}catch(Exception e) {}
    Motor.C.stop();
}
```

Il metodo *suppress()* interrompe semplicemente la rotazione dei motori:

```
public void suppress {
  Motor.A.stop();
  Motor.C.stop();
}
```

Segue il codice completo per il behavior *HitWall:*

```
import lejos.subsumption.*;
import lejos.nxt.*;

public class HitWall implements Behavior {
   public TouchSensor touch = new TouchSensor(SensorPort.S!);

   public boolean takeControl() {
      return touch.isPressed();
   }
…
```

```
…
public void action() {
      // Back up:
      Motor.A.backward();
      Motor.C.backward();
      try{Thread.sleep(1000);}catch(Exception e) {}

      // Ruota fermando uno dei due motori:
      Motor.A.stop();
      try{Thread.sleep(300);}catch(Exception e) {}
      Motor.C.stop();
   }
}
```

Abbiamo definito i nostri due behavior, ed è un problema da poco creare una classe con un metodo *main* per far iniziare il tutto; tutto ciò che abbiamo bisogno di fare è di creare un array di oggetti *Behavior*, creare un'istanza dell'*Arbitrator* e farlo partire:

```
import lejos.subsumption.*;

public class BumperCar {
   public static void main(String [] args) {
      Behavior b1 = new DriveForward();
      Behavior b2 = new HitWall();
      Behavior [] bArray = {b1, b2}; // b2 di priorità
                                     // maggiore di b1
      Arbitrator arby = new Arbitrator(bArray);
      arby.start();
   }
}
```

Le prime due righe del metodo *main* creano le istanze dei behavior, la terza li inserisce nell'array mettendo quello con priorità più bassa in una posizione con indice minore rispetto all'altro. La quarta riga crea l'Arbitrator e la quinta lo fa partire.

Vediamo ora come inserire un terzo behavior senza alterare pezzi di codice nelle due precedenti classi.
Creiamo un behavior che controlla il livello di batteria, genera un segnale acustico quando è minore di un certo livello e interrompe l'esecuzione del programma.

Ecco direttamente il codice completo:

```
import lejos.subsumption.*;
import lejos.nxt.*;

public class BatteryLow implements Behavior {
   private float LOW_LEVEL;

   private static final short [] note = {
      2349,115, 0,5, 1760,165, 0,35, 1760,28, 0,13, 1976,23,
      0,18, 1760,18, 0,23, 1568,15, 0,25, 1480,103, 0,18,
      1175,180, 0,20, 1760,18, 0,23, 1976,20, 0,20, 1760,15,
      0,25, 1568,15, 0,25, 2217,98, 0,23, 1760,88, 0,33, 1760,
      75, 0,5, 1760,20, 0,20, 1760,20, 0,20, 1976,18, 0,23,
      1760,18, 0,23, 2217,225, 0,15, 2217,218};

   public BatteryLow(float volts) {
      LOW_LEVEL = volts;
   }

   public boolean takeControl() {
      float voltLevel = Battery.getVoltage();
      System.out.println("Voltage " + voltLevel);

      return voltLevel < LOW_LEVEL;
   }

   public void suppress() {
      // Nothing to suppress
   }

   public void action() {
      play();
      try{Thread.sleep(3000);}catch(Exception e) {}
      System.exit(0);
   }
  …
```

```
…
public static void play() {
      for(int i=0;i <note.length; i+=2) {
         final short w = note[i+1];
         Sound.playTone(note[i], w);
         try {
            Thread.sleep(w*10);
         } catch (InterruptedException e) {}
      }
   }
}
```

La melodia completa è conservata nell'array *note* alla riga numero 6 e il metodo per suonare le note è alla riga 30. Questo behavior prende il controllo solamente se il livello della batteria è sotto il voltaggio definito nel costruttore.
Il metodo *takeControl* visualizza anche la carica della batteria sul display LCD; il metodo action suona la melodia e poi termina e, dato che questo behavior ferma l'esecuzione del programma, non è necessario creare un metodo *suppress*.
Ecco come inserire il nuovo behavior:

```
import lejos.subsumption.*;

public class BumperCar {
   public static void main(String [] args) {
      Behavior b1 = new DriveForward();
      Behavior b2 = new BatteryLow(6.5f);
      Behavior b3 = new HitWall();
      Behavior [] bArray = {b1, b2, b3};
      Arbitrator arby = new Arbitrator(bArray);
      arby.start();
   }
}
```

Consiglio: quando si crea un sistema di controllo a behavior è meglio programmare e collaudare un behavior alla volta. Creando e caricando tutti i behavior in una volta, gli eventuali bug nel codice possono rendere più difficile la loro individuazione.

Capitolo 3

Applicazione della logica fuzzy a Java

3.1 Introduzione

In questo capitolo sarà illustrato come è stata realizzato un controllore fuzzy in Java. Per semplicità tutto il codice necessario alla realizzazione delle funzioni di appartenenza, della fase di fuzzificazione, inferenza e defuzzificazione sono stati inclusi in un'unica classe che è stata chiamata *MembershipFunction*.
Segue il dettaglio di come è stata implementata tale classe e i suoi metodi.

3.2 Costruttore

Il costruttore della classe *MembersipFunction* crea un'istanza della funzione di appartenenza di un insieme fuzzy.
Data la non necessità di creare funzioni di appartenenza complicate, è stato scelto in fase di progettazione di dare la possibilità di utilizzare tale classe per creare uno dei seguenti tipi di funzione:
a) *Sigma Right-open*
b) *Sigma Left-open*
c) *Triangolare*
d) *Trapezoidale*
Il costruttore in pratica riceve in ingresso cinque parametri, dei quali i primi quattro indicano l'ascissa dei punti necessari alla costruzione della funzione di appartenenza rappresentata, mentre il quinto indica l'ordinata massima raggiunta dalla funzione stessa. L'utilità di questo parametro sarà chiarita più avanti, visto che inizialmente le funzioni di appartenenza assumeranno tutte un valore massimo pari a 1.

```
public MembershipFunction(int a, int b, int c, int d, double h) {
      min1 = a;
      max1 = b;
      max2 = c;
      min2 = d;
      alt = h;
}
```

Come si può capire osservando la figura 3.1, con al più quattro punti si riesce a definire qualsiasi dei 4 tipi di funzioni di appartenenza sopra elencati.

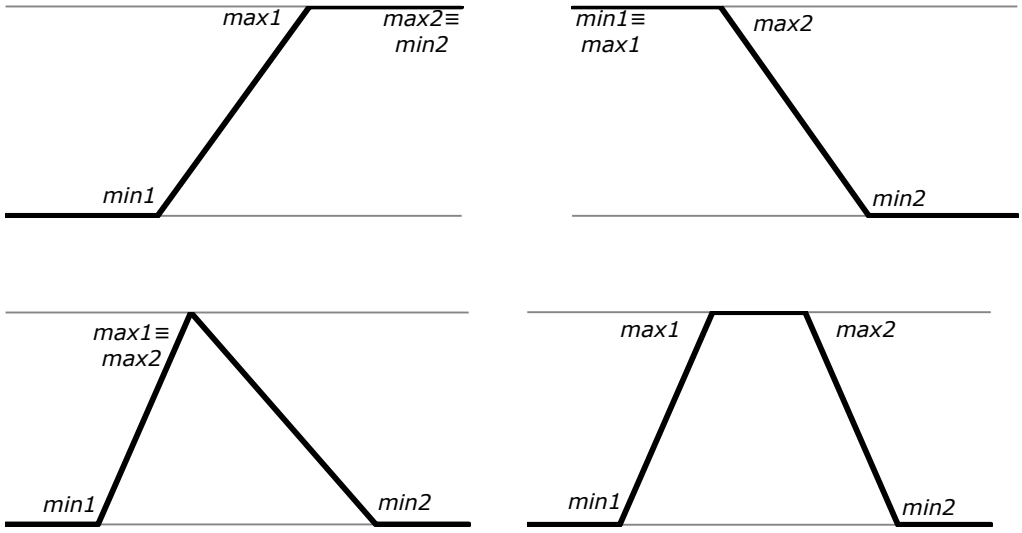

Figura 3.1: Punti necessari alla rappresentazione delle principali funzioni di appartenenza

Da qui in poi questa denominazione dei punti fondamentali delle funzioni di appartenenza sarà usata per spiegare il codice, dove necessario.
Quando viene creata un'istanza di un oggetto di tipo *MembershipFunction,* il programma non sa ancora di quale tipo sia la funzione di appartenenza ed a dire il vero non è importante finché non dovranno essere eseguite le operazioni per il calcolo degli output del controllore fuzzy.

3.3 Fuzzificazione

Per la realizzazione della prima parte del controllore fuzzy, ovvero la fuzzificazione degli input, è stato implementato un metodo *fuzzification* che riceve come parametro un numero intero che rappresenta il valore dell'input e restituisce un valore *double* compreso tra 0 e 1 che indica il suo grado di appartenenza all'insieme fuzzy rappresentato dall'oggetto *MembershipFunction*.

```
public double fuzzificate(int x) {
…
}
```

Come è facile notare, quando il valore dell'input è minore del punto *min1* oppure è maggiore del punto *min2* (*x≤min1 or x≥min2*), non è necessario effettuare alcun calcolo sul grado di appartenenza visto che esso assumerà necessariamente il valore 0 oppure 1. Nello specifico esso assumerà il valore 1 solamente in due casi:

i. Se *max2* ≡ *min2* e *input* ≥ *max2* (oppure *min2*)

ii. Se *max1* ≡ *min1* e *input* ≤ *max1* (oppure *min1*)

```
if (x<=min1 || x>=min2) {
      if ((max2==min2 && x>=min2) || (max1==min1 && x<=min1))
           return 1;
      else
           return 0;
}
```

Nel caso molto semplice in cui il valore dell'input sia tra i punti *max1* e *max2* compresi (*x≥max1 and x≤max2)*, il grado di appartenenza assumerà sicuramente il valore massimo 1.

```
if (x >= max1 && x <= max2)
     return 1;
```

Rimangono solamente 2 casi da analizzare:

1) *min1≤x≤max1*
   In questo caso, essendo già stata esclusa dai controlli precedenti la possibilità di trovarsi in un punto di massimo della funzione di tipo Left-open, ci troviamo necessariamente nella parte crescente di uno degli altri tre tipi di funzioni considerate; quindi il grado di appartenenza è calcolato secondo la formula

$$y = \frac{(x - min1)}{(max1 - min1)}$$

2) *max2≤x≤min2*
   Simmetricamente al caso precedente, essendo già stata esclusa la possibilità di trovarsi in un punto di massimo di una funzione di tipo Right-open, dobbiamo trovarci nella parte decrescente di uno degli altri tre tipo di funzione:

$$y = \frac{(min2 - x)}{(min2 - max2)}$$

Il codice quindi risulta essere così:

```
double newX = (double) x;
if (x >= min1 && x <= max1)
     return ((newX - min1) / (max1 - min1));
else
     return ((min2 - newX) / (min2 - max2));
```

Da notare che in questa ultima parte di codice, per effettuare i calcoli dobbiamo effettuare un *cast* del valore intero di ingresso (in generale i metodi di lejOS che leggono i valori dei sensori forniscono interi) in un *double,* dato che la divisione tra soli dati di tipo *int*, fornirebbe in Java ancora un risultato arrotondato di tipo *int*, il quale sarebbe inutile ai nostri scopi.
Riassumendo, il codice per la fuzzificazione dell'input risulta essere:

```
public double fuzzificate(int x) {

     if (x <= min1 || x >= min2) {
        if ((max2==min2 && x>=min2) || (max1==min1 && x<=min1))
           return 1;
        else
           return 0;
     }

     if (x >= max1 && x <= max2)
           return 1;

     double newX = (double) x;
     if (x >= min1 && x <= max1)
           return ((newX - min1) / (max1 - min1));
     else
           return ((min2 - newX) / (min2 - max2));

}
```

3.4 Inferenza

Come è stato spiegato nel paragrafo 1.2.5, il motore di inferenza valuta ciascuna regola e crea per ognuna un insieme fuzzy di output. Il metodo di implicazione scelto per l'implementazione di questa fase è quello di *Mamdani*.
Il metodo in questione, che è stato chiamato *inference,* è un metodo dinamico che deve essere invocato sull'oggetto *MembershipFunction* che rappresenta la funzione di appartenenza del conseguente della regola presa in esame.
Per esempio, se la regola fosse:

If X is A then Y is B

allora il metodo andrebbe invocato sull'oggetto che rappresenta la variabile linguistica *B*.
Inoltre esso riceve come parametro di ingresso il grado di verità della regola e restituisce un oggetto di tipo *MembershipFunction*, l'insieme fuzzy dell'output della regola.

```
public MembershipFunction inference(double mu) {
…
}
```

Come accennato in precedenza, l'effetto dell'implicazione di Mamdani è quello di troncare la funzione di appartenenza all'altezza indicata dal valore del grado di verità della regola, come in figura.

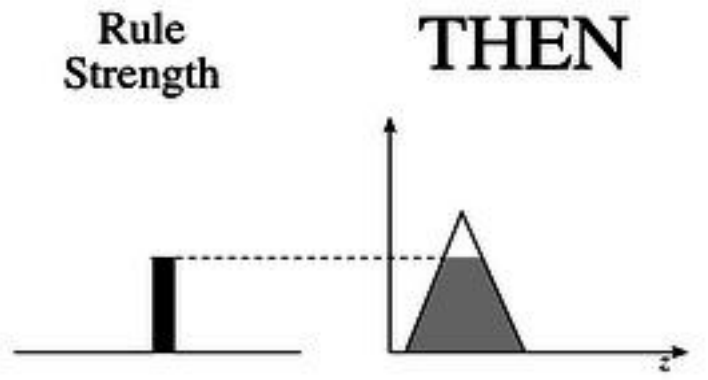

Figura 3.2: Implicazione di Mamdani

Il metodo inizia, innanzitutto creando le variabili che serviranno a caratterizzare la nuova funzione di appartenenza, ovvero i parametri di ingresso del costruttore di *MembershipFunction*.

```
int a,b,c,d;
double h;
```

A questo punto, come si può facilmente intuire, se *mu=1* la funzione di appartenenza non viene modificata e il metodo termina la sua esecuzione ritornano la funzione di partenza; in caso contrario possiamo cominciare ad inizializzare alcune variabili di cui sappiamo già sicuramente i valori, ovvero:

a) *min1* corrisponde a quello della funzione di partenza
b) *min2* corrisponde a quello della funzione di partenza
c) *h* (l'altezza massima della funzione) è uguale a *mu*

Segue, quindi il codice di questa frazione del metodo.

```
if (mu == 1)
      return this;

a = min1;
d = min2;
h = mu;
```

A questo punto si tratta di capire quale dei quattro tipo di funzione di appartenenza deve essere troncata, in modo da poter agire di conseguenza.
Analizziamo i vari casi che, in realtà, si riducono ad essere tre:

1) Se la funzione è Trapezoidale, anche la nuova funzione avrà la stessa tipologia; inoltre, come si può osservare dalla figura 3.1, anche la funzione triangolare, una volta troncata, diventa di tipo trapezoidale. Questi due tipi di funzioni si possono identificare con la caratteristica comune che:

$$min1 \not\equiv max1 \; e \; min2 \not\equiv max2$$

Quindi, per chiudere questo punto, basta calcolare i punti *max1* e *max2* applicando le formule inverse a quelle per ottenere il valore di appartenenza:

```
if (min1 != max1 && min2 != max2) {
      Double bTemp = mu*(max1 - min1) + min1;
      b = bTemp.intValue();
      Double cTemp = min2 - (mu*(min2 - max2));
      c = cTemp.intValue();
}
```

2) Se la funzione è di tipo Sigma Right-open, allora quella risultante dalla troncatura, appartiene allo stesso tipo.
   Questo tipo di funzione si può riconoscere dal fatto che

$$min2 \equiv max2$$

   Quindi l'unico punto che rimane da calcolare è *max1*, e che si può facilmente ottenere come nel caso precedente:

```
if (min2 == max2) {
      Double bTemp = mu*(max1 - min1) + min1;
      b = bTemp.intValue();
      c = d;
}
```

3) Il caso Sigma Left-open è simmetrico al punto 2:

```
else {
      b = a;
      Double cTemp = min2 - (mu*(min2 - max2));
      c = cTemp.intValue();
}
```

Infine, se il valore del grado di verità della regola è nullo (*mu=0*), risulta una funzione nulla, che rappresentiamo qui ponendo tutti i parametri a 0.

```
if (mu == 0) {
      a = b = c = d = 0;
      h = 0;
}
```

A questo punto possiamo creare, con le variabili finora definite, il nuovo oggetto di tipo *MembershipFunction*.

```
MembershipFunction result = new MembershipFunction(a,b,c,d,h);
```

Risulta a questo punto chiarito anche il motivo per cui, nella spiegazione del creatore della classe, abbiamo inserito il parametro che indica l'altezza massima della funzione di appartenenza.
Non rimane che unire il codice del metodo.

```
public MembershipFunction inference(double mu) {

      int a,b,c,d;
      double h;

      if (mu == 1)
            return this;

      a = min1;
      d = min2;
      h = mu;

      if (min1 != max1 && min2 != max2) {
            Double bTemp = mu*(max1 - min1) + min1;
            b = bTemp.intValue();
            Double cTemp = min2 - (mu*(min2 - max2));
            c = cTemp.intValue();
      }
      else {
            if (min2 == max2) {
                  Double bTemp = mu*(max1 - min1) + min1;
                  b = bTemp.intValue();
                  c = d;
            }
            else {
                  b = a;
                  Double cTemp = min2 - (mu*(min2 - max2));
                  c = cTemp.intValue();
            }
      }

      if (mu == 0) {
            a = b = c = d = 0;
            h = 0;
      }

   MembershipFunction result = new MembershipFunction(a,b,c,d,h);
   return result;
}
```

3.5 Defuzzificazione

Per l'implementazione dell'ultimo blocco del controllore fuzzy è stato scelto il metodo di defuzzificazione *Center of Gravity (COG)* che, come spiegato nel paragrafo 1.2.6, si basa sui fuzzy set delle singole regole presi separatamente, e ha come formula:

$$y_q^{crisp} = \frac{\sum_{i=1}^{R} b_i^q \int_{y_q} \mu_{\hat{B}_q^i}(y_q) dy_q}{\sum_{i=1}^{R} \int_{y_q} \mu_{\hat{B}_q^i}(y_q) dy_q}$$

Innanzitutto il metodo è stato chiamato *defuzCOG()*, è un metodo statico, e restituisce l'intero che rappresenta il valore di uscita dell'output dell'intero processo fuzzy.
Come parametri di input riceve i fuzzy set di uscita delle singole regole, sotto forma dei loro corrispondenti oggetti *MembershipFunction*.
Da notare che sono stati creati due di questi metodi, uno che riceve in ingresso tre parametri e uno che ne riceve quattro, visto che negli esempi che saranno presentati sono presenti al più quattro regole.

```
public static int defuzCOG(MembershipFunction a,
MembershipFunction b, MembershipFunction c) {
…
}
```

```
public static int defuzCOG(MembershipFunction a,
MembershipFunction b, MembershipFunction c, MembershipFunction d)
{
…
}
```

Guardando alla formula per calcolare y_q^{crisp} si può notare come il punto centrale stia nell'integrale per trovare le aree sottese dalla funzioni di appartenenza di uscita delle regole $\int_{y_q} \mu_{\hat{B}_q^i}(y_q) dy_q$.
Come è stato detto in precedenza, però, spesso si possono trovare delle formule semplici per calcolare tali aree, ed è il nostro caso dato che le funzioni di appartenenza utilizzate appartengono ad uno

dei quattro tipi per i quali l'area è finita e riconducibile alla somma di aree di semplici figure geometriche.
Vediamo ora nel dettaglio le tre aree che sommate danno l'area complessiva sottesa dalla funzione di appartenenza.

1) La prima parte è quella per $min1 \leq x \leq max1$

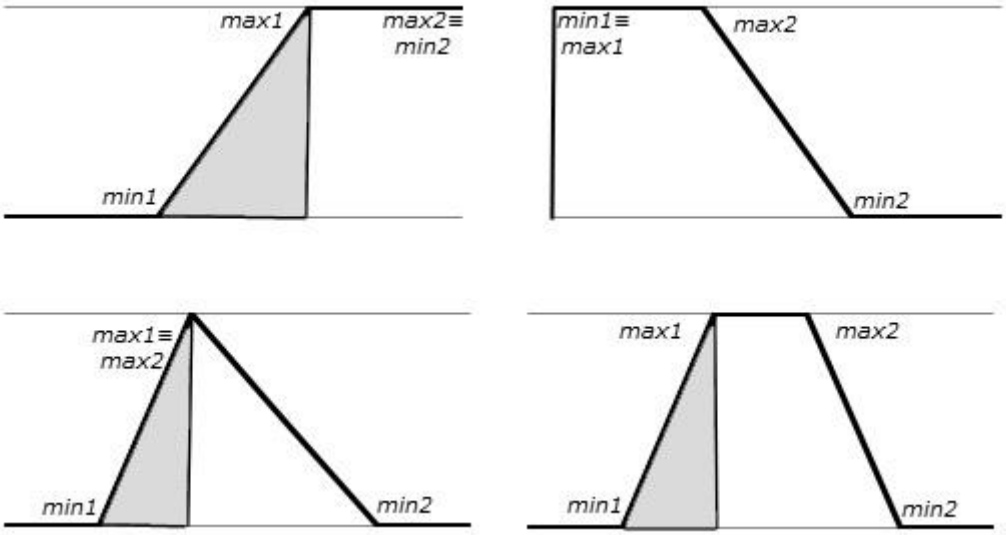

Figura 3.3: Aree comprese tra min1 e max1

Come è evidente, dalla fig. 3.3, questa sezione si può ottenere con la formula:

$$Area1 = \frac{(max1 - min1) * h}{2}$$

dove h è l'altezza massima della funzione di appartenenza.
Infatti nel caso del tipo Sigma Left-open la sottrazione al numeratore risulta uguale a 0 e di conseguenza anche l'area sarà nulla, mentre negli altri tre casi essa corrisponde all'area di un triangolo avente come base il segmento con estremi $min1$ e $max1$ e altezza h.

2) La seconda parte è quella per $max1 \leq x \leq max2$

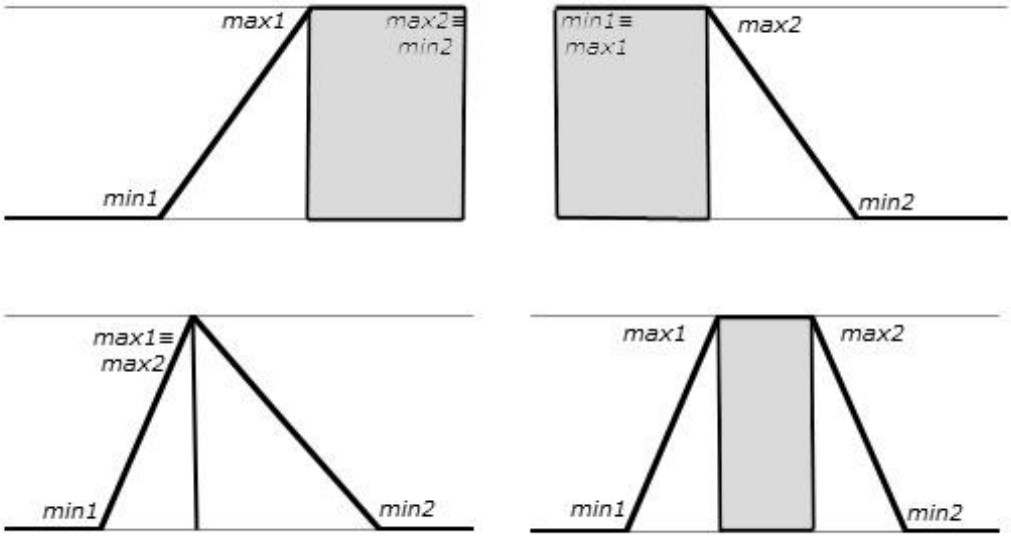

Figura 3.4: Aree comprese tra max1 e max2

In questo caso l'area è nulla nella funzione di tipo triangolare, in cui $max1 \equiv max2$, mentre negli altri tipi corrisponde all'area di un rettangolo avente come base il segmento con estremi $max1$ e $max2$ e come altezza h:

$$Area2 = (max2 - max1) * h$$

3) La terza parte è quella per cui $max2 \leq x \leq min2$

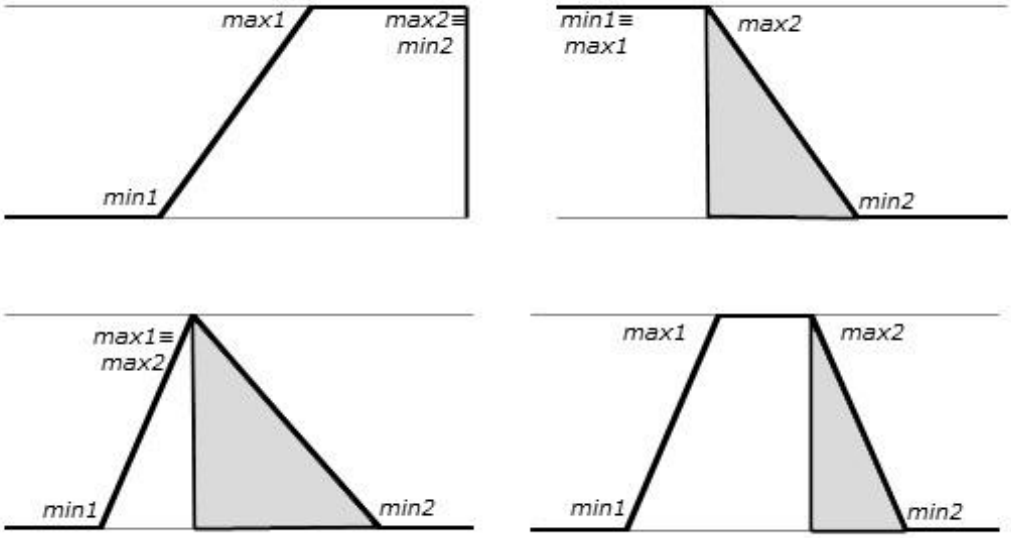

Figura 3.5: Aree comprese tra max2 e min2

Questa situazione è simmetrica a quella del punto 1; qui è la parte sottesa nella funzione Sigma Right-open ad essere nulla; per le altre tre, corrisponde all'area di un triangolo con base il segmento con estremi $max2$ e $min2$ e altezza h:

$$Area3 = \frac{(min2 - max2) * h}{2}$$

Fatto ciò non rimane che sommare le tre aree per ottenere quella totale sottesa dalla funzione di appartenenza, e tale operazione va evidentemente ripetuta per ognuno degli oggetti di input.

```
double areaA = ((a.max1-a.min1)*a.alt)/2 + (a.max2-a.max1)*a.alt
+ ((a.min2-a.max2)*a.alt)/2;
```

Per quanto riguarda il centro b_i^q dell'insieme fuzzy, esso corrisponde, per tutti i tipi di funzione, al punto medio tra $min1$ e $min2$; quindi:

```
double centroA = ((a.min2 - a.min1) / 2) +  a.min1;
```

Ora che tutti i dati necessari al calcolo della formula finale sono stati raccolti, è possibile applicare la formula stessa per ottenere il valore dell'output del controllore fuzzy.

```
Double cOG = (areaA*centroA + areaB*centroB + areaC*centroC) /
(areaA + areaB + areaC);
```

Come ultimo passo di questo metodo, visto che i valori da applicare ai metodi per controllare i motori del NXT in lejOS sono valori interi, il risultato viene approssimato all'unità più vicina.

```
return cOG.intValue();
```

Il codice completo, quindi, nell'esempio che riceve tre parametri di input, risulta:

```
public static int defuzCOG(MembershipFunction a,
MembershipFunction b, MembershipFunction c) {


      double areaA = ((a.max1 - a.min1)*a.alt) / 2 + (a.max2 -
a.max1)*a.alt + ((a.min2 - a.max2)*a.alt) / 2;
…
```

```
…
      double areaB = ((b.max1 - b.min1)*b.alt) / 2 + (b.max2 -
      b.max1)*b.alt + ((b.min2 - b.max2)*b.alt) / 2;

      double areaC = ((c.max1 - c.min1)*c.alt) / 2 + (c.max2 -
      c.max1)*c.alt + ((c.min2 - c.max2)*c.alt) / 2;

      double centroA = ((a.min2 - a.min1) / 2) +  a.min1;

      double centroB = ((b.min2 - b.min1) / 2) +  b.min1;

      double centroC = ((c.min2 - c.min1) / 2) +  c.min1;

      Double cOG = (areaA*centroA + areaB*centroB + areaC*centroC)
      / (areaA + areaB + areaC);

      return cOG.intValue();
}
```

Capitolo 4

Il line follower

4.1 Introduzione

Il primo dei due esempi su cui sono state studiate le differenze tra la programmazione a behavior e la logica fuzzy è un tipo di costruzione robotica comunemente chiamato *line follower*, il cui compito, come si può dedurre dal nome, è quello di seguire una linea.
Strutturalmente questo robot non è altro che un piccolo veicolo bimotore sul quale viene montato, nella parte anteriore, un sensore di luminosità puntato verso il suolo, sulle cui rilevazioni è incentrata la programmazione.

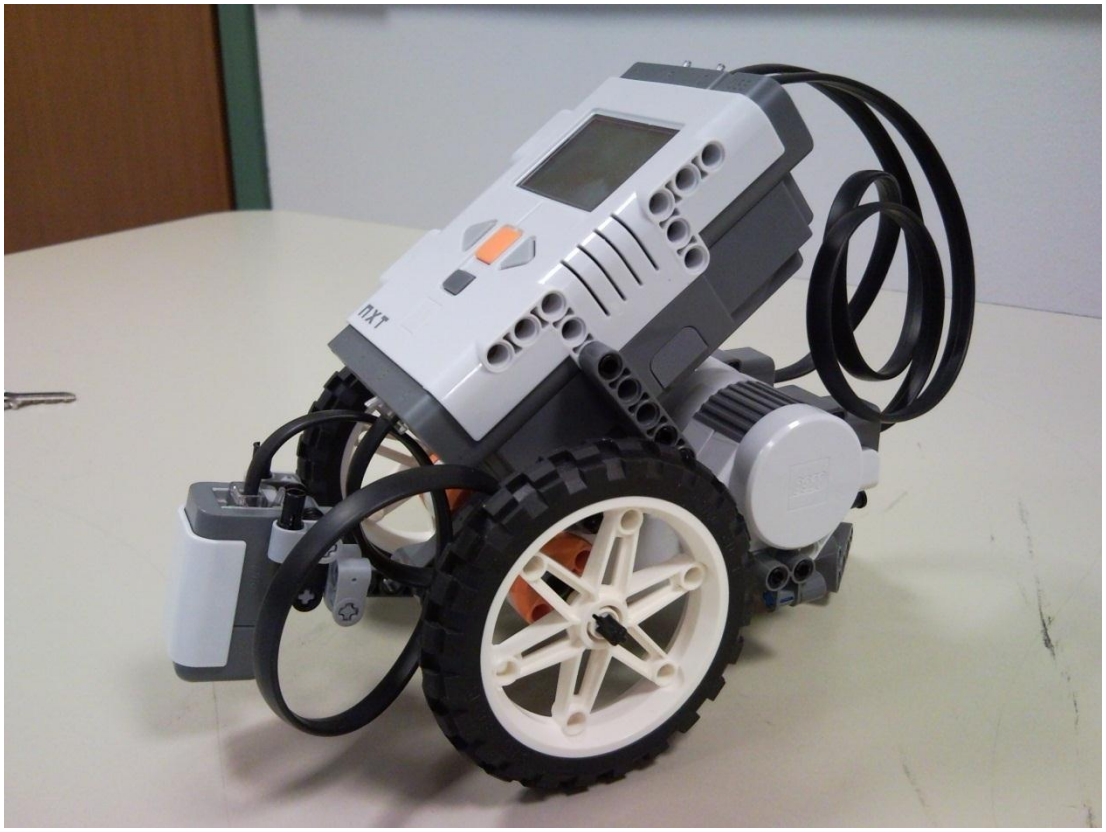

Figura 4.1: Robot Line Follower

I componenti fondamentali e i relativi metodi leJOS utilizzati per la realizzazione software in questa relazione sono i seguenti:

1. *Brick NXT*

Figura 4.2: Brick NXT

2. *Sensore di luminosità*

Figura 4.3: Sensore di luminosità

   a. *public LightSensor(ADSensorPort port, boolean floodlight)*
      Crea un oggetto sensore di luminosità connesso alla porta specificata e imposta lo stato del LED (acceso o spento)

   b. *public int getNormalizedLIghtValue()*
      Come il metodo precedente, ma consente una maggiore accuratezza ampliando l'intervallo in modo che il valore restituito sia teoricamente compreso tra 0 e 1023; in realtà va tipicamente da 145 a 890 nel caso di luce rifless a partire da quella emessa dal LED del sensore

3. *Motore NXT*

Figura 4.4: Motore NXT

a. *public void backward()*
   Imposta la rotazione all'indietro del motore

b. *public void forward()*
   Imposta la rotazione avanti del motore

c. *public void regulateSpeed(boolean yes)*
   Attiva o disattiva la regolazione della velocità del motore

d. *public void smoothAcceleration(boolean yes)*
   Fa in modo che il motore aumenti acceleri progressivamente senza overshoot quando viene usata la regolazione della velocità

e. *public void setSpeed(int speed)*
   Imposta la velocità angolare dell'asse esterno del motore in *gradi/sec*, fino ad un massimo di 900 con una tensione di 8 V

4.2 Il line follower implementato a behavior

Il comportamento che il robot deve tenere è quello di seguire una semplice linea nera; per ottenere ciò ci si basa sulla luminosità rilevata dal sensore per farlo girare a destra o a sinistra quando tende ad allontanarsi dalla linea.
Da notare che in realtà il robot non segue la linea in se stessa ma il bordo, in questo esempio si è scelto di fargli seguire quello destro.
Infatti viene programmato in modo che:

- se la luce è molto intensa, il robot gira velocemente a sinistra
- se la luce è intensa il robot gira lentamente a sinistra
- se la luce è compresa in un certo intervallo, in cui si considera il sensore posizionato sul bordo della linea, il robot prosegue diritto
- se la luce è debole il robot gira lentamente a destra
- se la luce è molto debole il robot gira velocemente a destra

Bisogna assegnare gli intervalli di luminosità che determinano in quale delle condizioni si trova il robot ad ogni istante.
Sperimentalmente, gli i valori assegnati per gli estremi degli intervalli sono i seguenti:

1. $Se\ luce \geq 550\ gira\ a\ sinistra\ velocemente$
2. $Se\ 495 < luce < 550\ gira\ a\ sinistra\ lentamente$
3. $Se\ 465 \leq luce \leq 495\ vai\ diritto$
4. $Se\ 390 < luce < 465\ gira\ a\ destra\ lentamente$
5. $Se\ luce \leq 390\ gira\ a\ destra\ velocemente$

Procediamo quindi con l'analisi delle classi necessarie all'implementazione di tale comportamento, nell'ordine in cui sono state elencate sopra.
La classe *LeftFast* implementa l'interfaccia *Behavior.*

```
public class LeftFast implements Behavior {
…
}
```

Per la lettura dei valori di luminosità abbiamo bisogno di un oggetto della classe LightSensor. Per crearne uno, che poi venga utilizzato da tutti i behavior del programma, è stato scelto di creare un'istanza di tale sensore nel metodo *main* della classe principale e di passarlo ai vari behavior. Quindi abbiamo bisogno di una

variabile di classe che memorizzi tale sensore e viene inizializzato tramite il costruttore del behavior.

```
private LightSensor sensor;

public LeftFast(LightSensor light) {
	sensor = light;
}
```

Ora dobbiamo realizzare il metodo che permette all'arbitrator di decidere quando il behavior può prendere il controllo e, come è stato detto al punto 1, questo avviene quando la luminosità è maggiore o uguale a 550.

```
public boolean takeControl() {
	int x = sensor.readNormalizedValue();
	return (x >= 550);
}
```

Passiamo ora al metodo *action*: deve far ruotare all'indietro il motore di sinistra (150 rad/s) e avanti velocemente quello di destra (900 rad/s), in modo che l'effetto sia quello che il robot sterzi a sinistra in maniera decisa, poi mettersi in attesa finchè la condizione di luce non cambia. Si è scelto anche però di ridurre tutte le velocità di un fattore 1/2, altrimenti il robot sbanda troppo.

```
public void action() {
	Motor.C.backward();
	Motor.B.forward();
	Motor.C.setSpeed(150*1/2);
	Motor.B.setSpeed(900*1/2);
	int x = sensor.readNormalizedValue();
	while (x >= 550 && !Button.ESCAPE.isPressed()) {
		x = sensor.readNormalizedValue();
	}
}
```

Il metodo *suppress*, infine, non esegue serve che esegua alcuna azione, in quanto le velocità dei motori sono modificate dagli altri behavior che prendono il controllo.

```
public void suppress() {}
```

Ecco il codice completo della classe.

```
import lejos.nxt.*;
import lejos.robotics.subsumption.*;

public class LeftFast implements Behavior {
	private LightSensor sensor;

	public LeftFast(LightSensor light) {
		sensor = light;
	}

	public boolean takeControl() {
		int x = sensor.readNormalizedValue();
		return (x >= 550);
	}

	public void suppress() {}

	public void action() {
		Motor.C.backward();
		Motor.B.forward();
		Motor.C.setSpeed(150*1/2);
		Motor.B.setSpeed(900*1/2);
		int x = sensor.readNormalizedValue();
		while (x >= 550 && !Button.ESCAPE.isPressed()) {
			x = sensor.readNormalizedValue();
		}
	}
}
```

Gli altri comportamenti sono molto simili e quindi non richiedono alcun chiarimento aggiuntivo. Si differenziano da quello appena proposto solamente per gli estremi dell'intervallo di valori in cui il behavior prende il controllo e per le velocità del motore impostate nel metodo *action*.
La classe *Left* fa virare il robot a sinistra lentamente.

```
import lejos.nxt.*;
import lejos.robotics.subsumption.*;

public class Left implements Behavior {
	private LightSensor sensor
…
```

```
…
	public Left(LightSensor light) {
		sensor = light;
	}

	public boolean takeControl() {
		int x = sensor.readNormalizedValue();
		return (x > 495 && x < 550);
	}

	public void suppress() {}

	public void action() {
		Motor.C.forward();
		Motor.B.forward();
		Motor.C.setSpeed(250*1/2);
		Motor.B.setSpeed(450*1/2);
		int x = sensor.readNormalizedValue();
		while (x < 550 && x >= 475 &&
		    !Button.ESCAPE.isPressed()) {
			x = sensor.readNormalizedValue();
		}
	}
}
```

La classe *Straight* fa proseguire il robot diritto.

```
import lejos.nxt.*;
import lejos.robotics.subsumption.*;

public class Straight implements Behavior {
	private LightSensor sensor;

	public Straight(LightSensor light) {
		sensor = light;
	}

	public boolean takeControl() {
		int x = sensor.readNormalizedValue();
		return (x >= 465 && x <= 495);
	}

	public void suppress() {}
…
```

```
…
	public void action() {
		Motor.C.forward();
		Motor.B.forward();
		Motor.C.setSpeed(450*1/2);
		Motor.B.setSpeed(450*1/2);
		int x = sensor.readNormalizedValue();
		while (x < 485 && x >= 475 &&
		     !Button.ESCAPE.isPressed()) {
		        x = sensor.readNormalizedValue();
		}
	}
}
```

La classe *Right* fa girare il robot a destra lentamente

```
import lejos.nxt.*;
import lejos.robotics.subsumption.*;

public class Right implements Behavior {
	private LightSensor sensor;

	public Right(LightSensor light) {
		sensor = light;
	}

	public boolean takeControl() {
		int x = sensor.readNormalizedValue();
		return (x > 390 && x < 465);
	}

	public void suppress() {}

	public void action() {
		Motor.C.forward();
		Motor.B.forward();
		Motor.C.setSpeed(450*1/2);
		Motor.B.setSpeed(250*1/2);
		int x = sensor.readNormalizedValue();
		while (x > 390 && x < 475 &&
		     !Button.ESCAPE.isPressed()) {
		        x = sensor.readNormalizedValue();
		}
	}
}
```

Infine la classe *RightFast* fa virare il robot a destra velocemente.

```
import lejos.nxt.*;
import lejos.robotics.subsumption.*;

public class RightFast implements Behavior {
	private LightSensor sensor;

	public RightFast(LightSensor light) {
		sensor = light;
	}

	public boolean takeControl() {
		int x = sensor.readNormalizedValue();
		return (x <= 390);
	}

	public void suppress() {}

	public void action() {
		Motor.C.forward();
		Motor.B.backward();
		Motor.C.setSpeed(900*1/2);
		Motor.B.setSpeed(150*1/2);
		int x = sensor.readNormalizedValue();
		while (x <= 390 && !Button.ESCAPE.isPressed()) {
			x = sensor.readNormalizedValue();
		}
	}
}
```

Teoricamente potremmo anche concludere qui la realizzazione dei behavior; tuttavia, per comodità, ne inseriamo anche un terzo che permette di interrompere l'esecuzione del programma in qualsiasi momento premendo il pulsante ESCAPE sul robot.
Non approfondiamo l'analisi di questo behavior in quanto non è indispensabile, basta dire che quando il pulsante *Button.ESCAPE* viene premuto, il valore che determina se questo behavior può prendere il controllo viene impostato a *true* e quando prende effettivamente il controllo semplicemente termina l'esecuzione del programma chiamando *System.exit(0).*

```
import lejos.robotics.subsumption.*;
import lejos.nxt.*;

public class Exit implements Behavior, ButtonListener {
	boolean pressed;

	public Exit() {
		pressed = false;
		Button.ESCAPE.addButtonListener(this);
	}

	public void buttonPressed(Button b) {
		pressed = true;
	}

	public void buttonReleased(Button b) {
		pressed = true;
	}

	public boolean takeControl() {
		return pressed ;
	}

	public void suppress() {}

	public void action() {
		System.exit(0);
	}
}
```

A questo punto non rimane che realizzare il metodo *main* che crea le istanze dei vari behavior, dell'arbitrator e li manda in esecuzione. Esso deve innanzitutto creare un oggetto di tipo *LightSensor*.

```
LightSensor light = new LightSensor(SensorPort.S3, true);
```

Successivamente deve anche creare le istanze dei behavior che verranno poi passate all'arbitrator usando, dove necessario, l'oggetto *light* appena creato.

```
Behavior leftFast = new LeftFast(light);
Behavior left = new Left(light);
Behavior straight = new Straight(light);
Behavior right = new Right(light);
Behavior rightFast = new RightFast(light);
Behavior exit = new Exit();
```

Ora, bisogna decidere le priorità dei behavior, in particolare:

- *exit* deve avere priorità massima, dato che quando il pulsante *ESCAPE* viene premuto, tutti gli altri behavior devono essere soppressi e il programma terminato
- gli altri behavior sono interscambiabili in quanto non divengono mai eseguibili contemporaneamente, data la natura delle condizioni dei loro metodi *takeControl*

La realizzazione dell'array contenente i behavior e l'inizializzazione dell'arbitrator è quindi la seguente:

```
Behavior[] bArray = {leftFast,left, straight, right, rightFast,
   exit};
Arbitrator arby = new Arbitrator(bArray);
```

Non dimentichiamo inoltre che per poter regolare la velocità dei due motori è necessario impostare il relativo valore:

```
Motor.B.regulateSpeed(true);
Motor.C.regulateSpeed(true);
```

Infine non resta che avviare l'arbitrator come conclusione del metodo.

```
arby.start();
```

Ecco il codice completo per il metodo *main*; da notare che è stato inserito nella classe LineFollower, che è quella che viene caricata nel robot.

```
import lejos.robotics.subsumption.*;
import lejos.nxt.*;

public class LineFollowerNew {
      public static void main(String[] args) {
              LightSensor light = new
                  LightSensor(SensorPort.S3, true);
              Behavior leftFast = new LeftFast(light);
              Behavior left = new Left(light);
…
```

```
…
                Behavior straight = new Straight(light);
                Behavior right = new Right(light);
                Behavior rightFast = new RightFast(light);
                Behavior exit = new Exit();
                Behavior[] bArray = {leftFast,left, straight,
                     right, rightFast, exit};
                Arbitrator arby = new Arbitrator(bArray);
                Motor.B.regulateSpeed(true);
                Motor.C.regulateSpeed(true);
                Motor.B.smoothAcceleration(true);
                Motor.C.smoothAcceleration(true);
                arby.start();
        }
}
```

4.3 Il line follower fuzzy

4.3.1 Progettazione

Prima di procedere con la spiegazione del codice, è necessario, nel caso della logica fuzzy, chiarire le scelte che sono state effettuate in fase di progettazione.
Non c'è bisogno di parlare del metodo di implicazione e di defuzzificazione scelti, dato che sono già stati spiegati nei paragrafi 3.4 e 3.5, quindi le scelte fondamentali rimaste per la realizzazione del controllore fuzzy sono:

a) le regole fuzzy

b) le funzioni di appartenenza gli inisiemi fuzzy

Iniziamo col dire che, al contrario della programmazione a behavior, in questo caso si è scelto di utilizzare l'intervallo di luminosità più ampio fornito dal metodo *getNormalizedLightValue*, vista la maggiore precisione necessaria.
Le variabili linguistiche utilizzate sono tre:

1. *Luce* per rappresentare la luminosità rilevata dal sensore, divisa in cinque insiemi fuzzy:
   - *molto debole*
   - *debole*
   - *media*
   - *forte*
   - *molto forte*
2. *MotorB* per rappresentare la velocità del motore destro, suddivisa in quattro insiemi fuzzy
   - *indietro*
   - *lento*
   - *medio*
   - *veloce*
3. *MotorC* per rappresentare la velocità del motore sinistro, suddivisa allo stesso modo della variabili *MotorB*

Anche in questo caso si è scelto di far seguire al robot il bordo destro della linea, quindi il motore di destra deve aumentare la velocità man mano che la luminosità rilevata aumenta e diminuire

man mano che la luminosità rilevata diminuisce; per il motore di sinistra vale evidentemente il contrario.
La tabella seguente riassume le condizioni, che sono le stesse utilizzate nella programmazione a behavior.

Luce	MotorB	MotorC
molto debole	indietro	veloce
debole	lento	medio
media	medio	medio
forte	medio	lento
molto forte	veloce	indietro

Da questa si possono ricavare il sistema di regole che sono state divise per i due output. Esse risultano così strutturate:

- $if\ \boldsymbol{Luce}\ is\ \boldsymbol{molto\ debole}\ then\ \boldsymbol{MotorB}\ is\ \boldsymbol{indietro}$
- $if\ \boldsymbol{Luce}\ is\ \boldsymbol{debole}\ then\ \boldsymbol{MotorB}\ is\ \boldsymbol{lento}$
- $if\ \boldsymbol{Luce}\ is\ \boldsymbol{media}\ or\ \boldsymbol{forte}\ then\ \boldsymbol{MotorB}\ is\ \boldsymbol{medio}$
- $if\ \boldsymbol{Luce}\ is\ \boldsymbol{molto\ forte}\ then\ \boldsymbol{MotorB}\ is\ \boldsymbol{veloce}$
- $if\ \boldsymbol{Luce}\ is\ \boldsymbol{molto\ debole}\ then\ \boldsymbol{MotorC}\ is\ \boldsymbol{veloce}$
- $if\ \boldsymbol{Luce}\ is\ \boldsymbol{debole}\ or\ \boldsymbol{media}\ then\ \boldsymbol{MotorC}\ is\ \boldsymbol{medio}$
- $if\ \boldsymbol{Luce}\ is\ \boldsymbol{forte}\ then\ \boldsymbol{MotorC}\ is\ \boldsymbol{lento}$
- $if\ \boldsymbol{Luce}\ is\ \boldsymbol{molto\ forte}\ then\ \boldsymbol{MotorC}\ is\ \boldsymbol{indietro}$

Attraverso alcuni test, si è trovato che in condizioni di luce normali, la luminosità rilevata col metodo *getNormalizedLIghtValue è circa 370* al centro della linea, circa 570 nella parte bianca del foglio e circa 500 nella zona in cui si dovrebbe mantenere il robot.
Le funzioni di appartenenza risultano quindi così come illustrato nel grafico.

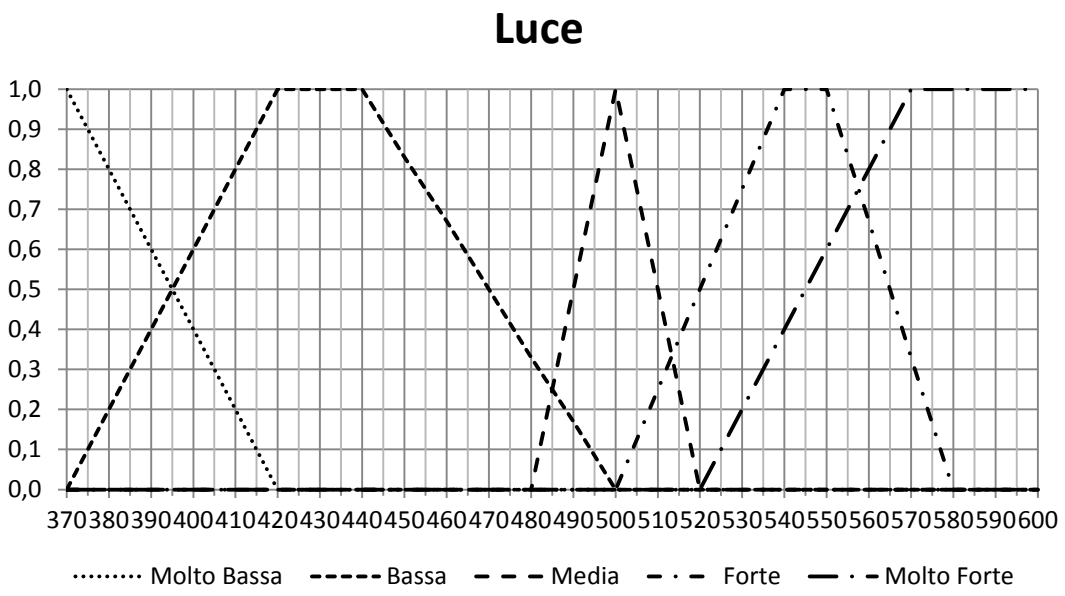

Figura 4.5: Funzioni di appartenenza variabile Luce

Per quanto riguarda i motori, dato che per regolarne la velocità è stato usato il metodo *setSpeed*, l'intervallo dei valori scelto va da -300 a 900.
Le funzioni di appartenenza degli insiemi fuzzy delle variabili $\boldsymbol{MotorB}$ e $\boldsymbol{MotorC}$ sono raffigurate nel grafico.

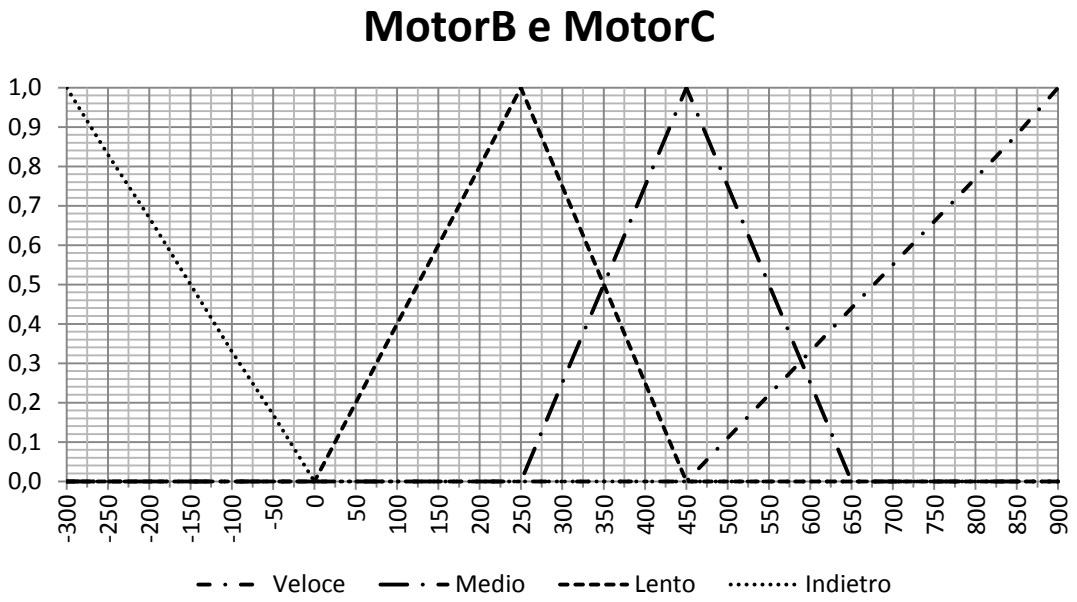

Figura 4.6: Funzioni di appartenenza variabili MotorB e MotorC

4.3.2 Realizzazione

Passiamo ora alla realizzazione del codice. Innanzitutto è stata creata la classe *FuzzyFollower* che implementa l'interfaccia *ButtonListener* e i relativi metodi, i quali fanno in modo che, quando il pulsante per uscire viene premuto, sia possibile che tale evento venga memorizzato anche se è in corso l'esecuzione di un altro pezzo di codice, e che, appena si presenti il controllo che verifica se si deve terminare l'applicazione, esso risulti positivo.

```
public class FuzzyFollower implements ButtonListener {
	private boolean exit;

	public FuzzyFollower() {
		exit = false;
		Button.ESCAPE.addButtonListener(this);
	}

	public void buttonPressed(Button b) {
		exit = true;
	}

	public void buttonReleased(Button b) {}
…
}
```

Successivamente inizia il metodo main, che ha come prime istruzioni quelle che inizializzano l'oggetto *LightSensor* collegato alla porta numero 3, che rappresenta il sensore di luminosità, e l'oggetto *FuzzyFollower*, del quale verrà controllata la variabile *exit* come condizione del ciclo presente più avanti, per sapere quando l'esecuzione deve terminare.

```
LightSensor sensor = new LightSensor(SensorPort.S3, true);
FuzzyFollower f = new FuzzyFollower();
```

Ora è necessario creare ed inizializzare le variabili che contengono le funzioni di appartenenza che saranno utilizzate per tutto il processo fuzzy, come oggetti della classe *MembershipFunction*, con i parametri di cui si è discusso nel paragrafo 4.3.1 riguardante la progettazione. Qui si sono utilizzati dei nomi abbreviati per gli oggetti rappresentanti i vari livelli di luminosità, cioè *hB* (molto

debole), *b* (debole), *m* (media), *w* (forte), *hW* (molto forte); i termini riguardanti le velocità del motore sono indicati dai relativi vocaboli in lingua inglese.

```
MembershipFunction hB = new MembershipFunction(370,370,370,420,1);
MembershipFunction b = new MembershipFunction(370,420,440,500,1);
MembershipFunction m = new MembershipFunction(480,500,500,520,1);
MembershipFunction w = new MembershipFunction(500,540,550,580,1);
MembershipFunction hW = new MembershipFunction(520,570,600,600,1);

MembershipFunction reverse = new MembershipFunction(-300,-300,-300,0,1);
MembershipFunction slow = new MembershipFunction(0,250,250,450,1);
MembershipFunction medium = new MembershipFunction(250,450,450, 650,1);
MembershipFunction fast = new MembershipFunction(450,900,900,900,1);
```

Prima di entrare nel ciclo che calcolerà di volta in volta, a seconda dell'ingresso, i valori delle uscite, è necessario attivare il controllo della velocità tramite il relativo metodo applicato ad entrambi i motori.

```
Motor.B.regulateSpeed(true);
Motor.C.regulateSpeed(true);
```

La condizione di ingresso del ciclo è quella che non sia stato premuto il pulsante *Button*.ESCAPE sul robot.
Come prima istruzione viene memorizzato in una variabile il valore rilevato dal sensore.

```
int x = sensor.readNormalizedValue();
```

Poi viene eseguita la prima parte del processo fuzzy, ovvero la fuzzificazione dell'input, che viene trasformato in valori compresi tra 0 e 1 che indicano i gradi di appartenenza agli insiemi fuzzy della variabile *Luce*.

```
double muHB = hB.fuzzificate(x);
double muB = b.fuzzificate(x);
double muM = m.fuzzificate(x);
double muW = w.fuzzificate(x);
double muHW = hW.fuzzificate(x);
```

Successivamente, con la fase di inferenza, si ottengono le funzioni di appartenenza di uscita separate, come è previsto per l'applicazione del metodo di defuzzificazione scelto (par 3.5).

In ogni chiamata del metodo *inference*, viene passato come argomento il grado di verità della regola rispettiva; in questo caso tutte le regole, tranne due, hanno un'antecedente semplice e quindi i gradi di verità di quelle regole sono dati dai gradi di appartenenza dell'input agli antecedenti di quelle regole.
Per le due regole che invece hanno l'antecedente composto il grado di verità si calcola così:

a. $if\ \boldsymbol{Luce}\ is\ \boldsymbol{media}\ or\ \boldsymbol{forte}\ then\ \boldsymbol{MotorB}\ is\ \boldsymbol{medio}$: $\max(muM, muW)$
b. $if\ \boldsymbol{Luce}\ is\ \boldsymbol{debole}\ or\ \boldsymbol{media}\ then\ \boldsymbol{MotorC}\ is\ \boldsymbol{medio}$: $\max(muB, muM)$

Il codice sarà così:

```
//MOTORE B
MembershipFunction slowB = slow.inference(muB);
MembershipFunction fastB = fast.inference(muHW);
MembershipFunction reverseB = reverse.inference(muHB);
MembershipFunction mediumB = medium.inference(Math.max(muM,muW));

//MOTORE C
MembershipFunction reverseC = reverse.inference(muHW);
MembershipFunction mediumC = medium.inference(Math.max(muB,muM));
MembershipFunction fastC = fast.inference(muHB);
MembershipFunction slowC = slow.inference(muW);
```

Rimane ora l'ultima fase, ovvero la defuzzificazione che è molto semplice visto che tutto il codice è già stato scritto nella classe *MembershipFunction*; basta quindi chiamare il metodo *defuzCOG* passando come parametri gli oggetti che rappresentano le funzioni di appartenenza degli output.

```
int motorB = MembershipFunction.defuzCOG(slowB, reverseB,
     fastB, mediumB);
     int motorC = MembershipFunction.defuzCOG(reverseC, mediumC,
     fastC, slowC);
```

Per applicare il risultato ottenuto e terminare il codice basta impostare le direzione dei motori in base al segno dei risultati ottenuto e le velocità in base ai moduli degli stessi.

```
if (motorB < 0)
      Motor.B.backward();
else
      Motor.B.forward();
…
```

```
…
if (motorC < 0)
      Motor.C.backward();
else
Motor.C.forward();

Motor.B.setSpeed(Math.abs(motorB)*1/2);
Motor.C.setSpeed(Math.abs(motorC)*1/2);
```

Segue il codice completo della classe:

```
import lejos.nxt.*;

public class FuzzyFollower implements ButtonListener {
      private boolean exit;

      public FuzzyFollower() {
            exit = false;
            Button.ESCAPE.addButtonListener(this);
      }

      public void buttonPressed(Button b) {
            exit = true;
      }

      public void buttonReleased(Button b) {}

      public static void main(String[] args) {

            LightSensor sensor = new
                  LightSensor(SensorPort.S3, true);
            FuzzyFollower f = new FuzzyFollower();

            MembershipFunction hB = new
                  MembershipFunction(370, 370, 370, 420, 1);
            MembershipFunction b = new
                  MembershipFunction(370, 420, 440, 500, 1);
            MembershipFunction m = new
                  MembershipFunction(480, 500, 500, 520, 1);
            MembershipFunction w = new
                  MembershipFunction(500, 540, 550, 580, 1);
            MembershipFunction hW = new
                  MembershipFunction(520, 570, 600, 600, 1);
…
```

```
…
                MembershipFunction reverse = new
                      MembershipFunction(-300, -300, -300, 0, 1);
                MembershipFunction slow = new
                      MembershipFunction(0, 250, 250, 450, 1);
                MembershipFunction medium = new
                      MembershipFunction(250, 450, 450, 650, 1);
                MembershipFunction fast = new
                      MembershipFunction(450, 900, 900, 900, 1);

                Motor.B.smoothAcceleration(true);
                Motor.C.smoothAcceleration(true);
                Motor.B.regulateSpeed(true);
                Motor.C.regulateSpeed(true);

                while (!f.exit) {

                      int x = sensor.readNormalizedValue();

                      double muHB = hB.fuzzificate(x);
                      double muB = b.fuzzificate(x);
                      double muM = m.fuzzificate(x);
                      double muW = w.fuzzificate(x);
                      double muHW = hW.fuzzificate(x);

                      //MOTORE B
                      MembershipFunction slowB =
                          slow.inference(muB);
                      MembershipFunction fastB =
                          fast.inference(muHW);
                      MembershipFunction reverseB =
                          reverse.inference(muHB);
                      MembershipFunction meiumB =
                          medium.inference(Math.max(muM, muW));

                      //MOTORE C
                      MembershipFunction reverseC =
                          reverse.inference(muHW);
                      MembershipFunction mediumC =
                          medium.inference(Math.max(muB, muM));
                      MembershipFunction fastC =
                          fast.inference(muHB);
                      MembershipFunction slowC =
                          slow.inference(muW);
…
```

```
…
                        int motorB =
                          MembershipFunction.defuzCOG(slowB,
                          reverseB, fastB, mediumB);
                        int motorC =
                          MembershipFunction.defuzCOG(slowC,
                          mediumC, fastC, slowC);

                        if (motorB < 0)
                            Motor.B.backward();
                        else
                            Motor.B.forward();

                        if (motorC < 0)
                            Motor.C.backward();
                        else
                            Motor.C.forward();

                        Motor.B.setSpeed(Math.abs(motorB)*1/2);
                        Motor.C.setSpeed(Math.abs(motorC)*1/2);
                } //fine while
        } //fine main
} //fine FuzzyFollower
```

Come ultima annotazione c'è da dire che anche in questo caso gli output delle velocità dei motori vengono ridotti, sia per fare in modo che essi siano confrontabili con la programmazione a behavior, sia perché vari fattori non consentono una velocità troppo elevata, per esempio il tempo con cui il sensore rileva la luminosità oppure la limitata potenza di calcolo del sistema.

4.4 Confronto

Il robot è stato collaudato, con entrambi i metodi di programmazione, su un una linea nera uniforme chiusa a formare un circuito, la cui forma è in figura.

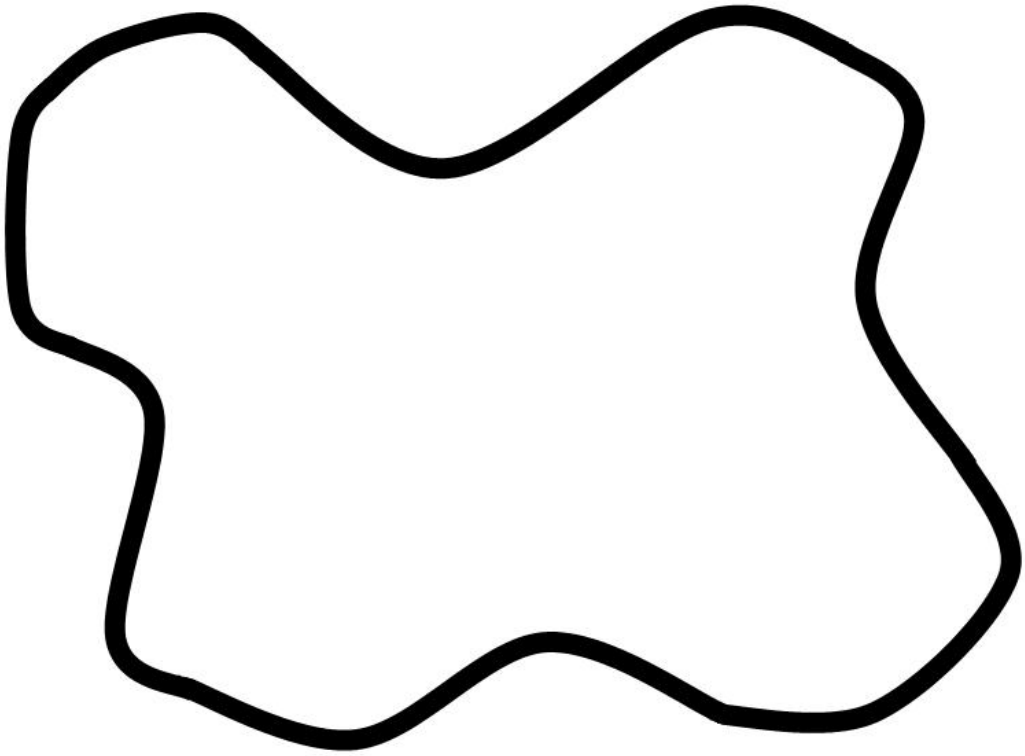

Figura 4.7: Percorso nero continuo

I due termini di paragone utilizzati per confrontare la programmazione a behavior e la logica fuzzy sono:

1. Il tempo di percorrenza del circuito
2. Quanto sono *smooth* i movimenti del robot

Segue la tabella confronto dei tempi di percorrenza.

	Tempi di percorrenza (sec)	
Numero giro	**Behavior**	**Fuzzy**
1	29,91	26,81
2	29,70	27,80
3	29,59	28,20
4	29,51	27,92
5	29,59	28,67
Media	**29,66**	**27,88**

Come si può osservare dalla tabella il tempo di percorrenza medio è migliorato, con l'approccio fuzzy, di ben *1.78 secondi*.
Dal punto di vista dei movimenti, i miglioramento non è molto evidente.

A questo punto, il robot programmato è stato fatto girare lungo una linea con i bordi sfumati, ovvero che dal centro nero si sbiadisce ai lati fino a scomparire nel bianco del foglio.
Ecco la tabella dei tempi con il nuovo tipo di linea.

	Tempi di percorrenza (sec)	
Numero giro	**Behavior**	**Fuzzy**
1	29,82	24,71
2	29,69	24,84
3	29,37	24,83
4	29,47	24,98
5	30,18	25,57
Media	**29,71**	**24,99**

Dalla nuova rilevazione dei tempi si può osservare come, con la linea sfumata, la differenza tra i tempi di percorrenza medi sia aumentata ancora, portandosi a 4.72 secondi.
Anche nei movimenti del robot il miglioramento è evidente, soprattutto in presenza di curve più accennate, dove il robot programmato a behavior effettua numerose oscillazioni per ritornare nella posizione corretta.

In conclusione si può giungere alla considerazione che l'esempio della linea continua porta dei miglioramenti lievi, in quanto essenzialmente il problema non è propriamente adatto ad essere risolto tramite la logica fuzzy, e pur rilevando i vari gradi di luminosità, rimane sempre il fatto che il bordo della linea si può considerare un oggetto binario, cioè c'è un taglio netto tra nero e bianco, 1 e 0.
Invece l'esempio della linea con i bordi che degradano verso il bianco si presta molto meglio all'approccio fuzzy per la sua natura appunto "sfumata" e porta un miglioramento più evidente del caso precedente.

Capitolo 5

Il braccio meccanico

5.1 Introduzione

Il secondo esempio che è stato utilizzato è quello di un braccio meccanico fisso il cui scopo è quello di raccogliere una palla da una posizione predefinita tramite coordinate angolari e di portarla in un posto oppure in un altro a seconda che la pallina sia rossa o blu.
Il braccio è dotato di tre articolazioni, ognuna delle quali controllata da un motore:

1. *Motor.A* apre e chiude la mano
2. *Motor.B* alza e abbassa il braccio
3. *Motor.C* ruota il braccio

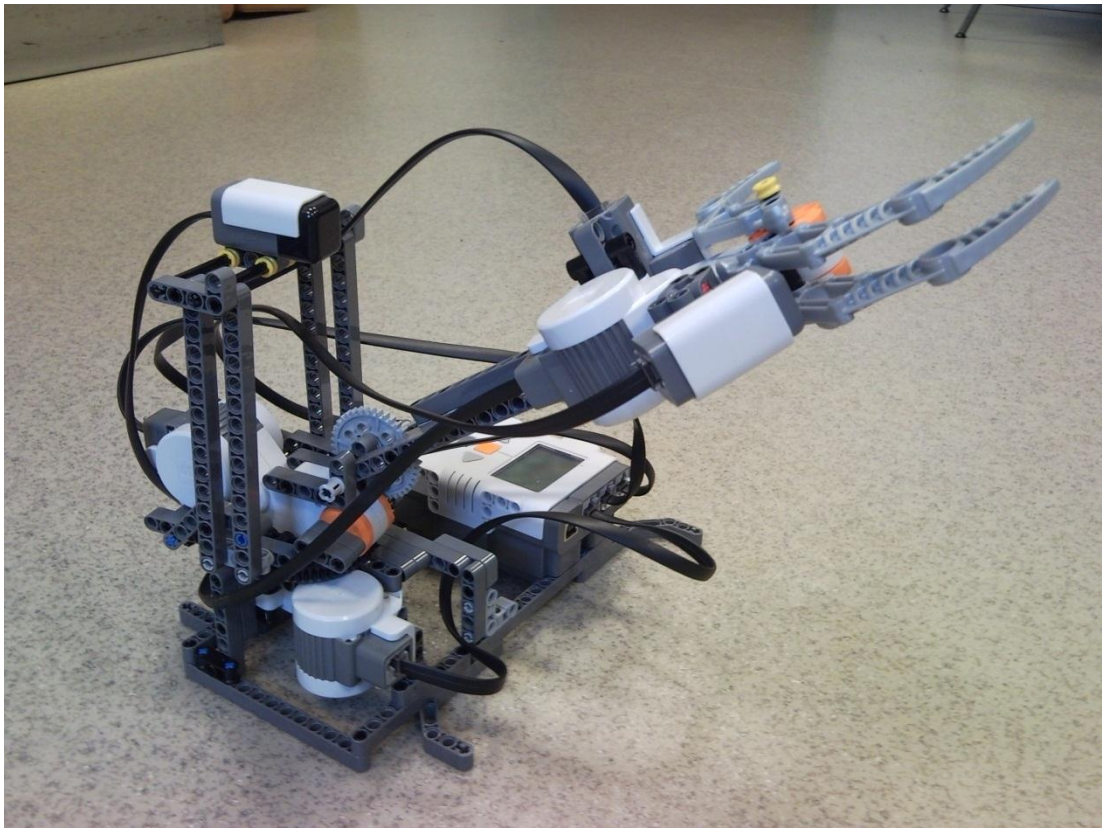

Figura 5.1: Robot Arm

I componenti e i relativi metodi LejOS utilizzati per la realizzazione di questo robot, oltre a quelli già visti nel precedente esempio, sono i seguenti:

1. Sensore di campo magnetico terrestre

Figura 5.2: Sensore di campo magnetico terrestre

a. *public CompassSensor(ADSensorPort port)*
   Crea un oggetto sensore campo magnetico terrestre connesso alla porta specificata

b. *public float getDegrees()*
   Restituisce il valore in gradi dell'angolo misurato, ponendo come angolo a 0° il nord. L'angolo cresce in senso orario.

2. Sensore di distanza ad ultrasuoni

Figura 5.3: Sensore di distanza ad ultrasuoni

a. *public UltrasonicSensor(ADSensorPort port)*
   Crea un oggetto sensore di distanza ad ultrasuoni connesso alla porta specificata

b. *public int getDistance()*
   Restituisce la distanza in centimetri approssimata all'unità più vicina. Rileva oggetti che si trovano a meno di 255 cm, quindi se nessun oggetto viene individuato restituisce il valore 255.

5.2 Il braccio meccanico implementato a behavior

Il comportamento del braccio meccanico, per come vuole essere realizzato non è altro che una serie di behavior che si attivano in sequenza, senza particolari condizioni di loro soppressione.
In serie il braccio deve compiere i seguenti ordini:

1. Ruotare, se necessario, fino all'angolo in cui deve trovarsi la la palla, che in questo caso è stato impostato per 150° rispetto al nord
2. Abbassare, se necessario, il braccio fino a posizionare la mano in modo che poter prendere la palla; sperimentalmente, per come è posizionato il sensore di distanza, si è trovato che la distanza rilevata dal sensore deve essere di 7 centimetri
3. Chiudere la mano ed afferrare la palla
4. Rialzare il braccio in modo da evitare eventuali ostacoli posti a livello del terreno; sono stati scelti 23 centimetri
5. Ruotare fino alla posizione designata per il deposito della palla, a seconda del suo colore; è stato scelto un angolo di 50° nel caso la palla sia rossa, di 250° nel caso sia blu
6. Abbassare il braccio fino a terra
7. Aprire la mano per rilasciare la palla
8. Rialzarsi

La prima classe che analizziamo è *Rotate* e implementa l'interfaccia *Behavior*.

```
public class Rotate implements Behavior{
…
}
```

Creiamo subito tre variabili necessarie al suo funzionamento:

- La variabile *degrees* di tipo *float* che deve memorizzare la destinazione, in gradi, della rotazione
- La variabile *done* di tipo *boolean* che ha il compito di controllare se tale behavior è stato eseguito, dato che, trattandosi di una sequenza, verrà eseguito solo una volta. Tale variabile è presente in tutte le classi di questa applicazione e quindi il suo significato non verrà rispiegato per i behavior successivi

- La variabile *compass* di tipo *CompassSensor* in cui viene memorizzato l'oggetto che rappresenta il sensore di campo magnetico terrestre

```
private float degrees;
private boolean done;
private CompassSensor compass;
```

Il creatore degli oggetti di tipo *Rotate* semplicemente inizializza le variabili appena viste, a seconda degli input ricevuti. Ovviamente la variabile *done* deve essere impostata a *false* in quanto, quando il behavior viene creato, non può essere stato già eseguito.

```
public Rotate(float gradi, CompassSensor sensor) {
         degrees = gradi;
         compass = sensor;
         done = false;
        }
```

Il metodo *takeControl* indica all'arbitrator che il behavior deve prendere il controllo se non è stato ancora eseguito.

```
public boolean takeControl() {
         return (!done);
        }
```

Il metodo *suppress* risulta vuoto, in quanto il metodo non viene soppresso ma termina in maniera naturale la sua esecuzione.
Il metodo *action* inizia con un controllo sull'angolo a cui si trova inizialmente il braccio per decidere in quale senso farlo girare, dato che farlo girare sempre in uno dei due sensi avrebbe creato dei problemi con gli angoli che dopo essere arrivati a 359 ricominciano da 0 e con i cavi che creano alcuni impedimenti in tal senso.
In base al controllo precedente, il metodo imposta la direzione, la velocità di rotazione del motore e manda il programma in un loop vuoto finché è stato raggiunto l'angolo desiderato oppure è stato premuto il pulsante di uscita dal programma *Button.ESCAPE*.
Infine imposta la variabile *done* = *true* e blocca il motore.

```
public void action() {
         if (compass.getDegrees() < degrees) {
                 Motor.C.backward();
                 Motor.C.setSpeed(500);
                 while (compass.getDegrees() < degrees &&
                    !Button.ESCAPE.isPressed()) {}
         }
        else {
                 if (compass.getDegrees() > degrees) {
                          Motor.C.forward();
                          Motor.C.setSpeed(500);
                          while (compass.getDegrees() > degrees+5 &&
                            !Button.ESCAPE.isPressed()) {}
                 }
         }
         done = true;
         Motor.C.stop();
}
```

Ecco il codice completo della classe:

```
import lejos.robotics.subsumption.*;
import lejos.nxt.*;
import lejos.nxt.addon.CompassSensor;

public class Rotate implements Behavior{
         private float degrees;
         private boolean done;
         private CompassSensor compass;

         public Rotate(float gradi, CompassSensor sensor) {
                 degrees = gradi;
                 compass = sensor;
                 done = false;
         }

         public boolean takeControl() {
                 return (!done);
         }

         public void suppress() {}

         public void action() {
                 if (compass.getDegrees() < degrees) {
                          Motor.C.backward();
                          Motor.C.setSpeed(500);
                          while (compass.getDegrees() < degrees &&
                            !Button.ESCAPE.isPressed()) {}
                 }
                 else {
…
```

```
…
                    if (compass.getDegrees() > degrees) {
                        Motor.C.forward();
                        Motor.C.setSpeed(500);
                        while (compass.getDegrees() > degrees+5 &&
                            !Button.ESCAPE.isPressed()) {}
                    }
                }
                done = true;
                Motor.C.stop();
        }
}
```

La classe *goUpDown* ha il compito di abbassare e alzare il braccio meccanico.
Innanzitutto le variabili create sono quattro:

- La variabile *to* di tipo *int* che memorizza la distanza da raggiungere
- La variabile *verse* di tipo *int* in cui viene memorizzato se il braccio deve alzarsi o abbassarsi; 1 indica che deve abbassarsi, 2 che deve alzarsi
- La variabile *done*
- La variabile *sensor* di tipo *UltrasonicSensor* che riceve il sensore di distanza

```
private int to;
private int verse;
private boolean done;
private UltrasonicSensor sensor;
```

Il creatore riceve i parametri di ingresso e inizializza le variabili:

```
public GoUpDown(UltrasonicSensor distance, int cm, int direction) {
        done = false;
        sensor = distance;
        to = cm;
        verse = direction;
}
```

I metodi *takeControl* e *suppress* sono identici alla classe *Rotate*.

Il metodo *action* rimane, in base alla direzione impostata, avvia il motore e lo lascia girare fino a quando non ha raggiunto la distanza desiderata, oppure è stato premuto il tasto *Button.ESCAPE;* infine imposta la variabile *done* = *true* e ferma il motore.

```
public void action() {
          if (verse == 1) {
                  Motor.B.backward();
                  Motor.B.setSpeed(500);
                  while (sensor.getDistance() > to &&
                     !Button.ESCAPE.isPressed()) {}
          }
          else {
                  Motor.B.forward();
                  Motor.B.setSpeed(500);
                  while (sensor.getDistance() < to &&
                     !Button.ESCAPE.isPressed()) {}
          }
          done = true;
          Motor.B.stop();
}
```

Conclude il codice completo della classe:

```
import lejos.robotics.subsumption.*;
import lejos.nxt.*;
import lejos.nxt.addon.*;

public class GoUpDown implements Behavior{
         private int to
         private int verse;
         private boolean done;
         private UltrasonicSensor sensor;

          public GoUpDown(UltrasonicSensor distance, int cm, int
              direction) {
                  done = false;
                  sensor = distance;
                  to = cm;
                  verse = direction;
         }

         public boolean takeControl() {
                  return (!done);
         }

         public void suppress() {}
…
```

```
…
        public void action() {
                if (verse == 1) {
                        Motor.B.backward();
                        Motor.B.setSpeed(500);
                        while (sensor.getDistance() > to &&
                            !Button.ESCAPE.isPressed()) {}
                }
                else {
                        Motor.B.forward();
                        Motor.B.setSpeed(500);
                        while (sensor.getDistance() < to &&
                            !Button.ESCAPE.isPressed()) {}
                }
                done = true;
                Motor.B.stop();
        }
}
```

Per effettuare la rotazione che porta la palla raccolta nel posto indicato dalle specifiche non si è utilizzata la classe *Rotate*, ma si è deciso di avere la classe *BringBack*. Ciò essenzialmente perché in questo caso non è deciso dall'utente l'angolo di destinazione, bensì dipende dalla luminosità rilevata dal relativo sensore riguardo al colore della palla.
Tale classe inizia con la creazione delle variabili necessarie al funzionamento del behavior, che sono:

- La variabile *compass* di tipo *CompassSensor* che memorizza l'oggetto sensore di campo magnetico terrestre
- La variabile *color* di tipo *LightSensor* che memorizza l'oggetto sensore di luminosità
- La variabile *done*

```
private CompassSensor compass;
private LightSensor color;
private boolean done;
```

Il creatore riceve i due sensori necessari alle rilevazioni e inizializza le rispettiva variabili, *takeControl* verifica il valore della variabile *done*.

```
public BringBack(CompassSensor sensor, LightSensor sensor1) {
          compass = sensor;
          color = sensor1;
          done = false;
}
         public boolean takeControl() {
                  return (!done);
         }

         public void suppress() {}
```

L'*if* presente all'inizio del metodo *action* controlla il livello di luminosità della palla, sperimentalmente con la pallina utilizzata si è trovato che tenendo come soglia il valore 360 si può distinguere il fatto che sia rossa (>360) o blu (<360).
A seconda del valore trovato il braccio ruoterà per portarsi nella posizione adatta utilizzando il sensore di campo magnetico, poi imposterà *done* = *true* è fermerà il motore.

```
public void action() {
          if (color.getNormalizedLightValue() > 360) {
                  Motor.C.forward();
                  Motor.C.setSpeed(500);
                  while (compass.getDegrees() > 50 &&
                      !Button.ESCAPE.isPressed()) {}
          }
          else {
                  Motor.C.backward();
                  Motor.C.setSpeed(500);
                  while (compass.getDegrees() < 250 &&
                      !Button.ESCAPE.isPressed()) {}
          }
          done = true;
          Motor.C.stop();
}
```

Interamente il codice della classe è così:

```
import lejos.nxt.*;
import lejos.nxt.addon.*;

public class BringBack implements Behavior{
	private CompassSensor compass;
	private LightSensor color;
	private boolean done;

	public BringBack(CompassSensor sensor, LightSensor sensor1) {
		compass = sensor;
		color = sensor1;
		done = false;
	}

	public boolean takeControl() {
		return (!done);
	}

	public void suppress() {}

	public void action() {
		if (color.getNormalizedLightValue() > 360) {
		    Motor.C.forward();
		    Motor.C.setSpeed(500);
		    while (compass.getDegrees() > 50 &&
		        !Button.ESCAPE.isPressed()) {}
		}
		else {
		    Motor.C.backward();
		    Motor.C.setSpeed(500);
		    while (compass.getDegrees() < 250 &&
		        !Button.ESCAPE.isPressed()) {}
		}
		done = true;
		Motor.C.stop();
	}
}
```

Il behavior che apre e chiude la mano è stato denominato *Hand*. Questa classe ha solo due variabili:

- La variabile verse di tipo *int* che memorizza se la mano deve aprirsi o chiudersi
- La variabile *done*

Il creatore riceve come parametro l'intero che indica il senso di rotazione del motore e imposta di conseguenza la variabile *done*, *takeControl* controlla se il behavior è già stato eseguito e *suppress* non esegue alcuna istruzione.

```
public Hand(int direction) {
          done = false;
          verse = direction;
         }

public boolean takeControl() {
         return (!done);
}

public void suppress() {
}
```

Il metodo *action* è leggermente diverso da quello delle altre classi: anche questo inizia con un *if* che controlla il valore della costante *VERSE* per impostare la rotazione del motore (1 per l'apertura della mano) ma, non avendo disponibile alcun sensore che controlli se la mano è aperta o chiusa, si è scelto di impostare che il motore giri per il tempo necessario affinché ciò avvenga, tempo che è stato valutato in maniera empirica e che risulta di circa 300 millisecondi alla velocità predefinita dei metodi *forward* e *backward.*

```
public void action() {
          if (verse == 1)
                  Motor.A.forward();
          else
                  Motor.A.backward();
          long start = System.currentTimeMillis();
          while (System.currentTimeMillis() < (start + 300) &&
              !Button.ESCAPE.isPressed()) {}
          done = true;
          Motor.A.stop();
}
```

La classe completa risulta come segue.

```
import lejos.robotics.subsumption.*;
import lejos.nxt.*;

public class Hand implements Behavior{
         private boolean done;
         private int verse;
}
…
```

```
…
        public Hand(int direction) {
                done = false;
                verse = direction;
        }

        public void suppress() {}

        public boolean takeControl() {
                return (!done);

         public void action() {
                if (VERSE == 1)
                        Motor.A.forward();
                else
                        Motor.A.backward();
                long start = System.currentTimeMillis();
                while (System.currentTimeMillis() < (start + 300) &&
                    !Button.ESCAPE.isPressed()) {}
                done = true;
                Motor.A.stop();
        }
}
```

L'ultimo behavior da analizzare è quello che viene eseguito quando tutti gli altri comportamenti sono già avvenuti e cioè quello appartenente alla classe che è stata denominata *End*.
Molto semplicemente il metodo *takeControl* di questa classe ritorna sempre *true* e il metodo *action* effettua la chiamata a *System.exit(0)*, terminando quindi l'esecuzione del programma.

```
import lejos.robotics.subsumption.*;
import lejos.nxt.*;

public class End implements Behavior{

        public boolean takeControl() {
                return true ;
        }

        public void suppress() {}
…
```

```
…
        public void action() {
                System.exit(0);
        }
}
```

Il passo finale nella stesura del codice consiste nella classe *RobotArm*, costituita dal solo metodo *main* dal quale sarà avviato l'arbitrator.
La prima cosa che tale metodo deve effettuare è creare i tre oggetti per i sensori utilizzati dal braccio meccanico.

```
CompassSensor compass = new CompassSensor(SensorPort.S2);
LightSensor color = new LightSensor(SensorPort.S1);
UltrasonicSensor distance = new UltrasonicSensor(SensorPort.S3);
```

Successivamente vengono inizializzate tutte le istanze dei behavior da inserire nell'array da passare all'arbitrator:

1. un'istanza della classe *Esci*

```
Esci exit = new Esci();
```

2. un'istanza della classe *Rotate* passando al costruttore l'angolo di 150° e il *CompassSensor*

```
Rotate rotation = new Rotate(150, compass);
```

3. due istanze della classe *GoUpDown* passando al costruttore l'*UltrasonicSensor*, la distanza dal suolo a cui portarsi e il verso di rotazione 1 (abbassamento); una verrà utilizzata per abbassarsi nel momento in cui deve essere raccolta la palla, l'altra nel momento in cui deve essere depositata

```
GoUpDown down1 = new GoUpDown(distance, 7, 1);
GoUpDown down2 = new GoUpDown(distance, 7, 1);
```

4. due istanze della classe *Hand*, una con verso di rotazione 2 per aprire la mano robotica e l'altra con verso 1 per chiuderla

```
Hand prendi = new Hand(2);
Hand rilascia = new Hand(1);
```

5. un'istanza di *BringBack* passando al costruttore i due sensori di campo magnetico e di luminosità

```
BringBack riporta = new BringBack(compass, color);
```

6. altre due istanze di *GoUpDown* sempre passando al costruttore il sensore di distanza ad ultrasioni; però impostando questa volta il verso di rotazione 2 (elevamento) e la distanza dal suolo di 23 centimetri

```
GoUpDown up1 = new GoUpDown(distance, 23, 2);
GoUpDown up2 = new GoUpDown(distance, 23, 2);
```

7. Un'istanza della classe *End*

```
End fine = new End();
```

A questo punto tutti gli oggetti behavior devono essere inseriti nell'array *b* da passare all'arbitrator; da notare che l'oggetto di tipo *Esci* deve avere la priorità maggiore in quanto, se viene premuto il pulsante per l'uscita, deve avere la precedenza su tutti gli altri behavior, mentre quello di tipo *End* deve avere priorità minore in quanto può sempre essere mandato in esecuzione, ma deve gli deve essere effettivamente passato il controllo solo se tutti gli altri movimenti sono già stati effettuati. Gli altri behavior sono disposti in ordine di priorità decrescente a seconda del posto nella sequenza dell'azione.

```
Behavior[] b = {fine, up2, rilascia, down2, riporta, up1, prendi, down1,
    rotation, exit};
```

Possiamo ora terminare con il codice completo per la classe *RobotArm*:

```
import lejos.robotics.subsumption.*;
import lejos.nxt.*;
import lejos.nxt.addon.*;

public class RobotArm {
        public static void main(String[] args) {
                CompassSensor compass = new
                   CompassSensor(SensorPort.S2);
                LightSensor color = new LightSensor(SensorPort.S1);
                UltrasonicSensor distance = new
                   UltrasonicSensor(SensorPort.S3);
                Esci exit = new Esci();
                Rotate rotation = new Rotate(150, compass);
                GoUpDown down1 = new GoUpDown(distance, 7, 1);
                GoUpDown down2 = new GoUpDown(distance, 7, 1);
…
```

```
…
                Hand prendi = new Hand(2);
                Hand rilascia = new Hand(1);
                BringBack riporta = new BringBack(compass, color);
                GoUpDown up1 = new GoUpDown(distance, 23, 2);
                GoUpDown up2 = new GoUpDown(distance, 23, 2);
                End fine = new End();

                Behavior[] b = {fine, up2, rilascia, down2, riporta,
                    up1, prendi, down1, rotation, exit};
                Arbitrator arby = new Arbitrator(b);
                arby.start();
        }
}
```

5.3 Il braccio meccanico fuzzy

5.3.1 Progettazione

Prima di stendere il codice è necessaria una fase di progettazione per determinare le variabili linguistiche necessarie, i conseguenti insiemi fuzzy e l'insieme delle regole che determinano il comportamento del braccio meccanico.
Visto che le varie azioni compiute sono indipendenti una dall'altra, analizziamole separatamente una per volta.
Cominciamo con la fase di rotazione che deve posizionare il braccio meccanico sopra la pallina. Le variabili linguistiche utilizzate sono:

1. *DegreeL* che rappresenta quanto lontano è il braccio dalla posizione di destinazione nel caso esso si trovi inizialmente ad un angolo minore; essa è suddivisa in tre insiemi fuzzy:
   - *farL*
   - *mediumL*
   - *nearL*
2. *DegreeR* che ha lo stesso utilizzo della variabile precedente ma nel caso in cui si il braccio si trovi inizialmente ad un angolazione maggiore:
   - *farR*
   - *mediumR*
   - *nearR*
3. *MotorC* per rappresentare la velocità del motore che regola la rotazione, suddivisa in:
   - *slow*
   - *medium*
   - *fast*
4. *Time* che indica il tempo trascorso da quando è cominciata la rotazione:
   - *veryEarly*
   - *early*
   - *late*

Come si può notare dall'inserimento della variabile linguistica *Time*, diversamente dall'esempio del line follower, si è scelto in questo caso di far dipendere ogni movimento del braccio da due input, dei quali uno è sempre il tempo trascorso dall'inizio del movimento. Ciò per creare un effetto di accelerazione smooth (che con il metodo *SmoothAcceleration* è quasi irrilevante).
La tabella quindi riassume le condizioni di funzionamento per questa prima parte.
Bisogna chiarire che sia per la rotazione in un verso che per quella nell'altro sono usati sempre valori di velocità positivi; basterà poi usare uno dei due metodi per impostare il senso di rotazione dei motori per causare il movimento del braccio in una direzione o nell'altra.

Degrees L/R	**Time**	**MotorC**
farL	veryEarly	slow
	early	medium
	late	fast
mediumL	veryEarly	slow
	early	medium
	late	medium
nearL	veryEarly	slow
	early	slow
	late	slow
nearR	veryEarly	slow
	early	slow
	late	slow
mediumR	veryEarly	slow
	early	medium
	late	medium
farD	veryEarly	slow
	early	medium
	late	fast

Dalla tabella si possono ricavare le regole fuzzy, elenchiamo prima quelle del caso in cui il braccio si trovi ad un angolazione minore:

- $if\ \boldsymbol{DegreeL}\ is\ \boldsymbol{farL}\ and\ \boldsymbol{Time}\ is\ \boldsymbol{Late}$
  $then\ \boldsymbol{MotorC}\ is\ \boldsymbol{fast}$
- $if\ (\boldsymbol{Degreel}\ is\ \boldsymbol{farL}\ and\ \boldsymbol{Time}\ is\ \boldsymbol{Early})\ or$
  $(\boldsymbol{Degree}\ is\ \boldsymbol{mediumL}\ and\ \boldsymbol{Time}\ is\ \boldsymbol{Early})\ or$
  $(\boldsymbol{DegreeL}\ is\ \boldsymbol{mediumL}\ and\ \boldsymbol{Time}\ is\ \boldsymbol{Late})$
  $then\ \boldsymbol{MotorC}\ is\ \boldsymbol{medium}$
- $if\ \boldsymbol{DegreeL}\ is\ \boldsymbol{nearL}\ or\ \boldsymbol{Time}\ is\ \boldsymbol{veryEarly}$
  $then\ \boldsymbol{MotorC}\ is\ \boldsymbol{slow}$

Nel caso in cui l'arto meccanico si trovi ad un angolazione maggiore le regole sono speculari:

- $if\ \boldsymbol{DegreeR}\ is\ \boldsymbol{farR}\ and\ \boldsymbol{Time}\ is\ \boldsymbol{Late}$
  $then\ \boldsymbol{MotorC}\ is\ \boldsymbol{fast}$
- $if\ (\boldsymbol{DegreeR}\ is\ \boldsymbol{farR}\ and\ \boldsymbol{Time}\ is\ \boldsymbol{Early})\ or$
  $(\boldsymbol{DegreeR}\ is\ \boldsymbol{mediumR}\ and\ \boldsymbol{Time}\ is\ \boldsymbol{Early})\ or$
  $(\boldsymbol{DegreeR}\ is\ \boldsymbol{mediumR}\ and\ \boldsymbol{Time}\ is\ \boldsymbol{Late})$
  $then\ \boldsymbol{MotorC}\ is\ \boldsymbol{medium}$
- $if\ \boldsymbol{DegreeR}\ is\ \boldsymbol{nearR}\ or\ \boldsymbol{Time}\ is\ \boldsymbol{veryEarly}$
  $then\ \boldsymbol{MotorC}\ is\ \boldsymbol{slow}$

Come per la realizzazione a behavior, l'angolo in cui è posizionata pallina è di 150°. A riguardo c'è da dire che entrambe le funzioni di appartenenza degli insiemi fuzzy *nearL* e *nearR* oltrepassano tale numero di 5°, questo perché, impostando la gradazione esatta, spesso il motore arrivava quasi alla destinazione senza la potenza necessaria a fargli fare l'ultimo spostamento per arrivare a 150° (probabilmente a causa dell'attrito dovuto al peso della struttura). Così invece, tentando di andare oltre, la potenza è sufficiente, poi ci penserà il controllo di ingresso al ciclo a farlo fermare effettivamente all'angolazione richiesta.

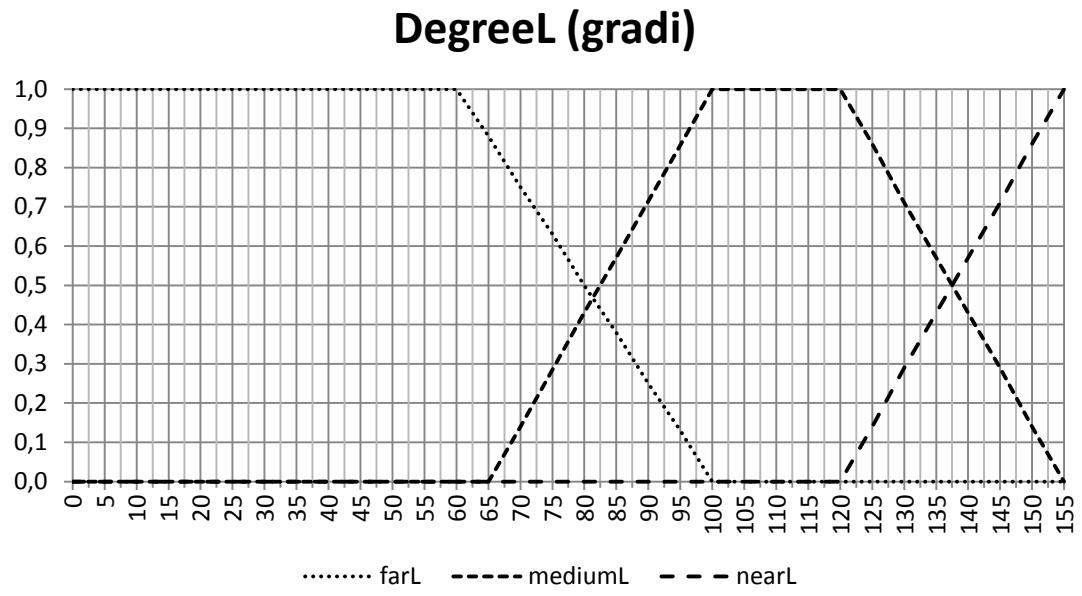

Figura 5.4: Funzioni di appartenenza variabile DegreeL (parte sinistra)

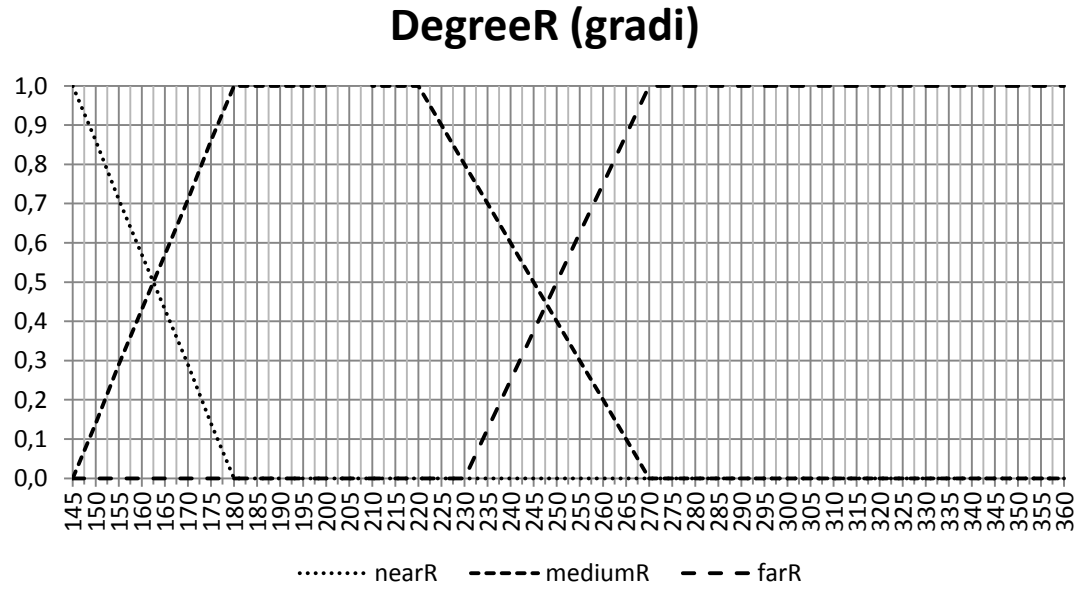

Figura 5.5: Funzioni di appartenenza variabile DegreeR (parte destra)

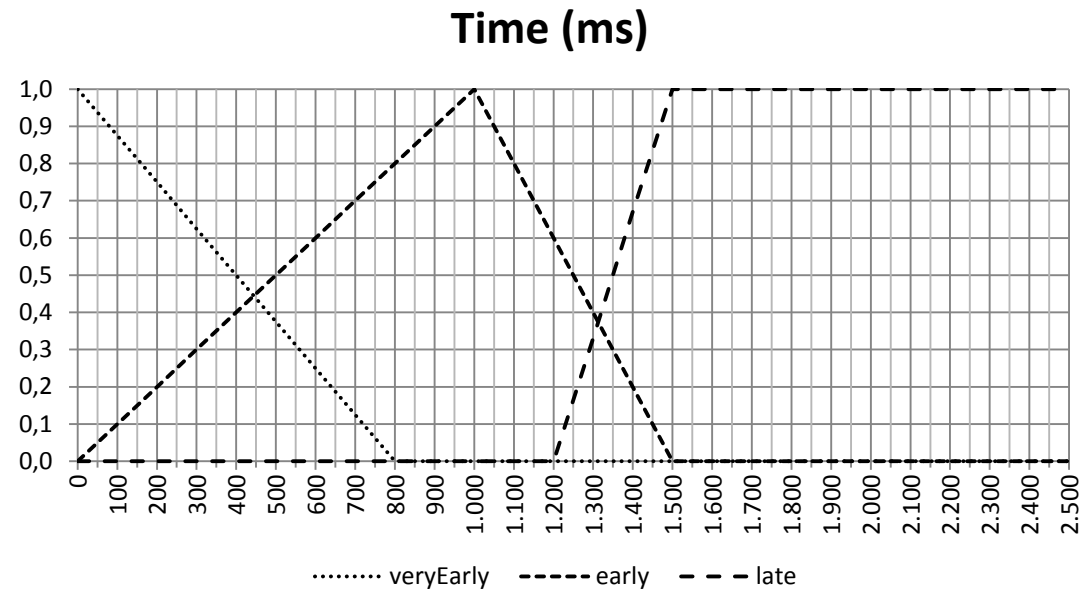

Figura 5.6: funzioni di appartenenza variabile Time

La funzione di appartenenza dell'insieme fuzzy *fast* arriva fino a 1200 volutamente, anche se il limite teorico della velocità è 900, per fare in modo che durante la fuzzificazione sia utilizzato 900 come centro della funzione. Lo stesso principio vale per *slow*, il cui centro deve essere 0.

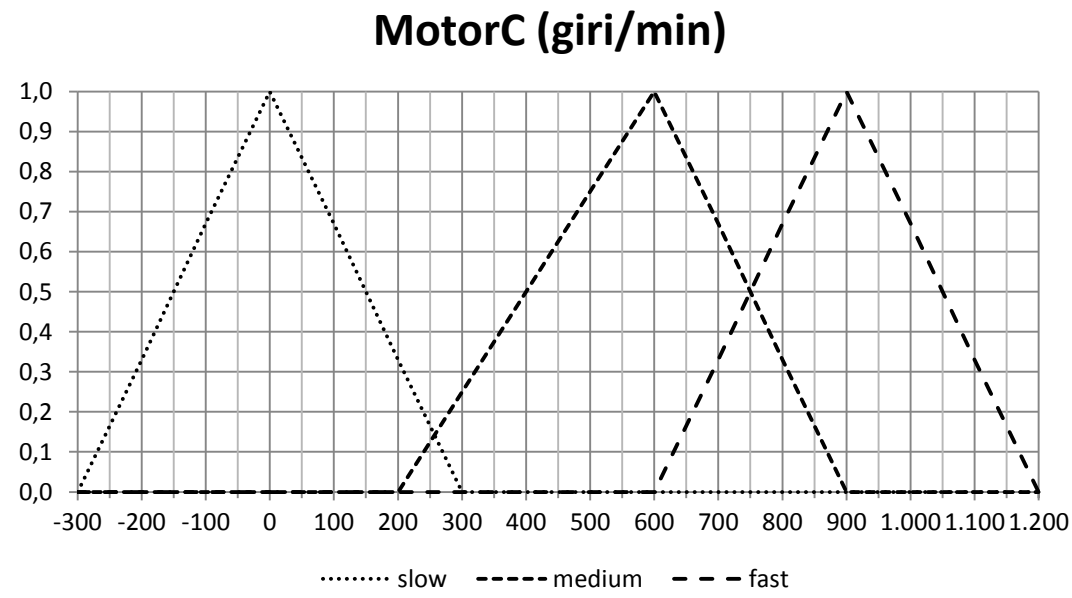

Figura 5.7: funzioni di appartenenza variabile MotorC

Analizziamo ora la fase di abbassamento. Come si può intuire, in realtà tutte le fasi sono abbastanza simili tra loro dal punto di vista delle regole, quelle che variano sono le variabili linguistiche e le relativa funzioni di appartenenza.
Le variabili linguistiche necessarie per far abbassare il braccio meccanico sono:

1. *Height* che rappresenta la distanza dal suolo misurata dal sensore di distanza ad ultrasuoni, divisa in:
   - *farDown*
   - *mediumDown*
   - *nearDown*
2. *Time* che ha la stessa valenza della fase precedente ma del quale sono stati modificati gli insiemi fuzzy, in quanto cambia la distanza da percorrere:
   - *veryEarlyUD*
   - *earlyUD*
   - *lateUD*
3. *MotorB* le cui funzioni di appartenenza risultato identiche a quelle per la rotazione

La tabella che riassume le dipendenze tra le variabili è la seguente.

Height	**Tempo**	**MotorB**
farDown	veryEarlyUD	slow
	earlyUD	medium
	lateUD	fast
mediumDown	veryEarlyUD	slow
	earlyUD	medium
	lateUD	medium
nearDown	veryEarlyUD	slow
	earlyUD	slow
	lateUD	slow

Le regole fuzzy per controllare la discesa del braccio sono quindi strutturate così:

- $if\ \boldsymbol{Height}\ is\ \boldsymbol{farDown}\ and\ \boldsymbol{Time}\ is\ \boldsymbol{LateUD}$
  $then\ \boldsymbol{MotorB}\ is\ \boldsymbol{fast}$
- $if\ (\boldsymbol{Height}\ is\ \boldsymbol{farDown}\ and\ \boldsymbol{Tempo}\ is\ \boldsymbol{EarlyUD})\ or$
  $(\boldsymbol{Height}\ is\ \boldsymbol{mediumDown}\ and\ \boldsymbol{Time}\ is\ \boldsymbol{EarlyUD})\ or$
  $(\boldsymbol{Height}\ is\ \boldsymbol{mediumDown}\ and\ \boldsymbol{Time}\ is\ \boldsymbol{LateUD})$
  $then\ \boldsymbol{MotorB}\ is\ \boldsymbol{medium}$
- $if\ \boldsymbol{Height}\ is\ \boldsymbol{nearDown}\ or\ \boldsymbol{Time}\ is\ \boldsymbol{veryEarlyUD}$
  $then\ \boldsymbol{MotorB}\ is\ \boldsymbol{slow}$

Sono forniti di seguito i grafici delle nuove funzioni di appartenenza.

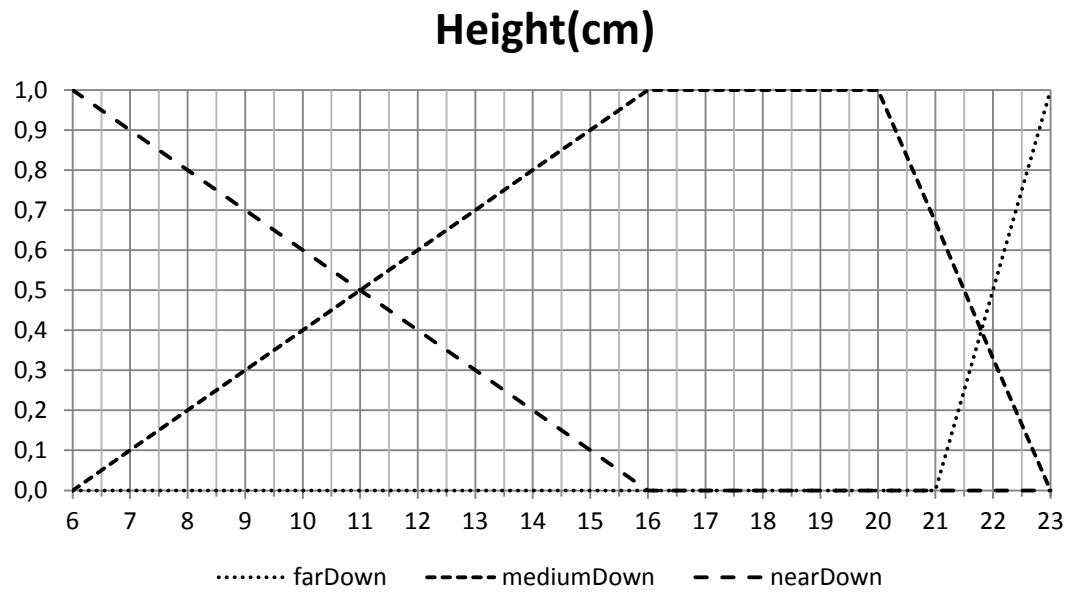

Figura 5.8: funzioni di appartenenza variabile Height

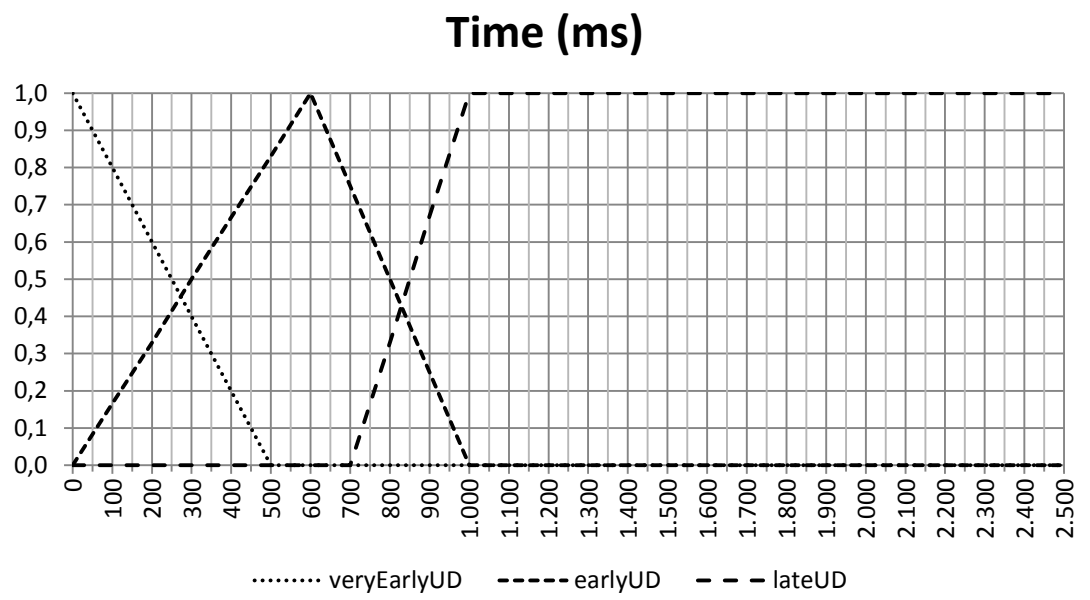

Figura 5.9: funzioni di appartenenza variabile Time

Per quanto riguarda l'elevamento, le variabili linguistiche sono le stesse della fase di abbassamento, le differenze stanno nel fatto che l'insieme fuzzy *slow* è stato sostituito da *slowUp* e gli insiemi riguardanti l'altezza sono stati sostituiti da:

- *farUp*
- *mediumUp*
- *nearUp*

Ovviamente la tabella e le regole fuzzy sono stati modificati visto che in il movimento è inverso al precedente.

Height	Tempo	MotorB
farUp	veryEarlyUD	slowUp
	earlyUD	slowUp
	lateUD	slowUp
mediumUp	veryEarlyUD	slowUp
	earlyUD	medium
	lateUD	medium
nearUp	veryEarlyUD	slowUp
	earlyUD	medium
	lateUD	fast

Regole fuzzy:

- if $\boldsymbol{Height}$ is $\boldsymbol{nearUp}$ and $\boldsymbol{Time}$ is $\boldsymbol{Late}$
  $then$ $\boldsymbol{MotorB}$ is $\boldsymbol{fast}$
- if $(\boldsymbol{Height}$ is $\boldsymbol{nearUp}$ and $\boldsymbol{Time}$ is $\boldsymbol{EarlyUD})$ or
  $(\boldsymbol{Height}$ is $\boldsymbol{mediumUp}$ and $\boldsymbol{Time}$ is $\boldsymbol{EarlyUD})$ or
  $(\boldsymbol{Height}$ is $\boldsymbol{mediumUp}$ and $\boldsymbol{Time}$ is $\boldsymbol{LateUD})$
  $then$ $\boldsymbol{MotorB}$ is $\boldsymbol{medium}$
- if $\boldsymbol{Height}$ is $\boldsymbol{farUp}$ or $\boldsymbol{Time}$ is $\boldsymbol{veryEarlyUD}$
  $then$ $\boldsymbol{MotorB}$ is $\boldsymbol{slowUp}$

Grafici delle nuove funzioni.

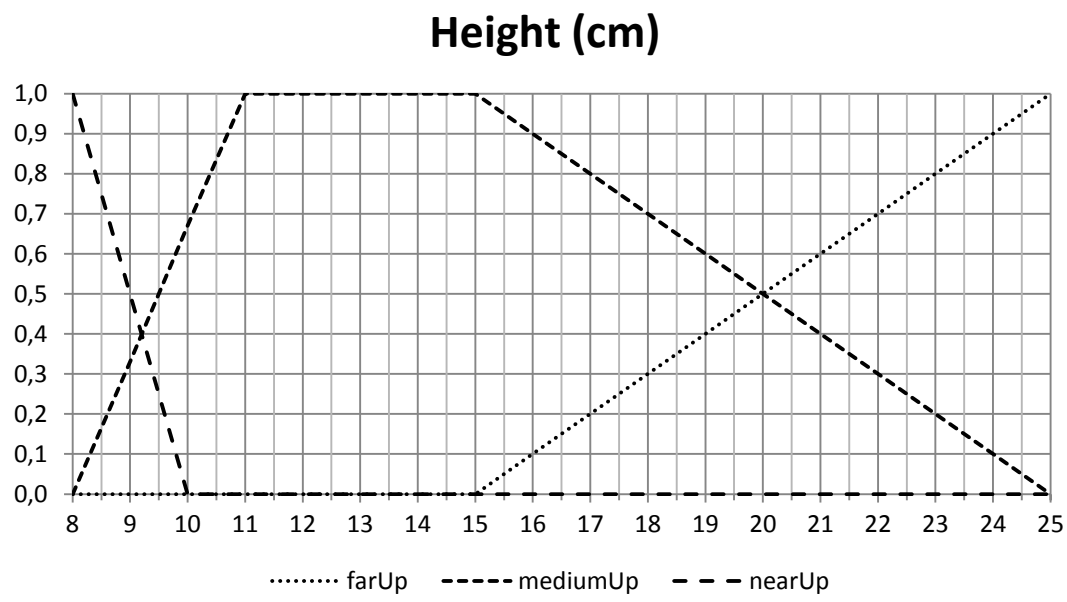

Figura 5.10: funzioni di appartenenza variabile Height (elevamento)

La funzione di appartenenza dell'insieme fuzzy *slowUp* è stata modificata in modo che il massimo sia posto in corrispondenza di 50 giri/min, questo perché durante la fase di elevamento il braccio si muove contro la forza di gravità e quindi se la velocità impostata è troppo ridotta, esso non riesce a raggiungere l'altezza fissata come destinazione.

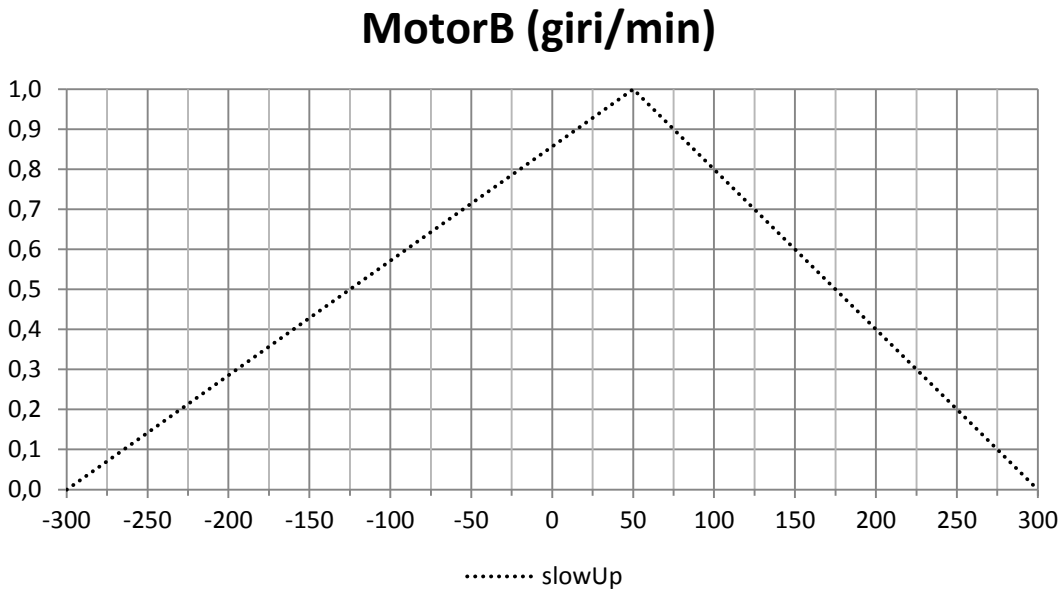

Figura 5.11: funzione di appartenenza insieme fuzzy slowUp

L'ultima fase rimasta è quella in cui il braccio, dopo avere già raccolto la palla, ruota per portarsi nella locazione di destinazione dipendente dal colore della pallina stessa.
Qui gli insiemi fuzzy della variabile Degree saranno suddivisi in due gruppi. Il primo gruppo nel caso la destinazione sia a 50° (pallina rossa):

- *nearRed*
- *mediumRed*
- *farRed*

Il secondo gruppo se la destinazione è a 250° (pallina blu):

- nearBlue
- mediumBlue
- farBlue

Infatti nel codice sarà presente un *if* che, in base al colore della palla, avvierà uno dei due comportamenti. Ormai non sono necessarie ulteriori spiegazioni, di seguito sono presenti le tabelle, le regole fuzzy e le funzioni di appartenenza per i due casi.

Palla di colore rosso
Tabella:

Degree (Red)	Time	MotorC
nearRed	veryEarly	slow
	early	slow
	late	slow
mediumRed	veryEarly	slow
	early	medium
	late	medium
farRed	veryEarly	slow
	early	medium
	late	fast

Regole fuzzy:

- $if\ \boldsymbol{Degree}\ is\ \boldsymbol{farRed}\ and\ \boldsymbol{Time}\ is\ \boldsymbol{Late}$
  $then\ \boldsymbol{MotorC}\ is\ \boldsymbol{fast}$
- $if\ (\boldsymbol{Degree}\ is\ \boldsymbol{farRed}\ and\ \boldsymbol{Time}\ is\ \boldsymbol{Early})\ or$
  $(\boldsymbol{Degree}\ is\ \boldsymbol{mediumRed}\ and\ \boldsymbol{Time}\ is\ \boldsymbol{Early})\ or$
  $(\boldsymbol{Degree}\ is\ \boldsymbol{mediumRed}\ and\ \boldsymbol{Time}\ is\ \boldsymbol{Late})$
  $then\ \boldsymbol{MotorB}\ is\ \boldsymbol{medium}$
- $if\ \boldsymbol{Degree}\ is\ \boldsymbol{nearRed}\ or\ \boldsymbol{Time}\ is\ \boldsymbol{veryEarly}$
  $then\ \boldsymbol{MotorC}\ is\ \boldsymbol{slow}$

Grafico delle funzioni di appartenenza:

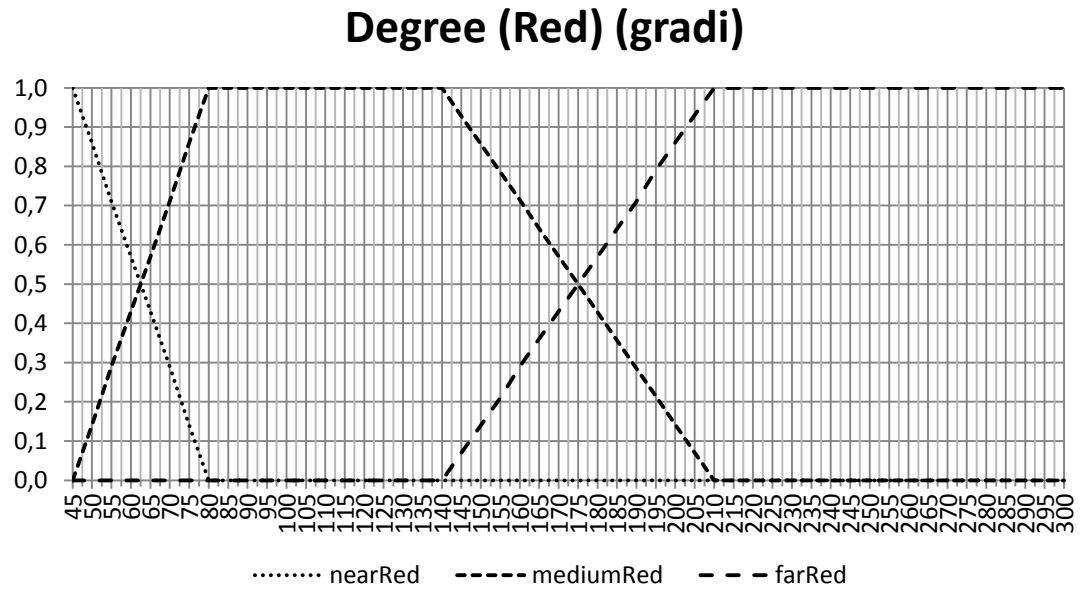

Figura 5.12: funzioni di appartenenza variabile Degree (palla rossa)

Palla di colore blu

Tabella:

Degree (Blue)	Time	MotorC
farBlue	veryEarly	slow
	early	medium
	late	fast
mediumBlue	veryEarly	slow
	early	medium
	late	medium
nearBlue	veryEarly	slow
	early	slow
	late	slow

Regole fuzzy:

- if $\boldsymbol{Degree}$ is $\boldsymbol{farBlue}$ and $\boldsymbol{Time}$ is $\boldsymbol{Late}$
  $then$ $\boldsymbol{MotorC}$ is $\boldsymbol{fast}$
- if $(\boldsymbol{Degree}$ is $\boldsymbol{farBlue}$ and $\boldsymbol{Time}$ is $\boldsymbol{Early})$ or
  $(\boldsymbol{Degree}$ is $\boldsymbol{mediumBlue}$ and $\boldsymbol{Time}$ is $\boldsymbol{Early})$ or
  $(\boldsymbol{Degree}$ is $\boldsymbol{mediumBlue}$ and $\boldsymbol{Time}$ is $\boldsymbol{Late})$
  $then$ $\boldsymbol{MotorB}$ is $\boldsymbol{medium}$
- if $\boldsymbol{Degree}$ is $\boldsymbol{nearBlue}$ or $\boldsymbol{Time}$ is $\boldsymbol{veryEarly}$
  $then$ $\boldsymbol{MotorC}$ is $\boldsymbol{slow}$

Grafico delle funzioni di appartenenza:

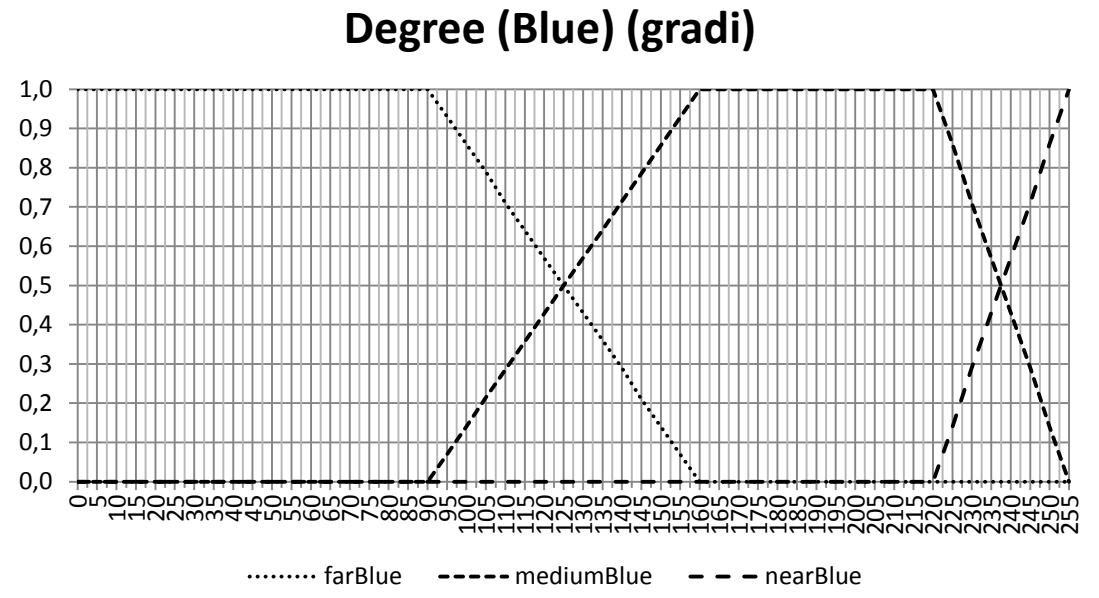

Figura 5.13: funzioni di appartenenza variabile Degree (palla blu)

5.3.2 Realizzazione

Come prima cosa è stata creata la classe *FuzzyArm,* che contiene il metodo *main* principale dell'applicazione. Essa inoltre ha una costante di classe *SOGLIA* che memorizza il valore limite per distinguere la palla di colore blu da quella di colore rosso.

```
public class FuzzyArmProva {

      public static final int SOGLIA = 360;

      public static void main(String[] args) {
      …
      }
}
```

Il metodo *main* comincia inizializzando gli oggetti sensori, inoltre crea anche un oggetto di tipo *Stopwatch* che verrà utilizzato per cronometrare il tempo trascorso per ogni azione.

```
LightSensor color = new LightSensor(SensorPort.S1, true);
CompassSensor compass = new CompassSensor(SensorPort.S2);
UltrasonicSensor sonar = new UltrasonicSensor(SensorPort.S3);
Stopwatch timer = new Stopwatch();
```

Prima di cominciare a creare tutte le funzioni di appartenenza che sono state illustrate nel paragrafo precedente impostiamo l'accelerazione smooth per il motore A, visto che è quello che controlla la mano e non c'è nessun sensore che ne manovrerà l'accelerazione: infatti il movimento che la apre e la chiude sarà gestito nello stesso modo dell'approccio a behavior. Inoltre attiviamo la regolazione della velocità per gli altri due motori.

```
Motor.A.smoothAcceleration(true);
Motor.C.regulateSpeed(true);
Motor.B.regulateSpeed(true);
```

L'ultimo passo prima dell'avvio della procedura di raccolta e spostamento della pallina è appunto l'inizializzazione di tutti gli oggetti *MembershipFunction* necessari alla realizzazione del comportamento desiderato.

```
MembershipFunction farL = new MembershipFunction(0, 0, 60, 100,
1);
MembershipFunction mediumL = new MembershipFunction(65, 100, 120,
155, 1);
MembershipFunction nearL = new MembershipFunction(120, 155, 155,
155, 1);


MembershipFunction nearR = new MembershipFunction(145, 145, 145,
180, 1);
MembershipFunction mediumR = new MembershipFunction(145, 180,
220, 270, 1);
MembershipFunction farR = new MembershipFunction(230, 270, 360,
360, 1);


MembershipFunction slowUp = new MembershipFunction(-300, 50, 50,
300, 1);
MembershipFunction slow = new MembershipFunction(-300, 0, 0,
300, 1);
MembershipFunction medium = new MembershipFunction(200, 500, 500,
900, 1);
MembershipFunction fast = new MembershipFunction(600, 900, 900,
1200, 1);


MembershipFunction farDown = new MembershipFunction(21, 23, 23,
23, 1);
MembershipFunction mediumDown = new MembershipFunction(6, 16, 20,
23, 1);
MembershipFunction nearDown = new MembershipFunction(6, 6, 6, 16,
1);


MembershipFunction farUp = new MembershipFunction(15, 25, 25, 25,
1);
MembershipFunction mediumUp = new MembershipFunction(8, 11, 15,
25, 1);
MembershipFunction nearUp = new MembershipFunction(8, 8, 8, 10,
1);


MembershipFunction nearRed = new MembershipFunction(45, 45, 45,
80, 1);
MembershipFunction mediumRed = new MembershipFunction(45, 80,
140, 210, 1);
MembershipFunction farRed = new MembershipFunction(140, 210, 300,
300, 1);
…
```

```
…
MembershipFunction nearBlue = new MembershipFunction(220, 255,
255, 255, 1);
MembershipFunction mediumBlue = new MembershipFunction(90, 160,
220, 255, 1);
MembershipFunction farBlue = new MembershipFunction(0, 0, 90,
160, 1);


MembershipFunction veryEarly = new MembershipFunction(0, 0, 0,
800, 1);
MembershipFunction early = new MembershipFunction(0,  1000, 1000,
1500, 1);
MembershipFunction late = new MembershipFunction(1200, 1500,
10000, 10000, 1);


MembershipFunction veryEarlyUD = new MembershipFunction(0, 0, 0,
500, 1);
MembershipFunction earlyUD = new MembershipFunction(0,  600, 600,
1000, 1);
MembershipFunction lateUD = new MembershipFunction(700, 1000,
10000, 10000, 1);
```

Passiamo ora alla prima fase di movimento, cioè la rotazione orizzontale per portarsi ad un angolo di 150° rispetto al nord. Visto che tutti gli altri movimenti saranno dal punto di vista strutturale molto simili, a parte ovviamente le differenze nelle variabili linguistiche e nelle funzioni di appartenenza utilizzate, solo questa prima fase sarà analizzata nel dettaglio, le altre poi saranno di facile comprensione anche alla luce di quanto già spiegato nel paragrafo precedente.
Innanzitutto bisogna eseguire un reset sull'oggetto *Timer* cosicché riparta da zero nel suo conteggio del tempo trascorso.

```
timer.reset();
```

A questo punto c'è la diramazione del codice dovuta alla posizione iniziale del braccio. Nel caso che l'angolo di partenza sia maggiore di 150° viene impostata la rotazione del motore in avanti, che causa una rotazione in senso antiorario dell'arto meccanico.

```
Motor.C.forward();
```

C'è ora l'ingresso nel ciclo di regolazione della velocità del motore, la cui è che l'angolo sia ancora superiore a 150° (in caso contrario significa che la destinazione è stata raggiunta) e che non sia stato premuto il pulsante *Button.ESCAPE.*

```
while (compass.getDegrees() > 150 && !Button.ESCAPE.isPressed()) {
…
 }
```

Le prime istruzioni che vengono eseguite riguardano la memorizzazione dei due input: il valore fornito dal sensore di campo magnetico terrestre e il tempo passato dalla chiamata *timer.reset()*.

```
int x = Float.valueOf(compass.getDegrees()).intValue();
int t = timer.elapsed();
```

Quindi inizia il processo di fuzzificazione in cui gli input vengono convertiti nei gradi di appartenenza alle singole funzioni.

```
double muNear = nearR.fuzzificate(x);
double muMedium = mediumR.fuzzificate(x);
double muFar = farR.fuzzificate(x);
double muEarly = early.fuzzificate(t);
double muVeryEarly = veryEarly.fuzzificate(t);
double muLate = late.fuzzificate(t);
```

La fase più intricata di questa semplice applicazione è probabilmente il calcolo dei gradi di verità delle regole tramite i connettivi logici. Analizziamo nel dettaglio la situazione.

1. La prima regola ha grado di verità dato da:

   $$\boldsymbol{DegreeR}\ is\ \boldsymbol{farR}\ and\ \boldsymbol{Time}\ is\ \boldsymbol{late}$$

   Quindi il grado di verità è dato da:

   $$\boldsymbol{Math}.\mathbf{min}(\boldsymbol{muFarR}, \boldsymbol{muLate})$$

2. La seconda regola ha grado di verità dato dall'OR tra:

   - $\boldsymbol{DegreeR}\ is\ \boldsymbol{farR}\ and\ \boldsymbol{Time}\ is\ \boldsymbol{early}$

     $\boldsymbol{Math}.\mathbf{min}(\boldsymbol{muFarR}, \boldsymbol{muEarly})$
   - $\boldsymbol{DegreeR}\ is\ \boldsymbol{mediumR}\ and\ \boldsymbol{Time}\ is\ \boldsymbol{early}$

     $\boldsymbol{Math}.\mathbf{min}(\boldsymbol{muMediumR}, \boldsymbol{muEarly})$
   - $\boldsymbol{DegreeR}\ is\ \boldsymbol{mediumR}\ and\ \boldsymbol{Time}\ is\ \boldsymbol{late}$

     $\boldsymbol{Math}.\mathbf{min}(\boldsymbol{muMediumR}, \boldsymbol{muLate})$

3. La terza regola ha grado di verità dato da:

$$\boldsymbol{DegreeR}\ \textit{is}\ \boldsymbol{nearR}\ \textit{or}\ \boldsymbol{Time}\ \textit{is}\ \boldsymbol{veryEarly}$$

$$\boldsymbol{Math}.\max(\boldsymbol{muNearR}, \boldsymbol{muVeryEarly})$$

Applicando poi i valori ottenuti per calcolare la fase di inferenza, il risultato è il seguente.

```
MembershipFunction fastC = fast.inference(Math.min(muFar,
      muLate));
MembershipFunction mediumC =
      medium.inference(Math.max(Math.max(Math.min(muFar, muEarly),
      Math.min(muMedium, muLate)), Math.min(muMedium, muEarly)));
MembershipFunction slowC = slow.inference(Math.max(muVeryEarly,
      muNear));
```

L'ultima operazione da eseguire è la defuzzificazione e l'applicazione del risultato tramite il metodo *setSpeed*.

```
int motorC = MembershipFunction.defuzCOG(slowC, mediumC, fastC);

Motor.C.setSpeed(motorC);
```

Il risultato per il frammento di codice di questa fase è il seguente.

```
timer.reset();
if (compass.getDegrees() > 150) {
      Motor.C.forward();
      while (compass.getDegrees() > 150 &&
          !Button.ESCAPE.isPressed()) {

          int x = Float.valueOf(compass.getDegrees()).intValue();
          int t = timer.elapsed();

          double muNear = nearR.fuzzificate(x);
          double muMedium = mediumR.fuzzificate(x);
          double muFar = farR.fuzzificate(x);
          double muEarly = early.fuzzificate(t);
          double muVeryEarly = veryEarly.fuzzificate(t);
          double muLate = late.fuzzificate(t);
…
```

```
…
          MembershipFunction slowC =
                slow.inference(Math.max(muVeryEarly, muNear));


          MembershipFunction mediumC =
                medium.inference(Math.max(Math.max(Math.min(muFar,
                muEarly), Math.min(muMedium, muLate)),
                Math.min(muMedium, muEarly)));
          MembershipFunction fastC =
                fast.inference(Math.min(muFar, muLate));


          int motorC = MembershipFunction.defuzCOG(slowC, mediumC,
                fastC);


          Motor.C.setSpeed(motorC);
}
```

Nel caso l'angolo di partenza il codice è speculare, come vale per tutti gli altri casi, cambieranno solo le funzioni di appartenenza utilizzate di volta in volta e le regole, mantenendo però la stessa struttura.
Quindi è presentato direttamente il codice completo per la classe *FuzzyArm.*

```
import lejos.nxt.*;
import lejos.nxt.addon.*;
import lejos.util.Stopwatch;

public class FuzzyArm {

     public static final int SOGLIA = 360;

     public static void main(String[] args) {
          LightSensor color = new LightSensor(SensorPort.S1,
                  true);
          CompassSensor compass = new
                  CompassSensor(SensorPort.S2);
          UltrasonicSensor sonar = new
                  UltrasonicSensor(SensorPort.S3);
          Stopwatch timer = new Stopwatch();


          Motor.A.smoothAcceleration(true);
          Motor.C.regulateSpeed(true);
          Motor.B.regulateSpeed(true);
…
```

```
…
          MembershipFunction farL = new MembershipFunction(0, 0,
                60, 100, 1);
          MembershipFunction mediumL = new MembershipFunction(65,
                100, 120, 155, 1);
          MembershipFunction nearL = new MembershipFunction(120,
                155, 155, 155, 1);


          MembershipFunction nearR = new MembershipFunction(145,
                145, 145, 180, 1);
          MembershipFunction mediumR = new MembershipFunction(145,
                180, 220, 270, 1);
          MembershipFunction farR = new MembershipFunction(230,
                270, 360, 360, 1);


          MembershipFunction slowUp = new MembershipFunction(-300,
                50, 50, 300, 1);
          MembershipFunction slow = new MembershipFunction(-300,
                0, 0,  300, 1);
          MembershipFunction medium = new MembershipFunction(200,
                500, 500, 900, 1);
          MembershipFunction fast = new MembershipFunction(600,
                900, 900, 1200, 1);


          MembershipFunction farDown = new MembershipFunction(21,
                23, 23, 23, 1);
          MembershipFunction mediumDown = new
                MembershipFunction(6, 16, 20, 23, 1);
          MembershipFunction nearDown = new MembershipFunction(6,
                6, 6, 16, 1);


          MembershipFunction farUp = new MembershipFunction(15,
                25, 25, 25, 1);
          MembershipFunction mediumUp = new MembershipFunction(8,
                11, 15, 25, 1);
          MembershipFunction nearUp = new MembershipFunction(8, 8,
                8, 10, 1);


          MembershipFunction nearRed = new MembershipFunction(45,
                45, 45, 80, 1);
          MembershipFunction mediumRed = new
                MembershipFunction(45, 80, 140, 210, 1);
          MembershipFunction farRed = new MembershipFunction(140,
                210, 300, 300, 1);
…
```

```
…
          MembershipFunction nearBlue = new
                MembershipFunction(220, 255, 255, 255, 1);
          MembershipFunction mediumBlue = new
                MembershipFunction(90, 160, 220, 255, 1);
          MembershipFunction farBlue = new MembershipFunction(0,
                0, 90, 160, 1);


          MembershipFunction veryEarly = new MembershipFunction(0,
                0, 0, 800, 1);
          MembershipFunction early = new MembershipFunction(0,
                1000, 1000, 1500, 1);
          MembershipFunction late = new MembershipFunction(1200,
                1500, 10000, 10000, 1);


          MembershipFunction veryEarlyUD = new
                MembershipFunction(0, 0, 0, 500, 1);
          MembershipFunction earlyUD = new MembershipFunction(0,
                600, 600, 1000, 1);
          MembershipFunction lateUD = new MembershipFunction(700,
                1000, 10000, 10000, 1);
//IL BRACCIO RUOTA FINO A POSIZIONARSI SOPRA LA PALLINA
          timer.reset();
          if (compass.getDegrees() > 150) {
               Motor.C.forward();
               while (compass.getDegrees() > 150 &&
                !Button.ESCAPE.isPressed()) {


                  int x =
                   Float.valueOf(compass.getDegrees()).intValue();
                  int t = timer.elapsed();


                  double muNear = nearR.fuzzificate(x);
                  double muMedium = mediumR.fuzzificate(x);
                  double muFar = farR.fuzzificate(x);
                  double muEarly = early.fuzzificate(t);
                  double muVeryEarly = veryEarly.fuzzificate(t);
                  double muLate = late.fuzzificate(t);


                  MembershipFunction slowC =
                       slow.inference(Math.max(muVeryEarly,
                       muNear));
…
```

```
…
                MembershipFunction mediumC =
                  medium.inference(Math.max(Math.max(Math.min(mu
                  Far, muEarly), Math.min(muMedium, muLate)),
                  Math.min(muMedium, muEarly)));
                MembershipFunction fastC =
                  fast.inference(Math.min(muFar, muLate));

                int motorC = MembershipFunction.defuzCOG(slowC,
                  mediumC, fastC);

                Motor.C.setSpeed(motorC);
            }
        }
        else {
            Motor.C.backward();
            while (compass.getDegrees() < 150 &&
                  !Button.ESCAPE.isPressed()) {

                int x =
                  Float.valueOf(compass.getDegrees()).intValue()
                  ;
                int t = timer.elapsed();

                double muFar = farL.fuzzificate(x);
                double muMedium = mediumL.fuzzificate(x);
                double muNear = nearL.fuzzificate(x);
                double muEarly = early.fuzzificate(t);
                double muVeryEarly = veryEarly.fuzzificate(t);
                double muLate = late.fuzzificate(t);

                MembershipFunction fastC =
                  fast.inference(Math.min(muFar, muLate));
                MembershipFunction mediumC =
                  medium.inference(Math.max(Math.max(Math.min(mu
                  Far, muEarly), Math.min(muMedium, muLate)),
                  Math.min(muMedium, muEarly)));
                MembershipFunction slowC =
                  slow.inference(Math.max(muVeryEarly, muNear));

                int motorC = MembershipFunction.defuzCOG(slowC,
                  mediumC, fastC);
…
```

```
…
                    Motor.C.setSpeed(motorC);
                }
            }
            Motor.C.stop();
//FINE ROTAZIONE

//IL BRACCIO SI ABBASSA FINO ALL'ALTEZZA DELLA PALLINA
            timer.reset();
            Motor.B.backward();
            while (sonar.getDistance() > 8 &&
                        !Button.ESCAPE.isPressed()) {

                int x = sonar.getDistance();
                int t = timer.elapsed();

                double muFar = farDown.fuzzificate(x);
                double muMedium = mediumDown.fuzzificate(x);
                double muNear = nearDown.fuzzificate(x);
                double muEarly = earlyUD.fuzzificate(t);
                double muVeryEarly = veryEarlyUD.fuzzificate(t);
                double muLate = lateUD.fuzzificate(t);

                MembershipFunction fastB =
                      fast.inference(Math.min(muFar, muLate));
                MembershipFunction mediumB =
                      medium.inference(Math.max(Math.max(Math.min(mu
                      Far, muEarly), Math.min(muMedium, muLate)),
                      Math.min(muMedium, muEarly)));
                MembershipFunction slowB =
                      slow.inference(Math.max(muVeryEarly, muNear));

                int motorB = MembershipFunction.defuzCOG(slowB,
                      mediumB, fastB);

                Motor.B.setSpeed(motorB*5/6);

            }
            Motor.B.stop();
//FINE ABBASSAMENTO
…
```

```
…
//IL BRACCIO CHIUDE LA MANO
          Motor.A.backward();
          try {
               Thread.sleep(400);
          }
          catch (InterruptedException e) {}
          Motor.A.stop();
//FINE CHIUSURA

//IL BRACCIO ALZA LA PALLA
          timer.reset();
          Motor.B.forward();
          while (sonar.getDistance() < 23 &&
                    !Button.ESCAPE.isPressed()) {

               int x = sonar.getDistance();
               int t = timer.elapsed();

               double muFar = farUp.fuzzificate(x);
               double muMedium = mediumUp.fuzzificate(x);
               double muNear = nearUp.fuzzificate(x);
               double muEarly = earlyUD.fuzzificate(t);
               double muVeryEarly = veryEarlyUD.fuzzificate(t);
               double muLate = lateUD.fuzzificate(t);

               MembershipFunction slowB =
                     slowUp.inference(Math.max(muFar,
                     muVeryEarly));
               MembershipFunction mediumB =
                     medium.inference(Math.max(Math.max(Math.min(mu
                     Near, muEarly), Math.min(muMedium, muLate)),
                     Math.min(muMedium, muEarly)));
               MembershipFunction fastB =
                     fast.inference(Math.min(muNear, muLate));

               int motorB = MembershipFunction.defuzCOG(slowB,
                     mediumB, fastB);

               Motor.B.setSpeed(motorB);
          }
          Motor.B.stop();
//FINE ELEVAMENTO
…
```

```
…
//CONTROLLA IL COLORE DELLA PALLINA PER DECIDERE DOVE PORTARLA
          if (color.getNormalizedLightValue() > SOGLIA) {
               timer.reset();
//RUOTA VERSO I 50°
               Motor.C.forward();
               while (compass.getDegrees() > 50 &&
                     !Button.ESCAPE.isPressed()) {

                   int x =
                     Float.valueOf(compass.getDegrees()).intValue()
                     ;
                   int t = timer.elapsed();

                   double muFar = farRed.fuzzificate(x);
                   double muMedium = mediumRed.fuzzificate(x);
                   double muNear = nearRed.fuzzificate(x);
                   double muEarly = early.fuzzificate(t);
                   double muVeryEarly = veryEarly.fuzzificate(t);
                   double muLate = late.fuzzificate(t);

                   MembershipFunction fastC =
                     fast.inference(Math.min(muFar, muLate));
                   MembershipFunction mediumC =
                     medium.inference(Math.max(Math.max(Math.min(mu
                     Far, muEarly), Math.min(muMedium, muLate)),
                     Math.min(muMedium, muEarly)));
                   MembershipFunction slowC =
                     slow.inference(Math.max(muVeryEarly, muNear));

                   int motorC = MembershipFunction.defuzCOG(fastC,
                     mediumC, slowC);

                   Motor.C.setSpeed(motorC);
               }
          }
          else {
               timer.reset();
//RUOTA VERSO I 250°
               Motor.C.backward();
…
```

```
…
              while (compass.getDegrees() < 250 &&
                    !Button.ESCAPE.isPressed()) {

                  int x =
                    Float.valueOf(compass.getDegrees()).intValue()
                    ;
                  int t = timer.elapsed();

                  double muFar = farBlue.fuzzificate(x);
                  double muMedium = mediumBlue.fuzzificate(x);
                  double muNear = nearBlue.fuzzificate(x);
                  double muEarly = early.fuzzificate(t);
                  double muVeryEarly = veryEarly.fuzzificate(t);
                  double muLate = late.fuzzificate(t);

                  MembershipFunction fastC =
                    fast.inference(Math.min(muFar, muLate));
                  MembershipFunction mediumC =
                    medium.inference(Math.max(Math.max(Math.min(mu
                    Far, muEarly), Math.min(muMedium, muLate)),
                    Math.min(muMedium, muEarly)));
                  MembershipFunction slowC =
                    slow.inference(Math.max(muVeryEarly, muNear));

                  int motorC = MembershipFunction.defuzCOG(fastC,
                    mediumC, slowC);

                  Motor.C.setSpeed(motorC);
              }
          }
          Motor.C.stop();
//FINE ROTAZIONE

//IL BRACCIO SI ABBASSA FINO ALL'ALTEZZA DELLA POSTAZIONE DELLA
PALLINA
          timer.reset();
          Motor.B.backward();
          while (sonar.getDistance() > 7 &&
              !Button.ESCAPE.isPressed()) {

               int x = sonar.getDistance();
               int t = timer.elapsed();
…
```

```
…
                double muFar = farDown.fuzzificate(x);
                double muMedium = mediumDown.fuzzificate(x);
                double muNear = nearDown.fuzzificate(x);
                double muEarly = earlyUD.fuzzificate(t);
                double muVeryEarly = veryEarlyUD.fuzzificate(t);
                double muLate = lateUD.fuzzificate(t);

                MembershipFunction fastB =
                      fast.inference(Math.min(muFar, muLate));
                MembershipFunction mediumB =
                      medium.inference(Math.max(Math.max(Math.min(mu
                      Far, muEarly), Math.min(muMedium, muLate)),
                      Math.min(muMedium, muEarly)));
                MembershipFunction slowB =
                      slow.inference(Math.max(muVeryEarly, muNear));

                int motorB = MembershipFunction.defuzCOG(slowB,
                      mediumB, fastB);

                Motor.B.setSpeed(motorB*5/6);

          }
          Motor.B.stop();
//FINE ABBASSAMENTO

//IL BRACCIO APRE LA MANO
          Motor.A.forward();
          try {
               Thread.sleep(400);
          }
          catch (InterruptedException e) {}
          Motor.A.stop();
//FINE APERTURA;

//IL BRACCIO SI ALZA COME CONCLUSIONE
          timer.reset();
          Motor.B.forward();
…
```

```
…
               while (sonar.getDistance() < 23 &&
                     !Button.ESCAPE.isPressed()) {

               int x = sonar.getDistance();
               int t = timer.elapsed();

               double muFar = farUp.fuzzificate(x);
               double muMedium = mediumUp.fuzzificate(x);
               double muNear = nearUp.fuzzificate(x);
               double muEarly = earlyUD.fuzzificate(t);
               double muVeryEarly = veryEarlyUD.fuzzificate(t);
               double muLate = lateUD.fuzzificate(t);

               MembershipFunction slowB =
                     slowUp.inference(Math.max(muFar,
                     muVeryEarly));
               MembershipFunction mediumB =
                     medium.inference(Math.max(Math.max(Math.min(mu
                     Near, muEarly), Math.min(muMedium, muLate)),
                     Math.min(muMedium, muEarly)));
               MembershipFunction fastB =
                     fast.inference(Math.min(muNear, muLate));

               int motorB = MembershipFunction.defuzCOG(slowB,
                     mediumB, fastB);

               Motor.B.setSpeed(motorB);
          }
          Motor.B.stop();
//FINE ELEVAMENTO
     }
}
```

5.4 Confronto

Come termine di paragone per confrontare le due realizzazioni del braccio meccanico non è probabilmente molto importante il tempo di esecuzione dell'azione. Il fattore principale consiste nelle vibrazioni della struttura e nella precisione con cui il braccio meccanico si ferma nei punti stabiliti.
Osserviamo cosa accade generalmente quando il braccio meccanico comincia un movimento e quando lo termina.

- **Programmazione a behavior**
  1. nelle fasi di *partenza* la struttura vibra a causa della forte accelerazione
  2. nelle fasi di *arrivo* la struttura vibra molto a causa del blocco improvviso del motore mentre si trova alla velocità massima impostata per l'azione
  3. la *velocità* è sempre alta
  4. nelle fasi di *arrivo* il braccio a volte non riesce a fermarsi nel punto desiderato ma lo fa leggermente oltre
- **Programmazione fuzzy**
  1. nella fase di *partenza* le vibrazioni sono praticamente annullate dato che è fornita un'accelerazione lieve tramite il controllo del tempo trascorso dall'inizio del movimento
  2. nella fase di *arrivo* le vibrazioni sono annullate dalla decelerazione graduale dovuta all'avvicinamento del braccio alla destinazione
  3. la *velocità* è alta nella parte centrale della percorrenza
  4. nelle fasi di *arrivo* il braccio si ferma sempre nel punto desiderato

In conclusione la programmazione a behavior sicuramente può fornire una velocità di esecuzione maggiore, ma deve anche pagare lo scotto delle forti vibrazioni della struttura che possono provocare problemi nel trasporto di certi tipo di oggetti; invece la programmazione tramite logica fuzzy pur riuscendo a garantire una buona velocità di esecuzione riesce ad annullare quasi del tutto le vibrazioni con accelerazione e decelerazioni accettabili.

Bibliografia

[1] http://doc.studenti.it/tesina/intelligenza-artificiale/fuzzy-logic-storia-applicazioni.html

[2] http://www.giorgiotave.it/wikigt/os/Logica_fuzzy

[3] Lofti A. Zadeh. Fuzzy Sets*. Information and control,* 8:338-353, 1965

[4] http://www.mathworks.com/access/helpdesk/help/toolbox/fuzzy/fp351dup8.html

[5] http://www.di.uniba.it/~ig/lezioni-08-09/schedule.htm

[6] http://en.wikipedia.org/wiki/Fuzzy_control_system

[7] http://www.generation5.org/content/2001/falcon.asp

[8] Kevin M. Passino, Stephen Yurkovich. Fuzzy Control. *Addison-Wesley*

[9] http://automatica.ing.unibs.it/mco/cgsa/neuro-fuzzy/paragrafi_teoria/

[10] http://lejos.sourceforge.net/

[11] http://www.cs.princeton.edu/courses/archive/fall07/cos436/HIDDEN/Knapp/fuzzy004.htm

[12] http://www.nxtprograms.com/robot_arm/index.html

[13] Andrea Piccolo. Analisi e sperimentazione della piattaforma LejOS su minirobot LEGO® MINDSTORMS™ NXT.

[14] Alberto Giovanni Busetto, Fabio Dalla Libera, Fabrizio Lana, Manuela Mantione, Martina Molin, Luca Polin, Paolo Semenzato. La logica fuzzy.

[15] http://www.ce.unipr.it/people/bianchi/Research/ProgettoFuzzy/home.html