ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA IN INGEGNERIA ELETTRONICA
E DELLE TELECOMUNICAZIONI

COSTRUZIONE DI UN ROBOT CON ARCHITETTURA BEHAVIOR BASED

Elaborato in
FONDAMENTI DI INFORMATICA LB

Relatore

Andrea Roli
……...

Presentata da

Matteo Bezzi
……...

Sessione Prima
Anno Accademico 2012/2013

Indice

Introduzione

Erano gli anni ’80 quando il professor Brooks[1], partendo da un modello reattivo come i veicoli di Braitenberg[2], iniziò a formalizzare quella che oggi è l’architettura behavior based. Le semplici azioni reattive su cui si basavano tali veicoli sono state sostituite da più complessi ed articolati comportamenti, organizzati in livelli e in grado di interagire tra loro.

L’obiettivo che ci si è posti in questa tesi è stato costruire un robot *path follower*[3] con architettura behavior based. Si è voluto costruire un modello reale che dovesse confrontarsi con variazioni dell’ambiente, errori di lettura e spostamento. Per prima cosa è stato costruito un modello olonomico, lo si è dotato di un sensore luminoso per trovare la traccia e tre sensori ultrasonici per percepire gli ostacoli; successivamente si è iniziato ad implementare il controller. Per fare questo ci si è ispirati ai lavori di Brooks, andando pian piano a definire i vari comportamenti ed evolvendo il sistema gradualmente. La progettazione ha dovuto tener conto di molti fattori come errori di spostamento dovuti alla diversa aderenza di ruote e terreni, errori di lettura da parte dei sensori, limiti fisici dei componenti, possibili movimenti di un sistema olonomico, priorizzazione dei comportamenti e comportamento emergente dalla loro interazione. Grazie a questa interazione il robot cercherà la traccia ed inizierà a seguirla, evitando il contatto con altri corpi ed eventuali ostacoli.

Per la costruzione del modello si è scelto di usare il sistema LEGO© Mindstorms©. La scelta, in primo luogo determinata dalla disponibilità all’interno del laboratorio di questa piattaforma, si è confermata valida per la praticità nella costruzione e modifica dei modelli, la reperibilità di accessori o pezzi di ricambio e la disponibilità su Internet di moltissimi progetti, esperienze e discussioni sull’argomento.

Nei primi capitoli della tesi abbiamo fornito alcune informazioni sulle caratteristiche che erano già decise all’inizio del progetto: vedremo alcuni tipi di controller soffermandoci su quello utilizzato per programmare il robot; descriveremo la piattaforma LEGO Mindstorms di cui disponiamo nel laboratorio e vedremo più nel dettaglio cosa significhi avere un robot olonomico. Successivamente passeremo alla parte progettuale della tesi, vedendo in che contesto si è testato il robot, come si è costruito il modello, quali soluzioni sono state efficaci per farlo muovere ed interfacciare con l’ambiente ed infine una descrizione dei comportamenti del robot e delle relative implementazioni.

---

[1] Rodney A. Brooks (1950), professore emerito del Massachusetts Institute of Technology, per il quale ha svolto e diretto fino al 2007 una proficua attività ricercativa, ricoprendo nel corso degli anni numerose cariche istituzionali.

[2] Valentino Braitenberg (1926-2011), neuropsichiatra e studioso di cibernetica italiano, fondatore e dal 1968 al 1994 direttore dell’istituto di Biologia Cibernetica del Max-Planck-Institut di Tubinga. Nel 1984 scrisse un libro intitolato "I veicoli pensanti. Saggio di psicologia sintetica" in cui descriveva 14 ipotetici veicoli i cui movimenti, semplici funzioni dei sensori, potevano essere interpretati, da un osservatore esterno, come comportamenti causati da stati psicologici quali paura, amore o ottimismo. [8] [9]

[3] Così vengono comunemente chiamati i robot il cui scopo principale è seguire un percorso.

1 Architettura del controller

Nella progettazione di un robot, fin dai primi passi ci si trova di fronte a numerose scelte. Che tipo di struttura dovrà avere, se e quale sistema di locomozione avrà, con che sensori sarà equipaggiato e che tipo di attuatori potrà utilizzare per agire sullo spazio circostante. Struttura, Attuatori e Sensori costituirebbero solo un esoscheletro comandabile a distanza se non ci fosse il *controller*[4].
Di seguito abbiamo analizzato le tipologie di controller più comuni basandoci sulle definizioni date da R.A.Brooks [1] e M.Mataric [2].

1.1 Deliberative Controller

E' il tipo di controller classico, una delle prime concezioni di intelligenza artificiale. Può essere rappresentato da una serie di blocchi orizzontali che formano un unico algoritmo. Il robot prende gli input in ingresso, immagazzina dati sull'ambiente circostante e li utilizza per costruirne una rappresentazione simbolica. Basandosi su questa rappresentazione, elabora poi una soluzione che risponda alle specifiche, pianificando in maniera ottimale percorsi da seguire o azioni da compiere. I punti di forza di questa architettura sono la precisione e l'efficienza della risposta, ma il prezzo è un tempo di elaborazione, relativamente elevato, che determina vulnerabilità a modifiche repentine nell'ambiente come lo spostamento di ostacoli ed obiettivi.
Esempio: Il robot è dotato di una telecamera, prima di muoversi controlla lo spazio circostante, identifica le pareti, i corridoi e le porte, rileva l'obiettivo ed eventuali ostacoli poi, dopo aver costruito una mappa, pianifica possibili percorsi ed azioni da eseguire, sceglie la soluzione migliore ed infine inizia a muoversi.

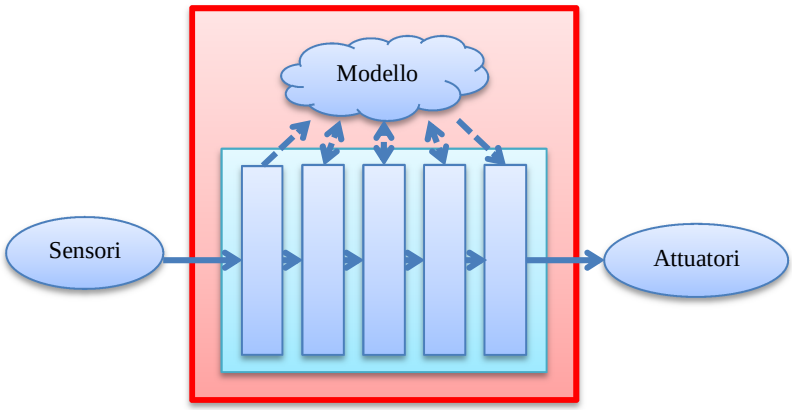


Figura 1: Rappresentazione di un controller deliberativo.

[4] Utilizzeremo il termine controller in quanto ci stiamo riferendo ad un componente specifico, così comunamente chiamato anche in lingua italiana.

1.2 Reactive Controller

Questo controller si comporta in modo opposto al Deliberative. I dati rilevati dai sensori vengono forniti in parallelo a molti blocchi che in un tempo breve forniranno un uscita per gli attuatori. Non vi è alcun tipo di rappresentazione simbolica del mondo circostante e, a parte qualche variabile da confrontare, non vengono memorizzati dati sull'ambiente. In questo caso il robot si comporta in maniera "istintiva" e la problematica principale è priorizzare questi "istinti" in modo che, ad esempio, il primo istinto sia quello di fermarsi ad un ostacolo. Per gestire queste priorità ed evitare conflitti i segnali in uscita dai blocchi non vengono quindi mandati direttamente agli attuatori ma passano attraverso un blocco che può decidere di farne passare solo alcuni o sommarli tra loro.
Esempio: il robot è dotato di sensori ottici. Inizia a muoversi appena vede la luce per inseguirla e si ferma se smette di vederla.

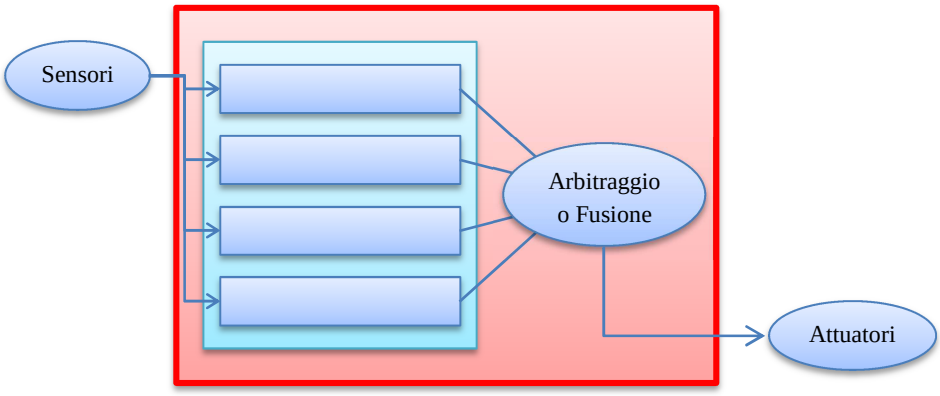


Figura 2: Rappresentazione di un controller reattivo.

L'organizzazione dei vari blocchi paralleli può essere strutturata a livelli per rendere più semplice la progettazione e robusto il sistema. Parliamo in questo caso di architettura a sussunzione: siccome ci troviamo ad avere numerosi blocchi paralleli con la problematica di decidere come priorizzarli, con quest'architettura andiamo a dividere il sistema in vari livelli che forniranno intrinsecamente una gerarchia di priorità.
Iniziamo con il progettare i blocchi dei livelli più semplici ed andiamo successivamente e gradualmente ad "evolvere" il nostro sistema aggiungendo i livelli superiori. Ogni livello sarà autosufficiente nel senso che non avrà bisogno dei livelli superiori ma si appoggerà ai blocchi dei livelli inferiori che di fatto andrà ad inglobare. I blocchi dei livelli inferiori restano sempre attivi. Per interfacciarsi al sistema i livelli superiori possono disabilitare i segnali in ingresso ai blocchi sottostanti e/o sovrascriverne l'uscita. In questo modo il nostro sistema risulta semplice da progettare in quanto ogni livello dovrà solo tener conto

dei livelli precedenti e robusto in quanto, immaginando i vari blocchi su macchine separate, il malfunzionamento di un blocco non andrà a compromettere il funzionamento di tutti i livelli sottostanti.

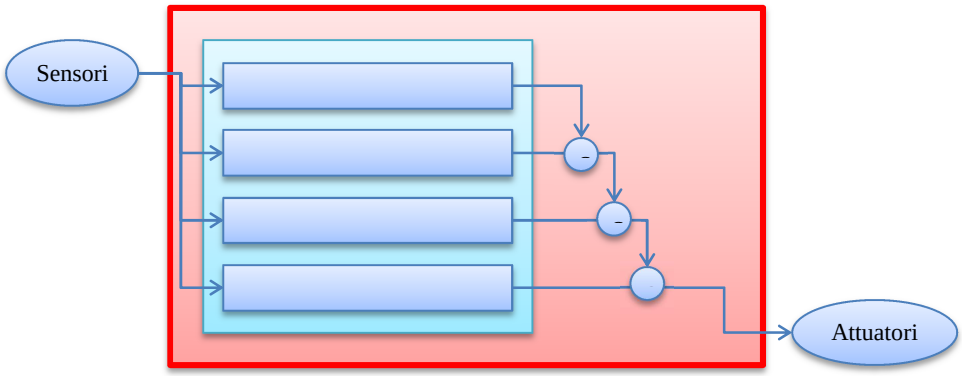


Figura 3: Controller reattivo con architettura a sussunzione.

1.3 Hybrid Controller

Nasce dalla fusione dei due controller appena citati: La parte reattiva è sempre attiva e all'arrivo dei primi input inizia subito a far reagire il robot. Nel frattempo gli stessi input vengono passati anche alla parte deliberativa che inizia a pianificare una strategia per arrivare all'obiettivo nel modo più efficiente. A dialogare tra questi due blocchi troviamo il blocco ibrido che si occuperà di gestire le differenze nei tempi dei due blocchi e dovrà decidere se far aspettare la parte reattiva o ignorare la parte deliberativa.
Esempio: il robot è dotato di una telecamera, una mappa del posto e sensori di prossimità. Appena il robot riceve l'input, la parte reattiva si mette in moto andando, se possibile, in direzione dell'obiettivo. Nel frattempo il blocco deliberativo inizia a calcolare il percorso ottimale. Quando il blocco deliberativo è pronto il blocco ibrido decide se far seguire al robot il percorso pianificato o se farne calcolare un altro perché ad esempio, vagando in modalità reattiva ha già visto che una strada lungo il percorso pianificato è bloccata.

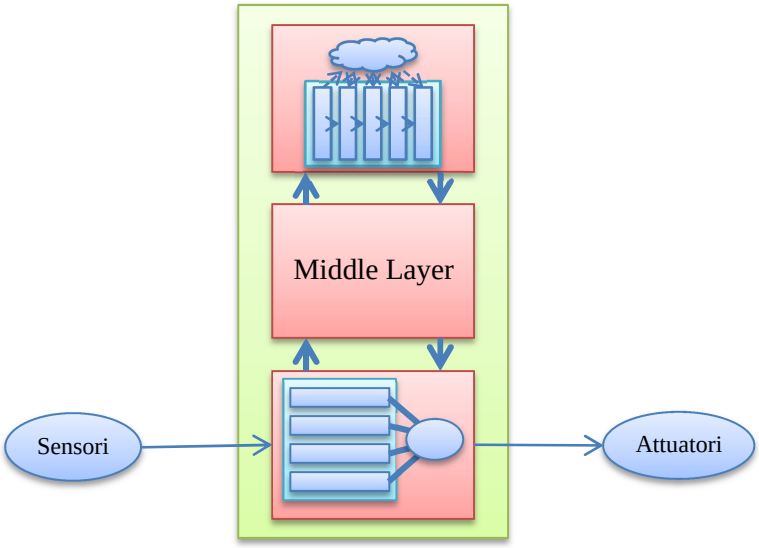


Figura 4: Controller ibrido formato sia da blocchi reattivi che deliberativi.

Esistono anche controller ibridi che sono di fatto semplici reattivi nei quali la parte deliberativa ha solo un numero finito di possibilità e tutte le possibili soluzioni sono già state calcolate off-line. Questi robot risultano nel funzionamento molto efficaci ma devono essere sempre tenuti aggiornati per quanto riguarda mappe o percorsi affinché rimangano tali.

1.4 BehaviorBased Controller

Cronologicamente è il tipo di controller più recente e ancora al centro di molti studi. Proprio per questo si possono notare tra i ricercatori pareri non sempre concordi nel definire le caratteristiche di questo controller.
L'idea principale è di realizzare un controller di derivazione reattiva ispirato ai sistemi biologici; in tal senso vediamo che l'architettura a sussunzione si adatta perfettamente ad un' idea evolutiva, consentendoci di partire da comportamenti semplici ed istintivi come preservare l'integrità del robot fermandosi in caso di collisione o fuggendo da determinati stimoli, per poi sviluppare comportamenti sempre più complessi. Ogni comportamento è caratterizzato da un determinato obiettivo che cercherà sempre di raggiungere e mantenere. Il robot si troverà quindi ad avere obiettivi multipli solitamente in conflitto tra loro, la cui importanza relativa spesso dipenderà dal contesto[5].
Esempio: Il robot di Brooks cui ci si è ispirati per la realizzazione di questa tesi.

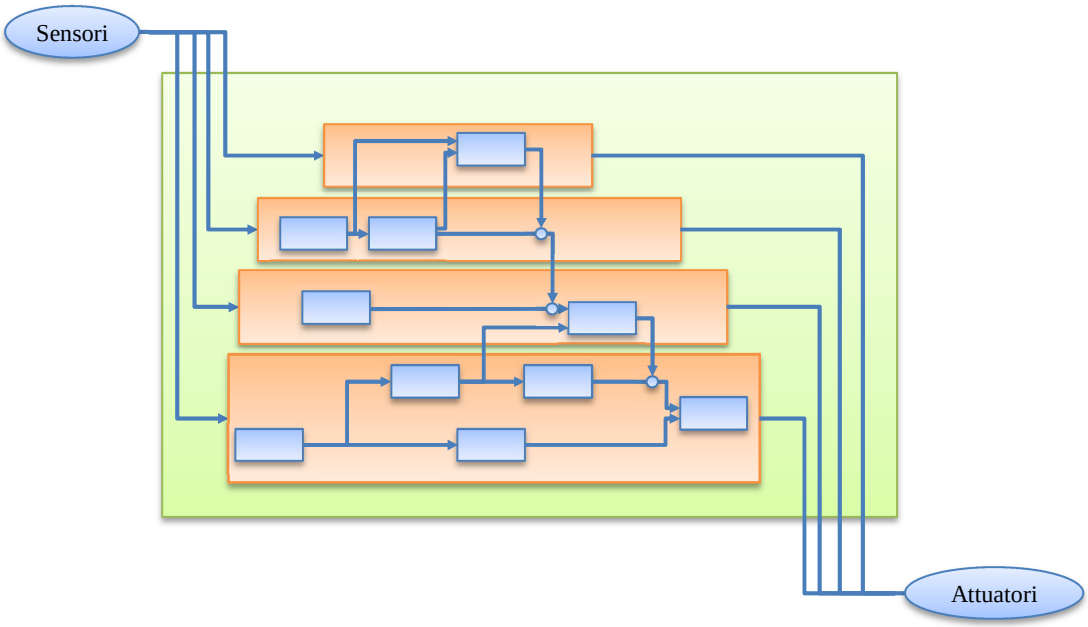


Figura 5: Controller behavior based con architettura a sussunzione.

La natura dei blocchi che compongono i vari comportamenti secondo la maggior parte dei ricercatori è di tipo reattivo ma entrando nel merito della questione si nota che ad elevati livelli di complessità questa assunzione diventa meno chiara o in certi casi vincolante.
Nel testo della Mataric si dice esplicitamente che nell'architettura a sussunzione non devono esserci blocchi deliberativi e che ogni blocco è rappresentabile come una macchina a stati finiti aumentata. Nei lavori di Brooks però notiamo come i livelli più alti dei progetti implichino la creazione di una rappresentazione che viene poi utilizzata anche dai livelli

[5] "Often the relative importance of goals will be context dependent" [1], pag 2 par 1.1.

superiori per il raggiungimento degli obiettivi. Come sappiamo, seppur i sistemi reattivi possano conservare alcune informazioni sull'ambiante, l'utilizzo di una vera e propria rappresentazione simbolica del mondo è caratteristico dei sistemi deliberativi e da questo si evince che l'architettura basata sul comportamento deve contenere anche dei livelli più vicini al modello deliberativo, costituendo di fatto un sistema ibrido. In questo sistema però non avremo né dei livelli puramente deliberativi nè un livello che amministri le differenze tra parte reattiva e deliberativa ma una gerarchia nella quale i livelli diventano via via più complessi per il raggiungimento di obiettivi più strutturati. La rappresentazione del mondo del layer deliberativo probabilmente non si troverà localizzata in un solo componente ma emergerà dall'interazione dei vari blocchi.

Le differenze di tempistica tra i livelli non sono un problema, ogni blocco è autonomo. I blocchi destinati a svolgere compiti più complessi e quindi ad essere più lenti, saranno quelli superiori ma questi, fino a quando non avranno dei dati utili da produrre, non andranno ad interferire con i livelli sottostanti che continueranno ad operare per mantenere i propri obiettivi.

Avremo quindi alla base dei livelli puramente reattivi ma, salendo di livello ed aumentando di complessità troveremo via via caratteristiche più deliberative. Saranno i livelli stessi a gestire tempi e priorità sostituendo il blocco ibrido.

Proseguendo con questa ipotesi potremo vedere il sistema di apprendimento umano come un sistema a sussunzione nel quale livelli superiori vengono inizialmente implementati in maniera deliberativa e, mano a mano che ci abituiamo a certi comportamenti, vengono sostituiti con livelli reattivi: basti pensare ad esempio alla guida nella quale inizialmente dobbiamo ragionare step by step su quello che stiamo facendo, immaginando magari cosa farà la macchina in risposta alle nostre azioni, ma poi pian piano la nostra risposta diventa intuitiva, come nel caso delle macchine ibride con livello deliberativo precompilato.

2 LEGO Mindstorms

Il progetto LEGO Mindstorms trova le sue origini nella collaborazione tra il Massachusetts Institute of Technology (MIT) e la danese LEGO che fin dal 1985 sponsorizza i lavori del MIT Media Lab. [21]
Il primo progetto ad essere completato fu il LEGO tc Logo nel 1988. Commercializzato solo per scopi didattici, consentiva agli studenti di programmare le loro costruzioni ma aveva bisogno di restare collegato al computer sul quale effettivamente giravano i programmi. Lo stesso anno i ricercatori del Media Lab iniziarono a sviluppare il brick programmabile al quale poi si ispirò il progetto del primo brick progettato e prodotto in casa LEGO. [22] [23] [12] [17] [4]
Il primo programmable brick di LEGO fu l'RCX 1.0 che nel 1998 venne commercializzato con kit LEGO Mindosrm RIS (Robotic Invencion System). Negli anni successivi l'RCX venne aggiornato e commercializzato in altre versioni più economiche[6]. Nel 2006 iniziò la produzione del nuovo mattoncino NXT che introdusse un nuovo sistema di connessione dei componenti e capacità molto maggiori da parte del predecessore. Anche di questa generazione sono già state rilasciate due versioni. Entro la fine del 2013 dovrebbe essere rilasciata la nuova piattaforma che introdurrà il brick EV3 (EVolution) e nuove periferiche, mantenendo però lo stesso sistema di connettori dell'NXT e la retro compatibilità. [27]
Vediamo ora più nel dettaglio i brick RCX ed NXT che sono presenti all'interno del laboratorio.

Figura 6: Brick RCX 1.0 del 1998.

[6] RCX 1.0 (1998), RCX 1.5 (1999), RCX 2.0 (2001), Scout (1999), Microscout (1999), Spybot (2002), Cybermaster (1998). [18]

2.1 RCX

L’RCX (Robotic Command eXplorers) Brick è stato il cervello della prima generazione di LEGO Mindsorms. L’RCX si basava su un micro controller a 8 bit Renesas H8/300 a 16 MHz con512 byte di RAM interna affiancato da ulteriori 32kB di RAM e 16kB di ROM. La programmazione avveniva scaricando il codice compilato direttamente sulla memoria RAM grazie all’apposita interfaccia IR LEGO Tower. Il sistema s’interfacciava con l’ambiente circostante tramite 6 contatti a forma di mattoncino (3 sensori e 3 attuatori) e poteva comunicare con un computer o con altri Brick tramite una porta ad infrarossi che però sfruttava un protocollo proprietario rendendo impossibile la comunicazione con altre periferiche irda. L’interazione diretta dell’utente era possibile grazie a 4 pulsanti, uno schermo LCD posto al centro del brick ed un altoparlante. L’alimentazione del primo brick era possibile con 6 batterie di tipo AA o con un connettore per collegare un alimentatore esterno, connettore la cui presa venne purtroppo rimossa nei successivi aggiornamenti. [12] [13]

2.1.1 Programmazione

Per la programmazione del brick inizialmente venne fornito RCX Code, un ambiente grafico dalla LEGO che era incluso nella confezione. Successivamente venne sviluppato ROBOLAB, anch’esso grafico e basato su LabVIEW. Pian piano con il diffondersi del prodotto e la nascita di numerosi progetti, nella rete si affiancarono altri linguaggi di programmazione, alcuni dei quali potevano essere compilati direttamente per il mattoncino originale, altri dei quali richiedevano per il funzionamento o per la possibilità di sfruttare caratteristiche avanzate la modifica del firmware. Tra questi troviamo ad esempio QuiteC, LeJos o NotQuiteC. Vedremo questi dettagli parlando del blocco NXT. [12]

2.1.2 Sensori

Abbiamo visto l’unità centrale del sistema che però senza la possibilità di acquisire dati dall’ambiente esterno o interagire con esso non ci avrebbe certo consentito di costruire un robot. I sensori di cui l’RCX si avvaleva possono essere divisi in due categorie: i sensori attivi, che hanno bisogno di essere alimentati poter fornire delle letture e quelli passivi. Vediamo ora quali sensori erano stati prodotti da LEGO per equipaggiare l’RCX.

Figura 7: Sensori dell'RCX.: sensore di contatto, sensore di luminosità, sensore di tempertura e sensore di rotazione.

Sensore di contatto

E' il sensore passivo più semplice ed è in grado di segnalare al robot il contatto di un oggetto mediante la pressione di un pulsante. Durante la programmazione possiamo impostare per questo sensore varie modalità di lettura: un valore booleano (0,1) a seconda che il pulsante sia premuto o meno, un valore intero che conta il numero di pressioni o un valore intero che conta il numero di variazioni di stato. Nella confezione RIS originale erano presenti due di questi sensori. [6]

Sensore di luminosità

Questo sensore è attivo ed è formato principalmente da un led rosso ed un fototransistor. Quando ci si trova lontani da superfici il fototransistor all'interno del circuito ci consente di rilevare la luce ambientale. Quando siamo in prossimità di oggetti invece la luce del LED verrà riflessa dalle superfici vicine, tornando al sensore che ne misurerà l'intensità, fornendo informazioni sulla superficie riflettente o la sua distanza. Il sensore riesce a misurare fonti luminose in tutto lo spettro del visibile ma mostra un picco di sensibilità per le basse frequenze nella regione rossa e infrarossa. Il brick per questo sensore ci può fornire in lettura un valore percentuale o un valore raw di intensità luminosa. Uno di questi sensori era presente nella confezione base del RIS. [4] [6]

Sensore di temperatura

Questo sensore si distingue subito dagli altri a causa del cilindro metallico della sonda di temperatura che fuoriesce dal mattoncino. Come possiamo facilmente dedurre questo sensore è in grado di fornirci la temperatura ambientale. Il valore letto sarà in gradi ma dovremo inizializzare il sensore definendo che scala utilizzare tra Celsius e Fahrenheit.
Questa periferica non era fornita insieme al kit dell'RCX e doveva essere acquistata separatamente. [4]

Sensore di rotazione

Questo sensore, grazie al foro a forma di croce che lo attraversa, può essere accoppiato ad un motore o ad una qualunque barretta di trasmissione all'interno del meccanismo. Facendo ruotare la barretta nel foro viene messo in movimento un cilindro con quattro alette che, passando attraverso due fotocellule, consente di ricavare la posizione della barretta e il numero di giri. La lettura dell'angolo statico è abbastanza grossolana poiché con soli 16 livelli abbiamo circa 22,5° tra un livello e l'altro. In compenso se vogliamo misurare la velocità di rotazione può fornire dati abbastanza precisi a patto che la velocità non sia troppo bassa.
Neanche questo sensore era fornito di base nella confezione del RIS, nonostante risultasse un componente fondamentale in molte costruzioni di complessità elevata e doveva esser ordinato separatamente. [4] [5]

LEGO Vision Command

All'interno del kit LEGO Vision Command era presente una telecamera. Questo accessorio, elencato tra gli altri sensori non è in realtà direttamente gestito dal robot ma richiede l'utilizzo di un computer. La telecamera è infatti una normale webcam Logitech Quick Cam© inserita in un case LEGO creato per poter essere meccanicamente interfacciato agli altri componenti. Una volta collegata via USB al computer per comunicare con l'RCX questa telecamera ha bisogno del software di corredo Vision Command capace di riconoscere diverse luci, movimenti e colori interfacciandosi poi con il brick. [12]

Figura 8: La telecamera inclusa nel kit Vision Command.

2.1.3 Attuatori

Gli attuatori disponibili nella confezione dell'RCX originariamente erano motori e luci. I contatti di output di fatto non servivano per comunicare con gli attuatori ma trasmettevano solo l'alimentazione interamente gestita dal Brick.

Motori

I motori utilizzati dall'RCX sono i classici LEGO Electric Technic Mini-Motor a 9V già affermatisi anni prima della comparsa dell'RCX. Questo motore include già un sistema di ingranaggi grazie al quale ha una velocità a vuoto di 340 rpm ed una coppia di 5,5N cm [7]. Di fatto non abbiamo elettronica di controllo sul motore e tutte le configurazioni per selezionare direzione e potenza della torsione avvengono sul Brick. Questa scelta rende possibile anche l'utilizzo di tutti i motori a 9V prodotti precedentemente, dotati di diverse caratteristiche.

Luci

Come i motori anche le luci utilizzate erano già presenti nella linea LEGO. Collegabili agli stessi contatti dei motori, possono essere affiancate al sensore di luce per fornire valori più precisi ma anche semplicemente per illuminare o segnalare.

2.2 NXT

La seconda generazione di LEGO Mindstorms, ha introdotto numerosi cambiamenti rispetto all'RCX. Primo fra tutti il nuovo brick NXT (NeXT) decisamente più potente dell'RCX e dotato di numerose altre possibilità di interfacciamento.
Il cuore del brick NXT è un processore a 32 bit Atmel AT91SAM7S256 (classe ARM7) a 48 MHz, con 256k flash memory e 64k RAM, supportato da un coprocessore 8 bit Atmel ATmega48 (classe AVR: è un RISC a 8 bit) a 8 MHz, con 4k flash e 512 byte RAM. Per interfacciarsi con il mondo esterno dispone di uno schermo LCD con una risoluzione di 60x100 pixel, un altoparlante, una porta USB 2.0 e connettività Bluetooth per comunicare con altri brick, computer e cellulari. Per comunicare con le varie periferiche dispone di quattro porte di ingresso e tre di uscita ma, avendo delle connessioni digitali, sarà possibile aumentarne il numero con dei moduli esterni e appositi moltiplicatori di ingressi. I connettori non sono gli stessi dell'RCX ma utilizzano un attacco proprietario simile ad un connettore RJ-12. Il mattoncino per essere alimentato richiede 6 batterie di tipo AA oppure una batteria al litio opzionale. [14] [15] [4]

Figura 9: Brick NXT nella classica versione bianca e grigia.

2.2.1 Programmazione

Nella versione retail del kit NXT, LEGO ha incluso nuovamente un software di programmazione grafico basato, come lo era ROBOLAB, su LabVIEW di nome NXT-G.
Questo software risulta inizialmente molto intuitivo ed include alcuni importanti tools come quello per l'aggiornamento del firmware o quello per il controllo remoto. Come tipico di LabVIEW ogni componente fisico deve avere un suo equivalente strumento virtuale che nel nostro caso è già stato creato per ogni componente del kit, includendo non solo i pezzi LEGO ma anche numerosi sensori prodotti da terzi. Questo linguaggio può essere interessante per le prime prove ma per progetti complessi si rivela lento e difficile da gestire.
Come il blocco RCX anche l'NXT può essere programmato in vari modi, nativamente o grazie ad una modifica al firmware. Vediamone alcuni.

ROBOLAB

Ritroviamo il software di programmazione grafica ROBOLAB nella versione 2.9 che ora supporta anche l'NXT. Seppur sia stata ufficialmente annunciata da LEGO la dismessa di questo prodotto può essere ancora scaricato e ci sono numerose risorse a disposizione nella rete. [12]

BricxCC

BricxCC è un IDE, originariamente per programmare l'RCX con NQC, consente di scrivere e compilare programmi anche per NXT utilizzando i linguaggi NBC o NXC. Per programmare con questi linguaggi non è necessaria alcuna modifica al firmware seppur sia possibile farlo per aggiungere funzionalità. Tratteremo meglio questo argomento in seguito, essendo stato scelto per il progetto. [12]

leJOS NXJ

Si tratta della versione di leJOS per NXT, una JVM concepita per girare su questo brick. Per installare leJOS dobbiamo ovviamente modificare il firmware ma fatto questo si può programmare il robot in Java utilizzando le apposite api.

ROBOTC

Robotc è un linguaggio di programmazione basato su C e progettato a fini didattici per diverse piattaforme tra le quali Mindstorms. Per funzionare richiede il caricamento di un firmware proprietario che sembra essere molto performate. La modifica del firmware e la programmazione avvengono grazie all'apposito IDE. E' un prodotto commerciale.

Matlab e Simulink

Con Matlab è possibile comandare il robot tramite USB o porta seriale bluetooth grazie ad alcune toolbox reperibili gratuitamente. Con Simulink invece, sempre grazie all'installazione di software aggiuntivo gratuitamente reperibile, si può modellare graficamente un controller, simularlo, ottenere il corrispondente in C e compilarlo nel robot.

2.2.2 Sensori

Nella confezione base dell'NXT sono inclusi 4 sensori: Il sensore di contatto, il sensore acustico, il sensore di luminosità ed il sensore ultrasonico. La grande differenza tra i sensori dell'RCX e quelli dell'NXT è che mentre i primi fornivano un valore e potevano essere al massimo alimentati, i nuovi sensori possono disporre anche di un collegamento digitale e comunicare direttamente con il brick.
Una importante distinzione da fare è quella tra il sensore fisico e l'implementazione del sensore nel codice: i sensori come il sensore di contatto o quello di luminosità forniscono fisicamente solo un valore come il contatto aperto/chiuso, una resistenza o una tensione. La conversione e le varie modalità di lettura avvengono tutte sul brick. Di fatto impostando il sensore dal codice andiamo a scegliere "come farlo leggere" al brick. Per il sensore ultrasonico invece abbiamo una vera e propria comunicazione digitale tra il controller del sensore e l'NXT.

Figura 10: Sensori dell'NXT. Partendo da sinistra vediamo sensore di contatto, sensore acustico, sensore di luminosità, sensore di colori, sensore ultrasonico e sensore di temperatura.

Sensore di contatto

Il funzionamento di questo sensore è in sostanza lo stesso di quello della serie precedente: E' un sensore passivo e può percepire pressione e rilascio del pulsante fornendo il valore corrente. Ci sono però numerosi cambiamenti: il connettore non è più a mattoncino; la struttura utilizza ora il case comune a molti sensori della nuova serie e la parte mobile del pulsante, grazie ad un foro a croce, può essere collegata direttamente ad altri pezzi. C'è anche una resistenza in serie al pulsante per evitare cortocircuiti in caso di errato collegamento ad una porta di output del controller.

Sensore acustico

Al contrario di come si potrebbe pensare, non otterremo da questo sensore un segnale audio analogico da utilizzare poi per varie elaborazioni ma solo un valore corrispondente alla pressione sonora istantanea.

La misurazione può essere impostata in dB per avere un valore in decibel del suono percepito dal microfono o in dBA per avere un valore pesato secondo la curva A di sensibilità equivalente per l'udito umano. La massima pressione sonora misurabile è di 90dB ed il valore letto sarà comunque percentuale, fissando attorno al 5% una stanza silenziosa, fino al 30% una conversazione a voce normale e oltre il 90% gli urli o un battito di mani energico nei pressi del sensore. Il sensore acustico è stato una grande mancanza nell'RCX e per questo è stato spesso auto costruito dagli utenti.

Sensore di luminosità

Il sensore di luminosità dell'NXT è stato aggiornato rispetto a quello dell'RCX dando la possibilità di accendere e spegnere il led rosso via software. In questo modo possiamo in ogni momento scegliere se misurare la luce ambientale o quella riflessa. Per quanto riguarda le modalità di utilizzo e le caratteristiche del foto-transistor, non ci sono grandi differenze. Il valore letto dal brick può essere percentuale o raw.
Nell'utilizzo di questo sensore bisogna comunque tenere sempre presente che la sensibilità del fototransistor è maggiore per le frequenze del rosso ed inferiori. Una luce calda come quella di una lampada incandescente darà quindi un valore molto maggiore di luminosità rispetto, ad esempio, ad una luce fluorescente, dando al robot una "visione" molto distorta rispetto a quella umana.

Sensore ultrasonico

Il sensore ultrasonico consente di rilevare la distanza dagli oggetti circostanti. Tra i sensori inclusi nella confezione è sicuramente quello più complesso, tanto da avere bisogno di un microprocessore dedicato al suo interno, che comunica con il brick tramite il protocollo I2C e fornisce direttamente una lettura in cm o pollici. Il funzionamento di questo sensore si basa sul principio del sonar: Esso rileva la distanza degli oggetti emettendo un onda sonora a 40kHz e calcolando il tempo necessario all'onda per raggiungere l'oggetto, riflettersi e tornare indietro. E' possibile misurare distanze in un range tra 0 e 255 centimetri con una precisione di +/- 3cm. [11]

Le pareti ed in generale gli oggetti più grandi danno letture più attendibili mentre oggetti piccoli, con superfici curve o molto fono assorbenti risultano difficili da rilevare e possono dare letture molto fuorvianti. Nel caso si utilizzino più sensori ultrasonici sullo stesso robot ci possono essere errori di lettura ed interferenze dovute ad esempio alla ricezione da parte di un sensore del segnale prodotto dagli altri. Vedremo che nel nostro caso, essendo i sensori posti con un angolo di 120° di distanza non ci sono stati grossi problemi ma in caso contrario avremmo dovuto, ad esempio, effettuare una sola misurazione per volta.

Sensore di temperature

Presente come accessorio per l'RCX è stato prodotto un sensore di temperatura anche per l'NXT per il quale hanno molto migliorato le caratteristiche: il range di temperatura misurabili va ora da -20°C a 110°C e il cilindro di metallo contenente la sonda è stato molto allungato anche per proteggere il sensore dalle alte temperature che può arrivare a misurare. Per questo stesso motivo non troveremo un connettore dietro al sensore ma direttamente il cavo saldato ed isolato.

Sensore di colori

Uno degli ultimi sensori prodotti da LEGO per NXT è il sensore di colori. Questo sensore di fatto è una versione migliorata del sensore di luminosità nel quale il led rosso è stato sostituito da un led RGB. Il case presenta nella parte anteriore tre rientranze ma solo due di queste sono effettivamente destinate rispettivamente al led e al fototransistor mentre la terza è solo per motivi estetici.

La funzione di rilevamento dei colori è semplice: il led RGB viene acceso e cambia ciclicamente colore. Misurando il livello di luce riflessa ad ogni stato del led possiamo capire di che colore è la traccia in quanto se ad esempio la traccia è verde la luce verde verrà riflessa con più intensità di quella rossa o blu. In questa modalità la lettura avrà un valore intero compreso tra 1 (nero) e 6 (bianco).

Le funzioni del precedente sensore di luminosità possono essere assolte semplicemente lasciando il led spento (ambiente) o utilizzando un solo colore del led RGB (ad esempio il rosso).

Un' ultima caratteristica di questo sensore è che il led tri colore può essere utilizzato anche come dispositivo di Output, potendo noi scegliere di accendere un colore fisso.

Sensore di rotazione

Nella serie RCX i sensori di rotazione erano stati un accessorio da acquistare separatamente e, seppur molto utili, andavano ad occupare alcuni dei pochi contatti in ingresso, costringendo a rinunciare ad altri sensori. Nella serie dell' NXT i sensori di rotazione sono stati inseriti direttamente all'interno dei motori. Ogni motore ha quindi il proprio sensore di rotazione che può misurare il numero di rotazioni o la posizione del motore in gradi, con una precisione di +/- un grado [11].

2.2.3 Attuatori

Il kit NXT si avvale, nella sua configurazione base di tre servomotori con integrati un sensore di rotazione e la meccanica di riduzione. Rispetto a quelli che erano i motori della precedente serie vediamo che il progetto è radicalmente cambiato dando al blocco motore anche una funzione strutturale. Se da un lato la meccanica incorporata ha discretamente penalizzato la velocità di rotazione del motore con al massimo 170 rpm, dall'altro lato gli ha dato una coppia massima di ben 50N cm, un ordine di grandezza maggiore del motore a corredo del RIS. Il sensore di rotazione incorporato inoltre, oltre a poter fornire delle letture dirette, viene utilizzato in automatico dal sistema per fornire dei controlli avanzati al motore, consentendo vari tipi di sincronismi ed un controllo PID.

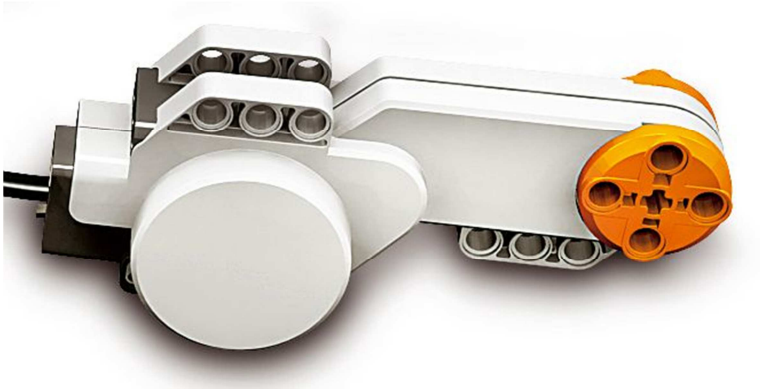


Figura 11: Motore dell'NXT. Questo componente è sia un sensore che un attuatore.

2.2.4 Retrocompatibilità

Grazie all'apposito cavo (LEGO #8528) fornito della confezione il blocco NXT può gestire sia i sensori che gli attuatori della serie precedente.

2.2.5 Componenti aggiuntivi

Con la diffusione dei Mindstorms in numerose applicazioni si è presto iniziata a sentire l'esigenza di ulteriori sensori ed in generale pezzi personalizzati per rispondere alle più disparate esigenze. Si sono sviluppate aziende terze che producono parti meccaniche, sensori aggiuntivi, moltiplicatori di ingressi e veri e propri kit per la creazione di sensori personalizzati da collegare ai Brick.

2.3 Ruote Rotacaster

La prima cosa che salta all'occhio guardando per la prima volta il robot è la strana forma delle tre ruote. Se fossero ruote normali la loro particolare disposizione non consentirebbe al robot altri movimenti che girare su se stesso. Si tratta, invece, di un componente aggiuntivo acquistato presso HiTechnics, una azienda certificata da LEGO che produce sensori ed accessori per LEGO Mindstorms.
Ogni Rotacaster, questo è il loro nome, è formata da due ruote uguali sfasate di 45 gradi. Ogni ruota ha, disposti lungo la circonferenza, 4 cilindri in gomma liberi di ruotare attorno al proprio asse. In totale abbiamo quindi 8 cilindretti disposti alternatamente su due ruote in modo tale che almeno un cilindretto abbia sempre aderenza a terra, consentendo alle ruote di girare come due comuni ruote di gomma, con la differenza che non opporranno alcuna resistenza agli spostamenti paralleli all'asse di rotazione. [28]

Figura 12: Ruota Rotacaster, venduta da HiTechnics.

3 Olonomicità

La posizione di un corpo solido in uno spazio tridimensionale può essere univocamente descritta tramite 6 parametri. Se tutti questi parametri sono liberi da vincoli, diciamo che il corpo ha 6 gradi di libertà altrimenti andremo a diminuirli man mano che vengono aggiunti vincoli fino a scendere a 0, quando il corpo è vincolato in tutti i modi e non ha gradi di libertà. I sei gradi di libertà sono tre di tipo traslativo e tre di tipo rotativo e possono assumere diversi nomi in base all'ambito nel quale ci troviamo, noi li chiameremo x, y, z, rollio, beccheggio e imbardo (in inglese roll, pich, yaw).
Un veicolo con la capacità di muoversi, in base al tipo di propulsione e al modo in cui è meccanicamente interfacciato allo spazio in cui si trova potrà cambiare la propria posizione intervenendo su uno o più di questi parametri. Diremo quindi che ogni veicolo ha un certo numero di gradi di libertà che può controllare.
Chiamando TGDL il numero totale di gradi di liberà di un veicolo e CGDL il numero di gradi che è in grado di controllare, possiamo classificare i veicoli in tre modi: Non olonomico, olonomico e ridondante. [3]

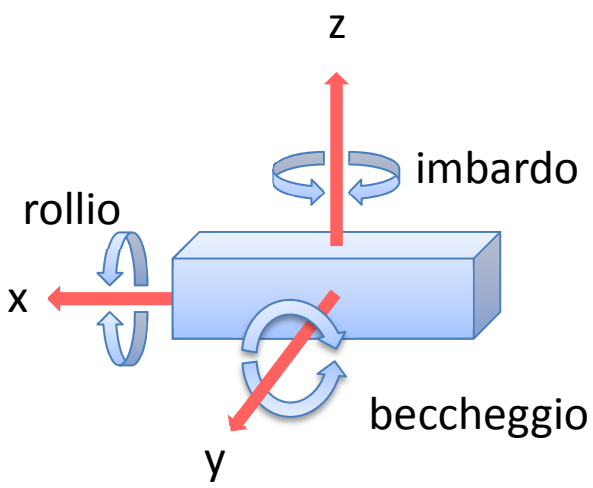


Figura 13: I gradi di libertà di un corpo rigido

Olonomico
Chiameremo olonomico un veicolo nel quale TGDL=CGDL, ovvero nel quale il veicolo può controllare tutti i gradi di libertà di cui dispone.
Il primo esempio che ci può venire in mente è ovviamente l'elicottero che riesce a muoversi in tutti gli assi e ruotare in tutte le direzioni separatamente grazie all'interazione delle due eliche di cui dispone. TGDF=CGDL=6

Il robot progettato in questa tesi è olonomico in quanto, vincolato al piano, ha 3 gradi di libertà (x, y ed imbardo) e grazie all'interazione delle 3 particolari ruote può controllare tutti e 3 questi gradi. TGDF=CGDL=3.
Se vogliamo anche il treno è un veicolo olonomico in quanto è vincolato ad una linea che idealmente gli dà solo un grado di libertà e può scegliere come muoversi lungo questa linea. TGDL=CGDL=1

Non Olonomico

Chiameremo invece non-olonomico un veicolo nel quale TGDL>CGDL, il numero di gradi totali è maggiore del numero di gradi controllabili. Questa è la situazione più comune.
Un'automobile su un piano ad esempio ha tre gradi di libertà, x, y ed imbardo mentre a causa della reazione vincolare del piano, e della rigidità del corpo sono bloccate z, rollio e beccheggio[7].
Di questi tre gradi di libertà il guidatore può controllarne solo due: x ed imbardo[8]. TGDL=3 > CGDL=2
Gli aerei di linea hanno 6 gradi di libertà ma di questi possono solo controllarne 4 in quanto z ed y non possono essere controllati direttamente né con i motori né grazie a flap, coda o alettoni. TGDL=6 > CGDL=4
Altri esempi sono i veicoli di strada, motoscafi, mongolfiera, biciclette, moto,….

Ridondante

Il terzo ed ultimo caso possibile è che il numero di gradi controllabili sia maggiore dei gradi effettivi, TGDL<CGDL. Quest'ultimo caso risulta spesso difficile da immaginare ed in effetti per spiegarlo ci dobbiamo servire di oggetti più complessi di un corpo solido, formati da più segmenti e giunti. Un esempio è il braccio umano.

[7] Stiamo ovviamente considerando un caso semplificato, nella realtà l'automobile non è un corpo rigido, ha degli ammortizzatori e si muove su strade che purtroppo hanno ben poco da spartire con un piano

[8] In realtà la sterzata è una forma di rototraslazione che coinvolge anche l'asse y, visto che il centro della rotazione non corrisponde con il centro della macchina.

4 Descrizione del progetto

Il progetto è quello di costruire un robot olonomico utilizzando i LEGO Mindstorms e successivamente programmarlo con l'architettura behavior based. Si è scelto questo particolare tipo di robot poiché si stavano già svolgendo alcune ricerche e simulazioni al riguardo. Facendo una breve ricerca ci si è resi conto che mancavano le numerose soluzioni canoniche che si possono trovare per il controllo di veicoli a due ruote motrici.

4.1 Ambiente delle prove

Tutte le prove del robot si sono svolte all'interno del laboratorio, avendo sempre cura di non far arrivare sui percorsi di prova luci rasenti che avrebbero messo in difficoltà il sensore luminoso. Come traccia si è utilizzata una stampa di grandi dimensioni di una figura curvilinea chiusa. La maggior parte degli esperimenti si è svolta riutilizzando un percorso già presente nel laboratorio che si è rivelato ben presto troppo piccolo rispetto alle dimensioni del robot che trovava immediatamente la traccia e talvolta arrivava a saltare parti di percorso o a perderlo seppur vi fosse praticamente sopra.

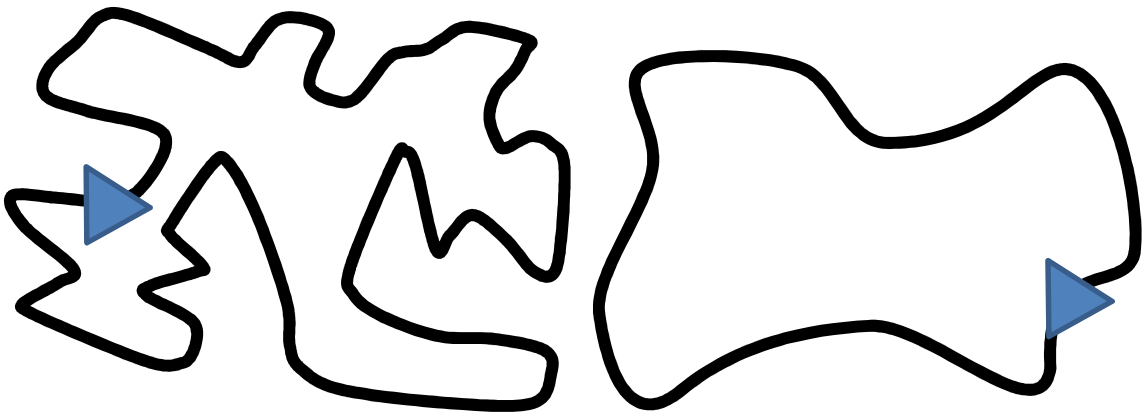


Figura 14: Tracciati del percorso utilizzato per i primi esperimenti e del percorso designato per i successivi test del robot, evitando curve a raggio troppo ridotto.

4.2 Costruzione del modello

Per la costruzione del prototipo ci si è serviti di uno schema fornito dallo stesso costruttore delle ruote a cui sono poi state apportate alcune modifiche. [28]
Il robot parte da una struttura triangolare. Lungo i lati del triangolo sono fissati i tre servo motori collegati direttamente alle ruote mentre al centro della struttura è montato un sensore di luminosità riflessa. Dopo aver svolto le prime prove per capire come far muovere il robot e come fargli seguire la traccia, il progetto è stato rivisto: ai tre vertici della struttura triangolare sono stati aggiunti altrettanti sensori ultrasonici che non erano negli schemi iniziali. I sensori sono stati messi tra una ruota e l'altra, ad un'altezza dal piano di circa 5 cm. Per fare spazio ai sensori aggiunti e ai connettori ad essi annessi, si è sollevata la posizione del brick di alcuni centimetri.

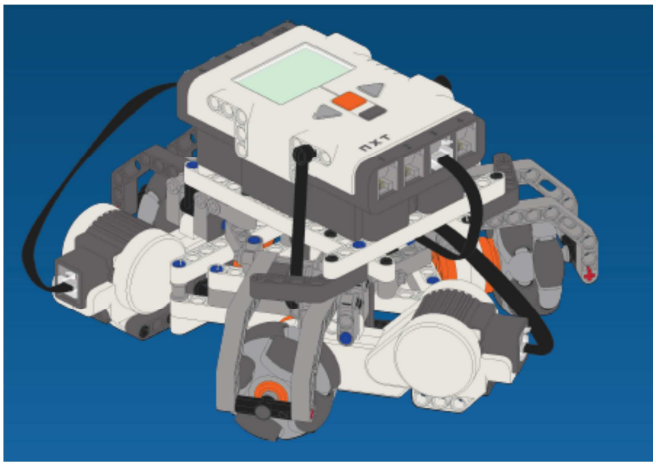


Figura 15: Progetto originale del Rotabot di HiTechnics

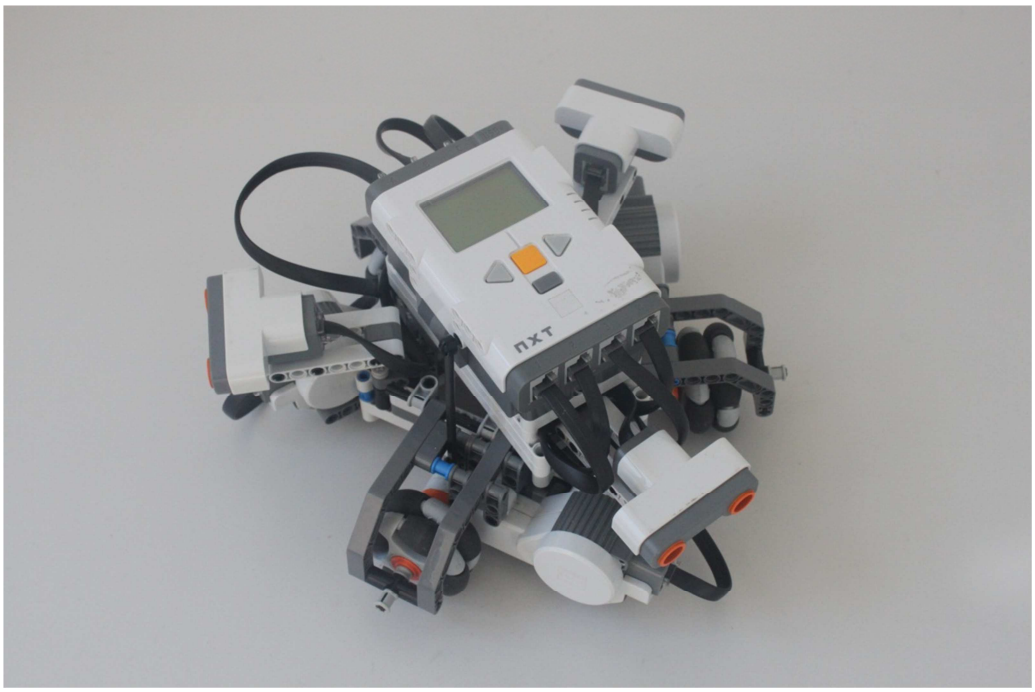


Figura 16: Foto del modello completo di sensori ultrasonici

4.3 Movimento del robot

Il primo ostacolo da affrontare è stato studiare il movimento del robot. Nella maggior parte dei progetti LEGO Mindstorms il robot si appoggia su tre ruote. Solitamente una delle ruote è passiva e per muovere le altre due sono utilizzati due soli motori. La guida è resa possibile dalla differenza di velocità tra la rotazione dei due motori grazie alla quale il robot si sposterà avanti indietro o sterzerà con raggio più o meno ampio. Il tutto è abbastanza intuitivo. Un'altra soluzione è avere un motore dedicato solo alla locomozione su una o più ruote e un motore dedicato solo allo sterzo. Anche questa soluzione è largamente testata.
Nel nostro caso i motori coinvolti nella deambulazione sono tre, giacenti sullo stesso piano e disposti con un angolo di 120 gradi tra loro. Bisognerà coordinare il movimento delle ruote in modo che componendolo il robot si sposti nella maniera desiderata.
La particolare struttura delle ruote rotocaster, consente di avere aderenza nella rotazione ma al tempo stesso di non opporre resistenza per quanto riguarda i moti paralleli all'asse di rotazione. Sfruttando questo comportamento vediamo che ogni ruota offrirà il proprio contributo allo spostamento solo perpendicolare all'asse di rotazione senza opporre alcuna resistenza agli spostamenti paralleli. Abbiamo quindi codificato alcuni movimenti standard per scomporre la rototraslazione.

4.3.1 Traslazione

In uno schema vettoriale possiamo rappresentare le forze agenti dalle ruote sul piano solo perpendicolari all'asse. La somma vettoriale di queste forze andrà a determinare la forza totale agente sul centro del robot che implicherà lo spostamento in quella direzione. Scelta una direzione di default da impostare a 0, abbiamo ricavato la forza che ogni ruota deve generare per ottenere lo spostamento in una qualunque direzione definita in gradi rispetto alla direzione 0.
Chiamando α l'angolo tra la direzione 0 e la direzione verso la quale vogliamo compiere lo spostamento avremo che la velocità di rotazione delle tre ruote sarà rispettivamente $V \cdot sin(\alpha)$, $V \cdot sin(\alpha+120)$, $V \cdot sin(\alpha+240)$ essendo V un valore float da -100 a 100. In questo caso non abbiamo fatto alcun tipo di calcolo quantitativo ma solo qualitativo: avremo, a meno di errori, uno spostamento proporzionale alla dimensione delle ruote e alla velocità di rotazione ma che sarà sicuramente nella direzione da noi decisa, senza ruotare il robot.

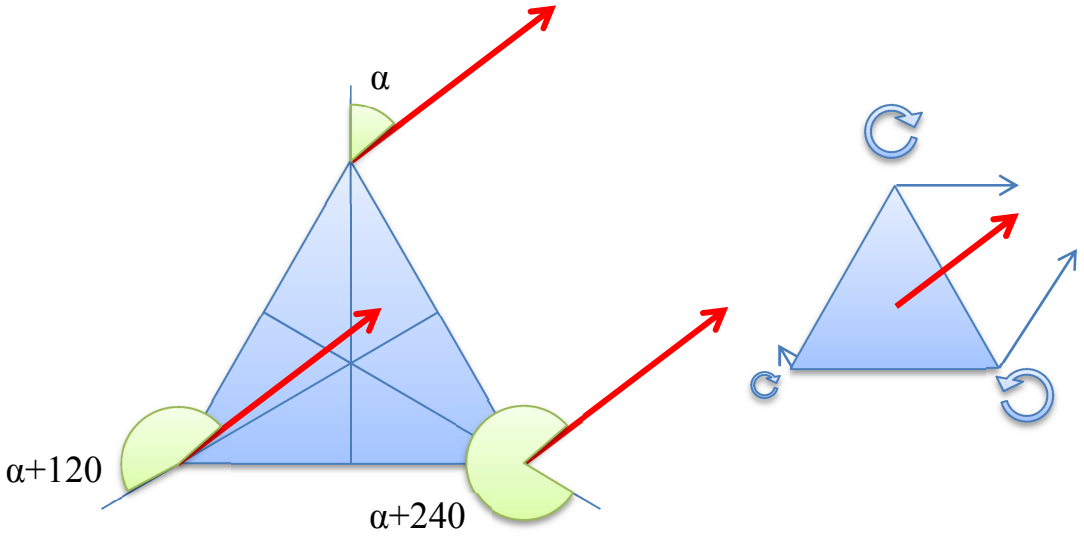


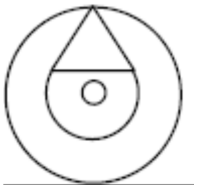
Figura 17: Movimento delle ruote in base alla direzione di movimento designata.

Se volessimo quantificare lo spostamento dovremmo per prima cosa conoscere la dimensione delle ruote e sfruttare i sensori di rotazione all'interno dei motori per conoscere o imporre l'esatta dinamica della rotazione di ogni ruota.

4.3.2 Rotazione

Per far ruotare il robot attorno al proprio centro geometrico che, visto il modello realizzato, corrisponde più o meno al punto in cui si trova il sensore di luminosità, basterà fare girare tutte e tre le ruote alla stessa velocità e nella stessa direzione. In questo caso l'unico dato qualitativo può essere il verso di rotazione. Se vogliamo sfruttare questo movimento dovremo calcolare a quanti giri della ruota corrisponderà un grado di rotazione del robot attorno al proprio centro.
Di fatto questo movimento non è stato sfruttato nel progetto e per questo motivo non si è approfondita la parte quantitativa.

4.3.3 Sterzata a una ruota



Questo movimento è già di tipo rototraslativo. Lasciando ferme due ruote e facendone girare una sola il robot descriverà tre circonferenze: la prima, più esterna, descritta dalla ruota che sta girando, la seconda descritta dal centro del robot e quindi dal sensore e la terza, più interna, sulla quale scorreranno le due ruote rimaste ferme.
Con questa soluzione, pilotando separatamente le altre due ruote otterremo un semplice veicolo non olonomico.

4.3.4 Sterzata a due ruote

Anche questo movimento è di tipo rototraslativo ma, a differenza dell'altro la circonferenza descritta dal centro del robot è molto più piccola implicando quindi un raggio di sterzata molto minore. In questo caso, la circonferenza più esterna è proprio quella descritta dalle due ruote, quella di mezzo è ancora del centro del robot e quella interna è della ruota centrale.
Come vedremo questo tipo di sterzata è stata utilizzata nella modellazione del controller.

4.3.5 Rototraslazione pura

Quest'ultimo movimento è quello più completo. Si tratta di far contemporaneamente ruotare e traslare il robot. La parte traslativa di questo movimento potrà essere svincolata da certe caratteristiche fisiche del robot e semplicemente fornire una direzione al movimento che sarà proporzionale alla velocità di rotazione, alle caratteristiche delle ruote e al tempo. La parte rotativa però, come già visto, deve essere rapportata al robot per sapere di quanti gradi stiamo esattamente spostando il robot. Componendo la traslazione e una rotazione in cui sappiamo esattamente di quanti gradi abbiamo ruotato, possiamo ottenere un movimento completo ed andare a ricoprire tutti i casi in precedenza visti.
In questo caso però il robot dovrà sempre sapere di quanto ha ruotato per cambiare la direzione della traslazione. E' questo il motivo principale per cui si è scelto di evitare tale soluzione e scomporre il movimento: ogni rotazione avrebbe introdotto ulteriori incertezze ed il robot avrebbe avuto sempre bisogno di ricordare o capire in che direzione fosse girato.
Le problematiche che ci hanno fatto escludere la rototraslazione affliggono comunque anche gli altri tipi di movimento: per qualunque operazione avremo sempre imprecisioni nel movimento dovute a perdite di aderenza, deformazioni dei componenti del robot ed imperfezioni del terreno. Quella che vorremmo fosse una rotazione pura comporterà comunque una leggera traslazione del centro di simmetria del robot mentre quella che vorremmo fosse una traslazione pura comporterà una leggera rotazione. Come esperimento si è provato ad impostare una traiettoria circolare e si è visto che già dopo alcuni giri il robot aveva iniziato a ruotare e la traiettoria del centro stesso non aveva seguito la forma di una circonferenza ma di varie ellissoidi. Come vedremo, questi errori di spostamento che in un'architettura deliberativa sarebbero stati grossi problemi, nel controller reattivo che abbiamo progettato in certi casi saranno addirittura del tutto irrilevanti.

4.4 Percezione del robot

Costruito il robot e datagli la facoltà di muoversi, ora bisogna capire come esso può percepire l'ambiente che lo circonda rendendo possibile un'interazione completa.

4.4.1 Sensore luminoso

Al centro del robot abbiamo posto un sensore di luminosità riflessa. Con tale sensore il robot potrà "guardare" a terra e riconoscere eventuali tracce. Mettendolo al centro non abbiamo dato al robot nessuna direzione preferenziale di movimento. L'unico dato che possiamo ottenere dalla lettura di tale sensore è un numero compreso tra 0 e 100 o, nel caso di lettura RAW, tra 0 e 1023. Distanza del sensore, luminosità ambientale, materiale della superficie e colore della traccia andranno ad influenzare il valore misurato.
Per evitare alterazioni significative delle letture tutti i test sono stati fatti sempre con le luci al neon sul soffitto del laboratorio accese e soprattutto senza mai nessuna sorgente di luce rasente il piano. Il piano del test è sempre stato un cartoncino bianco con una traccia nera ad inchiostro o formata con il nastro isolante nero. E' importante tenere presente che non stiamo misurando un valore del SI utilizzando un campione calibrato ma solo dei valori che ci serviranno più qualitativamente che quantitativamente.
La distanza ottimale[9] per ottenere la maggior differenza tra il valore registrato sul bianco e quello della pista nera è di circa 1 cm. A tale distanza e con l'illuminazione al neon del laboratorio abbiamo una differenza tra bianco e nero di oltre sessanta punti percentuali. La misurazione inoltre non avviene su un punto infinitesimo ma su un'area di circa 1 cm^2.
Grazie a questo sensore quindi il robot potrà capire in ogni momento se si trova al centro della traccia nera, se è sullo sfondo bianco o se si trova vicino o parzialmente sulla traccia.

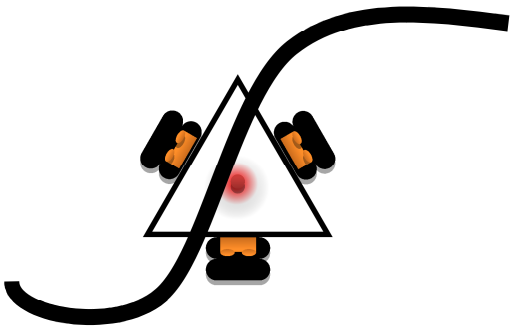


Figura 18: Posizione del sensore ed area di rilevamento

[9] Sono stati fatti dei test [20] con diverse colorazioni e diverse distanze evidenziando a questa distanza risultati più netti.

4.4.2 Sensori ultrasonici

Ai tre vertici del robot abbiamo posto altrettanti sensori ultrasonici. Con questi sensori il robot può percepire la distanza e, per quanto possibile, evitare gli ostacoli.
Come sappiamo l'NXT ha soltanto 4 porte di input ed essendone già presa una dal sensore di luminosità erano ancora disponibili solo tre porte. In aggiunta il laboratorio disponeva al massimo di quattro sensori ultrasonici ed acquistare un moltiplicatore di porte per aggiungere un solo sensore non sarebbe stato conveniente.
Purtroppo i sensori che avevamo a disposizione potevano misurare la distanza con un angolo visale di circa 60 gradi[10]. Ciò significa che siamo riusciti a coprire soltanto 180 sui 360 gradi del robot creando di fatto tre punti ciechi. In aggiunta i tre sensori non sono applicati al centro del robot ma ai suoi vertici, riducendo ulteriormente l'angolo di copertura alle brevi distanze. Questo fatto non è comunque da intendersi come un problema, ma al più un ulteriore specifica tecnica di cui tener conto in sede di definizione del comportamento del robot.

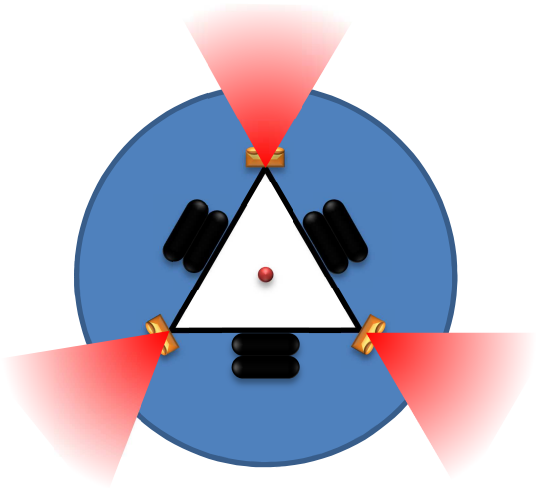


Figura 19: Disposizione dei sensori ultrasonici e raggio di rilevamento.

Le possibili soluzioni sono accettare la presenza di punti ciechi ed evitare per quanto possibile di muoversi in quelle direzioni oppure fare periodicamente ruotare il robot su se stesso di almeno 60° gradi per scambiare i punti ciechi. Volendo, avremmo comunque potuto aggiungere al progetto altri tre sensori ed un moltiplicatore di ingressi per dare una copertura a 360°. In questo caso però avremmo dovuto tener conto di altre problematiche come l'interferenza tra i sensori: mentre nel caso dei tre sensori abbiamo un angolo di 120 gradi tra un sensore e l'altro che, salvo per spazi molto stretti, ci consente di trascurare questo problema; nel caso di sei sensori, se utilizzati contemporaneamente, ognuno di essi riceverebbe non solo la propria onda riflessa ma anche quella trasmessa dagli altri sensori, fornendo al brick letture erronee e fuorvianti. La soluzione più semplice sarebbe accendere e spegnere alternativamente i sensori adiacenti in modo da evitare interazioni tra essi.

[10] Sono stati fatti dei test [19] fissando il sensore ed avvicinando oggetti alle varie angolazioni.

4.5 Programmazione del robot

Per il progetto si è scelto di usare l'IDE BricxCC con il linguaggio NXC: Not eXaclty C.
Come suggerisce il nome stesso, questo linguaggio di alto livello ha una sintassi molto simile al C ma si basa su NBC, Next Byte Codes, un linguaggio assembly.
Si è scelto di programmare il robot in questo modo per numerosi motivi:

1. Stavamo cercando una soluzione che non implicasse la programmazione a blocchi. Dovendo gestire numerosi thread che interagissero tra loro, un ambiente grafico sarebbe risultato complicato da gestire e comunque meno pratico per reperire esempi o altre fonti.
2. Quando il progetto è iniziato, si era parlato di supportare fisicamente delle simulazioni che erano state fatte con Matlab, di conseguenza utilizzare un codice basato su C avrebbe semplificato le cose.
3. A differenza di altre soluzioni, come ad esempio leJos, questa non rendeva necessario cambiare il firmware del brick che quindi è stato solo aggiornato con una delle ultime versioni di fabbrica. Un firmware modificato per sfruttare meglio le caratteristiche di NXC era presente e caricabile direttamente con BricxCC ma non necessario.
4. BricxCC si è rivelato un software molto versatile, ricco di strumenti utili e mediamente personalizzabile.
5. Sia il compilatore di NXC che BricxCC sono open source e non soluzioni commerciali come ROBOTC
6. Infine sia BricxCC che NXC sono progetti vivi, aggiornati ed in evoluzione, sostenuti da una discreta community di utenti. E' possibile inoltre trovare numerosi esempi e frammenti di codice, talvolta risultati utili per la progressione del progetto.

BricxCC è nato inizialmente come IDE per NotQiteC, linguaggio di programmazione dell'RCX ed è stato successivamente aggiornato all'NXT con l'aggiunta dei compilatori per NXC ed NBC.
Attualmente rilasciato nella versione 3.3, Bricx Command Center è un applicativo a 32 bit per ambiente Windows che supporta tutti i prodotti della famiglia Mindstorms. I prodotti RCX, Scout, Cybermaster, e Spybot possono essere programmati con NQC mentre per la linea NXT abbiamo NXC ed NBC. E' possibile decompilare i programmi per NXT in NBC [24] [25]. Il programma è versatile poiché lo si può utilizzare anche per numerosi altri linguaggi a patto di installare separatamente i compilatori necessari. Sono presenti numerosi strumenti che si rivelano molto utili in certe situazioni come lo schermo remoto, il controllo remoto, la possibilità di visualizzare il valore dei sensori o di certe variabili, l'explorer o lo strumento per caricare il firmware modificato.
Prima di iniziare ad utilizzare il brick NXT è comunque stato aggiornato il firmware alla versione 1.26.
L'aggiornamento con il firmware originale, scaricabile dal download center di LEGO Mindstorms, è stato eseguito utilizzando l'apposito tool in G-NXT

4.5.1 Breve accenno ad NXC

Ai fini di una miglior comprensione del codice con cui verranno tra breve implementati i componenti all'interno del controller, vediamo velocemente alcuni dei principali comandi per NXC. Le API e tutta la documentazione necessaria sono comunque disponibili all'indirizzo del progetto BricxCC [25] [26].

NXC si basa sulle task. L'esecuzione del programma parte dal task `main()` da cui poi possiamo lanciare le altre task con:

```
Precedes([nome task]);
```

Il linguaggio come sappiamo è molto simile al C quindi possiamo limitarci a vedere alcune funzioni importanti che coinvolgono l'hardware e che ci sono servite nel progetto.

Sensori

Prima di poter utilizzare un sensore dobbiamo inizializzarlo dichiarando che tipo di sensore è e che tipo di lettura vogliamo.

```
SetSensorType([porta sensore], [tipo sensore]);
SetSensorMode([porta sensore], [modalità sensore]);
```

Per indicare le 4 porte dei sensori si utilizzano

```
IN_1 IN_2 IN_3 IN_4
```

I sensori che abbiamo usato nel nostro progetto sono:

```
SENSOR_TYPE_LIGHT_ACTIVE    SENSOR_TYPE_LOWSPEED_9
```

La modalità è stata sempre RAW

```
SENSOR_MODE_RAW
```

Per indicare il valore di un sensore invece si usano

```
SENSOR_1 SENSOR_2    SENSOR_3      SENSOR_4
```

Motori

I comandi che abbiamo usato per i motori sono:

```
OnFwd([porta motore], [potenza])
OnFwdReg([porta motore], [potenza], [tipo di registrazione]);
```

Le tre porte dei motori e rispettive combinazioni sono:

```
OUT_A    OUT_B        OUT_C OUT_AB  OUT_BC OUT_AC         OUT_ABC.
```

La potenza è un float compreso tra -100 e 100 ed il tipo di registrazione può essere

```
OUT_REGMODE_SPEED      OUT_REGMODE_SYNC
```

In base a ciò che riteniamo più importante, se avere le ruote a velocità costante (il sensore di rotazione controllerà che velocità sia costante e, nel caso qualcosa opponga resistenza aumenterà la potenza) o avere le ruote che si muovono sincronizzate (le due ruote si muoveranno contemporaneamente e se una delle due dovesse essere bloccata si fermerebbe anche l'altra).

Display

Potremo comunicare tramite il display con il comando

```
NumOut([pixel di offset] ,LCD_LINE[riga],[dato]);
```

scegliendo con che distanza dall'inizio dello schermo iniziare a scrivere, in che riga e, ovviamente, che cosa.

Speaker

Ultimo ma non per questo meno importante il comando per far emettere dei suoni al nostro brick

```
PlayToneEx(TONE_[tono], [durata],[volume],[ripeti]);
```

il tono sarà una nota in notazione americana, quindi A5, A7, B5, …

La durata dovrà essere indicata in ms.

Il volume è un numero intero 0-4

La ripetizione del suono è un booleano.

5 Architettura del modello

Entriamo ora all'interno del controller e vediamo quali comportamenti il robot dovrà avere, come saranno strutturati ed implementati nel codice.

5.1 Comportamento del robot

Come deve comportarsi questo robot? Che cosa deve fare? Cosa è più importante?
Qui entra in gioco l'architettura a sussunzione. Piuttosto che elencare tutti i vari comportamenti e cercare di creare un unico blocco atomico che in base alle letture dei sensori esegua tutte le operazioni iniziamo a scomporre il problema partendo dai comportamenti più semplici ed essenziali per poi man mano rendere il modello più complesso.
Di seguito alcuni possibili comportamenti che potrebbero essere implementati da livelli appositi.

0. Evitare il contatto (allontanarsi o in caso di collisione fermarsi)
1. Evitare ostacoli lungo il percorso, qualunque esso sia
2. Saper cercare un percorso
3. Seguire il percorso
4. Mappare il percorso che si sta seguendo
5. Capire se si è già passati da un punto (ovvero orientarsi lungo il percorso)
6. Esplorare attorno al percorso (ex punti d'interesse etc)
7. Saltare parti di percorso o trovare strade alternative
8. Muoversi in libertà dentro al percorso nei limiti degli errori.

Nel progetto siamo arrivati ad implementare fino al terzo livello. Dal quarto livello però ci saremmo trovati di fronte ad altri problemi, non tanto tecnici, quanto concettuali.
Come detto all'inizio seppur ci sia chi ha dato delle definizioni nette per l'architettura behavior based, non tutti i pareri sono concordi. Il livello 4, infatti, implicherebbe una rappresentazione simbolica del mondo che poi sarebbe utilizzata dai successivi livelli ma proprio questo renderebbe il vari livelli oltre il quarto assumibili al tipo deliberativo. E' però Brooks stesso ad inserire in un architettura[11] behavior based livelli con funzioni simili a quelle appena descritte e per ciò, a meno che non si possa creare una mappa rimanendo reattivi ed usarla in modo reattivo, stiamo dicendo che il behavior based può contenere dei blocchi deliberativi o, al più, che il behavior based è esso stesso un controller ibrido.

[11] Riferimento [1] pag 13 par 4.3

5.1.1 Evitare il contatto – Livello 0

Per prima cosa, per proteggere sé stesso e gli altri oggetti o le persone, un robot deambulatore deve evitare in ogni modo gli urti. Prendiamo quindi questo comportamento come livello 0. Avremo un robot che in ogni momento si limiterà ad evitare gli urti allontanandosi da oggetti in avvicinamento e bloccando tutti i motori nel caso qualche ostacolo entri nella distanza di contatto. Questo livello impedirà soprattutto che il robot vada a sbattere contro il muro e cerchi di proseguire la corsa, andando a sovraccaricare i motori o facendo slittare le ruote.
Vediamo quali blocchi saranno necessari a realizzare questo livello.

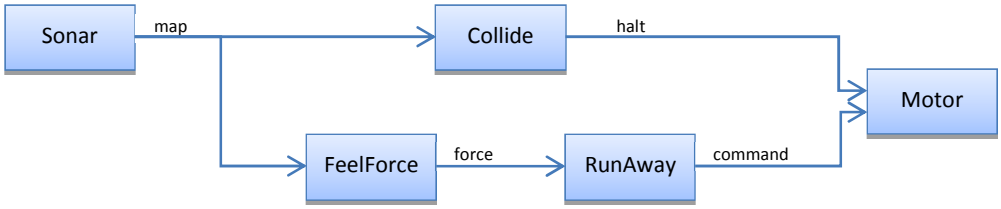


Figura 20: Grafo del livello 0

Motor

Per prima cosa ci servirà un blocco che vada a controllare gli attuatori. Chiameremo Motor questo blocco che si dovrà occupare di gestire in maniera esclusiva i motori. Teoricamente ogni blocco potrebbe prendere in input il valore dei sensori e comandare gli attuatori ma, dividendo i compiti e dedicando alcuni blocchi solo alla gestione dei motori ed altri solo alla gestione degli input, eviteremo molti problemi di gestione.

Sonar

Successivamente introduciamo il blocco Sonar che gestirà gli input dai sensori ultrasonici. Nel nostro caso, siccome abbiamo solo tre sensori ultrasonici che non si disturbano a vicenda e che forniscono continuamente dei dati senza problemi di tempistica o sovrascrittura da gestire, questo blocco risulterà non necessario. Per ora, per rimanere fedeli al modello di Brooks e dare uno schema più generale lo introdurremo ugualmente: dopotutto se avessimo avuto 6 sensori o se questi fossero stati in qualche modo da sincronizzare o leggere solo a determinati intervalli di tempo, questo blocco sarebbe stato indispensabile.

Collide

Vediamo ora il primo blocco che effettivamente faccia reagire il nostro robot a determinati stimoli. Il blocco Collide ha la semplice funzione di bloccare i motori nel caso il robot si avvicini ad un qualunque oggetto violando la soglia minima di distanza per evitare la collisione. Questo blocco è molto importante in quanto, durante tutta la sperimentazione garantirà che il robot non sbatta mai e che arrivato in prossimità di un muro si fermi e non cerchi di spostarsi oltre in quella direzione.

In input avrò i valori dei sensori acquisiti dal blocco sonar ed in uscita avrò semplicemente un segnale di halt da inviare a Motor. Starà ovviamente a Motor saper gestire in modo appropriato questo halt, dandogli priorità massima.

FeelForce

Il blocco sonar, come abbiamo detto ci fornisce solo i valori dei sensori. Feelforce si dovrà occupare di leggere ed interpretare tali valori per fornire un segnale force. Il robot quindi grazie a questo blocco percepirà una forza, come se gli oggetti producessero una azione repulsiva nei suoi confronti.
In progetti behavior based avanzati i movimenti del robot sono stati definiti addirittura con un campo vettoriale.

RunAway

Quest'ultimo blocco si occuperà di far fuggire il robot dagli ostacoli in movimento o, se possibile, di mantenerlo ad una distanza di sicurezza da muri ed altri oggetti. In ingresso prenderà il segnale dal blocco FeelForce mentre in uscita manderà dei comandi a Motor. Per assolvere le sue funzioni questo blocco dovrà semplicemente fare muovere il robot in base alla forza percepita, sarà un comportamento emergente il fatto che il robot fugga dagli ostacoli o cerchi sempre di fermarsi in un posto abbastanza lontano dagli ostacoli.

Come abbiamo visto studiando le percezioni del robot, i sensori ultrasonici che abbiamo posto ai tre lati non riescono a monitorare tutta l'area attorno al robot ma lasciano dei punti ciechi. Per quanto riguarda il rilevamento delle pareti questo non è un problema in quanto, anche nel caso peggiore in cui il robot si stia dirigendo verso una parete rivolto con uno dei punti ciechi, i sensori posti ai lati riusciranno a rilevarla. Il problema diventa invece critico se ci troviamo ad avere oggetti di minori dimensioni o spigoli minori di 120 gradi: in questo caso è impossibile che i sensori rilevino l'ostacolo.

Nel corso di questo studio non si è voluto implementare un comportamento che tenesse monitorati anche i punti ciechi per non complicare troppo il progetto. Si era comunque studiata la possibilità di far continuamente ruotare il robot su se stesso o alternare ciclicamente due posizioni che consentissero di coprire anche i punti ciechi. Il motivo principale per cui non si fatto è che gli errori di spostamento derivanti da un movimento continuo sarebbero stati molto rilevanti, tali da non poter mai considerare il robot fermo.
L'implementazione attuale non esclude comunque la possibilità di un futuro aggiornamento modificando il blocco RunAway e, come vedremo tra poco, i blocchi che andranno a sopprimerne l'uscita.
Vedremo, in definitiva che il problema dei punti ciechi è stato parzialmente risolto con alcuni semplici accorgimenti come mantenere una buona distanza da qualunque ostacolo rilevato e cercando di prediligere direzioni di spostamento coperte dai sensori.

5.1.2 Evitare o allontanarsi da ostacoli – Livello 1

Ora che il nostro robot è in grado di evitare gli urti e fuggire dagli ostacoli vogliamo dargli la possibilità di muoversi autonomamente. Per fare questo il nuovo livello dovrà fornire una direzione di movimento e, tenendo conto anche degli ostacoli, decidere come far muovere il robot.

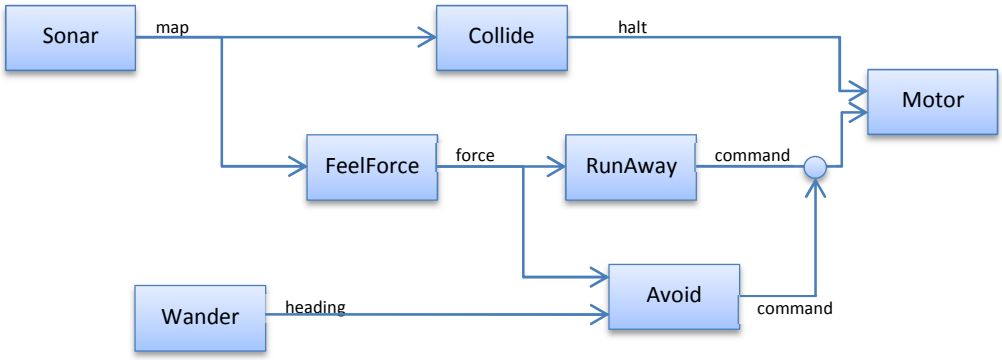


Figura 21: Grafo dei livelli 1 e 0.

Wander

Il primo blocco che andremo ad aggiungere sarà il Wander che molto semplicemente fornirà delle direzioni random al robot. Di fatto questo blocco servirà solo in questo livello ed anticipiamo che il suo segnale di uscita sarà praticamente sempre soppresso dai blocchi dei livelli superiori.

Avoid

Dobbiamo ora gestire il robot in modo che continui a mantenere i comportamenti del livello precedente ma al tempo stesso inizi muoversi autonomamente nella direzione fornitagli dal Wander.

La maniera più elegante per fare questo sarà considerare la forza percepita da FeelForce e la direzione fornita da Wander come due vettori omogenei e sommarli vettorialmente. Per evitare loop e stalli dovremo anche avere alcuni piccoli accorgimenti come rendere uno dei due vettori prevalente, trascurare la forza se inferiore ad una soglia ed inserire dei valori random che rendano il comportamento leggermente diverso ogni volta.

Come vedremo nel progetto il comportamento scelte sarà leggermente differente ma ugualmente efficace.

5.1.3 Trovare un percorso da seguire – Livello 2

Abbiamo un robot che può vagare liberamente per la stanza, che non va "mai" a sbattere e che fugge da eventuali ostacoli. Adesso possiamo iniziare a comandargli delle direzioni per svolgere compiti più articolati. Vogliamo che il nostro robot giri per la stanza alla ricerca di un traccia da rilevare con il sensore di luminosità. Aggiungiamo quindi tre nuovi blocchi.

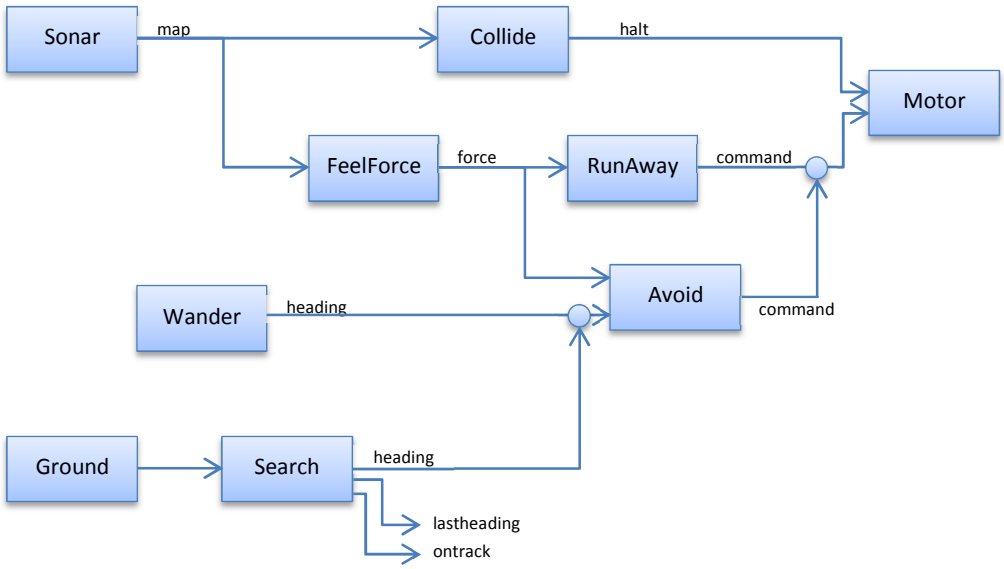


Figura 22: Grafo del secondo livello, integrato a quelli inferiori.

Ground

Come per Sonar questo blocco viene introdotto solo per motivi formali ma possiamo senza grossi problemi bypassarlo e fornire direttamente ai blocchi interessati il valore del sensore.

Search

Questo blocco di fatto costituisce il vero fulcro del livello 2. Se il valore rilevato dal sensore di luminosità è sotto una certa soglia, significa che ci troviamo sulla traccia, altrimenti, essendo il bianco al di fuori della traccia più riflettente del nero, avremo un valore al di sopra della soglia.

Quando il robot si troverà sulla traccia starà fermo ma, se la dovesse perdere, inizierebbe a girare per il piano alla ricerca di tale traccia.

Per fare questa ricerca potremmo limitarci a far girare casualmente il robot finché una traccia non viene trovata oppure definire dei pattern di ricerca per essere, ad esempio, certi di trovare la traccia o trovarla il più velocemente possibile.

La navigazione casuale è stata presto sostituita da quadrati concentrici e successivamente da una spirale con inviluppo lineare o esponenziale.

In questo stadio l'olonomicità del robot è stata un vantaggio poiché abbiamo potuto decidere come far vagare il robot semplicemente scegliendo una velocità ed una direzione dello spostamento che variavamo istantaneamente per creare i vari pattern. In un robot non

olonomico sarebbe stato molto più complicato realizzare dei pattern poiché avremmo avuto dei vincoli come il minimo angolo di sterzata.

Da questo momento quindi il robot cercherà sempre di trovarsi sulla traccia e, nel caso la perda, ricomincerà a cercarla. Essendo l'autoconservazione del robot un istinto fondamentale (almeno è quello che abbiamo deciso) anche nel caso si trovi sulla traccia, se un oggetto si avvicina troppo, esso lascerà la traccia per fuggire, salvo poi ricominciare a cercarla, mantenendo le distanze dall'ostacolo. Anche questo non è un comportamento scritto ma emergente.

Possiamo notare che dal blocco Search esce un valore che non viene attualmente utilizzato e che quindi suppone già la presenza di un livello successivo. Come vedremo inoltre lo stesso blocco prenderà in ingresso un segnale che funzionerà a tutti gli effetti come un enable e gli verrà fornito da un livello superiore.
La verità è che abbiamo generato questo schema pensando già ai comportamenti futuri e come anche Brooks ha fatto, abbiamo violato alcune delle regole da lui stesso redatte per semplificare la realizzazione[12].
Potremmo comunque ovviare a questi problemi e rendere il progetto formalmente più coerente con poche modifiche come la sussunzione dell'input al blocco Search (che se il sensore luminoso è stato gestito da un livello dedicato risulterà semplice) o dividendo il blocco Search in più sotto blocchi in modo da poter accedere ai segnali interni, come è per FeelForce e RunAway.
Potremmo, ad esempio, mettere un blocco CheckTrack che continua sempre a verificare se il robot è sulla traccia ed in caso contrario abilita il blocco SpiralSearch ed aggiungere a valle un altro blocco che richieda per vari scopi i segnali che poi andranno forniti al livello successivo. Restano comunque soluzioni non particolarmente utili, generate a posteriori per un vezzo teorico.

[12] Riferimento [2] Pag 7 e 8: vediamo che legup in fig. 3 riceve tre segnali in input, uno dei quali deriva da alpha collide che è un livello superiore, in quanto come vediamo da fig. 4 nel quale non è presente.

5.1.4 Seguire il percorso – Livello 3

Ricapitolando abbiamo costruito un robot che vuole rimanere sempre sulla traccia e, se la perde, inizia a cercarla fino a quando non la ritrova, evita gli ostacoli e fugge agli oggetti in avvicinamento. Nella situazione di quiete il nostro robot sarà quindi sulla traccia e ad una distanza sufficiente da altri ostacoli, senza che nessuno si avvicini. Andiamo ora ad aggiungere un nuovo comportamento.

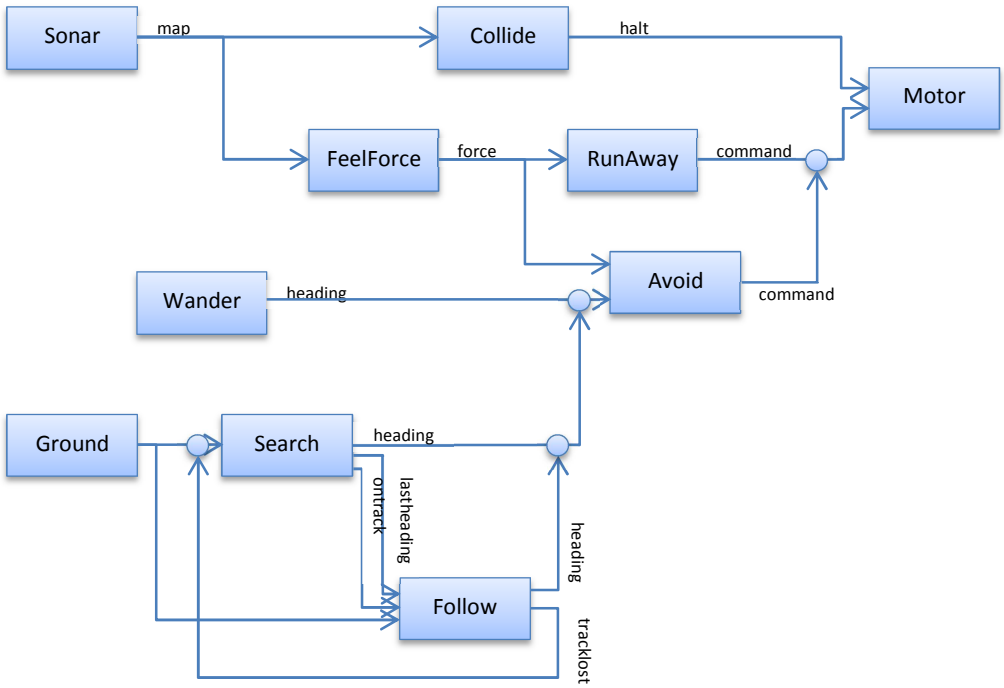


Figura 23: Grafo completo, comprensivo del terzo livello.

Follow

In questo caso c'è identità tra il blocco ed il livello. Vogliamo che il robot, una volta agganciata la traccia inizi a muoversi lungo questa. Per prima cosa dobbiamo andare in qualche modo ad escludere il comportamento di ricerca del percorso: siccome seguire la traccia potrebbe spesso significare perderla per brevi istanti e ritrovarla se il precedente comportamento rimanesse attivo, il robot inizierebbe subito a cercare la traccia andando ad ostacolare il nuovo comportamento. Faremo questo, come abbiamo accennato poco fa, mandando direttamente un segnale al blocco Search, che smetterà di cercare aspettando che la traccia venga persa.

A questo punto dobbiamo trovare la migliore strategia per seguire la traccia una volta che è stata agganciata.

Il nostro robot si può muovere in tutte le direzioni, ruotare su se stesso e sterzare ad 1 o 2 ruote. Purtroppo però dispone di un solo sensore e dovremo trovare un modo per sfruttarlo al meglio.

Arrivando alla traccia il robot ricorderà sempre in che direzione si stava spostando quando l'ha raggiunta. Quando inizieremo a muoverci, supporremo di aver raggiunto la traccia

perpendicolarmente, e quindi sceglieremo sempre l'ultima direzione fornita dal Search, variata di 90°.
A questo punto abbiamo sviluppato tre possibili tattiche per seguire il percorso.

1- Prima soluzione: Viene scelta, in base alla direzione in cui pensiamo si sviluppi la traccia, una direzione principale per il robot, che verrà mantenuta per tutto il percorso. Il robot si muoverà in direzione 0°, 120° o 240°: una delle tre direzioni preferite e coperte dal sensore ultrasonico. Riguardando la formula della traslazione noteremo che in questo modo ogni volta una delle tre ruote rimarrà ferma. Abbiamo quindi appena vincolato uno dei gradi di libertà rendendo il nostro robot non olonomico. L'inseguimento della traccia a questo punto funzionerà cosi: il robot si muove sempre nella direzione predefinita. Se perde la traccia inizia a cercarla sterzando con le due ruote a destra e a sinistra aumentando sempre l'ampiezza della sterzata. Quando riaggancia la traccia ricomincia a muoversi in quella direzione. Se per qualche ragione non dovesse riuscire a ritrovare la traccia[13], dopo un certo tempo o un certo numero di tentativi ricomincerà a cercare il percorso lasciando fare al livello sottostante.
2- Seconda soluzione: Partiamo in maniera molto simile alla prima, scegliendo la direzione principale ma in questo caso teniamo costantemente monitorata la traccia e, appena il valore cambia, senza aver perso la traccia, ci spostiamo leggermente a destra e a sinistra come nel caso precedente per trovare un valore migliore. In questo caso non aspettiamo di aver perso la traccia per cercare ma vogliamo sempre il valore migliore. Se perde la traccia ci comportiamo esattamente come nel caso precedente.
3- La terza soluzione che abbiamo preso in analisi cercava di mantenere, durante l'inseguimento della traccia, le caratteristiche olonomiche del robot. Scelta la direzione principale il robot inizia a muoversi a destra e a sinistra della traccia contando per quanto tempo resta a in una o nell'altra posizione. Andando a confrontare i due tempi decide come cambiare la propria direzione principale cercando di mantenere sempre i due valori simmetrici e quindi la traccia al centro.
4- Un'ultima soluzione che si era presa in considerazione poteva essere sempre di tipo olonomico ed implicava iniziare subito a muoversi nella direzione principale e appena venisse persa la traccia cominciare una sorta di ricerca a spirale in miniatura.

Il terzo caso, seppur inizialmente sembrasse una soluzione molto valida è stato scartato abbastanza velocemente a causa della complessità e della scarsa robustezza rapportati ai possibili vantaggi in termini di velocità e precisione: il movimento lungo il percorso sarebbe stato molto lento e ci sarebbero sicuramente stati numerosi problemi dovuti agli errori di spostamento e lettura.

[13] Ad esempio perché è arrivato ad un vicolo cieco, si è spostato troppo o, per evitare un ostacolo, si è dovuto proprio allontanare dal percorso.

Anche il quarto caso è stato scartato abbastanza velocemente principalmente perché avrebbe portato il robot a ballare attorno alla traccia: ogni volta che l'avesse persa, al suo ritrovamento non avrebbe avuto una direzione ben precisa da seguire ed avrebbe girato ciclicamente nei pressi dello stesso tratto di percorso, senza effettivamente seguirlo.
Sono quindi stati subito deprecati i movimenti olonomici. Probabilmente se avessimo avuto più sensori di luminosità avremmo potuto sfruttare meglio l'olonomicità del robot, rilevando di volta in volta in che direzione fosse la traccia e come fosse meglio seguirla.
Rimaste valide solo le prime strategie, tra le due si è scelto di sviluppare la prima in quanto più semplice ed intuitiva seppur la seconda fosse, almeno teoricamente, più efficace.

5.1.5 Possibili livelli successivi

A seguito di questi livelli potremmo aggiungerne altri, come abbiamo elencato all'inizio di questa sezione, per dare al robot comportamenti sempre più complessi. A questo punto però c'è un problema: per rendere più complesso il sistema dobbiamo aggiungere altri blocchi e segnali che però iniziano ad avvicinarci al mondo dei controller deliberativi. Se ad esempio volessimo che il robot mappasse il percorso per poi magari utilizzarlo anche nei livelli successivi dovremmo utilizzare una rappresentazione simbolica del mondo, un modello ricostruito all'interno del quale dovremmo pianificare le rotte da seguire per ottimizzare gli spostamenti.

5.2 Implementazione del controller

L'implementazione in codice del controller ha seguito lo schema a livelli che abbiamo appena prodotto anche se, come vedremo sono stati necessari alcuni accorgimenti.
In generale ogni blocco è stato implementato con una task. Per portare il segnale da un blocco all'altro abbiamo usato delle variabili globali e i blocchi S ed I sono stati portati all'interno di alcune task.

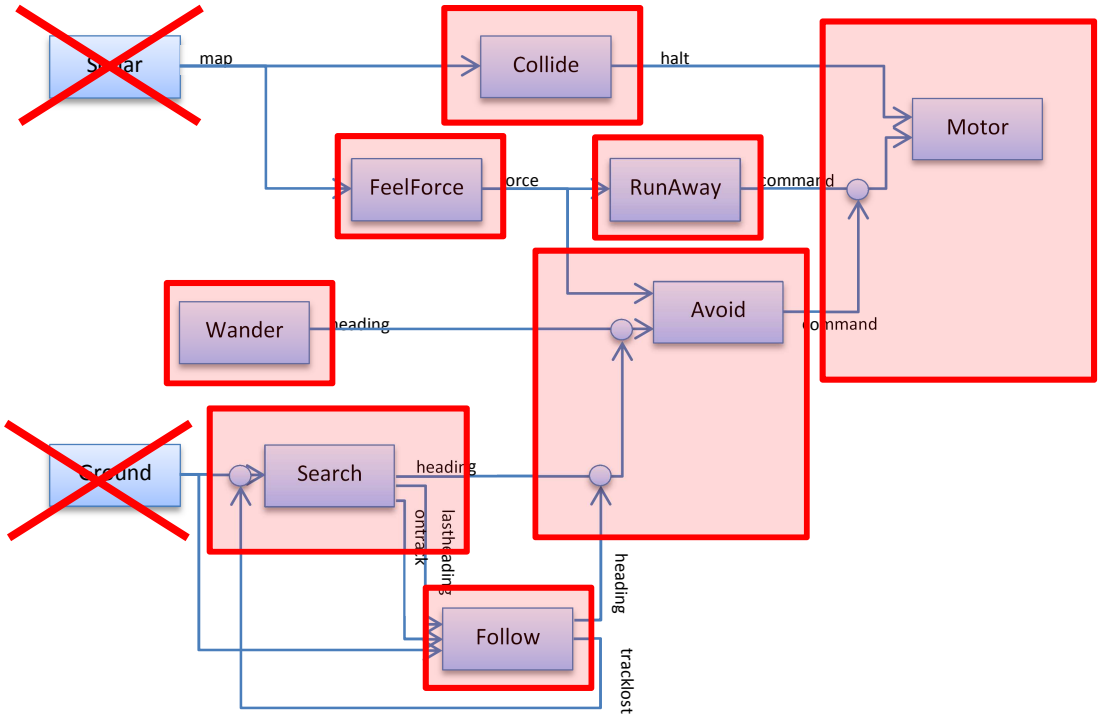


Figura 24 Schema del controller. I blocchi rossi rappresentano le task che sono state programmate, talvolta inglobando più componenti del grafo dei comportamenti.

5.2.1 Collegamenti

Come accennato tutti i collegamenti sono delle variabili. Per segnali semplici come halt ci siamo serviti di una semplice variabile intera. Per segnali più complessi come la direzione o i comandi per i motori invece è stato necessario creare una struttura dati che come vedremo ci consente sia di passare informazioni che di disabilitare il segnale stesso.

```
struct datalink{
  int mode;
  float speed;
  int direction;
  int rotation;
  int time;
  byte weel;
};
```

Ogni collegamento che nel grafo fosse stato interrotto da un blocco S è stato distinto tra segnale a monte e a valle del blocco.

```
int halt=0;
datalink force;
datalink heading;
datalink heading_wander;
datalink heading_spiral;
datalink heading_follower;
datalink command;
datalink command_run;
datalink command_block;
int lastheading=0;
int ontrack=0;
int tracklost=1;
```

Si aggiunge anche il blocco dei define poiché sono stati usati nel codice

```
#define MotorA OUT_A
#define MotorB OUT_B
#define MotorC OUT_C
#define AllMotors OUT_ABC
#define Sonic1 IN_1
#define Sonic2 IN_2
#define Sonic3 IN_3
#define Ground IN_4
#define Path SENSOR_4
```

5.2.2 Livello 0

Il blocco Sonar è stato completamente omesso in quanto i sensori non avevano bisogno di particolari controlli ed usare un task solo per costruire un eventuale vettore di 3 valori da passare così com'era costituiva solo uno spreco di risorse.

Dopo aver inizializzato i sensori è stato molto più semplice farne leggere il valore direttamente al blocco interessato.

Il blocco Collide è stato implementato nella task `AvoidCollision` e fa esattamente quello che ci si aspetta. Possiamo regolare la distanza critica con la variabile intera `collision`, inizializzata nel `main`.

```
task AvoidCollision(){
  while(true){
  if(  SensorUS(Sonic1)<collision||
    SensorUS(Sonic2)<collision||
    SensorUS(Sonic3)<collision  ){
      halt=1;
    }else{
      halt=0;
    }
  }
}
```

Il blocco FeelForce è stato implementato con l'omonima task. La generazione del vettore force, avendo noi a disposizione solo tre sensori, è stata semplificata molto, evitando di calcolare trigonometricamente la direzione e l'ampiezza della forza ma semplicemente scegliendo un valore di forza proporzionale alla distanza e impostando come direzione quella dell'ampiezza maggiore. Essendo tutto all'interno di un `while` che ogni volta aggiorna i valori, di fatto avremo comunque una somma vettoriale discreta del vettore spostamento. Come nel blocco precedente possiamo regolare la distanza regular alla quale il robot inizia a sentire una forza repulsiva verso gli ostacoli.

```
task FeelForce(){
  float a,b,c,rate,intensity;
  while(true){
    a= SensorUS(Sonic1);
    b= SensorUS(Sonic2);
    c= SensorUS(Sonic3);
    if (a<regular&&a<=b&&a<=c){
      rate= (a/regular)*100;
      intensity=100-rate;
      force.speed=intensity;
      force.direction=0;
    }else if (b<regular&&b<=c){
      rate= (b/regular)*100;
      intensity=100-rate;
      force.speed=intensity;
      force.direction=240;
    }else if (c<regular){
      rate= (c/regular)*100;
      intensity=100-rate;
      force.speed=intensity;
      force.direction=120;
    }else{
      force.speed=0.0;
      force.direction=0;
    }
  }
}
```

Vediamo ora come è stato realizzato il blocco RunAway, il primo la cui uscita è destinata ad essere soppressa dai blocchi sottostanti. Il blocco non darà in uscita il segnale `command` ma `command_run`, il proprio segnale. Tra un attimo vedremo come questo segnale viene gestito e perché viene posto il mode a 0.

```
task RunAway(){
  while(true){
    if(abs(force.speed)>15){
      command_run.mode=0;
            command_run.direction=force.direction+Random(30);
      command_run.speed=force.speed;
    }
  }
}
```

Il blocco Motor non è stato implementato dalla task `Move` che però ha di fatto inglobato anche i vari blocchi di sussunzione del segnale command. Guardando meglio questa task vediamo che è divisa in due segmenti. La parte bassa svolge il compito che avrebbe svolto Motor, andando a chiamare, in base alla modalità di movimento selezionata, una delle apposite procedure. La parte alta invece fa da arbitro gestendo i segnali che arrivano lungo `command` e dando la precedenza a quelli dei livelli più alti.
In questo caso dovremo modificare la task ogni volta che aggiungiamo un segnale su `command`. Nononstante questo, non stiamo infrangendo le regole dell'architettura a sussunzione: stiamo modificando una parte della task che svolge la funzione dei blocchi S[14] ma la parte che implementa Motor rimane invariata. Avremmo potuto anche separare queste due task ma così facendo non sarebbero state sincronizzate e ci sarebbe stato il rischio di cambiare dei valori in momenti sbagliati.
Tornando all'arbitro vediamo quindi che prima di tutto vengono inizializzate le modalità dei segnali a -1 così che vengano annullati. L'arbitro dovrà ovviamente essere caricato prima dei blocchi arbitrati. Mano a mano che aggiungeremo i blocchi dei livelli successivi, questi inizieranno ad inviare dei comandi al `Move`, cambiando il loro mode da -1 a quello dello spostamento voluto. A parte l'inizializzazione dell'arbiter nessun altro agente può impostare il mode di un segnale a -1 se non la task a cui tale segnale appartiene.

[14] Nel progetto non abbiamo usato blocchi di tipo I. Si sarebbe comunque potuto realizzare creando una task che disaccoppiasse il segnale inviato ad un blocco da quello effettivamente ricevuto. In ingresso dovrebbe avere il segnale ed un enable dai livelli superiori mentre in uscita fornirebbe un valore nullo o una replica del segnale in base al valore di enable.

```
task Move(){

  //INITIALIZE
  command_block.mode=-1;
  command_run.mode=-1;

  while(true){
  //ARBITER
    if(command_block.mode!=-1){
      command=command_block;
    }else if(command_run.mode!=-1){
      command=command_run;
    }

  //MOTOR
    if(halt==0){
      if(command.mode==0){
        shift(command.speed,command.direction,0);
      }else if(command.mode==1){
        rotate1w(command.speed,0,command.weel);
      }else if(command.mode==2){
        rotate2w(command.speed,0,command.weel);
      }else if(command.mode==3){
        rotate(command.speed,0);
      }
    }else{
      Off(AllMotors);
    }
  }
}
```

5.2.3 Livello 1

Questo livello introduce i due blocchi Avoid e Wander. Wander fa ciò che ci aspettiamo e dà in uscita il segnale `heading_wander`.

Come possiamo immaginare, siccome anche `heading` del wander verrà presto sovrascritto dal segnale `heading` del blocco Search, il blocco che andrà a ricevere `heading` dovrà contenere un altro arbitro.

Vediamo quindi come è implementato Avoid, contenuto all'interno della task `Block`.

```
task Block(){

  //INITIALIZE
  heading_wander.mode=-1;
  heading_spiral.mode=-1;
  heading_follower.mode=-1;

  while(true){
  //ARBITER
    if(heading_follower.mode!=-1){heading=heading_follower;}
    else if(heading_spiral.mode!=-1){heading=heading_spiral;}
    else if(heading_wander.mode!=-1){heading=heading_wander;}

  //AVOID
    if(abs(force.speed)>15){
      command_block.mode=0;
      command_block.direction=force.direction+Random(30);
      command_block.speed=force.speed;
      float lastforce=force.speed;
      long t0=CurrentTick();

  while(CurrentTick()<t0+Random(5000)&&force.speed<lastforce){}
    }else{
      command_block=heading;
    }
  }
}
```

Il comportamento di Avoid sarà molto simile a quello di RunAway ma vi si aggiungerà un `if` per gestire la forza.

Bisogna notare che grazie all'initialize che c'è in entrambi gli arbitri, anche se all'avvio non sono presenti i livelli superiori, che potremmo ad esempio avere commentato dal `main`, il sistema funziona ugualmente, prendendo semplicemente quei comportamenti ma lasciando funzionare quelli dei livelli inferiori.

5.2.4 Livello 2

Iniziamo ora la ricerca della traccia. Il blocco Search è stato implementato da `SpiralSearch` che come si può intuire cercherà la traccia descrivendo una traiettoria spiraliforme. Il verso della spirale è legato ad un valore random per evitare che il robot vada in loop.

Oltre all'input del sensore luminoso (`path`) abbiamo la variabile tracklost che, inizializzata ad 1, per ora non viene modificata da nessun blocco. In uscita SpiralSearch, prima di fermarsi, comunica di avere trovato la traccia (`ontrack=1`) e con che direzione vi è giunto.

Come viene avviata la task inizia a far percorrere al robot una spirale e si ferma non appena il valore del sensore luminoso scende sotto una certa soglia, impostata nel `main`. Se il robot è sulla traccia resta fermo ma appena viene allontanato da questa ricomincia a cercarla.

```
task SpiralSearch(){
  int i=0;
  long t0;

  while(true){
    heading_spiral.mode=0;
    int rnd= sign(100-Random(200));
    i=0;
    if(Path<=threshold){ontrack=1;}
    while(Path>threshold&&tracklost==1){
      ontrack=0;
      t0=CurrentTick();
      i=i+10;
      heading_spiral.direction=rnd*i;
      heading_spiral.speed=30.0;
      while(Path>threshold && CurrentTick()<(t0+i)){
      }
      if(Path<=threshold){
        PlayToneEx(TONE_F5, MS_500,VOL,FALSE);
        Off(AllMotors);
        heading_spiral.speed=0.0;
        lastheading=heading_spiral.direction;
        ontrack=1;
      }
    }
  }
}
```

5.2.5 Livello 3

Vediamo ora Follow, l'ultimo blocco che abbiamo implementato con la task `PathFollow`.

Come si nota questo comportamento è decisamente più articolato dei precedenti.

Per prima cosa, se il robot non pensa di trovarsi sulla traccia (`ontrack`=0), questo blocco si limita a disabilitare la propria uscita e dichiarare la traccia persa (`tracklost`=1), così che il blocco Search si attivi e possa inviare il proprio output ad Avoid.

```
task PathFollow(){
  while(true){
    if(ontrack==1){
      .
      .
      .
    }else{
      heading_follower.mode=-1;
      tracklost=1;
    }
  }
}
```

Il valore di `ontrack` viene inizializzato a 0 ed è modificato solo da `SpiralSearch`.

Quando la traccia viene trovata il blocco sceglie in che direzione muoversi per seguirla ed entra in un loop fino a quando non perde la traccia senza riuscire a ritrovarla.

```
    if(ontrack==1){
      heading_follower.mode=0;
      tracklost=0;
      Wait(3000);
      int dir=lastheading+90;
      while(dir>360){dir=dir-360;}
      while(dir<0){dir=dir+360;}
      if(dir>0&&dir<=120){
        heading_follower.weel=MotorB;
        dir=60;
      } else if(dir>120&&dir<=240){
        heading_follower.weel=MotorA;
        dir=180;
      }else{
        heading_follower.weel=MotorC;
        dir=300;
      }
      heading_follower.direction=dir;
      heading_follower.speed=25.0;
      while(ontrack==1){
        .
        .
        .
      }
    }
```

All'interno di questo loop il robot si muove sempre lungo la direzione principale e cambia comportamento solo quando perde la traccia, per iniziare a basculare e ritrovarla. Quando

la ritrova ricomincia a muoversi nuovamente lungo la direzione principale che però, come a breve vedremo, è ruotata rispetto alla traccia[15].
In questo punto del codice inizializziamo anche la variabile `rnd` che ci servirà per rendere aleatoria la direzione in cui il robot inizierà a basculare.

```
      while(ontrack==1){
        int rnd= sign(100-Random(200));
        if(rnd==0){rnd=1;}
        heading_follower.mode=0;
        heading_follower.speed=25.0;
        if(Path>threshold){
          .
          .
          .

        }
      }
```

La ricerca della traccia basculando è implementata con un ciclo nel quale possiamo entrare solo se il sensore supera una certa soglia e dopo aver fermato le ruote ed inizializzato le variabili per il ciclo.

Per uscire dal ciclo di ricerca della traccia ci sono due modi: il primo è trovare la traccia, il secondo è raggiungere il tempo di timeout, impostato a 8 secondi. All'uscita del ciclo ricontrolliamo la traccia e, se non vi siamo sopra, vuole dire che l'abbiamo persa[16].
In sintesi continuiamo a muoverci e ricercare la traccia ogni volta che la perdiamo, fino a quando riusciamo a trovarla: se andiamo in timeout e il robot non ha ritrovato la traccia, la dichiara persa e disabilita il proprio output ai motori.
Sarà `SpiralSearch` a confermare che la traccia è persa riportando `ontrack` a 0 per poi ricominciare a cercare.

[15] Questo comportamento mostra chiaramente la caratteristica di architettura reattiva e merita una breve riflessione: la grande differenza tra deliberativo e reattivo è che mentre nel primo la posizione e lo stato fisico del robot vengono registrati all'inizio ed usati per pianificare la migliore strategia d'intervento, nel secondo, seppur giochino un ruolo fondamentale, non sono rappresentati all'interno del programma e vengono comunque usati.

[16] Con questo sistema la traccia talvolta può essere considerata persa, anche se era stata trovata ma, per qualche problema di posizionamento o rumore, arrivati all'`if` non l'abbiamo riconosciuta.

```
        if(Path>threshold){
          heading_follower.speed=0.0;
          Wait(20);
          long t0=CurrentTick();
          int i=100;
          float power=40.0*rnd;
          while((Path>=threshold) &&(CurrentTick()<(t0+8000))){
            .
            .
            .
          }
          heading_follower.speed=0.0;
          Wait(20);
          if(Path>threshold){
            heading_follower.mode=-1;
            Wait(2000);
            tracklost=1;
            Wait(1000);
          }
        }
```

Il basculare è implementato con la sterzata a due ruote che, come abbiamo visto, trasla e ruota il robot. Per velocizzare la ricerca la potenza fornita ai motori che è stata inizializzata fuori dal ciclo, viene progressivamente aumentata fino a 90.0 come valor massimo.
Nel momento in cui la traccia viene riagganciata il robot smette di basculare; si sarà sicuramente girato, presumibilmente nel verso giusto, e potremo ricominciare ad avanzare.

```
          while((Path>=threshold) &&(CurrentTick()<(t0+8000))){
            heading_follower.mode=2;
            heading_follower.speed=power;
            long t1=CurrentTick();
            while((Path>threshold) &&(CurrentTick()<(t1+i))){
            }
            i=i+100;
            if(abs(power)<90){
              power=(power/100)*(100+(i/100));
            }
            power=power-(2*power);
          }
```

Conclusione

Abbiamo costruito e fatto funzionare, soddisfando gli obiettivi che ci si era posti all'inizio, un robot con architettura behavior based ispirato a quello di Brooks. Nella progettazione abbiamo potuto affrontare direttamente le numerose problematiche che un modello reale e l'implementazione di un controller di questo tipo possono nascondere, comprendendo meglio anche alcune scelte fatte da Brooks stesso. La scelta di un robot olonomico, inoltre, essendo disponibili relativamente poche esperienze di questo tipo implementate con architettura behavior based, ci ha costretti ad uno studio più approfondito del movimento e della sua possibile gestione. L'architettura behavior based e la sussunzione si sono mostrati versatili: seppur siano cambiate le implementazioni di alcuni blocchi, lo schema logico delle interazioni tra i comportamenti è rimasto invariato.
Nel progetto siamo stati limitati dal numero di porte e di sensori disponibili. In sviluppi futuri, introducendo un moltiplicatore di porte, sarebbe possibile utilizzare più sensori.
Con l'aggiunta di altri tre sensori ultrasonici si potrebbe avere una rilevazione in tempo reale di tutti i possibili ostacoli attorno al robot, rendendo più efficiente il meccanismo di evasione degli ostacoli. L'aggiunta di sensori di luminosità riflessa sotto al robot, invece, potrebbe rendere molto più efficaci la ricerca e l'inseguimento della traccia.
Conservando intatta la struttura del modello fino ad ora utilizzata si potrebbero aggiungere livelli più complessi che consentirebbero al robot di gestire maggiori informazioni sulla traccia, iniziando ad esempio a mapparla, riconoscendo alcuni punti, distinguendo obiettivi da ostacoli o riuscendo ad avere consapevolezza del punto della traccia in cui si trova e della direzione in cui la sta percorrendo. Grazie al bluetooth si potrebbe far comunicare il robot con altri brick NXT, ricevendo comandi o facendolo collaborare con altri robot. In tal caso avremmo un ulteriore comportamento emergente, non più del robot ma del team, generato dall'interazione dei comportamenti dei singoli individui. Potremmo far collaborare robot dotati di diversa struttura ed attuatori[17], oppure potremmo avere numerosi robot della stessa tipologia che cooperano sullo stesso piano[18] per un obiettivo comune.
Con questo lavoro si spera di aver fornito una base solida e coerente per gli eventuali sviluppi futuri.

[17] Un robot con un braccio che raccoglie le lattine ad esempio potrebbe essere montato sopra ad un versione evoluta del nostro robot, raccogliere le lattine a cui si avvicina e depositarle in un certo punto della traccia.

[18] Potrebbero semplicemente muoversi in fila mantenendo le distanze di sicurezza, esplorare contemporaneamente diverse parti della mappa o cercare di raggruppare e spostare degli oggetti all'interno del recinto come farebbero dei cani da pastore con il gregge.

Appendice

Si riporta in questa appendice il codice integrale con cui è stato programmato il robot, aggiornato all'ultima versione testata.

```
#define MotorA OUT_A
#define MotorB OUT_B
#define MotorC OUT_C
#define AllMotors OUT_ABC
#define Sonic1 IN_1
#define Sonic2 IN_2
#define Sonic3 IN_3
#define Ground IN_4
#define Path SENSOR_4
#define VOL 1
```

```
void shift(float power, int dir, int time){
  while(dir>360){
    dir=dir-360
  }
  while(dir<0){
    dir=dir+360;
  }
  int deg=dir;
  float a,b,c;
  a=Sin(deg);
  deg=dir+120;
  b=Sin(deg);
  deg=dir+240;
  c=Sin(deg);
  a=(a/100)*power;
  b=(b/100)*power;
  c=(c/100)*power;
  OnFwd(MotorA, a);
  OnFwd(MotorB, b);
  OnFwd(MotorC, c);

  if(time>0){
    Wait(time);
  Off(AllMotors);
  }
}
```

```
void rotatedeg(float power, int deg){
  RotateMotor(MotorA, power,deg);
  RotateMotor(MotorB, power,deg);
  RotateMotor(MotorC, power,deg);
}
```

```
void rotate(float power, int time) {
  OnFwdReg(MotorA, power,OUT_REGMODE_SPEED);
  OnFwdReg(MotorB, power,OUT_REGMODE_SPEED);
  OnFwdReg(MotorC, power,OUT_REGMODE_SPEED);
  if(time>0){
    Wait(time);
    Off(AllMotors);
  }
}
```

```
void rotate1w(float power, int time, byte weel){
  Off(AllMotors);
  OnFwd (weel, power);
  if(time>0){
    Wait(time);
    Off(weel);
  }
}
```

```
void rotate2w(float power, int time, byte weel){
  Off(weel);
  if(weel==MotorA){
    OnFwdReg(OUT_BC,power,OUT_REGMODE_SYNC);
  }else if(weel==MotorB){
    OnFwdReg(OUT_AC,power,OUT_REGMODE_SYNC);
  }else if(weel==MotorC){
    OnFwdReg(OUT_AB,power,OUT_REGMODE_SYNC);
  }
  if(time>0){
    Wait(time);
  Off(AllMotors);
  }
}
```

```
struct datalink{
  int mode;
  float speed;
  int direction;
  int rotation;
  int time;
  byte weel;
};
```

```
int halt=0;
datalink force;
datalink heading;
datalink heading_wander;
datalink heading_spiral;
datalink heading_follower;
datalink command;
datalink command_run;
datalink command_block;
int lastheading=0;
int ontrack=0;
int tracklost=1;
int threshold,threshold2;
int collision,alarm,warning,regular;
```

```
task ShowLCD(){
  while(true){
    NumOut(15,LCD_LINE1, SensorUS(Sonic1),true);
    NumOut(15,LCD_LINE2, SensorUS(Sonic2));
    NumOut(15,LCD_LINE3, SensorUS(Sonic3));
    NumOut(15,LCD_LINE4, Path);
    Wait(MS_400);
  }
}
```

```
task SenseSpace(){
  while(true){
    if(SensorUS(Sonic1)<10||
    SensorUS(Sonic2)<10||SensorUS(Sonic3)<10){
      PlayToneEx(TONE_A7, MS_100,VOL,FALSE);
    }else if(SensorUS(Sonic1)<18||
    SensorUS(Sonic2)<18||SensorUS(Sonic3)<18){
      PlayToneEx(TONE_A6, MS_20,VOL,FALSE);
    }else if(SensorUS(Sonic1)<23||
    SensorUS(Sonic2)<23||SensorUS(Sonic3)<23){
      PlayToneEx(TONE_A5, MS_40,VOL,FALSE);
    }else if(SensorUS(Sonic1)<30||
    SensorUS(Sonic2)<30||SensorUS(Sonic3)<30){
      PlayToneEx(TONE_A4, MS_60,VOL,FALSE);
    }
  }
}
```

```
task Move(){
//appena si avvia azzara tutti i segnali in ingresso. ci
penseranno i vari blocchi, se esistono a farsi sentire
  command_block.mode=-1;
  command_run.mode=-1;

  while(true){
  //ARBITER
    if(command_block.mode!=-1){
      command=command_block;
    }else if(command_run.mode!=-1){
      command=command_run;
    }
    if(halt==0){
      if(command.mode==0){
        shift(command.speed,command.direction,0);
      }else if(command.mode==1){
        rotate1w(command.speed,0,command.weel);
      }else if(command.mode==2){
        rotate2w(command.speed,0,command.weel);
      }else if(command.mode==3){
        rotate(command.speed,0);
      }
    }else{
      Off(AllMotors);
    }
  }
}
```

```
task AvoidCollision(){
  while(true){
    if(SensorUS(Sonic1)<collision||
    SensorUS(Sonic2)<collision||SensorUS(Sonic3)<collision){
      halt=1;
    }else{
      halt=0;
    }
  }
}
```

```
task FeelForce(){
  float a,b,c,rate,intensity;
  while(true){
    a= SensorUS(Sonic1);
    b= SensorUS(Sonic2);
    c= SensorUS(Sonic3);
    if (a<regular&&a<=b&&a<=c){
      rate= (a/regular)*100;
      intensity=100-rate;
      force.speed=intensity;
      force.direction=0;
    }else if (b<regular&&b<=c){
      rate= (b/regular)*100;
      intensity=100-rate;
      force.speed=intensity;
      force.direction=240;
    }else if (c<regular){
      rate= (c/regular)*100;
      intensity=100-rate;
      force.speed=intensity;
      force.direction=120;
    }else{
      force.speed=0.0;
      force.direction=0;
    }
  }
}
```

```
task RunAway(){
  while(true){
    if(abs(force.speed)>15){
      command_run.mode=0;
      command_run.direction=force.direction+Random(30);
      command_run.speed=force.speed;
    }
  }
}
```

```
task Wander(){
  while(true){
    heading_wander.mode=0;
    heading_wander.direction=Random(360);
    heading_wander.speed=40.0;
    Wait(SEC_2);
  }
}
```

```
task Block(){
  heading_wander.mode=-1;
  heading_spiral.mode=-1;
  heading_follower.mode=-1;

  while(true){
    if(heading_follower.mode!=-1){heading=heading_follower;}
    else if(heading_spiral.mode!=-1){heading=heading_spiral;}
    else if(heading_wander.mode!=-1){heading=heading_wander;}

    if(abs(force.speed)>15){
      command_block.mode=0;
      command_block.direction=force.direction+Random(30);
      command_block.speed=force.speed;
      float lastforce=force.speed;
      long t0=CurrentTick();
      while(CurrentTick()<t0+Random(5000)&&force.speed<lastforce){
      }
    }else{
      command_block=heading;
    }
  }
}
```

```
task SpiralSearch(){
  int i=0;
  long t0;

  while(true){
    heading_spiral.mode=0;
    int rnd= sign(100-Random(200));
    i=0;
    if(Path<=threshold){ontrack=1;}
    while(Path>threshold&&tracklost==1){
      ontrack=0;
      t0=CurrentTick();
      i=i+10;
      heading_spiral.direction=rnd*i;
      heading_spiral.speed=30.0;
      while(Path>threshold && CurrentTick()<(t0+i)){
      }
      if(Path<=threshold){
        PlayToneEx(TONE_F5, MS_500,VOL,FALSE);
        Off(AllMotors);
        heading_spiral.speed=0.0;
        lastheading=heading_spiral.direction;
        ontrack=1;
      }
    }
  }
}
```

```
task PathFollow(){
  while(true){
    if(ontrack==1){
      Wait(300);
      PlayToneEx(TONE_F4, MS_500,VOL,FALSE);
      Wait(300);
      PlayToneEx(TONE_A5, MS_500,VOL,FALSE);
      Wait(300);
      PlayToneEx(TONE_F5, MS_500,VOL,FALSE);
      Wait(300);
      heading_follower.mode=0;
      tracklost=0;
      Wait(3000);
      int dir=lastheading+90;
      while(dir>360){
        dir=dir-360;
      }
      while(dir<0){
        dir=dir+360;
      }
      if(dir>0&&dir<=120){
        heading_follower.weel=MotorB;
        dir=60;
      } else if(dir>120&&dir<=240){
        heading_follower.weel=MotorA;
        dir=180;
      }else{
        heading_follower.weel=MotorC;
        dir=300;
      }
      heading_follower.direction=dir;
      heading_follower.speed=25.0;
      while(ontrack==1){
        int rnd= sign(100-Random(200));
        if(rnd==0){rnd=1;}
        heading_follower.mode=0;
        heading_follower.speed=25.0;
        if(Path>threshold){
          heading_follower.speed=0.0;
          Wait(20);
          long t0=CurrentTick();
          int i=100;
          float power=40.0*rnd;
          while((Path>=threshold) &&(CurrentTick()<(t0+8000))){
            heading_follower.mode=2;
            heading_follower.speed=power;
            long t1=CurrentTick();
            while((Path>threshold) &&(CurrentTick()<(t1+i))){
            }
            i=i+100;
            if(abs(power)<90){
              power=(power/100)*(100+(i/100));
            }
            power=power-(2*power);
          }
          heading_follower.speed=0.0;
          Wait(20);
```

```
            if(Path>threshold){
              PlayToneEx(TONE_F5, MS_500,VOL,FALSE);
              Wait(300);
              PlayToneEx(TONE_A5, MS_500,VOL,FALSE);
              Wait(300);
              PlayToneEx(TONE_F4, MS_500,VOL,FALSE);
              Wait(300);
              heading_follower.mode=-1;
              Wait(2000);
              tracklost=1;
              Wait(1000);
            }
          }
        }
      }else{
        heading_follower.mode=-1;
        tracklost=1;
      }
    }
}
```

```
task main(){
  SetSensorType(Ground, SENSOR_TYPE_LIGHT_ACTIVE);
  SetSensorMode(Ground, SENSOR_MODE_RAW);
  SetSensorType(Sonic1, SENSOR_TYPE_LOWSPEED_9V);
  SetSensorMode(Sonic1, SENSOR_MODE_RAW);
  SetSensorType(Sonic2, SENSOR_TYPE_LOWSPEED_9V);
  SetSensorMode(Sonic2, SENSOR_MODE_RAW);
  SetSensorType(Sonic3, SENSOR_TYPE_LOWSPEED_9V);
  SetSensorMode(Sonic3, SENSOR_MODE_RAW);
  Wait(SEC_1);
  Precedes(ShowLCD);
  collision=8;
  alarm=15;
  warning=20;
  regular=25;
  Precedes(SenseSpace);
  Precedes(AvoidCollision);
  Precedes(Move);
  threshold=400;
  threshold2=350;
  Precedes(Block);
  Precedes(Wander);
  Precedes(FeelForce);
  Precedes(RunAway);
  Precedes(SpiralSearch);
  Precedes(PathFollow);

  PlayToneEx(TONE_C4, MS_100,VOL,FALSE);
  Wait(MS_300);
  PlayToneEx(TONE_C4, MS_100,VOL,FALSE);
  Wait(MS_300);
}
```

Bibliografia

[1] "A Robust Layered Control System for a Mobile Robot", Rodney A. Brooks, IEEE Journal of Robotics and Automation, RA-2, April 1986.

[2] "A Robot that Walks; Emergent Behaviors from a Carefully Evolved Network", Rodney A. Brooks.

[3] "The Robotics Primer", Maja Mataric, The MIT Press Cambridge, Massachusetts, London, England.

[4] "Extreme NXT: Extending the LEGO MINDSTORMS NXT to the Next Level", Michael Gasperi and Philippe "Philo" Hurbain, Apress

[5] http://www.philohome.com/sensors/legorot.htm (Luglio 2013)

[6] http://cs.brown.edu/~tld/courses/cs148/02/sensors.html (Luglio 2013)

[7] http://www.philohome.com/motors/motorcomp.htm (Luglio 2013)

[8] http://digilander.libero.it/beamweb/braitenberg.htm (Luglio 2013)

[9] https://it.wikipedia.org/wiki/Valentino_von_Braitenberg (Luglio 2013)

[10] http://thenxtstep.blogspot.it/2010/10/cool-color-sensor-pictures.html (Luglio 2013)

[11] http://www.LEGOengineering.com/nxt-sensors/ (Luglio 2013)

[12] http://en.wikipedia.org/wiki/Lego_Mindstorms (Luglio 2013)

[13] http://it.wikipedia.org/wiki/LEGO_Mindstorms (Luglio 2013)

[14] http://it.wikipedia.org/wiki/Lego_Mindstorms_NXT (Luglio 2013)

[15] http://en.wikipedia.org/wiki/Lego_Mindstorms_NXT (Luglio 2013)

[16] http://isodomos.com/technica/registry/9volt/9volt_8.php(Luglio 2013)

[17] http://isodomos.com/Lego-Sets/ (Luglio 2013)

[18] http://isodomos.com/technica/registry/9volt/9volt_7.php (Luglio 2013)

[19] http://www.tik.ee.ethz.ch/mindstorms/sa_nxt/index.php?page=tests_us (Aprile 2013)

[20] http://www.tik.ee.ethz.ch/mindstorms/sa_nxt/index.php?page=tests_light (Aprile 2013)

[21] http://www.media.mit.edu/sponsorship/getting-value/collaborations/mindstorms (Luglio 2013)

[22] http://l3d.cs.colorado.edu/systems/legosheets/Brick.html (Giugno 2013)

[23] http://el.media.mit.edu/logo-foundation/pubs/logoupdate/v7n1/v7n1-pbrick.html (Luglio 2013)

[24] http://bricxcc.sourceforge.net/ (Luglio 2013)

[25] http://bricxcc.sourceforge.net/nbc/ (Luglio 2013)

[26] http://bricxcc.sourceforge.net/nbc/nxcdoc/index.html (Luglio 2013)

[27] http://mindstorms.lego.com/en-us/News/ReadMore/Default.aspx?id=476243 (Luglio 2013)

[28] http://www.hitechnic.com/cgi-bin/commerce.cgi?preadd=action&key=HRC2148 (Luglio 2013)