Corso di Robotica del Prof. Antonio Chella
Assistente Ing. Filippo Lanza
a.a. 2018/2019

Progetto di Robotica

di
Filippo Guarina
Fabiano Limina
Marco Lo Piano

RoboCorp

UNIVERSITÀ
DEGLI STUDI
DI PALERMO

Progetto di Robotica

Introduzione

Il progetto consiste nella progettazione e realizzazione di un robot in grado di svolgere semplici operazioni in modo autonomo, come il riconoscimento di oggetti di un determinato colore, il calcolo della distanza da questi e la navigazione in una mappa con presenza di ostacoli non determinabili offline.

Il primo passo è stato quello della realizzazione hardware del robot, andandolo ad assemblare utilizzando il minor numero di dispositivi possibile per i compiti che deve svolgere.

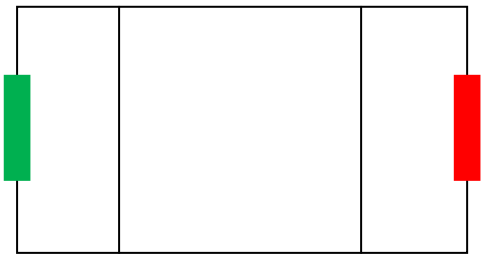
Abbiamo utilizzato uno chassis a due piani con due ruote poste sul piano inferiore, controllate ognuna da un motore e un "ruotino" per garantire stabilità e diminuire l'attrito al posteriore.

Il cervello del nostro robot è costituito da una Raspberry pi 3b+ a cui sono collegati:

- Due sonar, due infrarossi che fungono da evita - ostacoli
- Il motor controller per l'interfacciamento dei motori alla raspberry
- Una usb cam per il riconoscimento dell'oggetto
- Una bussola per direzionare il robot
- Un servo motore collegato ad un braccio che delimita un'area nella parte anteriore del robot, attivato da un sensore infrarossi.

Il robot deve essere in grado di partecipare ad una competizione le cui regole sono:

- Campo di dimensioni 3x2 m
- Due aree per lato distinte per colore
- Ostacoli sul percorso
- Due squadre
- Una pallina per squadra
- Tempo limite 15 minuti
- Penalità al punteggio finale per ogni ostacolo abbattuto



L'obiettivo è quello di nascondere il nostro oggetto all'interno del campo, prendere l'oggetto della squadra avversaria e portarlo nella porta avversaria per segnare. Una volta fatto ciò la pallina verrà rimessa esattamente nella posizione in cui il robot proprietario l'aveva nascosta e si dovrà ripetere il processo di ricerca e trasporto nell'area avversaria fino allo scadere dei 15 minuti. Vincerà chi riuscirà ad ottenere il punteggio finale più alto, calcolato in base al numero di goal, al materiale utilizzato e alle penalità ricevute.

La relazione che presentiamo per il progetto appena descritto è divisa in quattro parti:

- Specifiche hardware
- Specifiche software
- Specifiche architettura
- Schema riassuntivo nodi

Nella prima parte discutiamo in modo dettagliato dell'hardware che abbiamo utilizzato.

Nella seconda parte discutiamo invece del software che abbiamo scritto per la gestione delle componenti hardware, approfondendone le funzioni.

Nella terza parte discutiamo delle scelte progettuali che abbiamo effettuato per la realizzazione del nostro robot, facendo particolare attenzione alle catene stimolo-reazione che lo caratterizzano e al codice utilizzato per la gestione di queste.

Nell'ultima parte presentiamo uno schema riassuntivo dei nodi creati con ROS.

Specifiche hardware

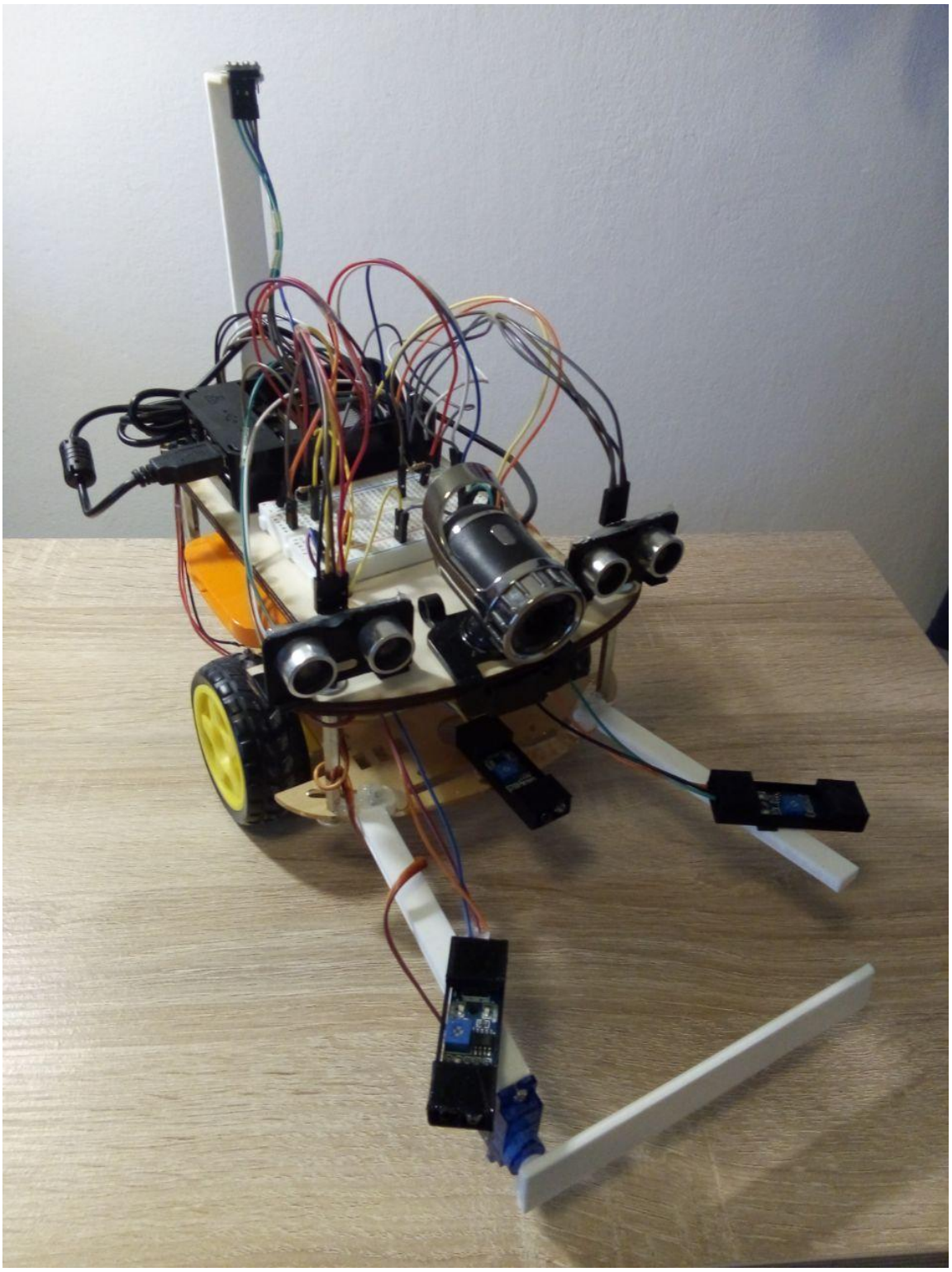


Figura 1: Dreambot

Iniziamo analizzando nel dettaglio l'hardware che abbiamo utilizzato, specificandone per completezza di informazioni anche le caratteristiche peculiari, e allegando i datasheet come collegamenti ipertestuali.

L'elemento che funge da cervello del nostro robot è la Raspberry Pi 3 Model B+, dotata di:

- CPU ARM Cortex-A53 1.4GHz
- 1GB SRAM
- Bluetooth 4.2 / BLE
- Ethernet up to 300Mbps
- Dual-band 2.4GHz and 5GHz wireless LAN
- 40-pin GPIO header

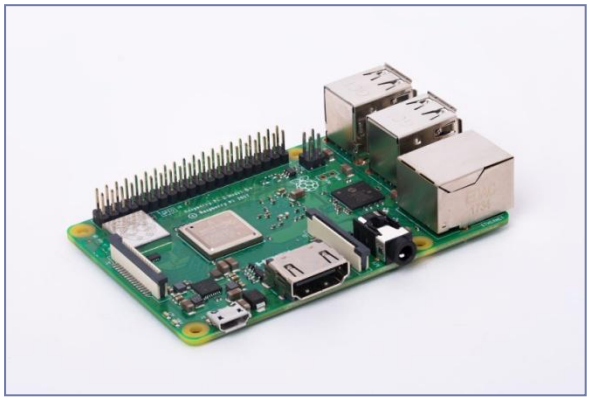


Figura 2: Raspberry Pi 3 Model B+

Questo è il componente più importante del robot, essendo demandato ad interfacciare tra loro tutti i successivi componenti, a gestirli e ad effettuare i calcoli necessari dal software.

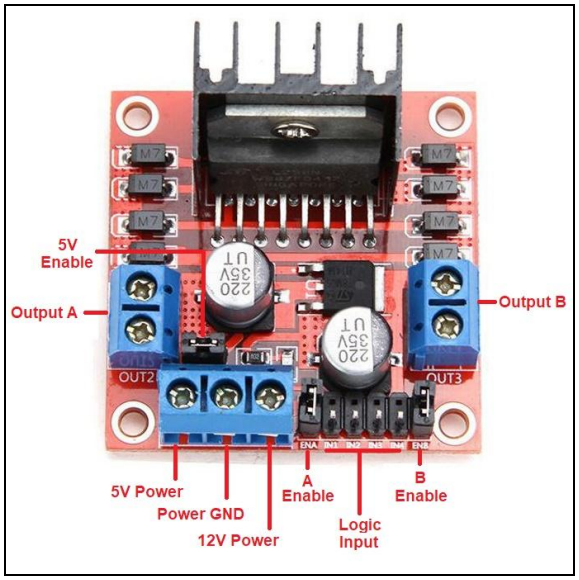


Figura 3: L298N

Il movimento delle due ruote collegate ai 2 motori è gestito da un L298N driver board, alimentato da un battery pack 4xAA.

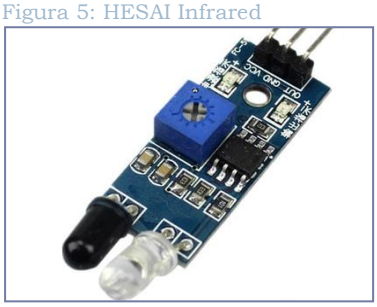
La webcam, che utilizziamo per il riconoscimento dell'oggetto e per il calcolo della distanza da esso, è una Alloyseed USB Cam con sensore CMOS,

risoluzione massima 640*480 e frame-rate di 30fps.

Figura 4: AS USB Cam

Per la funzione di evita - ostacoli utilizziamo invece 2 infrarossi della HESAI (alimentati a 3.3V) con potenziometro manuale per l'attivazione del sensore tra i 2 e i 6 cm e 2 sonar HCSR04 (a 5V).

Figura 5: HESAI Infrared



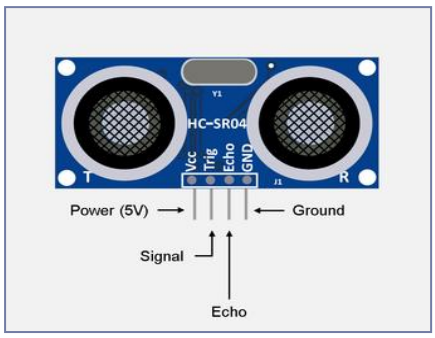


Figura 6: HCSR04 Sonar

La bussola per l'orientamento del robot è una QST QMC5883L a 3 assi con un'accuratezza di 1-2 gradi collegabile alla Raspberry tramite collegamento I2C e alimentata a 5V.

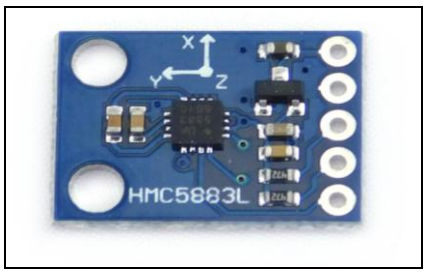
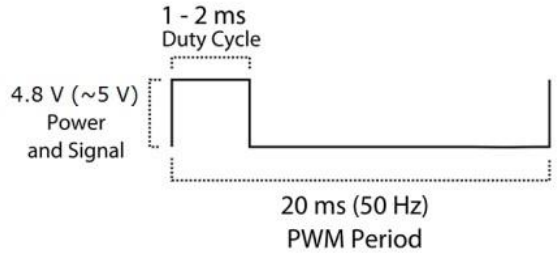


Figura 7: QST QMC5883L compass

Infine abbiamo utilizzato un servo motore Smart Electronics SG90 collegato tramite PWM alla Raspberry per la gestione della chiusura del braccio in prossimità dell'oggetto .

Figura 8: SG90 ServoMotor

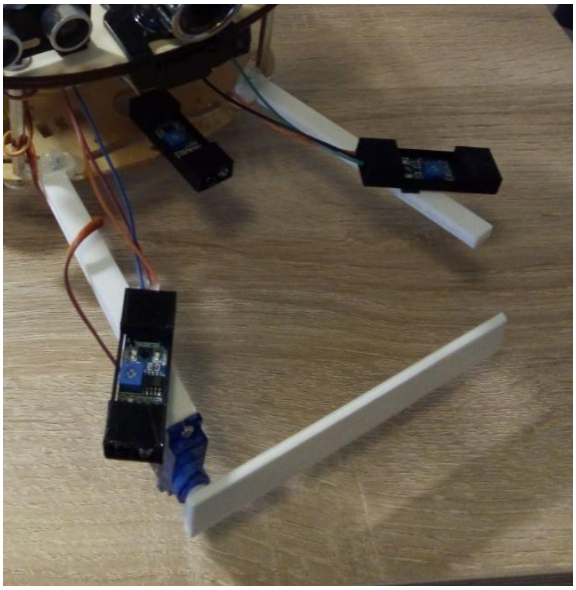
Il servo motore funziona tramite un segnale PWM sul cavo arancione.



Segnale che, come si vede in figura sopra, dovrebbe avere una frequenza di 50Hz, con un valore di Duty Cicle compreso tra 1 e 2 ms. Per impulsi di 1 ms il servo motore si trova a 0 gradi, per impulso di durata 2 ms si trova a 180 gradi.

Abbiamo deciso di utilizzare la configurazione visibile nella pagina precedente, ovvero di un doppio piano per migliorare la suddivisione dello spazio e quindi dei pesi, fondamentali per la stabilità della traiettoria del robot, e per una migliore visione della camera che inquadra più ambiente. La disposizione dei sonar e degli infrarossi inoltre è dovuta alle caratteristiche del campo e degli ostacoli che incontreremo, oltre che delle dimensioni dello chassis compreso di

braccio. Abbiamo quindi pensato che la soluzione migliore per evitare gli ostacoli e per poter costeggiare le pareti del campo, coprendolo meglio in ampiezza, sia quella di utilizzare due sonar frontali per gli ostacoli davanti il robot, anche a metà dello stesso, e due infrarossi laterali posti sul braccio necessari per evitare che il robot possa bloccarsi e/o colpire un oggetto durante le rotazioni. Infine abbiamo utilizzato l'ultimo sensore infrarosso per permettere al nostro robot di chiudere il braccio collegato al servomotore quando rileva un oggetto all'interno dell'area delimitata dallo chassis e dal braccio stesso.



Specifiche software

Il progetto che presentiamo necessita di molte funzioni che sono demandate al software. La telecamera, ad esempio, deve riuscire non solo a riconoscere l'oggetto solo se di un determinato colore ma anche a calcolare con buona approssimazione la distanza da esso. In questa sezione dunque presentiamo il software da noi scritto evidenziandone le funzioni fondamentali e le scelte progettuali.

Il primo software che andremo ad analizzare quello che utilizziamo per il semplice movimento del nostro robot.

```
# pins setup
def init():
    GPIO.setmode(GPIO.BOARD)
    GPIO.setup(7, GPIO.OUT)
    GPIO.setup(11, GPIO.OUT)
    GPIO.setup(13, GPIO.OUT)
    GPIO.setup(15, GPIO.OUT)
```

Con questa piccola funzione in python andiamo a definire i pin della GPIO a cui sono agganciati i motori, stando attenti che i pin 7 e 15 corrispondono al movimento in avanti, i pin 11 e 13 al movimento in direzione opposta.

```
def forward(t):
    GPIO.output(7, True)
    GPIO.output(15, True)
```

Ad esempio definendo questa funzione forward andrò ad attivare entrambi i pin delle due ruoti dedicati al movimenti in avanti, permettendo al robot di muoversi in linea retta. Continuando con questa logica abbiamo definito anche funzioni di turnLeft, turnRight, reverse e stop.
L'utilizzo di queste funzioni ci permette di muovere il robot secondo i criteri stabiliti nell'architettura.

Il secondo programma è quello della USB Cam, necessario come scritto prima al riconoscimento dell'oggetto e al calcolo della distanza da questo.
Utilizzando le librerie OpenCV3 in C++ abbiamo sfruttato funzioni che permettono di ottenere, tramite tuning manuale, i valori HSV (Hue, Saturation, Value) che corrispondono al colore specifico del nostro oggetto. Nel dettaglio le funzioni da noi utilizzate in enhanced_threshold.cpp sono:

- createTrackbar()
- inRange()
- erode()
- dilate()

La funzione inRange(), in collaborazione con la funzione di creazione trackbar, permette di modificare il frame in arrivo da una qualsiasi sorgente (nel nostro caso la cam) creando un'immagine binaria in cui gli elementi visibili sono quelli che stanno nel range HSV stabilito.
Utilizzando questa immagine binaria quindi è possibile trovare i valori HSV cercati del nostro oggetto.
Questi valori sono utilizzati nella funzione distance.cpp, per il calcolo della distanza del robot dall'oggetto stesso.
Le funzioni erode e dilate sono invece funzioni utilizzate per la pulizia

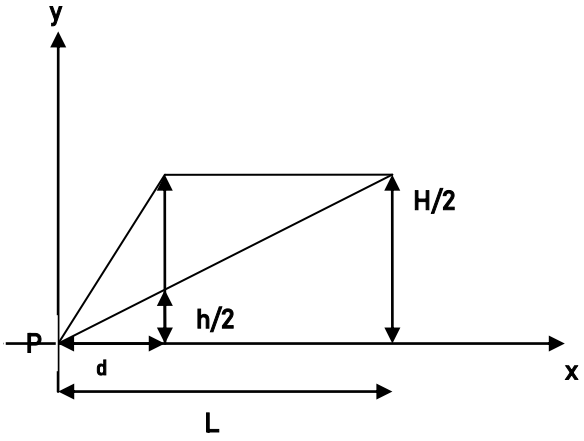
dell'immagine thresholdata andando a rimuovere rumore e isolando elementi singoli dell'immagine.

In distance.cpp le funzioni da noi utilizzate sono:

- findContours()
- contoursConvexHull()
- boundingRect()
- distance()

La funzione findContours() , estrae gli edge dall'immagine passatale in argomento e crea i contorni in un vettore di vettori di punti. Per migliorare la resa dei contorni così trovati e diminuire ulteriormente la possibilità di valori sparsi o di rumore, usiamo la funzione contoursConvexHull() che crea, sulla base dei valori usciti dalla findContours, il contorno chiuso che meglio approssima l'oggetto. Infine utilizziamo un'altra funzione di OpenCV3 che, dato un contorno chiuso, crea il minimo rettangolo includente, ritornandone centro e i due lati.

Una volta identificato e isolato l'oggetto, compreso di rettangolo minimi includente, non rimane che calcolarne la distanza dal nostro robot. La funzione distanza, da noi scritta, prevede lo sfruttamento di semplici regole geometriche di proiezione per un calcolo della distanza molto rapido e abbastanza fedele. Il nostro algoritmo inoltre, anche in situazioni di oggetto quasi totalmente occluso orizzontalmente, permette una stima accurata della distanza.



L'algoritmo procede calcolando la distanza L di un oggetto, conoscendone l'altezza H ad una distanza stabilita d.
Procediamo dunque tracciando il cono C di base H e altezza L; si è verificato che la dimensione $h/2$ rilevata dalla Cam dell'oggetto a distanza L è la semi base del cono ottenuto sezionando C con un piano perpendicolare al piano xy a distanza d da P.
Avendo dunque d, h e H posso trovare L con la regola della similitudine dei triangoli rettangoli :

$$\frac{D}{h} = \frac{L}{H} \Rightarrow L = D * \frac{H}{h}$$

```
double distance (double h) {
    double DistNota1=36;
//  double areaKnow1=41500;
    double heightKnow1=182;
    double dist;
//  return dist=sqrt(areaKnow1/pigreco)/r*DistNota1
    return dist=(heightKnow1/2)/h*DistNota1;
}
```

Proprio l'utilizzo dell'altezza come componente principale dell'algoritmo, ne permette l'utilizzo anche in situazioni di occlusione.

Utilizzando la funzione che calcolo il minimo rettangolo includente abbiamo anche la possibilità di calcolare la distanza tra il centro del frame della camera e il centro del rettangolo prima descritto, questo valore, denotato come centroid_dist lo usiamo poi per centrare il robot rispetto al nostro oggetto da prendere.

Come precedentemente scritto, abbiamo utilizzato insieme sonar e IR per il problema del rilevamento ostacoli. I codici, che abbiamo creato separatamente per IR e sonar, destro e sinistro (e che si appoggia, come descritto nell'architettura, a ROS) inserisce in un topic dedicato il valore specifico del dispositivo in questione. Nel caso dell'IR sinistro:

```
rate = rospy.Rate(4)

while not rospy.is_shutdown() :
    booleanValue = GPIO.input(LEFT_IR_PIN)

    pub.publish(booleanValue)
    rate.sleep()
```

creiamo quindi un valore booleano che avrà assegnato valore "falso" se l'IR rileva un ostacolo, vero altrimenti. Questo valore poi sarà pubblicato nel canale dedicato al sensore infrarosso sinistro ad un rate indicate dalla variabile rate.

Il codice del sonar (nello specifico di quello sinistro) risulta più complesso poiché non ritorna un valore booleano ma direttamente la distanza dall'oggetto più vicino nel raggio di rilevamento. abbiamo quindi due funzioni fondamentali oltre al main:

- trigModule
- distanceCalculation

il sonar verrà infatti attivato per un tempo determinato dalla variabile "impulse"

```
def trigModule() :
    GPIO.output(TRIG, True)
    time.sleep(IMPULSE)
    GPIO.output(TRIG, False)
```

Nel main invece abbiamo due cicli while che assegnano i valori di pulse_start e pulse_end nel momento in cui il componente echo del sonar rispettivamente rileva il segnale e non lo rileva più.

```
while GPIO.input(ECHO) == 0 :
    pulse_start = time.time()

while GPIO.input(ECHO) == 1 :
    pulse_end = time.time()
```

La funzione che calcola la distanza sulla base di questi dati è la distanceCalculation.

```
def distanceCalculation(p_start, p_end) :
    p_duration = p_end - p_start
    dist = (p_duration * SOUND_SPEED) / 2
    dist = round(dist, 2)
    return dist
```

Questa funzione prende per argomento i valori p_start e p_end e sulla base della velocità del suono ritorna il valore di distanza. Anche in questo caso, come negli IR utilizzeremo le librerie ROS per inviare ad una data frequenza il valore sul topic dedicato.

Il servo motore, che utilizziamo per aprire e chiudere il braccio per la presa dell'oggetto è gestito da questa funzione:

```
p.start(2)          # this is the neutral positi

# the ball is close
p.ChangeDutyCycle(7.5)      # turn towards 105 d
time.sleep(0.8)     # very important sleep

infinityForward()       # let's go get it

booleanValue = GPIO.input(36)
tic = time.time()
toc = tic

while booleanValue == True and (toc - tic) <= 3:
booleanValue = GPIO.input(36)
toc = time.time()

# the ball is inside (or maybe not)
p.ChangeDutyCycle(2)    # turn towards 0 degree
time.sleep(0.8)     # very important sleep

stop()

p.ChangeDutyCycle(0)    # to stop servo oscillat
time.sleep(0.5)     # very important sleep

time.sleep(1.5)
```

Partiamo assegnando il valore "neutrale" della posizione del servo motore; quando la palla è vicina cambiamo il valore del Duty Cicle a 7.5 e ci muoviamo in avanti, aspettando un input dal infrarossi interno del robot. Quando questo si attiva (valore False) chiudiamo in braccio risettando a 2 prima e infine a 0 il Duty Cicle per evitare oscillazioni del servo motore. Abbiamo inserito anche un timer di 3 secondi per fare chiudere il braccio in caso di cattura fallita della pallina.

Nel caso siamo in vista dell'oggetto ma non siamo centrati dobbiamo far fare al robot dei piccoli movimenti per il centramento rispetto all'obiettivo.

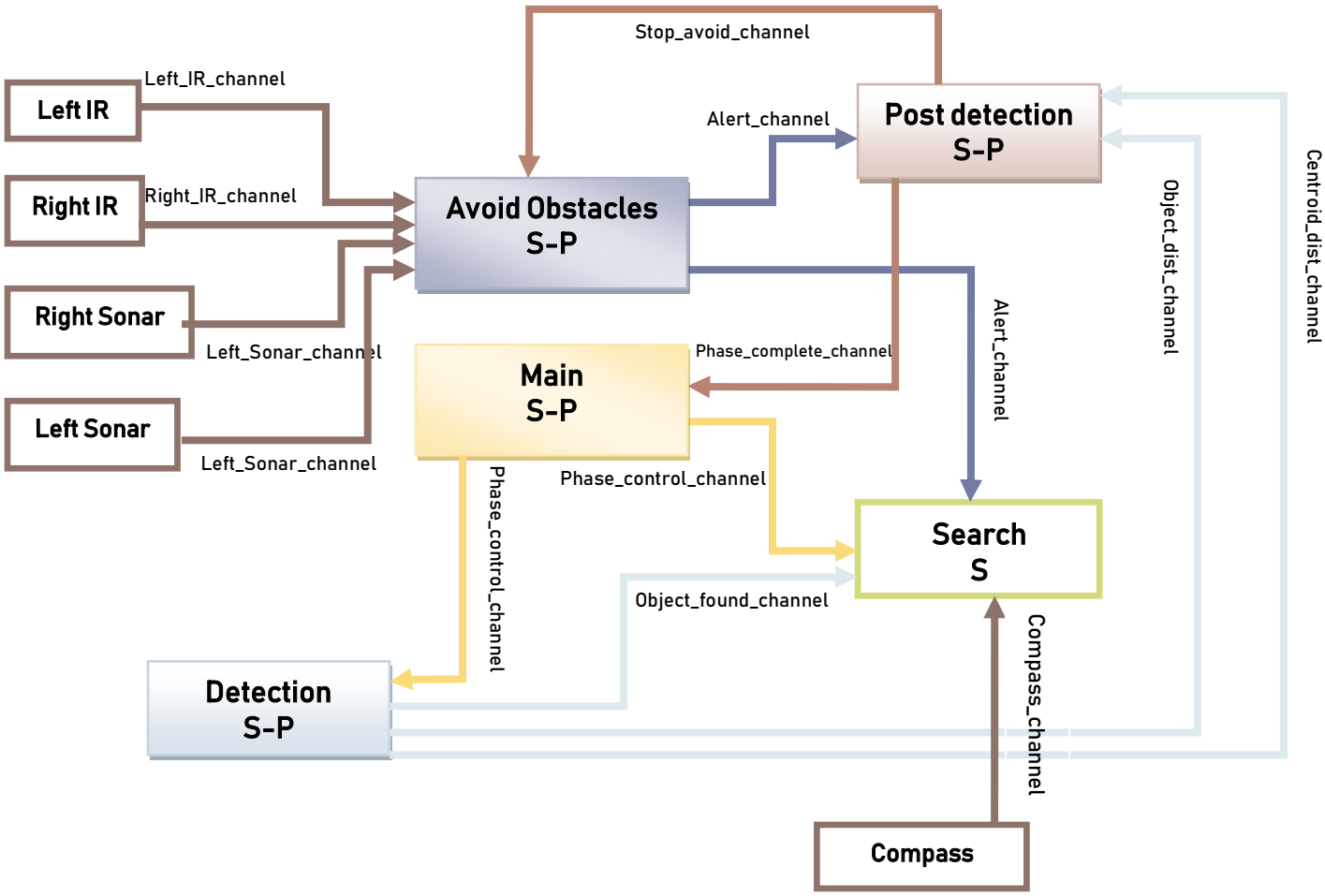
```
if state == 1 :
    turnRight(0.05)
    time.sleep(1)
    state = 0

elif state == 2 :
    turnLeft(0.05)
    time.sleep(1)
    state = 0
```

La bussola, invece, la gestiamo tramite una classe che abbiamo precedentemente installato sulla Raspberry, e che andiamo ad importare nel programma python di gestione della stessa. Abbiamo usato questa classe perché provvista di tutta una serie di funzioni, molto utili per l'utilizzo della bussola. Nel nostro codice abbiamo utilizzato la funzione costruttrice della classe, che permette anche di specificare la declinazione magnetica del luogo in cui ci si trova (utile per una migliore approssimazione) , e abbiamo usato la funzione get_bearing() che ritorna l'angolo tra gli assi della bussola e il nord magnetico.

```
angle = sensor.get_bearing()

pub.publish(angle)

rate = rospy.Rate(6)

while not rospy.is_shutdown() :
    # angle calculation
    angle = sensor.get_bearing()
    pub.publish(angle)

    rate.sleep()
```

```
direction = (angle - target_angle) % 360

while direction >= 5 and direction <= 355 :
    direction = (angle - target_angle) % 360
    if direction >= 5 and direction <= 180 :
    turnRight(0.2 * direction / 90)
    elif direction >= 181 and direction <= 355 :
    turnLeft(0.2 * (direction-180)/90)
```

Usiamo tre variabili che contengono valori di angoli prima descritti. La prima variabile è target_angle, che utilizziamo nella primissima fare di avvio della competizione, in cui siamo sicuri che il robot sarà nella direzione della porta avversaria, per salvarci proprio questo valore; il secondo invece è angle, che contiene l'attuale angolo tra direzione bussola e asse magnetico terrestre; il terzo è la direction che ci indica di quanto ci siamo spostati come angolazione rispetto alla porta avversaria. Moduliamo questo valore sui 360 gradi per evitare di dover gestire angoli negativi.
Utilizziamo poi i comandi turnLeft e turnRight all'interno di un ciclo while per ridirezionare il robot tramite la bussola in determinate situazioni che vedremo nella prossima sezione.
Chiaramente questo valore direction rappresenta una direzione molto approssimata.



Specifiche architettura

Per poter spiegare cosa rappresenta questo diagramma dobbiamo prima spiegare brevemente cos'è **ROS** e come funziona. Dal sito Ros.org:
"The Robot Operating System (ROS) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms".

ROS dunque è un ambiente di sviluppo dedicato interamente allo sviluppo di software nell'ambito della robotica; è organizzato in pacchetti (i progetti), suddivisi a loro volta in nodi (i singoli software) e canali, il luogo dove i nodi possono scambiarsi messaggi.
In questo diagramma di flusso abbiamo una rappresentazione schematica dell'organizzazione della nostra architettura. I rettangoli sono i nodi ROS, i connettori rappresentano la direzione di pubblicazione e il nome sopra i connettori indica il canale sul quale vengono pubblicati i messaggi. L'indicazione

S-P sta per nodo subscriber e publisher, S sta per nodo subscriber.

Abbiamo dunque costruito un robot di tipo **ibrido reattivo/deliberativo** su un'**architettura a sussunzione** governata dall'evita ostacoli, che blocca tutti gli altri comportamenti in caso di rilevamento di ostacolo e che implementa un algoritmo di ricerca basato sulla navigazione casuale del campo di gioco. Parliamo di un'**architettura ibrida** poiché abbiamo inserito una forma di deliberazione data dalla bussola che in determinate circostanze obbliga il robot ad effettuare dei ridirezionamenti lungo una direzione nota allo start della competizione.

Dal punto di vista architetturale e software abbiamo quindi usato ROS e diviso il software in nodi che logicamente rappresentassero le componenti minime necessarie per il compimento della competizione, stando attenti ai collegamenti tra questi nodi e ai problemi dovuti alla temporizzazione di questi e delle singole callback in ogni nodo. All'interno di questi abbiamo infatti utilizzato una rappresentazione dei vari comportamenti tramite macchina a stati, rappresentati da variabili globali, che sono modificate al raggiungimento di determinate condizioni.

Il nodo più importante, da un punto di vista gerarchico, è l'avoid obstacles; poiché rappresenta il software che avverte il robot di un ostacolo nelle sue vicinanze (grazie alla rete di sensori descritti nella parte hardware), questo deve essere in grado di stoppare il funzionamento di tutti gli altri nodi fino alla risoluzione dell'ostacolo. Abbiamo scritto dunque un codice differenziando i comportamenti in base al sensore che rileva l'ostacolo.
Abbiamo dunque uno schema del tipo:

Nel caso del sonar di destra abbiamo scritto un codice che inizializza una variabile randSonarR con una valore casuale compreso tra 0.2 e 0.3; questa è usata per indicare per quanto tempo le funzioni di reverse e turnLeft devono essere attivate.

Usiamo la stessa logica anche per i sensori infrarossi, modificando semplicemente la variabile tempo da inserire nelle funzioni di movimento.

Come anche indicato nel grafico di flusso, il nodo Avoid Obstacles è un publisher. Infatti, nel momento in cui un sensore della rete rileva un ostacolo, un valore booleano "true" è pubblicato sul canale alert_channel, su cui sono in ascolto i nodi Post_Detection e Search.

Il nodo Compass si occupa di leggere i valori della nostra bussola e di spedirli sul canale compass_channel. Il primo valore registrato dal nodo sarà salvato in una variabile chiamata target_angle, mentre gli altri verranno salvati nella variabile angle. Usiamo questa distinzione di variabili poiché abbiamo necessità (per come abbiamo implementato l'algoritmo di ricerca) di salvarci in una variabile separata la direzione della porta avversaria, che useremo poi successivamente per il direzionamento del robot nelle fasi avanzate della competizione.

Il nodo Main si occupa di informare i nodi Search e Detection della fase in cui si trova il robot, facendolo agire di conseguenza. Le fasi sono:

- Fase 1: prendi la nostra palla
- Fase 2: appena avvista la palla avversaria, rilascio della nostra palla e presa di quella avversaria
- Fase 3: rilascio della palla avversaria (Goal)
- Fase 4: ricerca della palla avversaria che è stata riposizionata

Abbiamo già parlato nella sezione precedente del codice C++ e OpenCV3 che ci permette di effettuare la threshold e calcolare l'approssimazione della distanza dall'oggetto e del calcolo del centroide. Vediamo adesso come questo codice è implemetato nell'architettura ROS. Come vediamo nel grafico il nodo Detection è sia un publisher che un subscriber. Questo nodo deve essere un subscriber poiché, dovendo andare alla ricerca di oggetti differenti, questi hanno threshold diverse e dunque dobbiamo variare in base all'oggetto che stiamo cercando questi valori. Abbiamo dunque (per la nostra pallina e quella avversaria):

```
void change_threshold_CB(const std_msgs::Int64::ConstPtr& msg){
    // phase 1 -> threshold for our ball
    if(msg->data == 1){
        low_H = 58;
        low_S = 75;
        low_V = 50;
        high_H = 90;
        high_S = 186;
        high_V = 255;
    }
    // phase 2 -> opponent's ball
    if(msg->data == 2){
        cout << "Valori cambiati!" << endl;
        low_H = 58;
        low_S = 75;
        low_V = 50;
        high_H = 90;
        high_S = 186;
        high_V = 255;
    }
}
```

La callback sopra inserita cambia dunque i valori della threshold in base al valore del messaggio nel canale phase_control_channel.
Il nodo invece manderà sui canali centroid_dist e object_dist le distanze rispettivamente del centroide e dell'oggetto, insieme ad un booleano sul canale object_found_channel che se True indica agli altri nodi in ascolto che l'oggetto attualmente cercato è stato visto dalla cam del robot.

```
std_msgs::Float64 msg1;
msg1.data = centroid_dist;
pub1.publish(msg1);

std_msgs::Float64 msg2;
msg2.data = object_dist;
pub2.publish(msg2);

std_msgs::Bool msg3;
msg3.data = true;
pub3.publish(msg3);
```

Il nodo Post Detection è subscriber ai canali centroid_dist, object_dist e alert_channel e publisher sul canale phase_complete. Questo nodo è quello che si occupa di prendere e rilasciare le due palline durante le varie fasi della competizione gestendo il braccio azionando il servomotore. Con i codici già visti nella sezione precedente gestiamo l'avvicinamento, il centramento e l'apertura e chiusura del braccio. È in questo nodo quindi che stabiliamo la politica di rilascio della pallina.

Utilizziamo quindi le callback per gestire le informazioni provenienti dai 3 canali a cui Post Detection è iscritto e cambiare stato. Gli stati utilizzati sono i seguenti:

- stato 1: oggetto rilevato ma deve centrarsi girandosi a destra
- stato 2: oggetto rilevato ma deve centrarsi girandosi a sinistra
- stato 3: oggetto rilevato e centrato
- stato 4: oggetto rilevato , controlliamo se abbiamo già la nostra pallina; se così è dobbiamo rilasciarla
- stato 5: oggetto rilevato, è centrato e a distanza inferiore a 30cm, quindi va avanti fino a raggiungerlo e catturala
- stato 6: stato di controllo per sapere se abbiamo la pallina; attendiamo per tre secondi dopo la chiusura del braccio se il sensore infrarosso centrale rileva un oggetto, se così è passa allo stato successivo, altrimenti ricomincia da capo

```
elif state == 6 :
    tic = time.time()
    toc = tic

    while (toc - tic) <= 3:  # 3 se
    booleanValue = GPIO.input(36)
    if booleanValue == False :
        state = 7        # it's ok.
        break
    else :
        state = 0        # d'oh!
    toc = time.time()    # 3 seconds
```

- stato 7: abbiamo la pallina e passiamo alla fase successiva della competizione; utilizziamo questo stato per ciclare tra fase 3 e 4 della competizione

```
elif state == 7 :
    if current_phase == 1 :
    pub.publish(current_phase)      #
    print("phase 1 completed")
    time.sleep(3)
    release = True                  # we h
    current_phase = current_phase + 1
    state = 0

    elif current_phase == 2 :
    pub.publish(current_phase)      #
    print("phase 2 completed")
    time.sleep(3)
    current_phase = current_phase + 1
    state = 0

    elif current_phase == 3 :
    pub.publish(current_phase)      #
    print("phase 3 completed")
    time.sleep(3)
    current_phase = current_phase + 1
    state = 0

    elif current_phase == 4 :
    pub.publish(current_phase)      #
    print("phase 4 completed")
    time.sleep(3)
    current_phase = current_phase - 1
    state = 0
```

- stato 8: rilasciamo la nostra pallina quando vediamo quella avversaria. Abbiamo scelto una politica che prevede il rilascio della nostra palla in caso di rilevamento della pallina avversaria o allo scadere di un timer . Il rilascio si svolge nei seguenti passaggi: ruotiamo il robot per 0.3s, apriamo il braccio, effettuiamo un reverse di 0.5s, richiudiamo il braccio e infine rieffettuiamo una rotazione di 0.3s nel senso opposto alla prima.

```
elif state == 8 :
    pubAvoid.publish(True)
    time.sleep(1.5)

    turnRight(0.3)

    p.start(2)

    p.ChangeDutyCycle(7.5)
    time.sleep(0.8)

    reverse(0.5)

    p.ChangeDutyCycle(2)
    time.sleep(0.8)

    p.ChangeDutyCycle(0)
    time.sleep(0.6)

    forward(0.5)

    turnLeft(0.3)
    release = False

    time.sleep(3)
    pubAvoid.publish(False)

    state = 0
```

- stato 9: rilasciamo la pallina avversaria nella porta avversaria facendo goal.

Il nodo Search si occupa della ricerca della nostra pallina, della pallina avversaria e della porta dove dobbiamo fare goal. In caso di ricerca della porta abbiamo implementato (come visto nella sezione precedente) un algoritmo di ridirezionamento verso il nord (la porta) dopo un certo lasso di tempo; in caso di ricerca della pallina avversaria dopo un goal invece abbiamo deciso di implementare un solo ridirezionamento verso il sud (direzione opposta alla porta) per evitare che il nostro robot continui a cercare nell'area avversaria.

Gli stati possibili in questo nodo sono 4:

- stato 0: controllo fase, se 1 o 2 passa allo stato successivo, altrimenti al 4
- stato 2: ricerca con un infinityForward() per un tempo determinato da un contatore, seguito da una turnLeft o turnRight random per numero di rotazioni casuale compreso tra 10 e 15.

```
if rand <= 0.5 :      # turn right
if count == randRot:
    count = 0
    choiceCounter = 0
    n = False
    state = 0

else :
    stop()
    turnRight(0.15)
    time.sleep(1.5)
    count = count + 1
    state = 0
else :           # turn left
if count ==  randRot:
    count = 0
    choiceCounter = 0
    n = False
    state = 0

else :
    stop()
    turnLeft(0.15)
    time.sleep(1.5)
    count = count + 1
    state = 0
```

- stato 3: ostacolo o oggetto rilevato, torna allo stato 0; arriviamo a questo controllo con la callback associata al canale object_found. Se il valore letto dal canale è False (oggetto non trovato) e mi trovo nello stato di normale funzionamento allora cambia stato e passa allo stato della ricerca vero e propria, altrimenti se l'oggetto è stato rilevato dalla cam (valore True) passa allo stato 3 che interrompe la ricerca.

```
objFunction_CB(msg) :
global state
global block
if state == 0 :
if msg.data == False and block == False :
    state = 2                    # object
elif msg.data == True and block == False:
    state = 3                    # object
```

- stato 4: ridirezionamento verso il nord
- stato 5: ridirezionamento verso sud

```
target_angle = (target_angle + 180) % 360
direction = (angle - target_angle) % 360

while direction >= 5 and direction <= 355 :
    direction = (angle - target_angle) % 360
    if direction >= 5 and direction <= 180 :
    turnRight(0.2 * direction / 90)
    elif direction >= 181 and direction <= 355 :
    turnLeft(0.2 * (direction-180)/90)

target_angle =target_angle - 180
```

Per il direziona mento verso il sud abbiamo quindi incrementato di 180 gradi il valore salvato all'inizio della competizione come target_angle.

```
def controlFunction(msg) :
    global state
    if state == 0 :
    if msg.data == 1 or msg.data == 2 :
        state = 1
    else :
        state = 4
```

Per la scelta tra ridirezionamento verso nord o sud utilizziamo la callback asso-

ciata al canale phase_control_channel. Il valore letto dal canale è assegnato alla variabile globale phase_control, che determina quale dei due stati raggiungere.

```
if state == 0:
if phase_control == 3 :
    if sampleNow == True or samplingTime > 350:
    block = True
    state = 4
elif phase_control == 4 :
    if sampleNow == True :
    block = True
    state = 5
samplingTime = samplingTime + 1
```

Schema riassuntivo nodi

Riassumendo in breve i vari collegamenti tra i nodi abbiamo:

- i 4 nodi per l'evita ostacoli, ciascuno deputato ad uno specifico sensore, gestiti contemporaneamente dall'Avoid Obstacles
- l'Avoid Obstacles stesso, che manda a tutti gli altri nodi il segnale di stop in caso di ostacolo rilevato
- il Main, che prende le informazioni provenienti dal Post Detection per segnalare a tutti gli altri nodi il completamento di una fase e l'avvio della successiva
- il Post Detection, che prende le informazioni dal Detection per avere informazioni sull'oggetto cercato riguardo a distanza e centroide. Informazioni che utilizza per centrarsi sull'obiettivo e avvicinarsi per poi catturarlo. È inoltre il nodo deputato al rilascio della nostra pallina quando quella avversaria è rilevata
- il Detection, che lavora con la Cam e in base alla fase in cui siamo effettua le operazioni di threshold e di calcolo della distanza e centroide
- il Compass, che legge i dati dalla bussola equipaggiata su un'antenna nel robot e li manda sul canale compass_channel, su cui è in ascolto il nodo Search
- il Search, che si occupa della ricerca vera e propria in caso di campo libero e oggetto non rilevato, effettuando movimenti di forward e rotazioni per cercare di spazzare più campo possibile alla ricerca dei nostri obiettivi.