RELAZIONE FINALE

INTERNET OF THINGS – PROGETTO AMERIGO 2.0

Corso di laurea magistrale in Informatica – Scuola di Scienze

Università di Bologna

Cod: 81683

Bruno Quintero Panaro Mat: 0000850912

Davide Nunzio Maccarrone Mat: 0000843825

2018/2019

SOMMARIO

INTRODUZIONE

Il progetto Amerigo 2.0, ideato come esercizio d'esame per il corso di internet of things, nasce dall'idea di voler creare da zero un automa capace di orientarsi in un ambiente sconosciuto e districarsi in un dedalo, evitando eventuali ostacoli lungo il percorso, seguendo un proprio obiettivo.

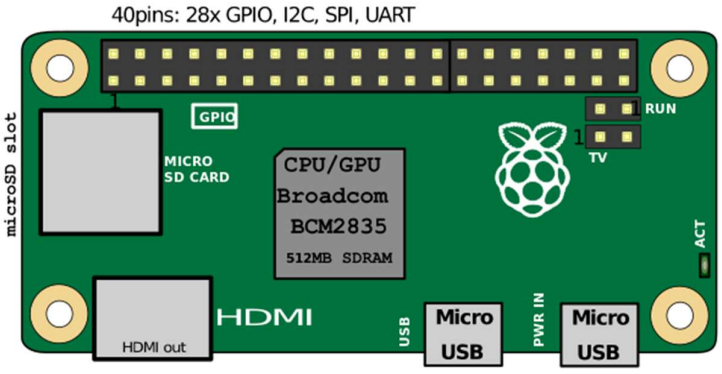
Il processo di sviluppo del progetto può essere idealmente diviso in tre parti diverse: lo studio dello stato dell'arte in ambito di robotica e sistemi di navigazione, in primis, la scelta, l'assemblamento e la corretta configurazione della componentistica, in una seconda fase e, infine, lo sviluppo di un algoritmo che, basandosi sugli studi fatti in precedenza, potesse risultare comunque innovativo e adatto alle scelte effettuate in fase di assemblamento. Ovviamente i tre step non vanno visti come sequenziali, ma il processo, spesso iterativo, ha richiesto più volte di rivedere alcune scelte effettuate in fasi precedenti, come la scelta di implementare nuove soluzioni hardware o concepire possibilità completamente diverse rispetto a quello che ci si era prefissati all'inizio.

In queste pagine di relazione si andrà ad esplicare la versione finale di Amerigo 2.0, senza disdegnare però qualche dettaglio emerso più avanti lungo il percorso di sviluppo, così come visioni o scelte alternative che non si son potute concretizzare e soluzioni alternative che sono state introdotte in loro vece e le motivazioni che hanno spinto verso queste scelte.

COMPONENTISTICA E HW

Per lo sviluppo del progetto sono stati impiegati diversi componenti, scelti accuratamente per le esigenze e gli scopi previsti dal progetto.

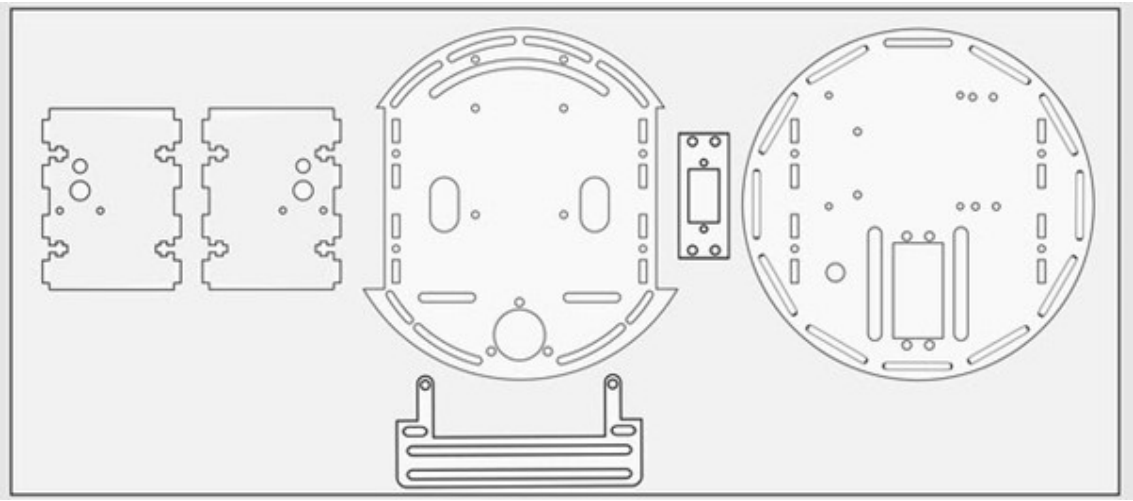
RASPBERRY



Gli obiettivi e le necessità realizzative del progetto hanno richiesto l'adozione di componenti dalle dimensioni ridotte e che richiedessero un dispendio energetico minimo. Secondo queste necessità è stato utilizzato un ***raspberry pi zero w*** dotato di scheda di rete WIFI, CPU ARM1176JZF-S 1 GHz e di 512 MB di memoria RAM. Inoltre il dispositivo dispone di sufficienti pin di connessione per la

gestione dei componenti successivamente installati sull'automa. Sul *raspberry pi zero w* è stata installata la distribuzione *Raspbian Buster Lite*.

CHASSIS E ALIMENTAZIONE

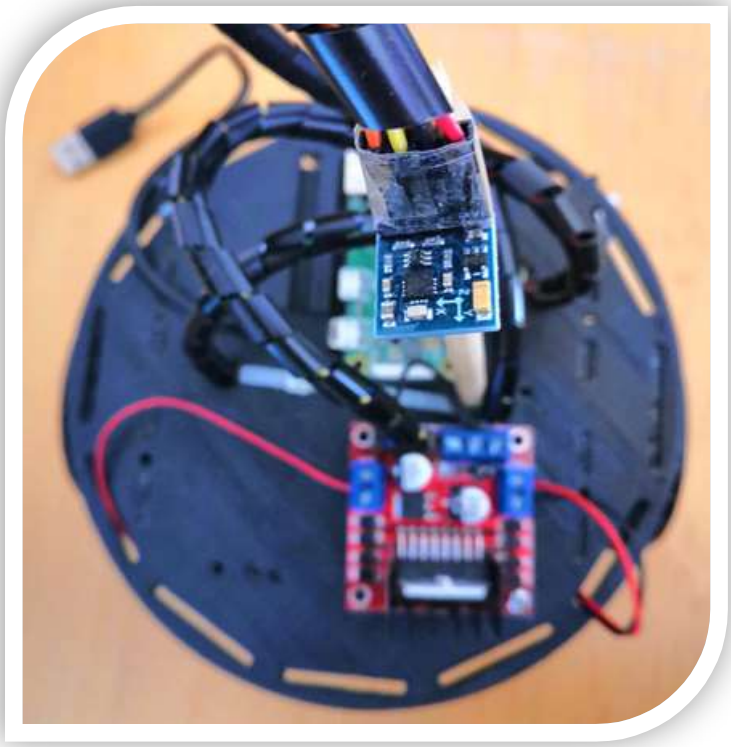


La scelta del telaio, dopo alcune considerazioni, si è indirizzata verso una forma circolare che permettesse un'elevata agilità nei movimenti tra gli ostacoli e un ridotto ingombro. Infatti, tale chassis, ha richiesto l'utilizzo di soltanto due ruote motrici permettendo così di effettuare rotazioni sul posto riducendo al minimo la possibilità di collidere con ostacoli durante le manovre. Il telaio è stato interamente prodotto utilizzando una stampante 3D e perfezionato secondo le esigenze richieste dal progetto.

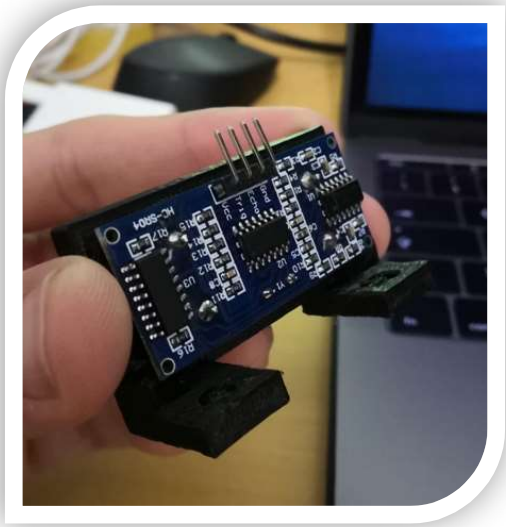
L’alimentazione è stata affidata ad una power bank modello ***Mophie Powerstation XL*** con una capacità di 12000mAh dotata di due ingressi usb, utilizzati rispettivamente per l’alimentazione dei motori e del raspberry pi zero w. La power bank è stata installata nella parte centrale del robot per ottenere un buon bilanciamento dei pesi considerando che il robot è provvisto di due ruote motrici laterali e un’unica ruota di appoggio anteriore.

BUSSOLA

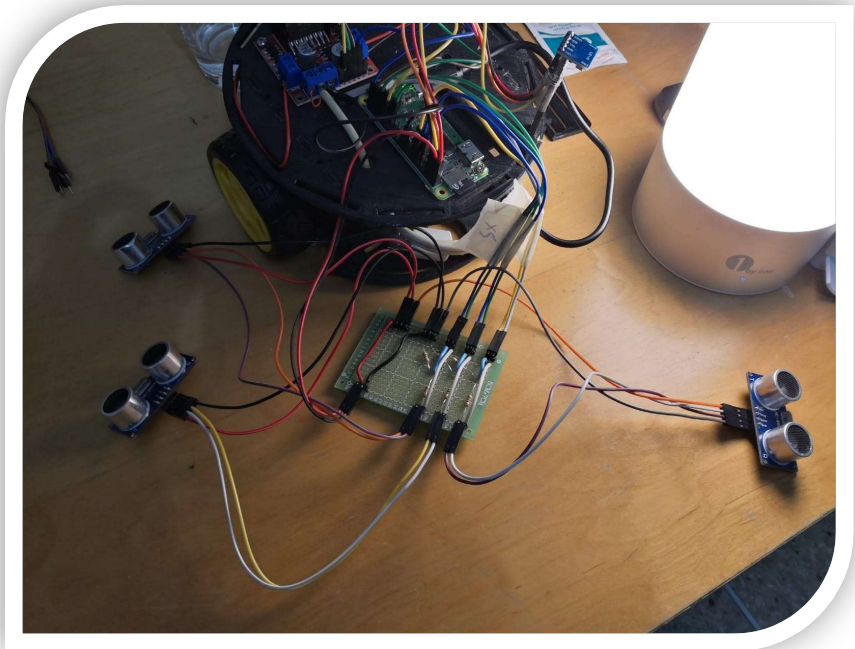
Per quanto riguarda la gestione dell’orientamento del dispositivo è stata impiegata una bussola digitale ***GY-271*** dotata di chip HMC5883L, disponendo di interfaccia I2C l’integrazione è stata piuttosto semplice e l’utilizzo di pin di comunicazione ridotto. Il dispositivo, di piccole dimensioni, è stato posto in una posizione il più possibile distanziata da fonti di interferenze elettromagnetiche che potessero interferire con le rilevazioni effettuate.



SENSORI DI PROSSIMITÀ



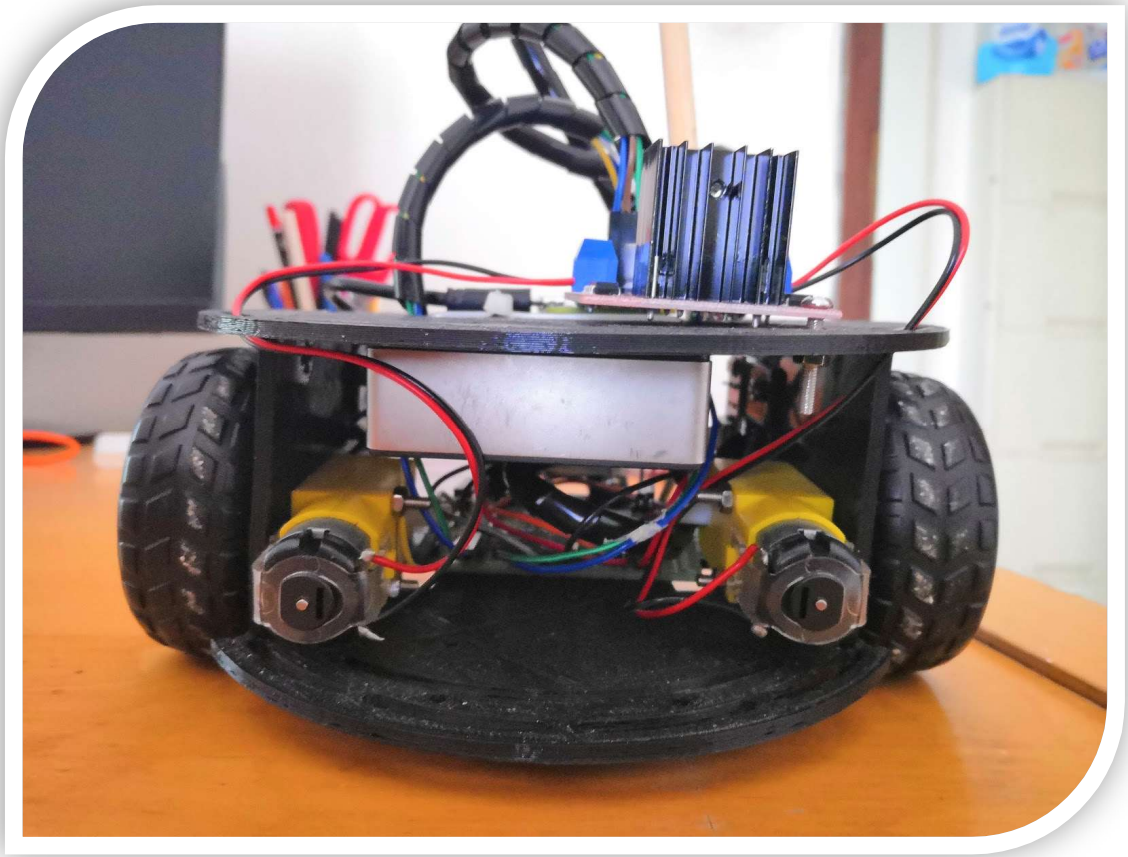
Per il monitoraggio dell'ambiente circostante al robot sono stati impiegati 3 sensori di prossimità ad ultrasuoni ***HC-SR04***. Tali sensori sono stati utilizzati per misurare la distanza del robot da eventuali ostacoli in tre diverse direzioni, rispettivamente: sinistra, fronte, destra. Per la loro installazione è stato pensato un circuito ad-hoc, dotato di 6 resistenze da 1K Ohm (2 per ogni *Echo*), che potesse utilizzare il minore numero di pin dal *raspberry pi zero w* e rendesse ordinato e compatto il cablaggio dei cavi di collegamento. Mediante il circuito costruito è stato possibile utilizzare un unico canale di alimentazione (positivo, neutro) per tutti e tre i sensori impiegati.



Dopo diversi test volti a verificare la corretta risposta dei sensori, il circuito è stato installato sul robot. Successivamente sono stati prodotti, mediante l'utilizzo di una stampante 3D, tutti i supporti necessari per l'alloggiamento dei sensori di prossimità nelle rispettive direzioni.

MOTORI E BOARD DI GESTIONE POTENZA

La parte motrice e direzionale è stata affidata a due motori elettrici ***DC*** con alimentazione 3V~ 6V. Inoltre, per la modulazione della potenza dei due motori si è resa necessaria l'implementazione di una board di controllo dotata di chip ***L298N***. La board e i motori sono poi stati alimentati alla power bank installata nella parte superiore del robot.



SOFTWARE

L'intero progetto è stato realizzato esclusivamente in linguaggio **Python**, inoltre per l'utilizzo dei componenti hardware e per alcune funzioni specifiche sono state utilizzate le seguenti librerie:

- *Pigpio*[1] libreria utilizzata per la gestione dei pin GPIO relativi a motori e sensori di prossimità.
- Drivers e tools per chip *HMC5883L*[2] utili per la configurazione, calibrazione e l'ottenimento di dati dalla bussola.
- *Matplotlib*[3] libreria utilizzata per la creazione e salvataggio della mappa ambientale generata autonomamente dal robot.

Il sistema è strutturato secondo diverse classi manager, ognuna destinata alla gestione di una specifica area del progetto:

- ***RouteManager*** è la classe manager in cui risiede la logica per la gestione della "rotta" o direzione verso uno specifico goal e la logica per l'aggiramento di eventuali ostacoli incontrati sul cammino del robot, in sintesi gli algoritmi per la navigazione.
- ***ProximityManager*** è la classe in cui risiedono i metodi necessari all'ottenimento dei dati relativi alla distanza del robot dagli ostacoli e ne gestisce eventuali eccezioni dettate dalla possibile collisione con questi ultimi.
- ***MapFileManager*** è la classe in cui risiedono i metodi per la registrazione dei dati necessari alla generazione della mappa ambientale.

Per semplicità e praticità nei vari test è stata creata una classe ***Configurator*** che si occupa di inizializzare tutti i pin necessari del *raspberry pi zero w* in fase di avvio del sistema. Tale classe inizializza i pin e il relativo hardware secondo quanto presente nel file di configurazione *conf.json* presente nella root del progetto.

```
{
  "Motors":{
    "RIGHT_MOTOR_FORWARDS" : 12,
    "RIGHT_MOTOR_BACKWARDS" : 16,
    "LEFT_MOTOR_FORWARDS" : 20,
    "LEFT_MOTOR_BACKWARDS" : 21
  },
  "Proximity":{
    "Triggers":{
      "LEFT" : 5,
      "FRONT" : 6,
      "RIGHT" : 13
    },
    "Echoes":{
      "LEFT" : 17,
      "FRONT" : 27,
      "RIGHT" : 22
    }
  }
}
```

Esempio di file di configurazione hardware. I pin sono numerati secondo la numerazione Broadcom

[1] http://abyz.me.uk/rpi/pigpio/
[2] https://github.com/RigacciOrg/py-qmc5883l
[3] https://matplotlib.org/

INTERAZIONE SW-HW

Il progetto include diverse classi che interagiscono con l'hardware installato. Queste, mediante l'esposizione di diversi metodi, permettono, inoltre, alle classi manager di gestire ad alto livello le azioni hardware necessarie.

BUSSOLA

La classe ***Compass*** si occupa di ottenere le rilevazioni riguardanti la posizione del robot mediante l'interrogazione della bussola. Inoltre, tale classe include un metodo di utilità tra cui *getRotationDegreeCosts* il quale una volta indicati i parametri, in gradi, dei punti di partenza e arrivo effettua il calcolo dei gradi che si dovrebbero percorrere in senso orario e in senso antiorario per raggiungere il punto di arrivo. Tale metodo è risultato molto utile per ottimizzare le rotazioni del robot nel verso che richiede un minor numero di gradi e quindi minor tempo nello spostamento verso un punto di arrivo ben specifico.

MOTORI

La classe ***Motors*** contiene i metodi per la gestione delle azioni fondamentali per lo spostamento del robot: spostamento in avanti, spostamento indietro, rotazione oraria, rotazione antioraria e arresto. A queste azioni sono poi state aggiunte altre ad hoc come la rotazione oraria di 90° e rotazione antioraria di 90°. Le rotazioni in senso orario e antiorario sono state realizzate ottenendo un movimento "sul posto" del robot consentendogli di ruotare in piccoli spazi. Questo è stato reso possibile invertendo i sensi di rotazioni dei due motori a velocità costanti.

La classe dispone, inoltre, di metodi per l'aggiornamento in tempo reale della potenza dei motori che si sono rivelati molto utili nella gestione della navigazione in modalità *normalMode* (algoritmo descritto nei paragrafi successivi).

SENSORI

La classe ***Proximity*** si occupa di interrogare i tre diversi sensori di prossimità (sinistra, destra, fronte) per ottenere informazioni circa la distanza degli oggetti dal robot. Utilizzando la libreria *pigpio* è stato possibile utilizzare dei meccanismi di callback configurati sui pin di *trigger* e *echo* dei sensori. Tali callback vengono richiamate ad ogni cambio di stato del pin ad esse associato, mediante tale meccanismo e tenendo traccia del *tick*, ovvero del numero di microsecondi trascorsi dal *boot* del device, è stato possibile calcolare con molta precisione il tempo impiegato dal segnale dal momento dell'invio alla sua ricezione e di conseguenza la distanza percorsa.

NAVIGAZIONE

Per risolvere la questione della navigazione (ovvero il problema proprio della robotica di trovare un percorso libero da collisioni che porti un robot da una configurazione di partenza a quella di arrivo) ci si è ispirati agli **algoritmi bug**, che si basano sulla capacità del robot di percepire gli ostacoli quando vicini, mediante sensori di prossimità o di contatto, per circumnavigare tali ostacoli e quindi aggirarli e puntare nuovamente verso l'obiettivo.

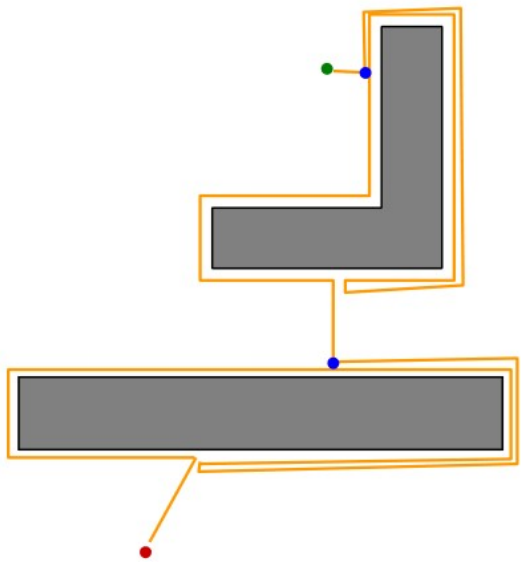
STATO DELL'ARTE: ALGORITMI BUG

Esistono diverse tipologie di algoritmi Bug, il cui nome deriva dalle strategie adoperate da alcuni insetti, come le formiche, per aggirare ostacoli improvvisi trovati lungo il cammino, tuttavia tutte si basano sulle seguenti ipotesi:

1. Il robot viene visto come un punto senza controllo sull'orientamento
2. Il robot conosce la sua posizione attuale, quella di partenza e quella di arrivo e la distanza (x,y) tra i vari punti.

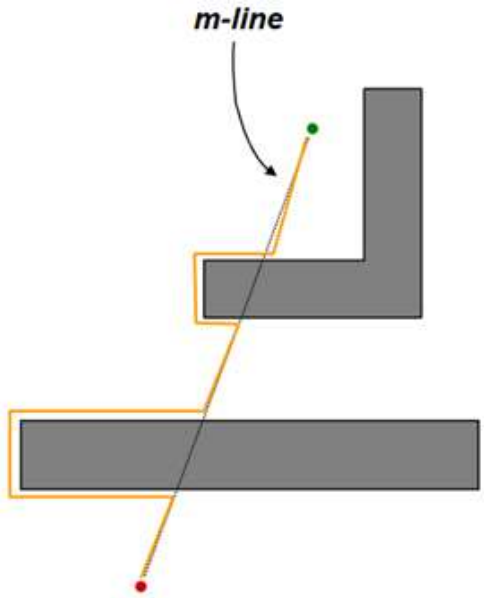
L'attuale stato dell'arte prevede due forme per tali algoritmi bug 1 e bug 2. Segue quindi una breve introduzione degli stessi:

L'algoritmo **Bug 1** prevede due tipi di comportamento per il robot: il ***motion to goal***, che a partire da un cosiddetto *leave point*, fa seguire il robot una linea retta (*m-line*) che collega il *leave point* al *goal,* ed il ***boundary following***, che si attiva qualora questo incontri un ostacolo, in un punto di primo contatto che viene definito *hit point*, da qui il robot circumnaviga l'ostacolo completamente fino a tornare al punto di partenza, l'*hit point,* lungo l'intero perimetro il robot sarà in grado di capire la sua distanza dal goal e quindi, una volta fatto il giro, potrà tornare, seguendo nuovamente gli spigoli dell'ostacolo, al punto più vicino al goal e quindi ripartire da lì col *motion to goal.*



Il **Bug 2**, d'altro canto, prova a risolvere gli ovvi problemi di tempo dovuti alla completa circumnavigazione degli ostacoli del primo algoritmo adottando una soluzione diversa: il nuovo

algoritmo la *m-line* resta fissa e nella fase di *boundary following* non compie un giro completo bensì, si stacca dall'ostacolo non appena avrà raggiunto nuovamente la *m-line* in un *leave point* dove tornerà al *motion to goal*.



Entrambi gli algoritmi sono tuttavia imperfetti e si dimostrano l'uno migliore dell'altro a seconda dei casi in analisi.

L'ALGORITMO DI NAVIGAZIONE DI AMERIGO 2.0: LE BASI DEL ROUTEMANAGER

Per quanto riguarda Amerigo 2.0, il semplice riportare un algoritmo Bug 1 o Bug 2 non sarebbe stato possibile per via della mancanza di diversi punti delle premesse iniziali di tali algoritmi, troppo idealistiche e poco definibili sul piano pratico con l'hardware discusso nei capitoli precedenti: Amerigo 2.0 non ha alcuna conoscenza della sua posizione (x, y) sul piano né di quella dell'obiettivo, se non solamente della direzione di quest'ultimo rispetto a sé. Inoltre il robot non può essere visto come un oggetto puntiforme ma il suo orientamento gioca un ruolo fondamentale nella sua struttura e nella definizione di tutti i suoi algoritmi di navigazione. Infine va sottolineato come il robot non disponga di sensori di contatto sulla sua intera superficie bensì di soli tre sensori di prossimità disposti sulla parte anteriore dello stesso con orientamento, rispettivamente, frontale, laterale destro e laterale sinistro, il che avrebbe reso quasi impossibile il concetto della circumnavigazione che sta alla base di entrambi gli algoritmi bug.

Quindi, prendendo ispirazione dall'algoritmo bug 2, è stato definito un algoritmo che consentisse al robot di orientarsi verso l'obiettivo ed, eventualmente, evitare ostacoli, anche complessi.

L'algoritmo di navigazione ideato per Amerigo 2.0 prevede anch'esso due tipi di comportamento, raffrontabili al *motion to goal* e *boundary following* dei normali algoritmi bug, qui chiamati rispettivamente ***Normal Mode*** e ***Bug Mode***. Il primo si occupa di guidare l'automa qualora non vi siano ostacoli lungo il suo percorso al raggiungimento dell'obiettivo definito da uno specifico grado rilevato dalla bussola (*goal degree*), mentre il *Bug Mode*, si attiva, invece, come il *boundary*

following, all'incontro (ad una distanza minore o uguale a 20cm) con un ostacolo frontale, e resta attivo fino a quando Amerigo 2.0 non avrà la via verso l'obiettivo libera.

L'ALGORITMO DI NAVIGAZIONE DI AMERIGO 2.0: IL NORMAL MODE

La logica della navigazione libera, ovvero in assenza di ostacoli, è stata implementata nel cosiddetto *normalMode*. L'algoritmo utilizza in modo combinato la bussola, per ottenere i gradi attuali a cui il robot punta in un esatto momento, e i motori riducendo o aumentando la potenza degli stessi per effettuare gli aggiustamenti necessari a perseguire la rotta stabilita. Più in dettaglio la gestione della potenza dei motori sinistro e destro avviene secondo la seguente logica:

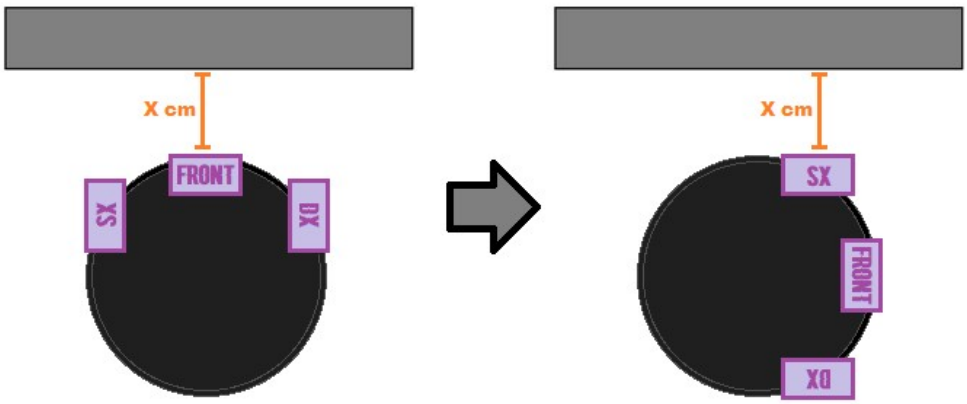
- *Obiettivo posizionato a sinistra del robot*, viene gradualmente ridotta la potenza del motore sinistro per permettere una leggera rotazione verso sinistra.
- *Obiettivo posizionato a destra del robot*, viene gradualmente ridotta la potenza del motore destro per permettere una leggera rotazione verso destra.
- *Obiettivo posizionato di fronte al robot*, viene gradualmente ripristinata la potenza di default a entrambi i motori per perseguire l'obiettivo prefissato.

Si veda quindi un esempio, nella pagina seguente dello pseudocodice alla base del funzionamento dell'algoritmo *NormalMode*.

```
#Caso di stop del thread o normalMode disabilitato
if status == STOPPED or normal_mode_status == 'DISABLED':
    return

#Rilevazione distanza oggetti da sensori di prossimità
proximity_manager.retrieveProximityData()

#Check lato frontale occupato da osacolo
if proximity_manager.getAvailability('FRONT') is False:
    normal_mode_status = 'DISABLED'
    bug_mode_status = 'ENABLED'

    #Reset variabile velocità motori
    motors.restoreMotorActualPowerToDefault()
    return

#Check stato motori, nel caso in cui siano fermi vengono attivati
if motors.getMotorsStatus() == STOPPED:
    motors.forward()

#Ottenimento gradi attuali rotta
degrees = compass.getDegress()

#Memorizzazione velocità attuale motori destro e sinistro
motor_left_actual_power = motors.getMotorLeftActualPower()
motor_right_actual_power = motors.getMotorRightActualPower()

#Caso in cui si è a sinistra dell'obiettivo
if ( degrees + compass_tolerance ) < goal_direction_degrees:

    #Check se la potenza motore è decrementabile ulteriomente, oltre la soglia di potenza per
    #cui si fermerebbe
    if motor_right_actual_power - motors_deceleration_step >= 140:
        motor_right_actual_power = motor_right_actual_power - motors_deceleration_step
        motors.updateMotorPower( 'RIGHT', motor_right_actual_power )

elif ( degrees - compass_tolerance ) > goal_direction_degrees:

    #Check se la potenza motore è decrementabile ulteriomente, oltre la soglia di potenza per
    #cui si fermerebbe
    if motor_left_actual_power - motors_deceleration_step >= 140:
        motor_left_actual_power = motor_left_actual_power - motors_deceleration_step
        motors_object.updateMotorPower( 'LEFT', motor_left_actual_power )

else: #Caso in cui obiettivo in posizione frontale

    #Ripristino potenza motori a velocità massima impostata
    motors_object.forward( True )
```

L'ALGORITMO DI NAVIGAZIONE DI AMERIGO 2.0: IL BUG MODE

Per ragioni di limitazioni hardware e divergenze con le premesse ideali degli algoritmi bug, e proprio a causa della disposizione "a T rovesciata" dei sensori di prossimità che consentono una visione limitata a tre soli punti dello spazio circostante al robot (con dei "coni ciechi" di 90° tra gli stessi), si è deciso di sostituire il concetto di circumnavigazione con delle rotazioni fisse di 90°. Così che, per fare un esempio, un eventuale ostacolo rilevato dal sensore frontale, dopo una rotazione standard in senso orario si trovasse a coincidere (entro un certo margine di tolleranza) con la rilevazione laterale sinistra, in modo da mantenere una certa continuità con quanto rilevato.

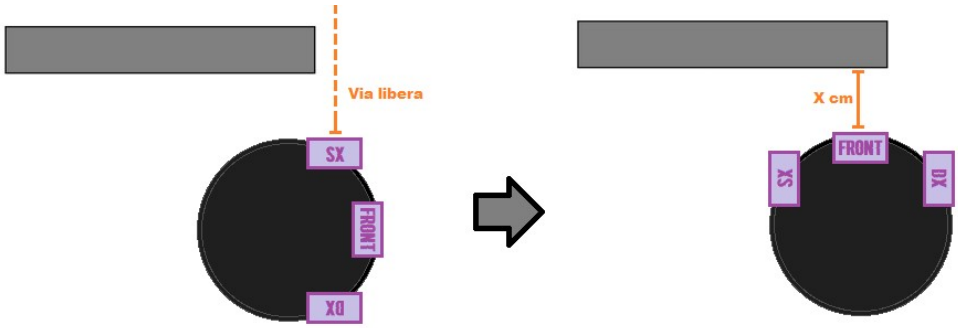


Il *BugMode* parte quindi all'occasione dell'incontro col primo ostacolo frontale, dall'*hitpoint* si troverà già in condizioni di dover scegliere come comportarsi attuando il cosiddetto *pathfinder*, algoritmo che si occuperà di verificare mediante i sensori laterali quale delle rispettive vie sia libera e, di conseguenza come voltarsi prima di proseguire. Nello specifico si avrà un giro antiorario per la sinistra, orario per la desta, due giri da novanta gradi (***u-turn)*** in caso di vicolo cieco e una scelta randomica tra le prime due in caso in cui entrambe le strade laterali siano libere. Quest'ultima possibilità è stata concepita per far sì che, vista l'assenza di una capacità mnemonica che permetta ad Amerigo 2.0 di capire di essere tornato ad un bivio precedentemente affrontato, sia garantita la possibilità di trovare, prima o poi, la strada "giusta" che consenta al robot di procedere.

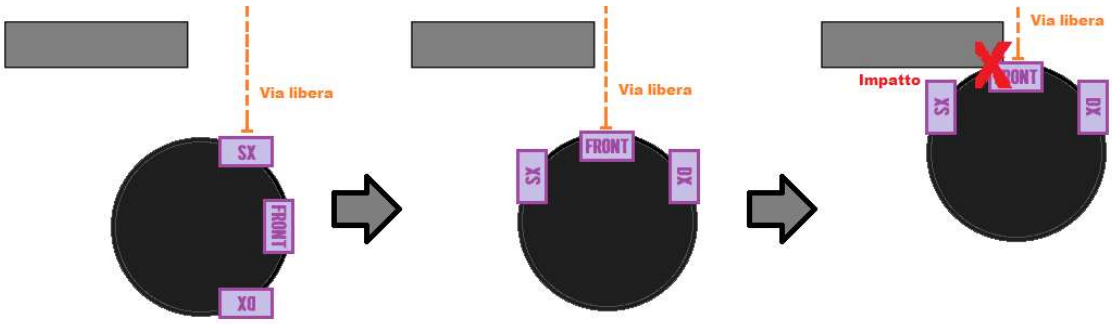
Lo scopo del *BugMode* sarà, da qui in poi, quello di trovare il prima possibile una strada libera che consenta al robot di tornare a girarsi verso il proprio obiettivo originale dato da un valore, in gradi rilevati dalla bussola, definito ***parent degrees,*** che viene salvato, alla prima svolta del robot, a partire dal ***goal degree*** iniziale, per poter garantire così l'uscita corretta dal *BugMode* e la riattivazione del *NormalMode.* Amerigo 2.0 tenderà quindi a muoversi dritto, orientandosi verso il nuovo *goal degree* (definito dopo essersi girato) fino a quando non troverà un nuovo ostacolo o una via libera verso l'obiettivo. Per garantire ciò è stato utilizzato il concetto di ***locked direction***, ovvero quella direzione (destra o sinistra) verso il quale il robot dovrà girare alla prima svolta libera, per cercare di tornare ad orientarsi verso il *parent degree*. Unico caso in cui la *locked direction* non viene registrata è per lo *u-turn,* laddove l'algoritmo farà sì che il robot proveniente da una inversione di marcia svolti verso la prima strada libera trovata, a prescindere che questa sia a destra o a sinistra.

Di base la *locked* direction è determinata dal verso opposto a quello del giro effettuato (E.g.: se l'automa inizialmente ha girato verso la sua sinistra, dovrà prendere la prima svolta a destra per tornare a rivolgersi verso l'obiettivo precedente) ad eccezione del caso in cui si provenga da uno *u-turn,* in tal caso l'algoritmo è programmato per settare, dopo aver girato, una locked direction concorde al senso di svolta così da poter completare l'angolo piatto e tornare a rivolgersi verso l'obiettivo.

Le prime fasi di testing hanno poi portato due importanti esigenze:

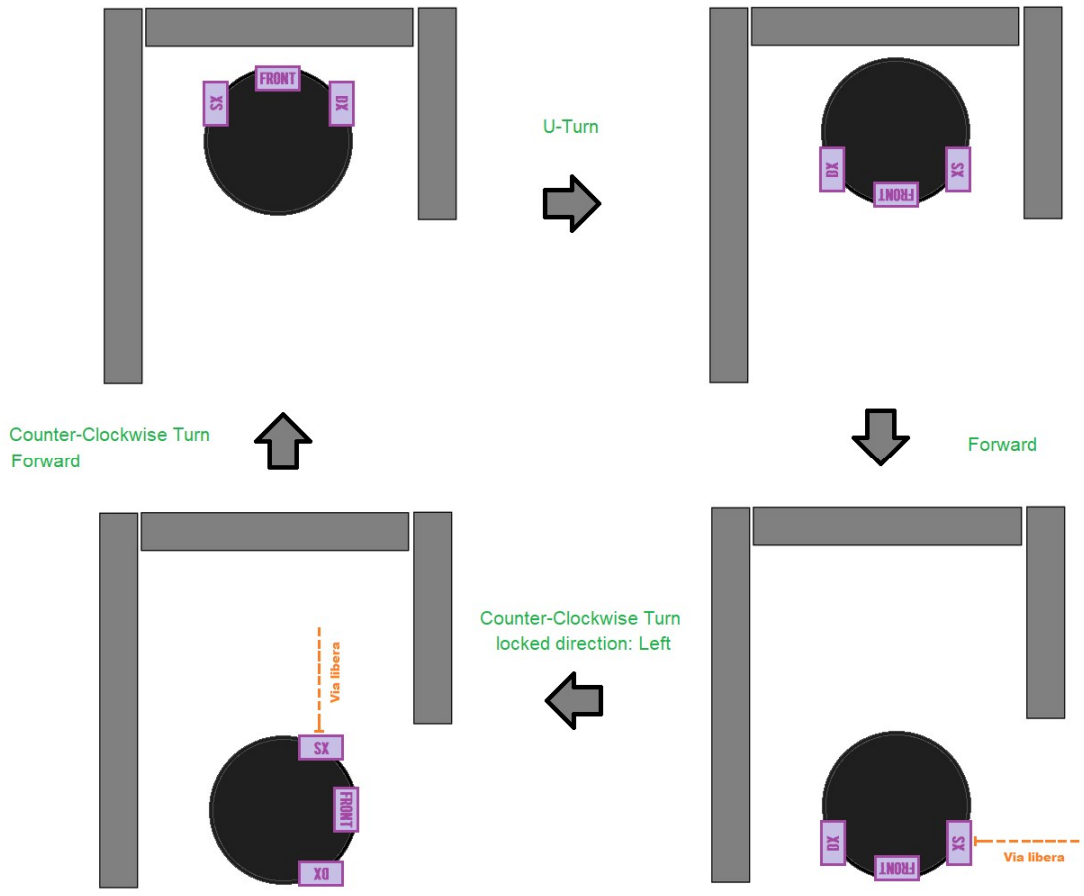


In primo luogo, come precedentemente indicato, si è notato che la struttura non puntiforme dell'automa, unita al posizionamento nella parte anteriore dello stesso dei tre sensori di prossimità, così come alla scelta progettuale di compiere le rotazioni tutte attorno all'asse centrale del corpo del robot e al ridottissimo "cono visivo" dei sensori, potesse portare ad alcune situazioni limite in cui una apertura individuata lateralmente venisse rilevata come un ostacolo dal sensore frontale dopo essersi girati (in quanto i sensori laterali sono disposti nella parte anteriore dell'automa e non alle estremità del suo diametro, vista la presenza delle ruote in tali posizioni) oppure, caso di gran lunga peggiore, individuata l'apertura laterale il robot imboccasse una strada attraverso la quale non potesse passare l'intera dimensione del suo corpo nonostante i sensori l'avessero individuata come libera (per intenderci, come un cane con un lungo bastone in bocca che cerca di passare attraverso una porta stretta e ci resta incastrato).



Per ovviare a questo primo problema è stato introdotto il concetto di ***before turn step***, un numero di passi da compiere in avanti prima di girarsi dopo aver visto una possibile via di uscita laterale. Tale valore viene resettato al suo valore massimale (2) ogniqualvolta l'uscita individuata non si dimostri sufficientemente ampia (venendo sostituita dalla rilevazione di un nuovo ostacolo laterale nel passo successivo). In questo modo si è arricchito l'algoritmo per far sì che Amerigo 2.0 si giri sempre in modo corretto verso le sole uscite da cui è certo di passare.

Secondo fattore critico è stato quello del giro dopo uno *u-turn*; per ovvie ragioni, un robot proveniente da una inversione di marcia al primo giro effettuato verso la prima strada libera trovata, come da algoritmo, avrebbe come *locked direction* la direzione concorde al giro effettuato, tuttavia qui si troverebbe a rivolgere il sensore corrispondente alla *locked direction* verso la strada appena percorsa, la quale sarebbe libera per ovvie ragioni, e quindi tenderebbe a tornare un'altra volta verso il vicolo cieco entrando così in un ciclo infinito.



Il problema è stato risolto introducendo un'altra forma di step, ***u-turn step***, utilizzato come contatore per fare un ulteriore *before turn step* in tali situazioni, facendo sì che Amerigo 2.0 imbocchi la nuova strada individuata prima di iniziare a cercare la via verso la *locked direction*.

Infine, situazioni più complesse e serie di ostacoli studiati a tavolino per portare all'estremo la difficoltà del dedalo (si veda nel paragrafo relativo ai test l'esempio della "scala inversa"), hanno fatto emergere la necessità di tener traccia per il *parent degree* (e conseguente *locked* direction) sia dell'angolo corrispondente al *goal* originale sia di quello del *goal degree* dovuto alla svolta precedente. Motivo per cui è stato necessario definire a livello di classe alcuni parametri come il ***goal_direction_degrees***, o la ***main_locked_direction***.

A seguire uno pseudocodice esplicativo del *bug Mode*, si noti, che nella versione attuale tutti i confronti coi dati rilevati dalla bussola sono soggetti ad una tolleranza e che l'ultima parte è solo un'analisi di due casistiche di svolte sulle quattro possibili (uturn, a sinistra, a destra e randomica).

```
Let goal_direction_degrees = Goal Degrees,   main_locked_direction = None,   before_turn_steps = 0, u_turn = False.

Funzione BugMode (goal_degrees, parent_degrees = goal_direction_degrees, locked_direction = None)

1   - Rilevo prossimità

2   - IF lockedDirection ≠ None →
        - IF Ostacolo rilevato nella locked_direction AND self.before_turn_steps ≠ Max →
            - self.before_turn_steps = Max
        - ELIF Via verso locked_direction Libera AND self.before_turn_steps == 0 →
            - IF locked_direction == right → Clockwise Turn
            - ELIF locked_direction == left → Counterclockwise Turn

            - IF Non sono rivolto verso l'obiettivo principale →
                - RETURN bugMode(parent_degrees, self.goal_direction_degrees, self.main_locked_direction)

            - IF main_locked_direction ≠ None →
                - main_locked_direction = None

            - RETURN bugMode(parent_degrees) #Caso in cui si torna a guardare l'obiettivo principale.#


3   - IF self.u_turn == false AND self.before_turn_steps == 0 AND Fronte Libero →
        - Enable normalMode
        - RETURN  #Fine BugMode

        ...
```

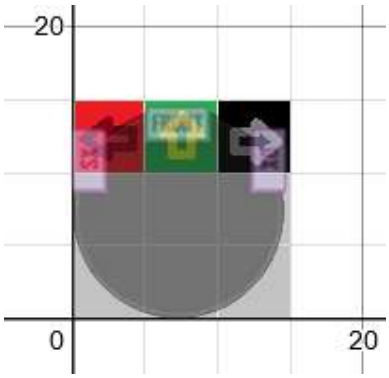
```
   ...

4   - IF Fronte libero →
        - IF self.before_turn_steps > 0 OR (self.u_turn == false AND lockedDirection == None OR via verso la
lockedDirection chiusa) OR (self.u_turn == false AND vie laterali tutte chiuse) →
            - Go Forward
            - self.before_turn_steps --
            - RETURN bugMode(goal_degrees, parent_degrees, locked_direction)


5   - self.before_turn_steps = Max #visto che, arrivato qui, sto per girare
    - IF Sinistra Occupata →
        - IF Destra Occupata →
            - Clockwise Turn
            - Clockwise Turn
            - self.u_turn = true
            - RETURN bugMode(Gradi attuali bussola, self.goal_direction_degrees)

#I goal degree sono i nuovi gradi rilevati dalla bussola, i gradi parent sono i vecchi goal direction, non ho
bisogno della locked in caso di uturn

        - ELSE →
            - IF self.u_turn == true AND self.u_turn_forward_steps == 0 →
                - self.before_turn_steps = self.max_before_turn_steps - 1
                            - self.u_turn_forward_steps = 1
                            - RETURN bugMode(goal_degrees, parent_degrees)

            - Clockwise Turn

            - IF self.u_turn == true →
                - self.u_turn_forward_steps = 0
                - self.before_turn_steps = Max
                - self.u_turn = false
                - RETURN bugMode(Gradi attuali bussola, self.goal_direction_degrees, right)
            - ELSE
                -IF (goal_degrees - parent_degrees) == 0 → self.main_locked_direction = left

            - RETURN bugMode(Gradi attuali bussola, self.goal_direction_degrees, left)


        ...
```

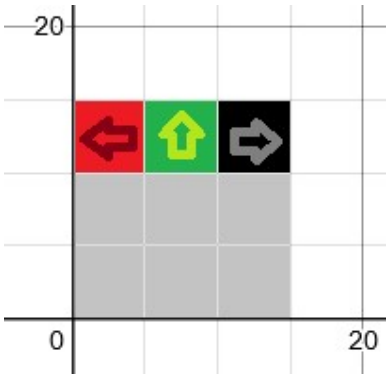
MAPPING

Ultimato l'algoritmo di navigazione si è deciso di implementare un'ultima funzionalità a Amerigo 2.0 che consentisse di mappare gli ostacoli, all'interno di un grafico cartesiano, ogniqualvolta venisse rilevato uno, partendo di conseguenza all'attivazione del *BugMode*.

Per favorire una visione omogenea, dando un certo margine d'errore alle varie rilevazioni (laddove tra piccole variazioni di potenza dei motori, oscillazioni varie e imprecisioni nelle rilevazioni dei sensori, uno stesso ostacolo potrebbe essere rilevato in posizioni leggermente differenti) è stato deciso di rapportare il tutto ad una **griglia di 5x5 cm** virtualmente posta sul piano, facendo quindi occupare ad Amerigo 2.0, coi suoi 15cm di diametro, un blocco di 3x3 quadrati ad ogni movimento. Al momento del lancio dell'algoritmo l'automa andrebbe quindi ad occupare lo spazio quadrato tra i punti (0,0), (0,15), (15,15), (15,0), quadrato suscettibile ad una traslazione di +/- 10 cm sull'ascissa o l'ordinata a seconda dell'orientamento del robot per ciascun movimento (ogni movimento frontale tra un ciclo di *bugMode* e l'altro, qualora venga effettuato, è infatti di 10 centimetri esatti).

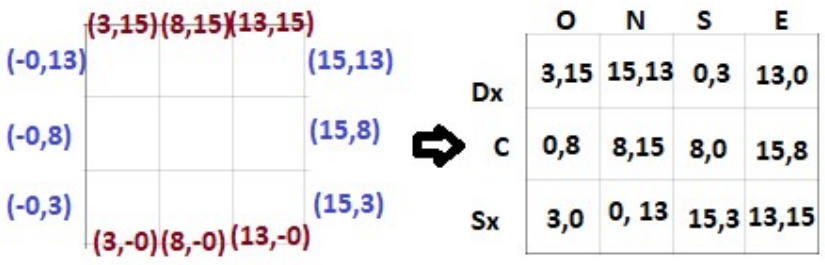
I sensori di quest'ultimo sono stati abbinati alle celle superiori del quadrato, ovvero, nel caso della prima rilevazione, ai quadrati (0,10), (0,15), (5,15), (5,10) per il sensore sinistro, (5,10), (5,15), (10,15), (10,10) per il centrale e (10,10), (10,15), (15,15), (15,10) per quello destro.





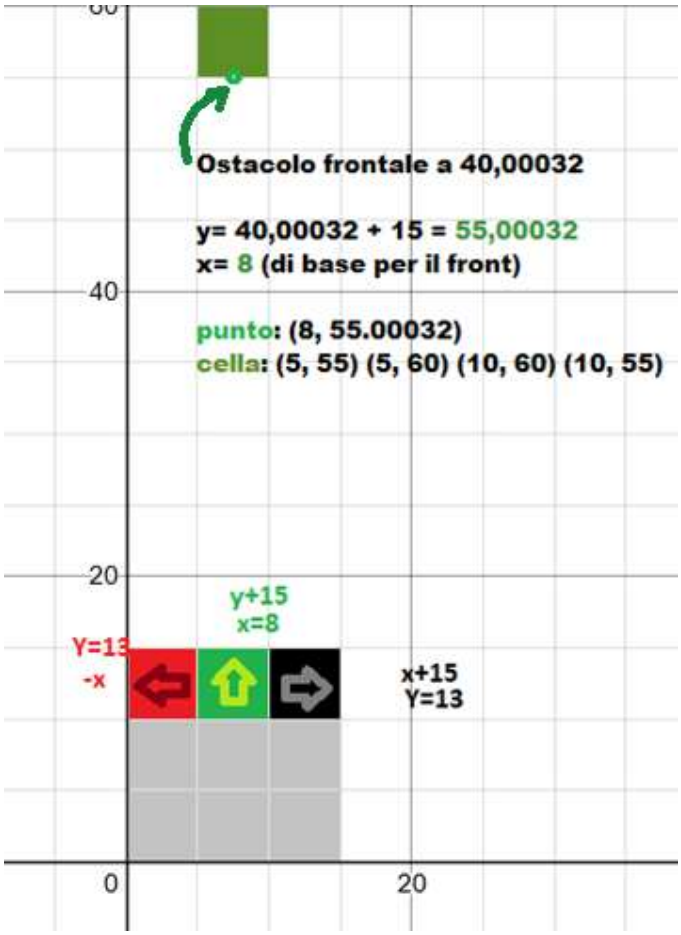
Di conseguenza, ogni rilevazione effettuata nel *BugMode* è stata salvata all'interno di un file m*ap_data.map* nella cartella *map_files* e, data la posizione dei sensori precedentemente esplicata, all'interno della classe ***MapAnalyzer*** è stato definito l'algoritmo capace di tracciare la mappa delle rilevazioni effettuate all'interno del piano cartesiano. Tali rilevazioni vengono modulate a seconda dell'orientamento del robot (ricavato dalle rotazioni effettuate combinate ad una matrice circolare) ed incrementando o diminuendo ciascun valore sulla base di una matrice ben definita i cui valori vengono a loro volta ri-calcolati in base al movimento effettuato dal robot.

Si veda un esempio della rappresentazione della matrice di modulazione delle misurazioni alla prima rilevazione, prima che l'automa effettui alcuno spostamento, le righe rappresentano i vari sensori mentre le colonne rappresentano le varie orientazioni possibili (definite dai punti cardinali individuati dall'orientamento del robot rispetto al cartesiano, infatti la prima rilevazione corrisponderà sempre ad un orientamento a nord).

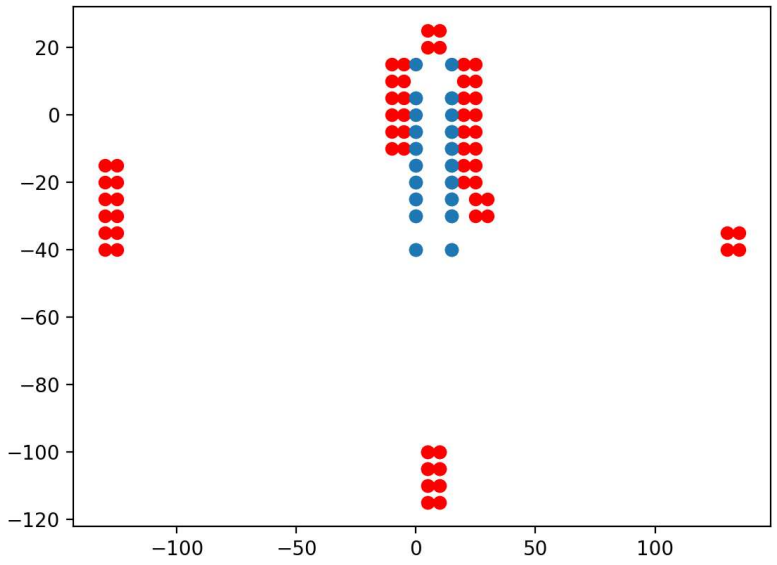


	O	N	S	E
Dx	3,15	15,13	0,3	13,0
C	0,8	8,15	8,0	15,8
Sx	3,0	0, 13	15,3	13,15

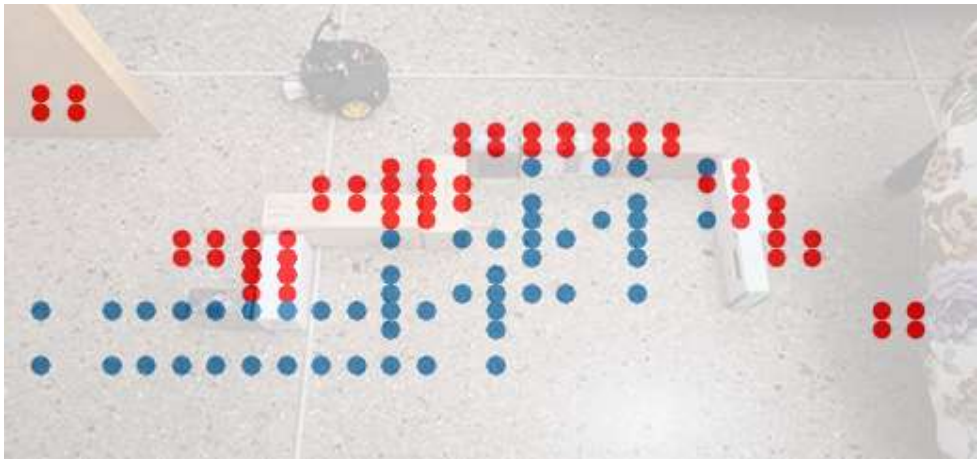
Da notare quindi un ultimo esempio, usato in fase di definizione dell'algoritmo, di come una rilevazione frontale iniziale vada corrisposta alla rispettiva cella della griglia 5x5.



I punti così ottenuti sono stati poi utilizzati mediante la libreria *Matplotlib* per strutturare una mappa ambientale nel grafico cartesiano, andando ad indicare con punti rossi i vertici delle celle individuate come ostacoli e, tramite dei punti azzurri, i vertici di quelle occupate dal robot lungo il suo moto nel dedalo. Andando ad ottenere così degli output del genere:



Esempio di mapping risultante dopo un vicolo cieco. In blu lo spazio occupato da Amerigo 2.0 durante il percorso.



Sovrapposizione tra la mappa ottenuta e l'effettiva disposizione degli ostacoli nell'esempio della "scala inversa".

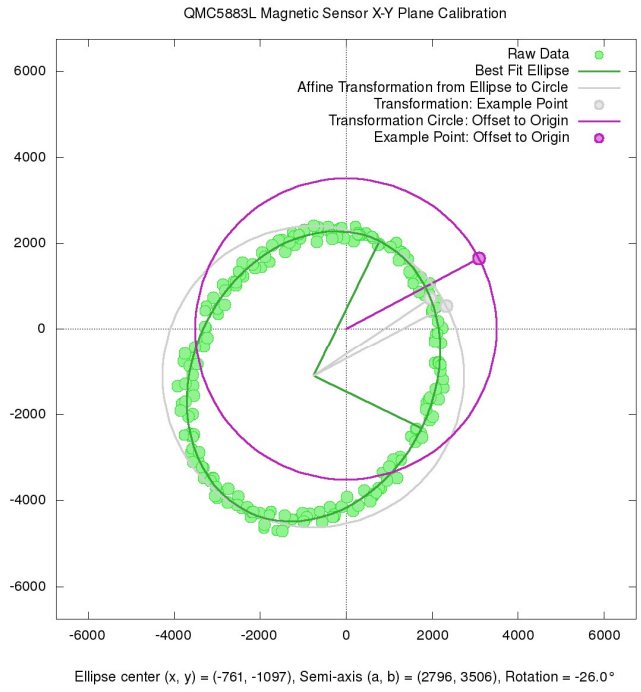
PROBLEMATICHE ED IDEE SCARTATE

Sin dall'inizio del progetto sono state incontrate diverse problematiche che hanno portato a scelte e soluzioni differenti da quanto prefissato in principio, inoltre alcune funzionalità pensate inizialmente si sono rivelate poi di difficile o addirittura impossibile implementazione con l'hardware a disposizione. Il corretto funzionamento della sensoristica ha rappresentato una delle più grandi difficoltà incontrate, probabilmente dovuta anche alla bassa qualità realizzativa dei componenti, considerata l'adozione di hardware dimostrativo a basso costo.

SENSORISTICA

La sensoristica ha richiesto molte ore di lavoro per il suo affinamento e la calibrazione di alcune componenti: per l'orientamento inizialmente era stato impiegato un giroscopio che potesse rilevare tutti gli eventuali spostamenti rotativi effettuati dal robot, tuttavia Il dispositivo presentava dei limiti tecnici, in quanto tra gli assi monitorati non era presente l'asse Z questo lo rendeva inadatto allo scopo finale del progetto.

Successivamente il giroscopio è stato sostituito da una bussola digitale che rilevasse i dati di posizionamento del robot nei tre assi tridimensionali. La bussola ha richiesto molto tempo per il suo corretto funzionamento: più volte si è resa necessaria la sua calibrazione rispetto all'asse magnetico terrestre per l'ottenimento di rilevazione affidabili, sebbene quest'ultima richiedesse diverso tempo per essere effettuata correttamente e di essere ripetuta parecchie volte nel corso del tempo per funzionare correttamente (e non riportare la bussola come un ellisse).



Proprio per questo motivo si è deciso di modulare ogni rotazione dell'automa sul tempo di attivazione dei motori e la potenza degli stessi piuttosto che sulla gradazione angolare rispetto al nord magnetico giacché un'ellisse decentrato per questioni di mancate, o errate calibrazioni sarebbe

stato troppo sensibile ad alcune variazioni di grado e poco ad altre (si veda un esempio di sistema di calibrazione nell'immagine sottostante e la forte differenza tra l'ellisse dato di base e la circonferenza corrispondente). È stato poi necessario distanziare la bussola digitale da qualsiasi fonte di interferenza elettromagnetica che ne potesse influenzare le rilevazioni.

I sensori di prossimità, infine, presentavano saltuariamente dei problemi durante le rilevazioni: vi era una certa percentuale di rilevazioni i cui dati erano inappropriati o assenti. Questo fenomeno presentava un grande problema in quanto il robot in movimento doveva in tempo reale avere informazioni attendibili circa la vicinanza di possibili ostacoli e, nel caso, evitarne la collisione. La problematica in questione è stata risolta adottando due diverse soluzioni: per ogni sensore (destro, sinistro, fronte) si è deciso di effettuare ogni volta effettuate due rilevazioni, scartati gli errori, tra queste viene poi selezionata la misura minore ovvero quella più attendibile. In questo caso si riduce la probabilità che un valore sia nullo o non corretto. La seconda soluzione è stata quella di mantenere fermo il robot in caso si verifichino eccezioni relative alle rilevazioni dei sensori di prossimità. Il robot ripartirà soltanto una volta ottenuti dati attendibili sulle distanze dagli oggetti.

BLUETOOTH

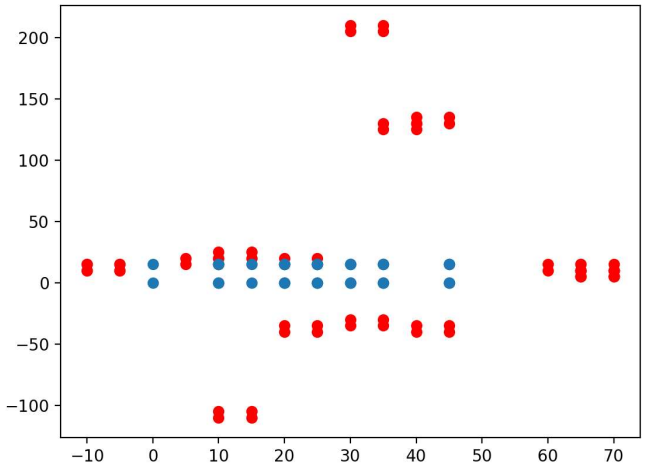
In origine si era pensato di identificare il goal come un beacon che, mediante il proprio segnale, riuscisse ad indicare la propria posizione al robot. È stata realizzata una piccola app per dispositivi Android che simulasse il comportamento di un beacon: tra i due dispositivi veniva stabilita una connessione mediante un socket che permettesse loro di scambiare informazioni sia posizionali che di navigazione. Dopo diversi test ci si è resi conto che la tecnologia Bluetooth in sé non permette la comunicazione di una specifica posizione esatta nello spazio, ma ricondurrà sempre ad un segnale che ne rappresenta approssimativamente la posizione. Lo scopo iniziale del progetto era di raggiungere in modo esatto l'obiettivo e per questo motivo la tecnologia Bluetooth è stata poi abbandonata in favore del perseguimento di una rotta ben precisa mediante bussola digitale.

TEST

Innumerevoli test di verifica, percorsi, dedali e strade più o meno irte di ostacoli sono stati fatti affrontare a Amerigo 2.0 sia nel corso che alla fine dello sviluppo del codice che ha portato al risultato finale. Tra i molteplici si vogliono riportare qui tre casi indicativi e la mappa tracciata dall'automa al loro completamento, la scelta non è casuale in quanto questi specifici esempi hanno fatto emergere in primo luogo delle difficoltà (alcune anche individuate a livello teorico addirittura prima di affrontare il percorso fisicamente) che sono state superate attraverso le scelte descritte precedentemente in queste pagine.

SERIE DI OSTACOLI

Il più semplice dei tre casi, prima prova pensata dopo il mero ostacolo frontale, consisteva in un dedalo costruito da un primo ostacolo frontale che portasse ad una prima svolta laterale e al ritorno all'orientamento verso l'obiettivo prima di raggiungere un secondo ostacolo frontale che bloccasse il percorso un'altra volta. Lo scopo del test era di verificare sia il corretto eseguimento del B*ugMode* che il corretto attivarsi e disattivarsi del N*ormalMode*.

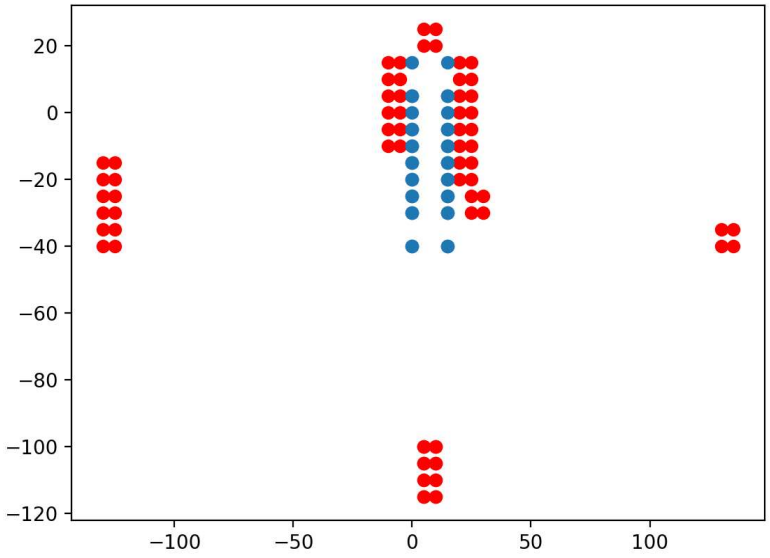


Mapping corrispondente alla seconda attivazione del bug mode

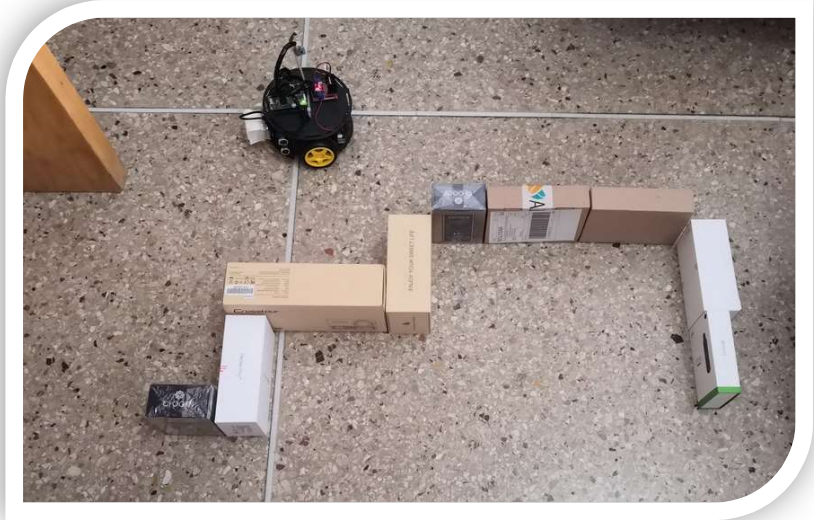
Si noti come, vista la possibilità di movimento randomico al primo ostacolo siano stati posizionati successivamente due ostacoli speculari questa serie di ostacoli ha portato ad un miglior affinamento delle interruzioni tra *Bug* e *Normal Mode*, oltre che a rendere necessari alcuni miglioramenti alla sensoristica descritti nel paragrafo precedente.

VICOLO CIECO

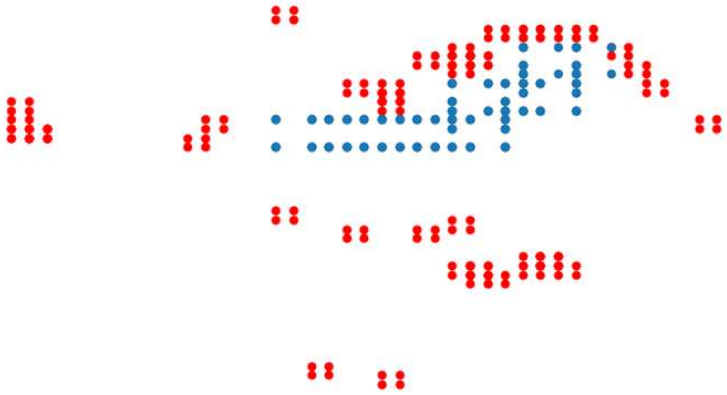
Caso ampiamente discusso in precedenza il vicolo cieco ha portato alla definizione dello u-turn nell'algoritmo di navigazione, all'esigenza di non incorporare, in questo caso, una locked direction per garantire la possibilità al robot di uscire dal labirinto imboccando la prima strada libera individuata e all'introduzione dello *step* specifico per lo *u-turn* per evitare di incappare nel caso del ciclo infinito descritto nel paragrafo sul *bug Mode*.



“SCALA INVERSA”



Ultimo caso, più complesso di tutti i precedenti, anche da trovare, è stato pensato appositamente per mettere in difficoltà l’automa, facendolo compiere forzatamente una serie di svolte che lo allontanassero sempre di più dall’obiettivo, senza mai farlo voltare verso la *locked direction* se non dopo serie di svolte in direzioni diverse. Tale processo ha portato alla luce l’esigenza di poter conservare in memoria sia il parametro del *goal direction degree,* che non sarebbe sempre potuto coincidere col *parent degree* come inizialmente ipotizzato, che mantenere una *main locked direction* da ricordare nel tempo in caso di svolte in direzioni opposte alla direzione augurata.



CONCLUSIONI E SVILUPPI FUTURI

Nonostante le problematiche incontrate e i cambi di rotta intrapresi per affrontarle, è lecito affermare che Amerigo 2.0 è egregiamente riuscito nel suo intento: il robot costruito interamente da zero con componentistica low-budget riesce completamente ad orientarsi in un ambiente sconosciuto e a districarsi in labirinti complessi, evitando ostacoli lungo il percorso e a seguire liberamente la sua strada verso un obiettivo stabilito. Inoltre, la scelta di implementare la funzionalità di mapping dello spazio esplorato apre virtualmente le porte a diversi scenari futuribili per Amerigo 2.0: potrebbe essere interessante, infatti, utilizzare tali mappe, unite ad uno specifico algoritmo di apprendimento per consentire al robot di muoversi con maggior sicurezza in un ambiente già esplorato, tenendo conto, per l'appunto degli ostacoli registrati e consentendo di determinare, con qualche approssimazione, la propria posizione al suo interno.

Alla luce di ciò un hardware più affidabile e preciso potrebbe sicuramente migliorare l'interazione del robot con l'ambiente circostante per generare delle mappe più affidabili, in ottica di apprendimento, ma, nonostante ciò, il progetto può dirsi concluso con successo.

Va sottolineato come nei mesi di sviluppo siano state molteplici le competenze, ben delineate in queste pagine, acquisite dal team di sviluppo sia sulla capacità di programmare un software orientato al paradigma dell'IoT sia in campi diversi come la robotica ed il making. Sicuramente un'esperienza del genere ha portato ad una forte ammirazione verso chi della progettazione e lo sviluppo di automi simili, ma di gran lunga più performanti di Amerigo 2.0, fa una ragione di vita: dai (non affatto) semplici robot aspirapolveri a chi progetta automi da inviare nell'ultima frontiera, ad esplorare nuovi mondi, per arrivare coraggiosamente là dove nessuno è mai giunto prima.