Ingegneria Informatica

Laboratorio di Automatica

Programmazione Python di un dispositivo
LEGO MINDSTORMS EV3

Docente:
Michele Basso

Studenti:
Abdullah Chaudhry
Paula Mihalcea

A.A. 2019 - 2020

Indice

1 Introduzione **3**
1.1 Programma di navigazione autonoma . . . 3
1.1.1 Inizializzazione . . . 3
1.1.2 Loop principale . . . 4
1.1.3 Loop interno . . . 4
1.2 Installazione . . . 5

2 LEGO EV3 MicroPython **8**
2.1 Programmazione MicroPython . . . 8
2.1.1 LED . . . 9
2.1.2 Pulsanti . . . 10
2.1.3 Sensore a ultrasuoni . . . 10
2.1.4 Motori . . . 10
2.2 Navigazione autonoma: MicroPython . . . 11

3 ev3dev **13**
3.1 Programmazione Python . . . 13
3.1.1 LED . . . 14
3.1.2 Pulsanti . . . 15
3.1.3 Sensore a ultrasuoni . . . 15
3.1.4 Motori . . . 15
3.2 Navigazione autonoma: Python . . . 17

4 Alcune osservazioni **19**
4.1 Considerazioni sui tempi di esecuzione . . . 19
4.1.1 Motori . . . 19
4.1.2 Loop . . . 20
4.2 Prime conclusioni . . . 20

5 Controllo remoto **21**
5.1 Architettura del sistema . . . 21
5.2 Acquisizione dell'input (client) . . . 22
5.3 Ricezione dell'input (server) . . . 22
5.4 Controlli . . . 23
5.4.1 Movimento con la levetta analogica . . . 24
5.5 Multithreading . . . 25
5.5.1 Rilevamento ostacoli . . . 25
5.5.2 Informazioni . . . 25

5.5.3 Logging 25
5.6 Plotting 26

6 Conclusioni **27**

7 Codice sorgente (controllo remoto) **28**
7.1 Server 28
7.2 Client 39
7.3 Plot 41

1 Introduzione

L'intento della presente relazione è quello di fornire una breve guida all'installazione di due diversi sistemi operativi su un dispositivo LEGO MINDSTORMS EV3[11], **LEGO EV3 MicroPython** e **ev3dev**, così come alla programmazione tramite i linguaggi **MicroPython** e **Python**, al fine di realizzare un confronto tra i due sistemi per valutarne pregi e difetti.

Inoltre, verrà descritto in dettaglio un programma di controllo remoto del brick attraverso una rete WiFi ed un gamepad, completo di rilevamento ostacoli e logging della velocità.

I sistemi ivi presentati consistono in due immagini (.img) da scaricare dai siti web dei rispettivi sviluppatori ed installare su una microSD card. La relazione dunque descriverà il procedimento per la loro configurazione e passerà alla descrizione di alcune funzioni MicroPython (*LEGO EV3 MicroPython*) e *Python* (*ev3dev*), specifiche delle librerie rispettivamente utilizzate da ogni sistema. Tali librerie verranno infine utilizzate per la creazione di un programma, in entrambi i linguaggi descritti, per meglio valutarne gli usi e le potenzialità.

Tutto il codice prodotto per il progetto, assieme ad alcuni grafici ed un file di log, sono liberamente disponibili su GitHub all'indirizzo `https://github.com/chabdullah/Lego-Ev3-Python`[10].

1.1 Programma di navigazione autonoma

Il programma implementato nei due linguaggi consentirà al dispositivo LEGO MINDSTORMS EV3[1] di navigare autonomamente in un ambiente, girando di 90° ogni volta che individua un ostacolo. Le varie fasi di controllo sono strutturate come nella figura 1, e vengono brevemente descritte di seguito in quanto sono generiche ed indipendenti dal linguaggio in cui saranno implementate.

1.1.1 Inizializzazione

Durante l'**inizializzazione** vengono definiti e configurati i motori ed i sensori necessari. Viene regolato anche il modo in cui l'EV3 gestisce gli ostacoli attraverso alcuni parametri, ovvero:

- la velocità dei motori;
- lo sterzo;
- la distanza minima rilevata alla quale il brick si dovrà fermare in presenza dell'ostacolo;

[1] Da qui in poi chiamato, per semplicità, *dispositivo LEGO*, *EV3* oppure *brick*, quest'ultimo termine derivato dalla parola inglese comunemente usata in rete per indicarlo.

- la velocità massima dei motori (solo LEGO EV3 MicroPython).

Viene utilizzata anche una variabile ausiliaria `started` come flag per indicare se il brick è fermo o in movimento, ai fini di permettere una corretta transizione tra gli stati.

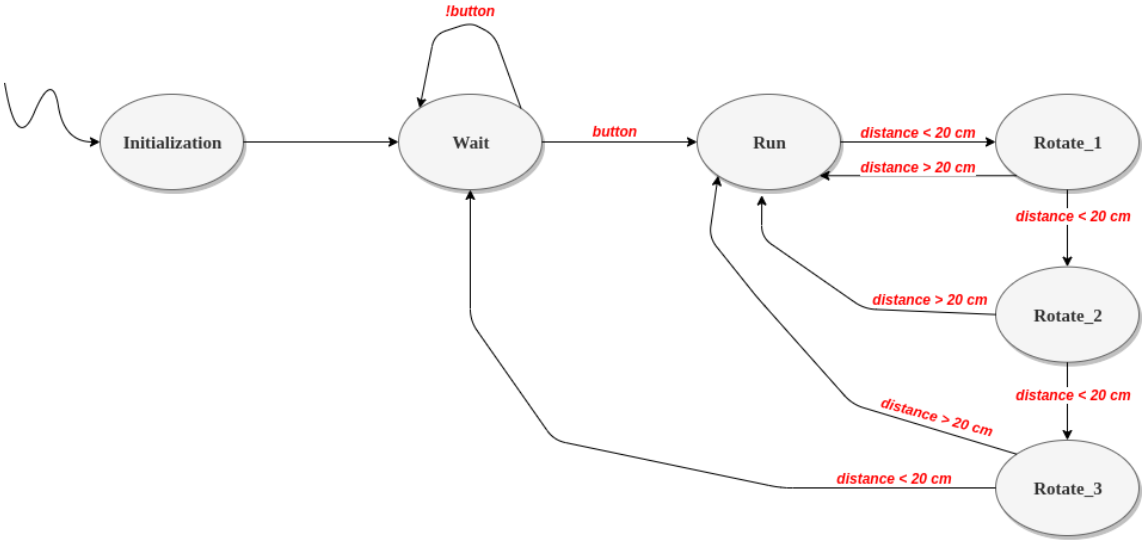

Figura 1: Diagramma di flusso del programma di navigazione autonoma.

1.1.2 Loop principale

Una volta inizializzati i parametri si entra in un ciclo *while* infinito, inizialmente in attesa che venga premuto un qualsiasi pulsante che avvii il brick facendone cambiare lo stato (*Wait* $\rightarrow$ *Run*); tale stato di attesa è indicato da entrambi i LED accesi di colore giallo.

Alla pressione di un tasto sull'EV3 i motori vengono avviati ed il flag `started` viene impostato a `True`.

1.1.3 Loop interno

Una volta che il brick è stato avviato (stato *Run*), esso richiede ciclicamente la distanza dell'ultima lettura effettuata dal sensore ad ultrasuoni. Se la distanza dell'EV3 è inferiore al valore di soglia (20 *cm* nel codice riportato), il brick si ferma e passa alla nuova fase (*Run* $\rightarrow$ *Rotate*) in cui il brick ruota sul posto per coprire un angolo di 90° con una certa velocità sterzo (in base ai parametri con cui è stato inizializzato).

Se dopo quattro letture successive il brick non trova alcun percorso disponibile (ovvero la variabile `d` è scesa a 0), significa che si trova racchiuso tra quattro mura,

e di conseguenza si ferma impostando il flag `started` a `False`, passando così nuovamente allo stato *Wait*. In caso contrario, l'EV3 avvia di nuovo i motori e ritorna nello stato *Run*.

1.2 Installazione

L'installazione dei due sistemi operativi sul brick è sostanzialmente identica nei passi base; cambiano, infatti, soltanto le immagini dei sistemi e le estensioni dell'ambiente di sviluppo Visual Studio Code che permettono la programmazione in MicroPython/Python. Prima di iniziare, tuttavia, sono necessari i seguenti componenti:

- un dispositivo LEGO MINDSTORMS EV3;
- una microSD (capienza minima 4GB (2GB per ev3dev), ma non eccedente i 32 GB o di tipo microSDXC, in quanto non supportata);
- un computer con adattatore microSD;
- un cavo mini-USB per comunicare con il LEGO MINDSTORMS.

Passo 1: download del software

Per cominciare è necessario scaricare dal sito LEGO l'immagine del sistema desiderato (LEGO EV3 MicroPython[9] oppure ev3dev[4]). Serve inoltre anche un tool per installare l'immagine sulla microSD; in questa documentazione si userà Etcher[2] (vedi figura 2).

Passo 2: flash della microSD

Si prosegue installando il tool per il flash della microSD e, una volta avviato, si seleziona l'immagine (che avrà un'estensione del tipo *.img*). Basterà a questo punto inserire la microSD card nell'adattatore del computer (che Etcher riconoscerà automaticamente), e proseguire con il flash.

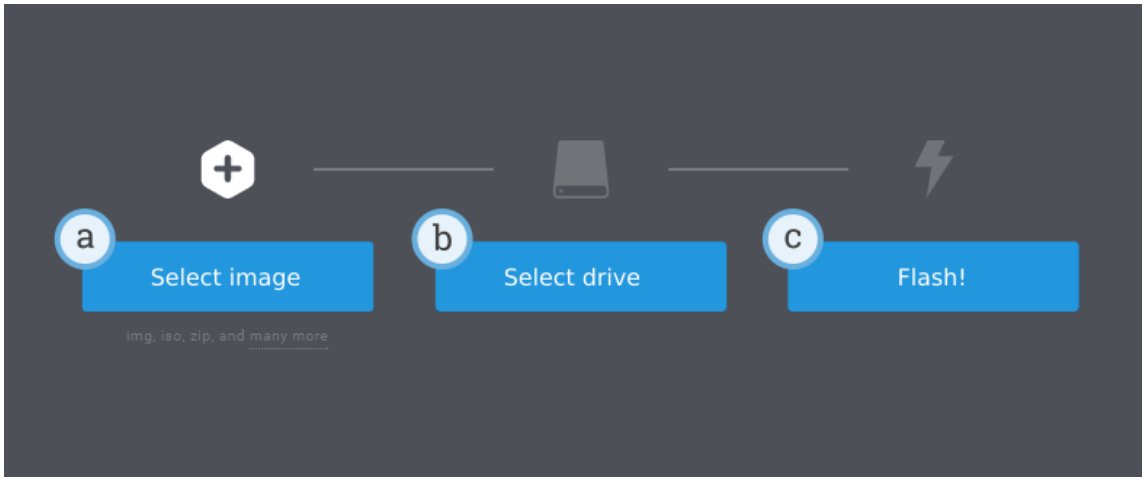

Figura 2: Schermata iniziale di Etcher.

Passo 3: primo avvio del brick

Completata l'installazione dell'immagine sulla microSD, la si inserisce nell'EV3 (da spento) e si accende il dispositivo (figura 3). Nel caso del sistema ev3dev il primo avvio richiederà diversi minuti, in quanto l'EV3 deve configurare l'host SSH ed altre impostazioni di base. Le schermate nelle figure 4 e 5 indicano la corretta installazione dei rispettivi sistemi.

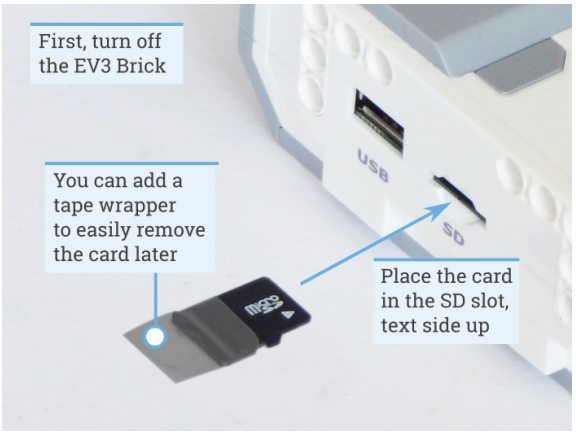

Figura 3: Inserimento della microSD card e primo avvio del brick.

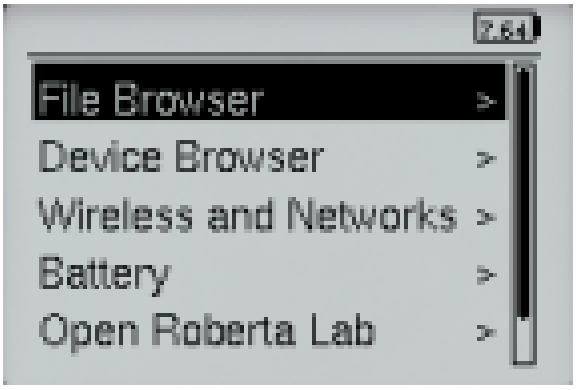

Figura 4: Schermata iniziale del sistema **LEGO EV3 MicroPython** all'avvio.

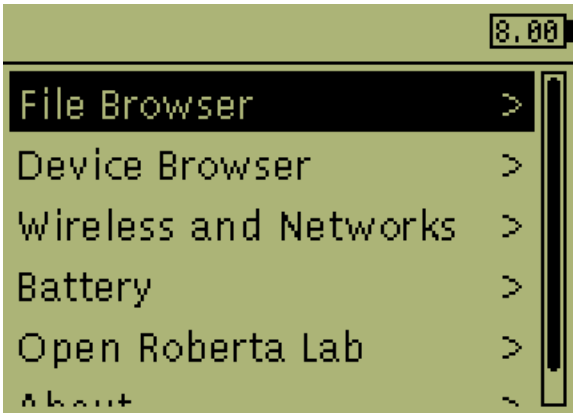

Figura 5: Schermata iniziale del sistema **ev3dev** all'avvio.

Passo 4: installazione di VS Code

Sia la programmazione in MicroPython che quella in Python richiedono l'ambiente di sviluppo **Visual Studio Code**, liberamente scaricabile dal sito ufficiale[16].

Per poter utilizzare le librerie di ciascun sistema e riconoscere ed inviare il codice al brick è necessaria anche l'installazione di due apposite estensioni, chiamate **EV3 MicroPython** e **ev3dev-browser**, reperibili nella sezione *Extensions* di VS Code inserendo nella barra di ricerca i loro nomi (figura 6).

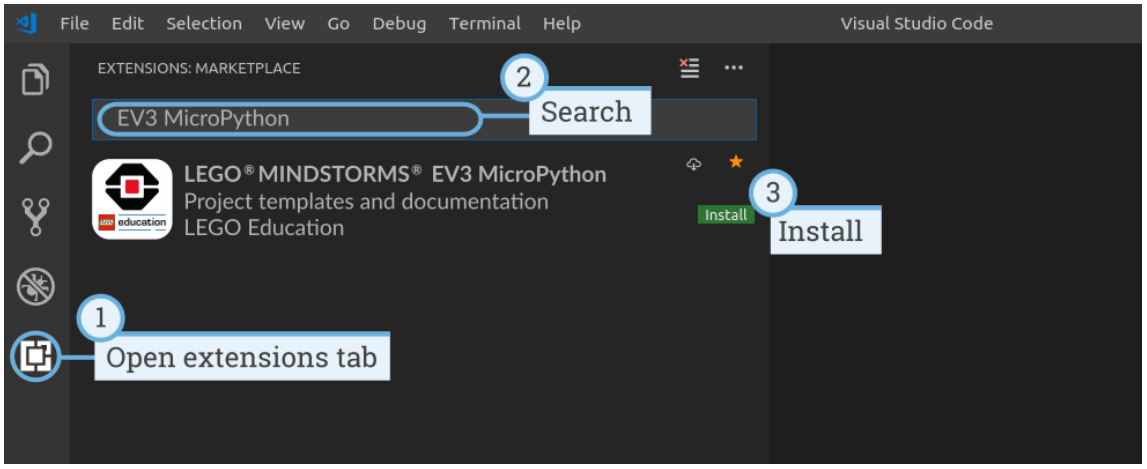

Figura 6: Installazione dell'estensione EV3 MicroPython in Visual Studio Code.

2 LEGO EV3 MicroPython

LEGO EV3 MicroPython è un sistema operativo ufficiale fornito da LEGO per consentire agli utenti del MINDSTORMS EV3 la programmazione di quest'ultimo in **MicroPython**, un'implementazione software del linguaggio Python 3 scritta in C ed ottimizzata per l'esecuzione su microcontrollori[12]. Non essendo open source, il sistema e le sue librerie sono da considerarsi *as is* e vanno presi così come si trovano sul sito LEGO, senza alcuna possibilità di sapere come effettivamente sono implementate le varie funzioni.

2.1 Programmazione MicroPython

Il sistema LEGO EV3 MicroPython permette la programmazione del brick nel linguaggio omonimo, ovvero il **MicroPython**. Per meglio illustrarne il funzionamento, seguono alcune classi e funzioni presenti nella documentazione ufficiale della libreria[7].

Istruzione shebang

```
#!/usr/bin/env pybricks-micropython
```

Quest'istruzione va sempre riportata nella prima riga del file di codice, in quanto è necessaria per specificare al sistema la corretta versione di MicroPython da utilizzare per l'esecuzione del programma.

Librerie

Subito dopo il shebang è necessario specificare le librerie da importare per poter utilizzare le funzioni del brick. In particolare, bisogna inserire le seguenti righe di codice per:

- specificare il dispositivo da utilizzare:

```
from pybricks import ev3brick as brick
```

- specificare gli ingressi e le uscite del brick (rispettivamente `S1`, `S2`, `S3`, `S4` e `A`, `B`, `C`, `D`):

```
from pybricks.parameters import (Port)
```

- importare le istruzioni necessarie per specificare la presenza di uno o più motori:

```
from pybricks.ev3devices import (Motor)
```

- utilizzare funzioni specifiche dei sensori montati a bordo del brick, in questo caso a infrarossi ed ultrasuoni (possibili opzioni includono anche: `TouchSensor`, `ColorSensor`, `GyroSensor`):

```
from pybricks.ev3devices import (InfraredSensor,
    UltrasonicSensor)
```

- controllare lo stato dei pulsanti sul brick:

```
from pybricks.parameters import (Button)
```

- specificare i parametri di alcune funzioni, rispettivamente direzione (`CLOCKWISE` o `COUNTERCLOCKWISE`), colore (del LED oppure rilevato dal sensore a infrarossi), azione dopo lo spegnimento del motore (`COAST`, `BRAKE`, `HOLD`);

```
from pybricks.parameters import (Direction, Color,
    Stop)
```

- inserire tempi di attesa attiva nel codice:

```
from pybricks.tools import wait
```

2.1.1 LED

Per cambiare il colore dei LED presenti a bordo dell'EV3 è sufficiente invocare la funzione `light(COLOR)` con il nome del colore desiderato al posto del parametro `COLOR` (valori possibili: `GREEN`, `YELLOW`, `RED` e `ORANGE`).

Si osserva che non è possibile controllare singolarmente i due LED (sinistro e destro) in quanto esiste un solo comando, che li controlla insieme.

```
brick.light(Color.RED)  # Brick LED is red
```

2.1.2 Pulsanti

Per controllare se un pulsante è premuto basta utilizzare la funzione `buttons()` insieme alla classe `Button` (che espone gli attributi `DOWN`, `LEFT`, `CENTER`, `RIGHT`, `UP`) per ottenere una lista dei pulsanti correntemente premuti, come nel seguente esempio:

```
Button.LEFT in ev3brick.buttons()  # Checks if the left button
    has been pressed
```

In alternativa è possibile invocare `buttons()` nel seguente modo per verificare se uno qualunque tra i pulsanti presenti sull'EV3 è stato premuto:

```
any(ev3brick.buttons())  # Checks if a button has been pressed
```

2.1.3 Sensore a ultrasuoni

Il sensore a ultrasuoni dell'EV3 si realizza, in MicroPython, con un oggetto ***UltrasonicSensor*** a cui va specificata la porta di input cui è collegato:

```
ultrasonic = UltrasonicSensor(Port.S2)
```

Il sensore ha un range compreso tra i 0 ed i 2550 mm e può essere interrogato tramite la funzione `distance()`, che al momento della chiamata restituisce una misura della distanza in millimetri e prosegue effettuando nuove rilevazioni in modo continuo, per metterle a disposizione di eventuali richieste future[2].

In alternativa è possibile richiedere una singola misurazione con `distance(silent=True)`, facendo però attenzione a non scendere sotto i 300 ms tra due letture successive, ovvero il tempo che il sensore impiega a spegnersi una volta eseguita la lettura (altrimenti potrebbe andare in errore).

2.1.4 Motori

I due motori montati sull'EV3 devono essere controllati separatamente, ed è quindi necessario creare due oggetti di tipo ***Motor***, specificando le porte di output a cui sono collegati:

```
motorL = Motor(Port.A)  # Left motor
motorR = Motor(Port.D)  # Right motor
```

Esistono diverse funzioni della classe *Motor* per far avanzare il brick, di cui la principale è `run()`, dove viene specificata la velocità angolare del motore:

```
motorL.run(motorSpeed)  # E.g. motorSpeed = 200 rad/s
motorR.run(motorSpeed)
```

[2]La frequenza di queste rilevazioni non è nota, ma in mancanza di ulteriori specifiche fornite nella documentazione ufficiale si può supporre che sia regolata in modo da non causare errori nelle letture, come accadrebbe se fosse troppo alta.

Altre varianti di `run()` permettono di specificare una modalità di avanzamento di ciascun motore in base ad un valore espresso in millisecondi oppure gradi:

```
run_time(speed, time)  # Run the motor at a constant speed for
    a given amount of time
run_angle(speed, rotation_angle)  # Run the motor at a
   constant speed (angular velocity) by a given angle
run_target(speed, target_angle)  # Run the motor at a constant
    speed (angular velocity) towards a given target angle
```

2.2 Navigazione autonoma: MicroPython

```
#!/usr/bin/env pybricks-micropython

from pybricks import ev3brick as brick
from pybricks.ev3devices import (Motor, UltrasonicSensor)
from pybricks.parameters import (Port, Stop, Button, Color)
from pybricks.tools import wait

# Initialization

brick.light(Color.RED)  # Brick LED is red during
   initialization

motorL = Motor(Port.A)  # Left motor
motorR = Motor(Port.D)  # Right motor
ultrasonic = UltrasonicSensor(Port.S2)  # Ultrasonic sensor

motorSpeed = 200  # Motor speed
steeringSpeed = 200  # Steering speed (to be used when turning
    around)
minDistance = 200  # Minimum distance (in mm) before the brick
    starts decelerating or stops to turn around if an obstacle
   is found
maxSpeed = 400

started = False  # Flag to be used as manual start/stop switch
   ; default is False (brick does not move)

while True:
    brick.light(Color.YELLOW)  # Brick LED is yellow when
       ready (waiting for a button press)
```

```
if any(brick.buttons()):  # If a button is pressed
    started = True  # The start/stop flag is set to True (
        start)
    brick.light(Color.GREEN)  # Brick LED becomes green
    # Run the motors up to 'motorSpeed' degrees per second
        :
    motorL.run(motorSpeed)
    motorR.run(motorSpeed)

while started is True:

    # Check if there is enough space; if not, stop, turn
        90 degrees, then start running again
    if ultrasonic.distance() < minDistance:
        # Stop the motors:
        tempSpeed = motorSpeed
        for i in range(1,100):
            tempSpeed = tempSpeed - 1
            motorL.run(tempSpeed)
            motorR.run(tempSpeed)
        motorL.stop()
        motorR.stop()

        # Turn 90 degrees until there is enough space to
            start running again:
        d = 4

        while ultrasonic.distance() < minDistance:
            motorL.run_angle(steeringSpeed, -174, Stop.
                COAST, False)
            motorR.run_angle(steeringSpeed, +174)
            d -= 1
            if(d == 0):
                started = False
                break

        # Run the motors up to 'motorSpeed' degrees per
            second:
        if d != 0:
            motorL.run(motorSpeed)
            motorR.run(motorSpeed)
```

3 ev3dev

ev3dev[3] è un sistema operativo basato su Linux Debian, creato in modo indipendente dalla comunità utilizzatrice dell'EV3 come alternativa al software ufficiale LEGO. Ricreando driver e librerie, si propone di fornire agli sviluppatori la capacità di programmare il brick sfruttando non solo le funzionalità già presenti sul dispositivo, ma anche quelle fornite in modo intrinseco dal sistema Linux. In questo modo è stata ampliata sia la gamma di funzioni a bordo dell'EV3 che quella dei dispositivi hardware con esso compatibili e dei linguaggi di programmazione supportati.

3.1 Programmazione Python

ev3dev permette, per sua natura, una grande flessibilità nella scelta del linguaggio di programmazione, come ad esempio Java, C e C++[5]. Tuttavia, per poter fare un confronto concreto con il linguaggio fornito ufficialmente da LEGO, ovvero il MicroPython, nella realizzazione del presente progetto è stato ritenuto opportuno utilizzare **Python**.

Istruzione shebang

```
#!/usr/bin/env python3
```

Similmente al sistema operativo descritto in precedenza, quest'istruzione va sempre riportata nella prima riga del file di codice per specificare la corretta versione di Python da utilizzare.

Librerie

Seguendo ancora il modello creato per il sistema LEGO EV3 MicroPython, anche per ev3dev bisogna specificare dopo il shebang le librerie da importare, ed in particolare quelle per:

- gli ingressi del brick (ogni stringa coincide con le porte numerate presenti sullo stesso):

  ```
  from ev3dev2.sensor import INPUT_1, INPUT_2, INPUT_3,
      INPUT_4
  ```

- gli output del brick:

  ```
  from ev3dev2.motor import OUTPUT_A, OUTPUT_B, OUTPUT_C
      , OUTPUT_D
  ```

- specificare la presenza di un motore *grande* ($18,33\ rad/s$) ed utilizzare le funzioni per farlo avanzare e sterzare:

```
from ev3dev2.motor import LargeMotor, MoveTank,
    MoveSteering
```

- alcune funzioni di conversione che permettono di inserire i valori della velocità per il motore in termini di gradi al secondo, gradi al minuto, rotazioni al secondo, rotazioni al minuto o percentuale[3]:

```
from ev3dev2.motor import SpeedDPS, SpeedDPM, SpeedRPM
    , SpeedRPS, SpeedPercent
```

- funzioni specifiche dei sensori presenti a bordo del brick (in questo caso a infrarossi ed ultrasuoni; altre possibili opzioni includono: `TouchSensor`, `ColorSensor`, `LightSensor`, `GyroSensor`, `SoundSensor`):

```
from ev3dev2.sensor.lego import InfraredSensor,
    UltrasonicSensor
```

- i pulsanti premuti sul brick:

```
from ev3dev2.button import Button
```

- i LED presenti sul brick:

```
from ev3dev2.led import Leds
```

- i tempi di attesa attiva nel codice (libreria Python):

```
from time import sleep
```

3.1.1 LED

Per controllare i LED presenti a bordo dell'EV3 è necessario creare un oggetto ***Leds***: `leds = Leds()`.

A questo punto per impostare i colori è possibile utilizzare la funzione `set_color (LED, COLOR)`, dove la variabile `LED` può essere `LEFT` o `RIGHT` (a seconda se si vuole controllare il LED sinistro o destro) ed il parametro `COLOR` può avere i seguenti valori: `BLACK, RED, GREEN, AMBER, ORANGE, YELLOW`[13]:

```
leds.set_color("LEFT", "RED")  # LED sinistro a rosso
leds.set_color("RIGHT", "YELLOW")  # LED destro a giallo
```

In alternativa è possibile impostare il livello, in percentuale (da 0 a 1), di rosso e verde, come ad esempio:

```
leds.set_color('LEFT', (1, 0))  # LED sinistro rosso chiaro
leds.set_color('RIGHT', (0, 1))  # LED destro verde chiaro
```

[3]Si tratta della percentuale x calcolata in base alla velocità massima stimata del motore, ottenibile con la seguente formula: $x * 18,33\ rad/s$.

3.1.2 Pulsanti

Per creare un oggetto ***Button*** basta scrivere il seguente codice:

```
button = Button()
```

La libreria **ev3dev** mette a disposizione diverse funzioni per il controllo dello stato dei pulsanti, ed in particolare per verificare se uno in particolare è stato premuto: `button.up()`, `button.down()`, `button.left()`, `button.right()`.

Esiste anche una funzione più generale che permette di eseguire tale controllo per uno qualunque tra i pulsanti sopra elencati (senza specificarli uno per uno): `button.any()`.

3.1.3 Sensore a ultrasuoni

Il sensore a ultrasuoni dell'EV3, realizzato con un oggetto ***UltrasonicSensor*** (`ultrasonic = UltrasonicSensor()`), presenta lo stesso range compreso tra i 0 ed i 255 cm e può essere interrogato tramite diverse funzioni.

Quella di maggiore utilità è sicuramente `ultrasonic.distance_centimeters`, che funziona esattamente come nell'implementazione nel sistema LEGO EV3 MicroPython, ovvero restituisce una misura della distanza (tuttavia in centimetri, non più millimetri) al momento della chiamata e continua effettuando nuove rilevazioni in modo continuo, mettendole a disposizione di eventuali richieste future.

È possibile, in alternativa, richiedere una singola misurazione con `ultrasonic.distance_centimeters`, facendo però attenzione a non scendere sotto i 250 ms tra due letture successive, in quanto il sensore potrebbe restituire un errore se interrogato troppo spesso.

3.1.4 Motori

Per il controllo della coppia di motori si utilizzano gli oggetti ***MoveTank()*** e ***MoveSteering()***, alla cui creazione si devono specificare le porte a cui sono collegati:

```
motor = MoveTank(OUTPUT_A, OUTPUT_D)
steer = MoveSteering(OUTPUT_A, OUTPUT_D)
```

La classe *MoveTank()* fornisce diverse funzioni per far avanzare il brick; tra queste, la principale è `on()`, dove vengono specificate le velocità per ciascun motore (ad esempio in percentuale):

```
motor.on(SpeedPercent(motorSpeed), SpeedPercent(motorSpeed))
```

Ci sono anche alcune varianti di `on()`, che permettono di specificare una modalità di gestione dei motori diversa in base a un valore espresso in gradi, rotazioni o secondi:

```
on_for_degrees(left_speed, right_speed, degrees)
on_for_rotations(left_speed, right_speed, rotations)
on_for_seconds(left_speed, right_speed, seconds)
```

MoveSteering() viene invece utilizzato per controllare lo sterzo; i due parametri principali consistono nella velocità, `speed`, e nel valore di sterzo, `steering`. Il valore `speed` si riferisce alla ruota più veloce (dato che durante una sterzata le ruote non hanno la stessa velocità), mentre la velocità della ruota più lenta è calcolata automaticamente in funzione di `speed` e `steering`[14].

La variabile `steering` controlla la traiettoria della sterzata, e può assumere valori compresi tra -100 e $+100$, con il seguente significato:

-100	=	il brick gira in senso antiorario sul posto;
-50	=	il brick gira in senso antiorario soltanto con la ruota destra (la sinistra è ferma);
0	=	il brick prosegue dritto;
+50	=	il brick gira in senso orario soltanto con la ruota sinistra (la destra è ferma);
+100	=	il brick gira in senso orario sul posto.

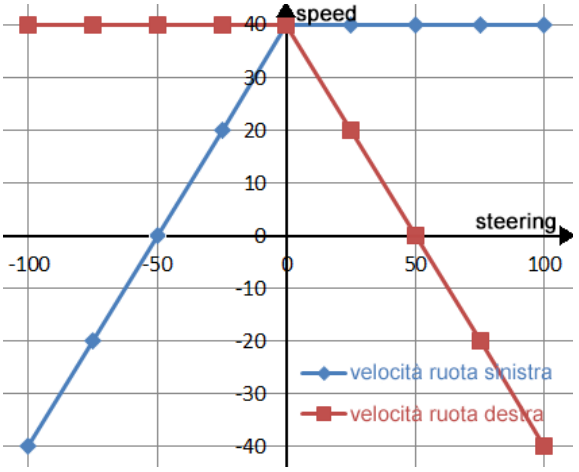

Figura 7: Velocità delle ruote in funzione di `steering`[6]

Esempio

Si presenta di seguito un breve esempio per meglio illustrare il funzionamento della sterzata.

Supponiamo di avere `speed = 40`; per qualsiasi valore di `steering` ci sarà sempre una ruota con velocità pari al valore di `speed` specificato:

- `steering = 0`: le due ruote hanno la stessa velocità (pari a *speed*);

- `steering = 25`: la ruota sinistra ha velocità pari a *speed*, mentre la ruota destra gira a *speed*/2;
- `steering = 50`: la ruota sinistra ha velocità pari a *speed*, mentre la ruota destra è ferma;
- `steering = 100`: la ruota sinistra ha velocità pari a *speed*, mentre la ruota destra gira a $-speed$.

3.2 Navigazione autonoma: Python

```
#!/usr/bin/env python3

from ev3dev2.motor import LargeMotor, OUTPUT_A, OUTPUT_D,
   MoveSteering
from ev3dev2.motor import SpeedPercent, MoveTank
from ev3dev2.sensor import INPUT_1, INPUT_2, INPUT_4
from ev3dev2.sensor.lego import UltrasonicSensor
from ev3dev2.button import Button
from ev3dev2.led import Leds
from time import sleep

def stopFunction(motor,button,started):
    if button.any():
        motor.off()
        started = False

# INITIALIZATION

# Brick LED is red during initialization
leds = Leds()
leds.set_color("LEFT", "RED")
leds.set_color("RIGHT", "RED")

# Ultrasonic sensor
ultrasonic = UltrasonicSensor()

# Drive using two motors
motor = MoveTank(OUTPUT_A, OUTPUT_D)
steer = MoveSteering(OUTPUT_A, OUTPUT_D)

# Buttons
button = Button()
```

```
# Parameters
motorSpeed = 30  # Motor speed (%)
steeringValue = -100  # Steering value (to be used when
   turning around); goes from -100 to 100
steeringSpeed = 30
steeringDegrees = 0.5
minDistance = 20  # Minimum distance (in cm) before the brick
   starts stops to turn around
# _maxSpeed = 400

# Flag to be used as manual start/stop switch; default is
   False (brick does not move)
started = False

while True:
    # Brick LED is yellow when ready (waiting for a button
       press)
    leds.set_color('LEFT','YELLOW')
    leds.set_color('RIGHT','YELLOW')

    if button.any():
        started = True
        leds.set_color('LEFT','GREEN')
        leds.set_color('RIGHT','GREEN')
        motor.on(SpeedPercent(motorSpeed),SpeedPercent(
           motorSpeed))

    while started is True:
        if ultrasonic.distance_centimeters < minDistance:
            motor.off()  # Stop motors
            d = 4
            while ultrasonic.distance_centimeters <
               minDistance:
                steer.on_for_rotations(steeringValue,
                   steeringSpeed, steeringDegrees)
                d -= 1
                if(d == 0):
                    started = False
                    break
            # Run the motors up to 'motorSpeed' degrees:
            if d != 0:
                motor.on(SpeedPercent(motorSpeed),SpeedPercent
                   (motorSpeed))
```

4 Alcune osservazioni

4.1 Considerazioni sui tempi di esecuzione

Data la natura del presente lavoro, si ritiene necessario fare alcune considerazioni sui tempi ed il flusso di esecuzione delle istruzioni, così come sul funzionamento dei metodi principali delle due librerie.

4.1.1 Motori

Le funzioni che fanno girare indefinitamente i motori, `run()` e `on()` (rispettivamente in MicroPython e Python), hanno l'effetto di avviarli e portarli gradualmente alla velocità specificata nei parametri, per mantenerla poi costante in modo automatico (anche in presenza di un eventuale carico). I motori continuano dunque a girare in background finché non viene chiamata una funzione che li fermi; nel frattempo il brick prosegue con le istruzioni successive, senza interrompere il flusso di esecuzione del programma.

L'interruzione del flusso di controllo del programma principale può tuttavia essere effettuata nelle funzioni secondarie di avvio dei motori:

- nel sistema LEGO EV3 MicroPython, esplicitando il parametro booleano `wait` e ponendolo a `True` (dato che, implicitamente, il suo valore di default è `False`) nelle funzioni `run_time()`, `run_angle()`, `run_target`; ad esempio:

  ```
  motor.run_angle(speed, angle, wait=True)
  ```

- in ev3dev, nello stesso modo ma utilizzando il parametro `block` (anch'esso posto a `False` di default) nelle funzioni `on_for_degrees()`, `on_for_rotations()`, `on_for_seconds()`:

  ```
  motor.on_for_degrees(left_speed, right_speed, degrees,
      block=True)
  ```

In questo modo il programma completa il movimento dei motori prima di continuare con l'esecuzione delle altre istruzioni.

Arresto

L'arresto dei motori, una volta avviati, viene effettuato attraverso due funzioni:

- in MicroPython, `stop(stop_type=Stop.PARAMETER)`. Questo metodo possiede tre parametri sostituibili a `PARAMETER`: `COAST`, che lascia fermare il motore per effetto dell'attrito; `BRAKE`, che oppone una resistenza passiva a piccole forze esterne; `HOLD`, che cerca attivamente di mantenere il motore in una posizione fissata al fine di fermarlo.

- in ev3dev, `off(brake)`; il parametro `brake` consente di specificare se l'EV3 cercherà di mantenere il motore in una posizione fissa, ovvero di fermarlo il prima possibile (`off(brake=True)` o `off()`, in quanto è `True` di default) oppure se lascerà che il brick si fermi per effetto dell'attrito (`off(brake=False)`).

Nei programmi presenti in questa relazione sono stati appositamente lasciati i valori di default, in quanto sono stati ritenuti adatti all'applicazione implementata.

4.1.2 Loop

Il linguaggio di programmazione Python, per sua natura, non è nato per essere utilizzato in sistemi embedded e/o a bassa potenza di calcolo. Tale inconveniente viene parzialmente arginato in MicroPython, appositamente nato per queste tipologie di macchine, ma dalle prestazioni ancora non paragonabili a linguaggi di livello più basso (come ad esempio il C). La conseguenza più importante di questo fatto è l'impossibilità di misurare e limitare accuratamente il tempo di esecuzione delle istruzioni, in quanto ne esistono in numero limitato e non è possibile sostituirle con versioni ottimizzate che facciano le stesse operazioni in meno tempo.

Il problema risulta particolarmente evidente nei loop, ovvero nei cicli *while*, la cui durata è aribitrariamente lunga in base al numero e tipo di istruzioni che contengono e che difficilmente si possono limitare in modo adeguato. Si tratta, però, di un inconveniente relativamente irrilevante nel caso della presente applicazione: l'EV3, con la sua CPU a 300 MHz ed i suoi 64 MB RAM, presenta infatti una potenza di calcolo sufficiente ad eseguire operazioni anche complesse[4] e supportare l'esecuzione non solo di un vero e proprio sistema operativo, ma anche di codice scritto in linguaggi di programmazione notoriamente poco adatti ai sistemi embedded (come appunto il Python, oppure Java).

Ne consegue che, al momento, la temporizzazione dei cicli *while* non risulta essere un aspetto critico nella programmazione dell'EV3; ulteriori sviluppi del progetto, illustrati nelle sezioni successive della presente relazione, hanno tuttavia evidenziato la necessità di limitare alcune istruzioni al fine di una corretta esecuzione dei programmi, così come la fattibilità, entro certi limiti, della temporizzazione dei loop.

4.2 Prime conclusioni

Il lavoro finora presentato rende evidente come i due sistemi analizzati vadano di pari passo ed eseguano in maniera estremamente simile le stesse funzioni, al punto che si potrebbero descrivere come due dialetti leggermente diversi della stessa lingua. Entrambi presentano il grande vantaggio di essere installabili su una microSD card,

[4] È paragonabile, ad esempio, ad un microcontrollore ESP8266, notoriamente utilizzato per un'ampia varietà di applicazioni *Internet of Things* anche molto complesse, inclusa l'intelligenza artificiale.[1].

eliminando il rischio di corrompere irrimediabilmente il firmware a bordo dell'EV3, ed hanno tempi di esecuzione delle istruzioni paragonabili - anche se è stata notata una maggiore responsività ai comandi (ad esempio alla pressione di un pulsante) nel sistema ev3dev.

Andando oltre l'effettiva somiglianza del codice nel programma di navigazione autonoma appositamente scritto per testare le librerie, si può notare già dalle premesse degli stessi sistemi come **ev3dev** presenti **maggiori potenzialità** di sviluppo e crescita.

A differenza di LEGO EV3 MicroPython, infatti, ev3dev è direttamente basato su Linux, permettendo così di sfruttare moltissime funzionalità che nell'altro sistema semplicemente non esistono (o nel migliore dei casi non sono in alcun modo documentate). Molto rilevante è la presenza del **SSH**, un protocollo di rete che consente la comunicazione remota col brick attraverso comandi da terminale. Non solo: la potenza di Linux, assieme alla gamma completa di funzioni Python, consentono l'integrazione con dispositivi che altrimenti non sarebbero compatibili, e che permetterebbero, ad esempio, di controllare l'EV3 in modi nuovi ed originali (come attraverso un gamepad, la tastiera di un pc, ecc.).

In conclusione, dunque, dai test effettuati ev3dev sembrerebbe essere il sistema migliore da scegliere per proseguire con lo sviluppo di nuovi programmi Python per ampliare l'orizzonte delle possibilità dell'EV3. Applicazioni future confermeranno questa deduzione.

5 Controllo remoto

Le prime osservazioni sul funzionamento dell'EV3 e dei due linguaggi di programmazione sono state utilizzate, come accennato nell'introduzione, per creare un programma di controllo remoto del brick attraverso una connessione WiFi ed un gamepad, al fine di concretizzare le conoscenze acquisite ed estenderle con un'applicazione pratica.

5.1 Architettura del sistema

Il sistema di controllo remoto si compone di due script Python principali che ricalcano una struttura generica client-server e di conseguenza sono stati semplicemente denominati *client.py* e *server.py*. Il primo può girare su una qualunque macchina Linux[5] a cui sia collegato il gamepad, ed ha la funzione di ricevere ed inviare in rete i comandi; il secondo, invece, consiste in un programma presente sul-

[5] Collegata alla stessa rete locale wireless dell'EV3.

l'EV3 che rimane in ascolto sulla stessa porta del client, riceve l'input ed esegue le funzioni di controllo e logging.

Lo scambio dei dati tra i due script avviene tramite il modulo `socket` della libreria **SocketServer**[15] (incluso in entrambi i programmi), che utilizza l'indirizzo IP ed una porta del brick per realizzare un collegamento estremamente semplice ed efficiente.

5.2 Acquisizione dell'input (client)

L'acquisizione dei comandi inviati dal gamepad avviene, nello script *client.py* attraverso la funzione `get_gamepad()` presente nella libreria **inputs**[8], che permette di ottenere con facilità tutti gli eventi generati dalla pressione di un pulsante. Ogni evento possiede un codice ed uno stato che lo identificano in modo univoco; questi dati sono dunque ciò di cui il brick necessita per poter eseguire correttamente i comandi di controllo.

Durante l'implementazione, tuttavia, è stato osservato che il numero di eventi generati dalla levetta analogica sinistra è troppo alto affinché il brick riesca a processarli correttamente senza lag. Per risolvere tale problema, l'input viene elaborato in modo da inviare all'EV3 soltanto un evento ogni cinque ricevuti dal gamepad, secondo una soglia empirica ritenuta come il miglior compromesso fra responsività del brick ed accuratezza dei comandi.

5.3 Ricezione dell'input (server)

All'avvio dello script `server.py` viene inizializzato, oltre a tutti i parametri necessari, anche un oggetto di tipo `TCPHandler`, definito nello stesso file. Questo contiene tutte le istruzioni fondamentali per il controllo del brick, e viene passato al server in modo da poter essere utilizzato per elaborare ogni input ricevuto tramite il metodo `server.serve_forever()`, che resta in ascolto sull'indirizzo IP e sulla porta specificati (ovvero quelli del brick) fino all'interruzione dello script stesso, pronta a ricevere qualunque evento dal client sotto forma di stringa contenente codice e stato del pulsante premuto sul gamepad collegato al computer.

La funzione che, nella pratica, si occupa della ricezione ed interpretazione dell'input è il metodo `TCPHandler.handle()`, che grazie ad una serie di *if* stabilisce quali sono i metodi da invocare corrispondenti ad ogni pulsante premuto.

Esempio

Supponiamo di premere (e non rilasciare) il pulsante **A** sul gamepad. Attraverso la connessione al client, il brick riceverà un input con codice `BTN_SOUTH` e stato `1`;

secondo la definizione dei comandi presente in `TCPHandler.handle()`, lo script `server.py` invocherà il metodo `TCPHandler.forward()`, che a sua volta eseguirà le operazioni necessarie al movimento in avanti dell'EV3 alla velocità corrente.

Supponiamo adesso di rilasciare il pulsante **A**; l'input avrà lo stesso codice ma stato `0`, perciò il server provvederà ad invocare il metodo `TCPHandler.stop()` per fermare il brick.

5.4 Controlli

Nello script `server.py`, ad ogni input ricevuto dal gamepad corrisponde una funzione definita nella classe `TCPHandler` che esegue le operazioni necessarie per eseguire il comando richiesto dall'utente, secondo il seguente schema:

- **A**: avanti[6];
- **B**: indietro[6];
- **Y**: inizia/ferma logging;
- **destra**[7]: gira a destra sul posto (45°);
- **sinistra**[7]: gira a sinistra sul posto (45°);
- **su**[7]: aumenta velocità (+10%, massimo 100%);
- **giù**[7]: diminuisce velocità (+10%, minimo 10%);
- **levetta analogica sinistra**: avanti/indietro/destra/sinistra (il brick si ferma se la levetta viene riportata alla posizione di riposo centrale);
- **SELECT**: stop.

Le funzioni corrispondenti ad ogni pulsante sono, rispettivamente:

- `forward()`: avvia i motori in avanti alla velocità di default oppure a quella impostata dall'utente attraverso la pulsantiera a croce attraverso la funzione: `motor.on(SpeedPercent(motor_speed), SpeedPercent(motor_speed);`
- `backward()`: avvia i motori con verso opposto alla velocità di default oppure a quella impostata dall'utente attraverso la pulsantiera a croce attraverso la funzione: `motor.on(SpeedPercent(-motor_speed), SpeedPercent(-motor_speed);`
- `log()`: vedi sez. 5.5.3;
- `right()`: gira a destra utilizzando la funzione `steer.on_for_rotations(steering_value, motor_speed, steering_degrees);`
- `left()`: gira a sinistra utilizzando la funzione `steer.on_for_rotations(-steering_value, motor_speed, steering_degrees);`
- `speed_up()`: aumenta la velocità del 10% ad ogni pressione del pulsante sommando 10 alla variabile globale `motor_speed`, fino ad un massimo di 100;

[6] Tenendolo premuto; appena lo si rilascia il brick si ferma. La velocità di default è del 30% della velocità massima erogabile dai motori, e può essere aumentata o diminuita con la pulsantiera a croce.

[7] Pulsantiera a croce.

- `speed_down()`: diminuisce la velocità del 10% ad ogni pressione del pulsante riducendo la variabile globale `motor_speed`, fino ad un minimo di 10;
- `move()`: vedi sez. 5.4.1;
- `stop()`: spegne i motori inviando loro velocità nulla con `motor.on(SpeedPercent (0), SpeedPercent(0))` e `motor.off()`; la doppia funzione serve come misura precauzionale per assicurare lo spegnimento dei motori, dato che sporadicamente, durante gli esperimenti, questi continuavano a girare (sebbene a velocità molto bassa).

Tutte le funzioni che inducono un movimento dei motori accendono di verde i LED per indicarne il corretto funzionamento, mentre la funzione `stop()` li colora di arancione per segnalare un brick completamente fermo.

I valori dei parametri `steering_value` e `steering_degrees`, rispettivamente posti a 100 e 0.554, sono stati ricavati sperimentalmente allo scopo di far girare il brick sul posto di esattamente 45°.

5.4.1 Movimento con la levetta analogica

Il movimento dell'EV3 attraverso la levetta analogica sinistra, effettuato grazie alla funzione `move()` si è rivelato particolarmente complesso, soprattutto a causa del grande numero di input che un qualunque spostamento di tale pulsante genera. Come specificato nella sezione 5.2, il server riceve dal client soltanto un evento ogni cinque per alleviare questo problema. La grande sensibilità della levetta ha reso comunque necessario attivare il movimento solo dopo il superamento di una certa soglia sull'input ricevuto dal gamepad, che consente lo spostamento solo per valori dello stato superiori a 1500 (rispetto all'ultimo stato ricevuto[8]) ed invoca invece la funzione `stop()` per valori inferiori.

Gli stati in ingresso sull'asse y della levetta che non rientrano nelle condizioni di arresto vengono normalizzati secondo la seguente funzione lineare, dove il valore 32.768 è stato ricavato sperimentalmente dal gamepad:

$$speed_y = \frac{-s}{32.768} \cdot 100$$

Se la velocità risultante è inferiore al 15% del massimo possibile, il brick ignora eventuali input sull'asse x della levetta e prosegue solo avanti oppure indietro. In caso contrario, l'ultimo valore in input ricevuto dall'asse x viene utilizzato per calcolare la velocità secondo le formule:

$$v = (100 - |speed_x|) \cdot \frac{speed_y}{100} + speed_y$$
$$w = (100 - |speed_y|) \cdot \frac{speed_x}{100} + speed_x$$
$$r = \frac{v + w}{2}$$

[8] Memorizzato in un'apposita variabile.

$$l = \frac{v - w}{2}$$

dove r ed l sono le velocità in percentuale che vengono successivamente inviate rispettivamente al motore destro ed a quello sinistro.

5.5 Multithreading

La necessità di far girare in parallelo due programmi, uno per controllare manualmente il brick e l'altro per rilevare eventuali ostacoli, ha portato durante lo sviluppo del progetto all'impiego della libreria Python **threading**, e dunque all'implementazione di un programma multi-threaded in grado di gestire con discreta affidabilità più task contemporaneamente.

5.5.1 Rilevamento ostacoli

Durante l'inizializzazione dello script `server.py`, prima dell'avvio del server vero e proprio, il programma avvia un thread indipendente che controlla periodicamente la presenza di eventuali ostacoli di fronte al brick, attraverso la continua interrogazione del sensore a ultrasuoni (con periodo pari a $350ms$, tale da non mandare in errore il sensore).

Se il thread dovesse rilevare un ostacolo a meno di $20cm$, esso ferma l'EV3 ed imposta a `True` un flag globale che impedisce al brick di rispondere ad eventuali comandi dell'utente che lo farebbero avanzare verso il suddetto ostacolo; il flag viene rimosso non appena un nuovo controllo della distanza rileva uno spazio libero di fronte al brick superiore ai $20cm$.

5.5.2 Informazioni

Un secondo thread che può essere facoltativamente avviato durante l'inizializzazione del server è il cosiddetto `InfoThread`, il cui compito è quello di stampare a video con periodo $1s$ le velocità correnti dei due motori, dopo aver visualizzato le loro velocità massime nominali.

5.5.3 Logging

L'ultimo thread avviato prima che il server cominci l'ascolto sulla porta di input dei comandi provenienti dal gamepad è quello addetto al logging della velocità.

Alla pressione del pulsante Y questo task avvia un ciclo temporizzato (con periodo pari a $150ms$) di rilevamento della velocità dei due motori[9] e successivo salvataggio dei valori in un apposito array.

Se l'utente preme nuovamente il pulsante Y, il logging viene interrotto ed i dati raccolti vengono salvati nel brick, che li salva in formato JSON nel file `log.json` assieme agli istanti di inizio e fine della loro misurazione.

5.6 Plotting

Lo script `plot.py` utilizza i dati salvati in `log.json` per creare tanti grafici quanti sono i log presenti nel suddetto file. Le figure così create, ritrovabili nella sottocartella *plots* presentano due plot, uno per motore, che mostrano l'andamento nel tempo della velocità target (ovvero quella impostata dall'utente) assieme alla velocità effettivamente rilevata dai sensori presenti sugli stessi motori (vedi fig. 8).

È interessante osservare dai grafici come la velocità massima effettiva raggiungibile dai motori si fermi a circa l'80%; questo fenomeno rientra perfettamente nella norma, dato che i motori dell'EV3, in quanto ovviamente dispositivi non ideali, non raggiungono mai il 100% del rendimento.

[9] La variabile di velocità è diversa a seconda se l'utente utilizza i pulsanti A/B/croce oppure la levetta analogica sinistra per controllare il brick. Di conseguenza, il thread di logging ha bisogno di sapere quale tra queste velocità registrare, e per farlo utilizza il flag `button` che viene adeguatamente impostato a `0` (A/B/croce) oppure a `1` (levetta sinistra), in base al tipo di controllo correntemente in uso.

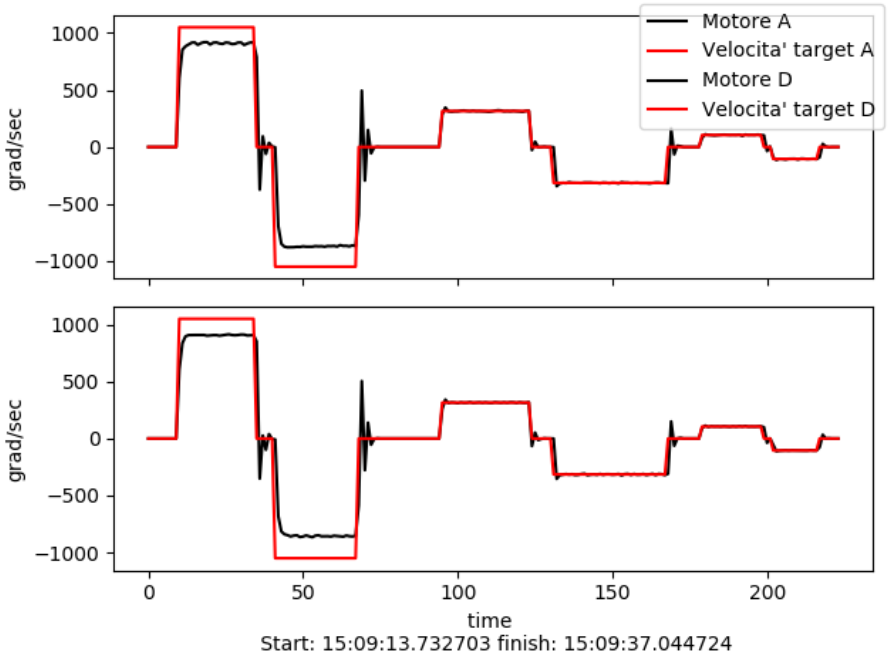

Figura 8: Grafico generato dallo script `plot.py`.

6 Conclusioni

La trasformazione dell'EV3 in un robot controllato da remoto con funzione di monitoraggio degli ostacoli e logging della velocità ha permesso una migliore comprensione del suo funzionamento, così come del suo potenziale e dei suoi limiti.

Questo dispositivo permette una grande flessibilità di programmazione, grazie alle moltissime librerie del linguaggio Python ed alla notevole quantità di informazioni ricavabile dai sensori montati sul brick. La presenza a bordo di un processore performante (considerando che si tratta pur sempre di un microcontrollore) ma non sufficientemente potente per elaborare efficacemente tutte le possibili istruzioni che potrebbero essergli inviate richiede tuttavia un certo grado di attenzione durante la programmazione, in quanto il rischio di lag aumenta velocemente con la complessità degli script.

Nonostante questo inconveniente, l'architettura client-server per la comunicazione remota col brick si è rivelata una strategia vincente in quanto leggera e funzionale, al punto che la difficoltà maggiore si è riscontrata nella limitazione dei comandi inviati dal gamepad piuttosto che nel loro invio (vedi sez. 5.2). Similmente, il movimento attraverso la levetta analogica sinistra ha aggiunto un ulteriore grado di complessità a causa della presenza di due motori sull'EV3 (uno per ruota), invece di uno sterzo

vero e proprio. Una volta risolte queste problematiche, però, la programmazione della ricerca continua di ostacoli e del logging si è rivelata assai semplice attraverso l'uso del multithreading, che non ha avuto alcun impatto rilevabile sulla velocità di esecuzione dei comandi ricevuti dal controllore.

È dunque possibile concludere che il **LEGO MINDSTORMS EV3** è un dispositivo dal grande potenziale e che, vista la varietà di funzioni presente nella libreria **ev3dev**, la sua programmazione in **Python** può essere a buona ragione considerata una valida alternativa ai linguaggi tradizionalmente utilizzati (come ad es. MATLAB). Escludendo le limitazioni hardware di questo microcontrollore, l'unico reale limite nella sua programmazione rimane soltanto nella fantasia del programmatore, dato che la flessibilità di Python, combinata all'inconsueta possibilità di modificare radicalmente anche la conformazione fisica del robot, aprono la strada ad una quantità virtualmente illimitata di possibili applicazioni.

7 Codice sorgente (controllo remoto)

Segue il codice Python del programma di controllo remoto, la cui suddivisione in sottosezioni riflette la ripartizione in più funzioni dello stesso (definite in file separati).

7.1 Server

```
#!/usr/bin/env python3
from __future__ import division  # Better divisions
from threading import Thread  # Multi-threading
import socketserver as socket  # Data transmission
from time import sleep, time  # Sleeping (to avoid lags) and
    timing
import datetime
import json  # JSON logging
from pprint import pprint  # Pretty print, for readable JSON [
    debug purposes only]

# LEGO MINDSTORMS EV3 ev3dev Python library
from ev3dev2.led import Leds
from ev3dev2.motor import Motor, MoveTank, MoveSteering,
    SpeedPercent, OUTPUT_A, OUTPUT_D
from ev3dev2.sensor.lego import UltrasonicSensor
from ev3dev2.sound import Sound
```

```
# TCP HANDLER - It is instantiated once, at the connection to
    the server
class TCPHandler(socket.BaseRequestHandler):

    # Setup
    global motor_speed  # Default speed for variant forward/
       backward function
    global motor_speed_log  # Variable used to store the speed
        value to be logged
    global s_old  # Variable used to store the previous state
       received from the gamepad (only needed for left analog
       stick)
    global speed_x  # Current speed on left analog stick x
       axis
    global speed_y  # Current speed on left analog stick y
       axis
    global log_thread_started  # Flag to check if the log
       thread is started

    motor_speed = 30  # Default motor speed
    s_old = 0  # Default (previous) left analog stick state
    speed_x = 0  # Default speed on left analog stick x axis
    speed_y = 0  # Default speed on left analog stick y axis

    # AUXILIARY FUNCTIONS
    # Left analog stick moving function (forward/backward)
    def move(self, s, axis):  # Left analog stick move
       function
        global button  # Flag necessary for the log function
           to know which button is being used to move the brick
            (0 = A/B buttons, 1 = left analog stick)
        global speed_y
        global speed_x
        global s_old
        global l
        global r
        global l_log
        global r_log
        button = 1  # Default move button is left analog stick
        l = 0
        r = 0
        l_log = 0
        r_log = 0
```

```
percent = 15  # Minimum percentage to be reached
   before activating x axis

self.leds_green()

if axis == 'ABS_Y':  # Speed on y axis
    if s > 0:
        reverse = True  # Flag to specific if the
           brick is moving forward or backward
    if s < 0 and (s - s_old) >= 1500:
        self.stop()
    elif s > 0 and (s - s_old) <= -1500:
        self.stop()
    else:
        speed_y = (-s / 32768.0) * 100.0
        if (-percent < speed_x) and (speed_x < percent
           ):
            speed_x = 0
            r = speed_y
            l = r
            motor.on(SpeedPercent(speed_y),
               SpeedPercent(speed_y))
        else:
            self.v = (100 - abs(speed_x)) * (speed_y /
                100) + speed_y
            self.w = (100 - abs(speed_y)) * (speed_x /
                100) + speed_x
            r = (self.v + self.w) / 2
            l = (self.v - self.w) / 2
            motor.on(SpeedPercent(l), SpeedPercent(r))
else:  # Speed on x axis
    if s < 0 and (s-s_old) >= 1500:
        self.stop()
    elif s > 0 and (s-s_old) <= -1500:
        self.stop()
    else:
        speed_x = (-s / 32768.0) * 100.0
        self.v = (100 - abs(speed_x)) * (speed_y /
           100) + speed_y
        self.w = (100 - abs(speed_y)) * (speed_x /
           100) + speed_x
        r = (self.v + self.w) / 2
        l = (self.v - self.w) / 2
```

```
                motor.on(SpeedPercent(l), SpeedPercent(r))
        s_old = s
        l_log = l
        r_log = r

    # Variant forward function (A button)
    def forward(self):
        global button
        global motor_speed
        global motor_speed_log
        button = 0

        self.leds_green()
        motor_speed_log = motor_speed
        motor.on(SpeedPercent(motor_speed), SpeedPercent(
           motor_speed))

    # Variant backward function (B button)
    def backward(self):
        global button
        global motor_speed
        global motor_speed_log
        button = 0

        self.leds_green()
        motor_speed_log = - motor_speed
        motor.on(SpeedPercent(-motor_speed), SpeedPercent(-
           motor_speed))

    # Stop function
    def stop(self):
        global motor_speed_log
        global l_log
        global r_log
        global speed_x
        global speed_y
        motor_speed_log = 0
        l_log = 0
        r_log = 0
        speed_x = 0
        speed_y = 0

        self.leds_orange()
```

```
        motor.on(SpeedPercent(0), SpeedPercent(0))
        motor.off()  # Stop motors

    # Variant right turn function (D-pad right button)
    def right(self):
        self.leds_green()
        steer.on_for_rotations(steering_value, motor_speed,
           steering_degrees)

    # Variant left turn function (D-pad left button)
    def left(self):
        self.leds_green()
        steer.on_for_rotations(-steering_value, motor_speed,
           steering_degrees)

    # Variant speed up function (D-pad up button)
    def speed_up(self):
        global motor_speed
        if motor_speed <= 90:
            motor_speed += 10
            print('Motor speed: ', motor_speed, '%')
        else:
            print('Maximum motor speed reached.')

    # Variant speed down function (D-pad down button)
    def speed_down(self):
        global motor_speed
        if motor_speed >= 20:
            motor_speed -= 10
            print('Motor speed: ', motor_speed, '%')
        else:
            print('Minimum motor speed reached.')

    # Led function (green)
    def leds_green(self):
        leds.set_color('LEFT','GREEN')
        leds.set_color('RIGHT','GREEN')

    # Led function (orange)
    def leds_orange(self):
        leds.set_color('LEFT','ORANGE')
        leds.set_color('RIGHT','ORANGE')
```

```
def log(self):
    global log_thread_started  # Flag to check if the log
       thread is started

    if log_thread_started == False:
        log_thread_started = True
        logThread = LogThread('Thread Log')
        logThread.start()
    else:
        log_thread_started = False

# Log file read function (here for debug purposes only)
def read_log(self):
    d = []
    with open('log_prova.json') as f:
        for line in f:
            d.append(json.loads(line))
        pprint(d)  # d is an array of Python dictionaries,
            each containing an instance of logged data


def handle(self):
    global reverse  # Flag to check if the brick is moving
        backwards

    self.data = self.request.recv(1024).strip()  # self.
       request is the TCP socket connected to the client

    self.code, self.state = self.data.decode('utf-8').
       split(',')  # Received data decoding

    # CONTROLS
    #
    # A button: forward
    # B button: backward
    # D-pad right button: turn right
    # D-pad left button: turn left
    # D-pad up button: speed up (+10%)
    # D-pad down button: speed down (-10%)
    #
    # Left analog stick: move (forward/backward/rotate)
    #
    # START button: emergency stop
```

```
#
# Y button: start/stop logging

if (self.code == 'BTN_WEST') and (self.state == '1'):
    # print('Y button pressed')
    self.log()

if (self.code == 'BTN_SOUTH') and (self.state == '1'):
    # print('A button pressed')
    self.forward()

if (self.code == 'BTN_SOUTH') and (self.state == '0'):
    # print('A button released')
    self.stop()

if (self.code == 'BTN_EAST') and (self.state == '1'):
    # print('B button pressed')
    reverse = True
    self.backward()

if (self.code == 'BTN_EAST') and (self.state == '0'):
    # print('B button released')
    self.stop()
    reverse = False

if (self.code == 'BTN_SELECT') and (self.state == '1')
   :
    # print('SELECT button pressed')
    self.stop()

if (self.code == 'ABS_HAT0X') and (self.state == '1'):
    # print('D-pad right button pressed')
    self.right()

if (self.code == 'ABS_HAT0X') and (self.state == '-1')
   :
    # print('D-pad left button pressed')
    self.left()

if (self.code == 'ABS_HAT0Y') and (self.state == '-1')
   :
    # print('D-pad up button pressed')
    self.speed_up()
```

```
        if (self.code == 'ABS_HATOY') and (self.state == '1'):
            # print('D-pad down button pressed')
            self.speed_down()

        if (self.code == 'ABS_Y'):
            # print('Left analog stick moved along y axis')
            self.move(float(self.state), 'ABS_Y')

        if (self.code == 'ABS_X'):
            # print('Left analog stick moved along x axis')
            self.move(float(self.state), 'ABS_X')


class UltrasonicThread(Thread):  # Distance checking thread
    def __init__(self):
        Thread.__init__(self)

    def stop(self):
        leds.set_color('LEFT','ORANGE')
        leds.set_color('RIGHT','ORANGE')
        motor.on(SpeedPercent(0),SpeedPercent(0))
        motor.off()

    def run(self):
        global wall
        global reverse
        period = 0.350

        while True:
            t = time()
            while True:
                if ultrasonic.distance_centimeters <
                   min_distance and not reverse:
                    self.stop()
                    wall = True  # Flag indicating that there
                       is an obstacle in front of the brick and
                        it cannot be started again until it is
                       removed from there
                else:
                    wall = False

                if time() - t >= period:
```

```
                    break
                else:
                    sleep(period - (time() - t))


class LogThread(Thread):
    def __init__(self, nome):
        Thread.__init__(self)
        self.nome = nome

    def run(self):
        global button
        global log_thread_started
        global motor_speed_log
        global motor_info_l
        global motor_info_r
        global l_log
        global r_log

        button = 1
        motor_speed_l = []  # A == L
        motor_speed_r = []  # D == R
        motor_target_l = []
        motor_target_r = []
        log_start = 0
        log_finish = 0

        period = 0.150

        log_start = str(datetime.datetime.now())

        print('Logging started...')

        while log_thread_started:
            t = time()
            motor_speed_l.append(motor_info_l.speed)
            motor_speed_r.append(motor_info_r.speed)

            if button == 1:  # Left analog stick
                motor_target_l.append(l_log)
                motor_target_r.append(r_log)
            else:  # A or B buttons
                motor_target_l.append(motor_speed_log)
```

```
                motor_target_r.append(motor_speed_log)

            if time() - t >= period:
                break
            else:
                sleep(period - (time() - t))

        log_finish = str(datetime.datetime.now())

        print('Logging ended.')

        data = {'start': log_start,
                'finish': log_finish,
                'motor_l': motor_speed_l,
                'motor_r': motor_speed_r,
                'target_l': motor_target_l,
                'target_r': motor_target_r
               }

        with open('log.json', 'a+') as f:
            print('Writing log to file...')
            f.write(str(data).replace("\'", "\"") + '\n')
            print('Done!')
            print()


class InfoThread (Thread):
    def __init__(self):
        Thread.__init__(self)
        print('InfoThread started.')

    def run(self):
        global speed_x
        global speed_y
        global motor_info_r
        global motor_info_l

        print('R maximum motor speed: ' + motor_info_r.speed +
            ' deg/s, L maximum motor speed: ' + motor_info_l.
           speed +' deg/s')

        while True:
            sleep(1)
```

```
            print('R motor speed: ' + motor_info_r.speed + '
               deg/s, L motor speed: ' + motor_info_l.speed + '
                deg/s')


if __name__ == '__main__':

    leds = Leds()  # Brick LEDs
    ultrasonic = UltrasonicSensor()  # Ultrasonic sensor
    motor = MoveTank(OUTPUT_A, OUTPUT_D)  # Drive using two
       motors (tank mode)
    motor_info_l = Motor(OUTPUT_A)  # Get info from left motor
    motor_info_r = Motor(OUTPUT_D)  # Get info from right
       motor
    steer = MoveSteering(OUTPUT_A, OUTPUT_D)  # Steer using
       two motors (tank mode)

    # Parameters
    steering_value = 100  # Steering value (to be used when
       turning around); goes from -100 to 100
    steering_degrees = 0.554  # Steering degrees value
       necessary to turn 45 degrees (empirical)
    min_distance = 20  # Minimum distance (in cm) before the
       brick starts decelerating or stops to turn around

    # Flags
    global log_thread_started
    log_thread_started = False
    wall = False
    reverse = False

    # Server settings
    host = '192.168.43.219'
    port = 12397

    # Server initialization
    print('Initializing server...')
    server = socket.TCPServer((host, port), TCPHandler)  #
       Creates the server, binding to the specified host and
       port

    # Secondary threads initialization
    print('Initializing threads...')
```

```
    ultrasonicThread = UltrasonicThread()
    ultrasonicThread.start()
    # infoThread = InfoThread()
    # infoThread.start()

    print('Ready.')
    print()
    readySound = Sound()
    readySound.tone(1000, 3)

    server.serve_forever()  # Activates the server, which will
        keep running until the user stops the program with Ctrl
       +C (KeyboardInterrupt exception)
```

7.2 Client

```
import socket  # Data transmission
from inputs import get_gamepad  # Gamepad input
from time import sleep


# Auxiliary function for easy data sending
def send_data(host, port, data):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as
       sock:
        sock.connect((host, port))  # Connects to server...
        sock.sendall(bytes(data, 'utf-8'))  # ...and sends
           data


# Setup
host = '192.168.43.219'  # Server IP
port = 12397  # Server port

stick_values_X = []  # Array used for collecting input events
   from the left analog stick (x axis)
stick_values_Y = []  # Array used for collecting input events
   from the left analog stick (y axis)

sleep(0.05)  # Avoids lagging related problems


# Main loop
```

```
while True:  # Listens for gamepad input events and sends them
    to server

    events = get_gamepad()  # Gets input events generated by
       the gamepad

    if events[0].ev_type != 'Sync':  # Only considers 'Key'
       type events

        code = events[0].code  # Event code
        state = events[0].state  # Event state (pressed,
           released or somewhere in between for non-binary
           buttons)

        commands_to_send = 5  # Number of commands to send to
           server (needs to be reduced for the left analog
           stick to avoid lagging)

        # Left analog stick (x axis)
        if code == 'ABS_X':

            stick_values_X.append(state)  # Appends the value
               of an input to the stick_values_X array

            if len(stick_values_X) == commands_to_send:  # If
               the maximum number of events to collect before
               sending them to the server has been reached...
                data = 'ABS_X' + ',' + str(stick_values_X[
                   commands_to_send-1])  # ...then the last
                   event is sent...
                stick_values_X = []  # ...and the array is
                   reset

                send_data(host, port, data)  # Actual sending
                   of the data

        # Left analog stick (y axis)
        elif code == 'ABS_Y':

            stick_values_Y.append(state)  # Appends the value
               of an input to the stick_values_Y array

            if len(stick_values_Y) == commands_to_send:  # If
```

```
                the maximum number of events to collect before
                sending them to the server has been reached...
                 data = 'ABS_Y' + ',' + str(stick_values_Y[
                    commands_to_send-1])  # ...then the last
                    event is sent...
                 stick_values_Y = []  # ...and the array is
                    reset

                 send_data(host, port, data)  # Actual sending
                    of the data

        # Every other button
        else:
            data = str(code) + ',' + str(state)  # Data to be
                sent

            send_data(host, port, data)  # Actual sending of
                the data

        sleep(0.001)  # Avoids lagging related problems
```

7.3 Plot

```
import datetime as datetime  # For time intervals
import numpy as np
import matplotlib.pyplot as plt


def read_log():
    with open('./log.json', 'r') as f:
        # Data reading
        data = f.read()
        data = data.split('\n')
        data.pop(-1)

        for i, reading in enumerate(data):
            info = eval(reading)

            # Data
            motor_l = info['motor_A']
            motor_r = info['motor_D']
            target_l = (np.array(info['target_A']) * 1050) /
                100
```

```
target_r = (np.array(info['target_D']) * 1050) /
    100

# Start/finish times
start = info['start']
start = start.split(' ')[1]
finish = info['finish']
finish = finish.split(' ')[1]

# Logging time interval
s = datetime.datetime.strptime(start.split(',')
    [0], '%H:%M:%S.%f')
f = datetime.datetime.strptime(finish.split(',')
    [0], '%H:%M:%S.%f')
interval = (f-s).total_seconds() * 10

# Plot time sample
time_sample = range(len(motor_l))

# Figure creation
plt.cla()
fig, (ax1, ax2) = plt.subplots(2, sharex=True)

# Log data plotting
ax1.set(xlim=(0, interval))
ax2.set(xlim=(0, interval))
ax1.plot(time_sample, motor_l, c='black', label='
    Left motor')
ax2.plot(time_sample, motor_r, c='black', label='
    Right motor')
ax1.plot(time_sample, target_l, c='r', label='Left
     motor target speed')
ax2.plot(time_sample, target_r, c='r', label='
    Right motor target speed')

# Plot labels
ax1.set(ylabel='deg/s')
ax2.set(xlabel='ms\n\nLogging start time: ' +
    start + ', logging finish time: ' + finish,
    ylabel='deg/s')

# Figure saving
fig.legend()
```

```
            fig.tight_layout()
            fig.savefig('./plots/' + start.replace(':', '-') +
                '.png')  # Replaces ':' character to avoid
                naming issues under Windows


read_log()
```

Riferimenti bibliografici

[1] ESP8266. https://it.wikipedia.org/wiki/ESP8266.

[2] Etcher. https://etcher.io/.

[3] ev3dev. https://www.ev3dev.org/.

[4] ev3dev Downloads. https://github.com/ev3dev/ev3dev/releases/download/ev3dev-stretch-2019-10-23/ev3dev-stretch-ev3-generic-2019-10-23.zip.

[5] ev3dev Programming Languages. https://www.ev3dev.org/docs/programming-languages/.

[6] EV3dev Python - Using Motors. https://sites.google.com/site/ev3devpython/learn_ev3_python/using-motors.

[7] Getting started with EV3 MicroPython. https://le-www-live-s.legocdn.com/sc/media/files/ev3-micropython/ev3micropythonv100-71d3f28c59a1e766e92a59ff8500818e.pdf.

[8] inputs - Cross-platform Python support for keyboards, mice and gamepads. https://pypi.org/project/inputs/.

[9] LEGO - EV3 MicroPython micro SD card image. https://le-www-live-s.legocdn.com/sc/media/files/ev3-micropython/ev3micropythonv100sdcardimage-4b8c8333736fafa1977ee7accbd3338f.zip.

[10] LEGO-EV3-Python - Progetto di laboratorio di automatica. https://github.com/chabdullah/Lego-Ev3-Python.

[11] LEGO MINDSTORMS Education EV3. https://education.lego.com/en-us/middle-school/intro/mindstorms-ev3.

[12] MicroPython. https://it.wikipedia.org/wiki/MicroPython.

[13] python-ev3dev - Leds. https://python-ev3dev.readthedocs.io/en/ev3dev-stretch/leds.html.

[14] Python language bindings for ev3dev - Motor classes. https://python-ev3dev.readthedocs.io/en/ev3dev-stretch/motors.html.

[15] SocketServer - A framework for network servers. https://docs.python.org/3/library/socketserver.html.

[16] Visual Studio Code. https://code.visualstudio.com/download.