UNIVERSITÀ DELLA CALABRIA

Dipartimento di ELETTRONICA,
INFORMATICA E SISTEMISTICA

Programmazione dei Sistemi Embedded per l'Automazione

Relazione Progetto Differential Drive

Studente: Giuseppe Emanuele Lisotti (216650)

Professore: Gianni Cario

20 febbraio 2022

Abstract

In questa relazione vengono esposti i vari passaggi per la realizzazione di un differential drive.

Si inizia da una descrizione esaustiva del modello matematico, con successiva sintesi di un controllore a tempo discreto. Si passa poi alla descrizione dei dispositivi utilizzati per realizzarlo, in primis la scheda di sviluppo STM32, i sensori e gli attuatori ed il piccolo modulo WiFi, utilizzato per la comunicazione. Infine vengono presentate le librerie ed i firmware, scritti in linguaggio C, realizzati utilizzando l'ambiente di sviluppo STM32CubeIDE.
Per quanto riguarda l'analisi dei dati, raccolti dal sistema durante i suoi movimenti, viene utilizzato il software MATLAB.

I vari progetti realizzati nell'ambiente STM32CubeIDE ed i vari script MATLAB implementati, sono visionabili nella repository GitHub consultabile al seguente link: https://github.com/FloydPeppe/Programmazione_Sistemi_Embedded_Automazione.

Indice

Capitolo 1

Modellizzazione e Sintesi Controllo

In questo capitolo vengono ricavate le equazioni che descrivono l'evoluzione di un differential drive, con la successiva sintesi di un controllo atto ad asservire una traiettoria desiderata.

1.1 Modello Matematico

Il sistema meccanico è composto da due ruote posteriori attive (ognuna munita di un motore indipendente) e da una ruota anteriore passiva, in corrispondenza del cerchio azzurro in figura 1.1.

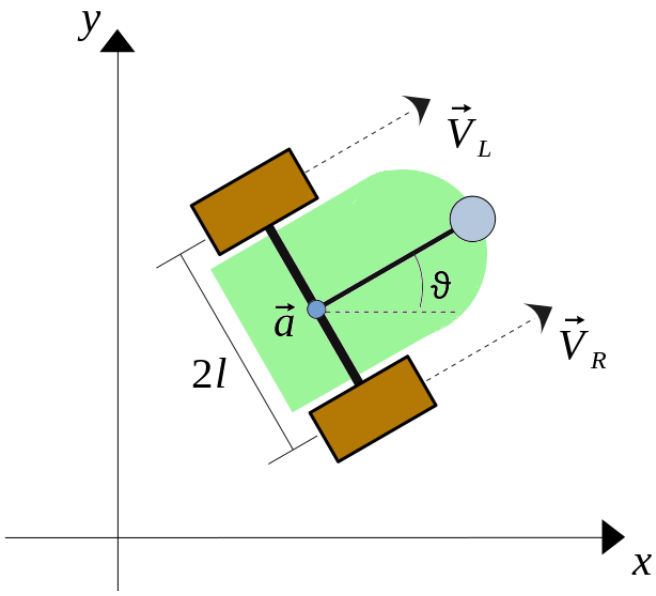

Figura 1.1: Schema Differential Drive visto dall'alto

Sempre facendo riferimento alla figura 1.1, per studiare il movimento del meccanismo è conveniente concentrarsi sull'evoluzione del punto $\vec{a}$, che possiamo scrivere in due modi distinti:

$$\begin{cases} \vec{a} = \vec{P}_R + \widehat{u}_\perp \cdot l \\ \vec{a} = \vec{P}_L - \widehat{u}_\perp \cdot l \end{cases} \tag{1.1}$$

in cui i vettori $\vec{P}_R$ e $\vec{P}_L$ sono le posizioni nel piano XY delle ruote del differential drive (R sta per "right" ed L per "left"). Inoltre si è posto:

$$\widehat{u}_\parallel = \begin{bmatrix} \cos(\vartheta) \\ \sin(\vartheta) \end{bmatrix}, \qquad \widehat{u}_\perp = \begin{bmatrix} -\sin(\vartheta) \\ \cos(\vartheta) \end{bmatrix} \tag{1.2}$$

Eguagliando le espressioni nella 1.1, derivando membro a membro ed eseguendo alcune semplici manipolazioni algebriche otteniamo la seguente relazione:

$$\vec{V}_R - \vec{V}_L = \widehat{u}_{\parallel} \cdot \dot{\vartheta} \cdot 2l \tag{1.3}$$

in cui si noti dalla 1.2 che $\widehat{u}_{\parallel} \cdot \dot{\vartheta} = -\dot{\widehat{u}}_{\perp}$
Ora, osservando che le due ruote possono spostarsi, istante per istante, solo nella direzione di $\widehat{u}_{\parallel}$ possiamo riscrivere i vettori velocità nel seguente modo:

$$\begin{cases} \vec{V}_R = V_R \cdot \widehat{u}_{\parallel} \\ \vec{V}_L = V_L \cdot \widehat{u}_{\parallel} \end{cases} , \qquad V_R,\, V_L \in \mathbb{R}$$

che ci permette di poter risolvere la 1.3 rispetto $\dot{\vartheta}$, ottenedo:

$$\dot{\vartheta} = \frac{V_R - V_L}{2l} \tag{1.4}$$

Per quanto riguarda il vettore $\vec{a}$, basta derivare una delle espressioni nella 1.1 ed eseguire alcune semplici sostituzioni per ottenere:

$$\dot{\vec{a}} = \frac{V_R + V_L}{2} \cdot \widehat{u}_{\parallel} \tag{1.5}$$

Una volta posto $\vec{a} = \begin{bmatrix} x_a \\ y_a \end{bmatrix}$ possiamo riassumere le relazioni cinematiche nel seguente modo:

$$\begin{cases} \dot{x}_a = \dfrac{V_R + V_L}{2} \cos(\vartheta) \\ \dot{y}_a = \dfrac{V_R + V_L}{2} \sin(\vartheta) \\ \dot{\vartheta} = \dfrac{V_R - V_L}{2l} \end{cases} \tag{1.6}$$

Una volta ottenute queste relazioni ci concentriamo sulle ruote del differential drive.
Come rappresentato in figura 1.2 supponiamo di avere due ruote di diametro $2r$ e ipotizziamo che esse siano sottoposte a vincolo di puro rotolamento. Ciò ci permette di scrivere la seguente relazione:

$$\begin{cases} V_R = r\ \omega_R \\ V_L = r\ \omega_L \end{cases} \tag{1.7}$$

dove ω_R ed ω_L rappresentano rispettivamente le velocità angolari della ruota destra e di quella sinistra (si noti che i segni delle rotazioni sono concordi con gli scalari V_R e V_L, ciò significa che, per soddisfare questa convenzione, nella realizzazione del sistema fisico bisognerà connettere i motori in modo opportuno). Possiamo ora riscrivere il sistema 1.6 nel seguente modo:

$$\begin{cases} \dot{x}_a = r \ \dfrac{\omega_R + \omega_L}{2} \ \cos(\vartheta) \\ \dot{y}_a = r \ \dfrac{\omega_R + \omega_L}{2} \ \sin(\vartheta) \\ \dot{\vartheta} = r \ \dfrac{\omega_R - \omega_L}{2l} \end{cases} \tag{1.8}$$

in cui viene reso esplicito il legame tra le velocità angolari delle ruote e le variazioni del vettore $\vec{a}$ e dell'orientamento ϑ del differential drive.

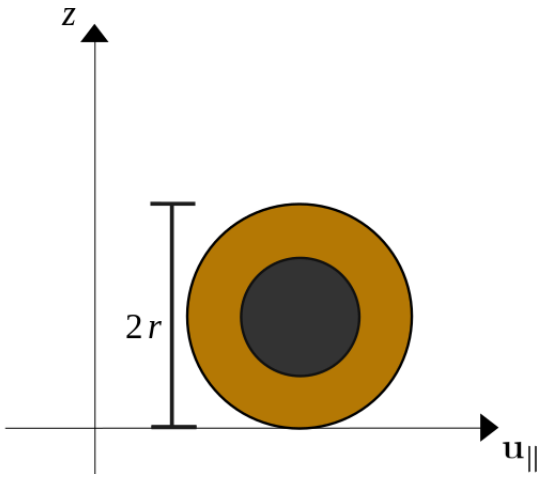

Figura 1.2: Schema Differential Drive visto dal lato ruota

1.2 Discretizzazione

Per la sintesi di una legge di controllo, che dovrà essere eseguita dal microcontrollore, è conveniente studiare il modello appena descritto in una versione tempo-discreta. Per fare ciò, fissato un tempo di campionamento T_s, un possibile approccio è quello di approssimare le derivate con dei rapporti incrementali. Nello specifico, utilizzando l'approssimazione di Eulero in avanti, possiamo scrivere la generica derivata come:

$$\dot{f}(t_k) \approx \frac{f(t_{k+1}) - f(t_k)}{T_s} \tag{1.9}$$

con $t_k = t_0 + k\ T_s$. Per alleggerire la notazione, da qui in avanti, verrà sottointeso $f(t_k) = f_k$.
Applicando questa approssimazione al sistema 1.8 otteniamo:

$$\begin{cases} x_{ak+1} = x_{ak} + T_s \ r \ \dfrac{\omega_{Rk} + \omega_{Lk}}{2} \ \cos(\vartheta_k) \\ y_{ak+1} = y_{ak} + T_s \ r \ \dfrac{\omega_{Rk} + \omega_{Lk}}{2} \ \sin(\vartheta_k) \\ \vartheta_{k+1} = \vartheta_k + T_s \ r \ \dfrac{\omega_{Rk} - \omega_{Lk}}{2l} \end{cases} \quad , \quad \begin{cases} \omega_{Rk} = \dfrac{\varphi_{Rk+1} - \varphi_{Rk}}{T_s} \\ \omega_{Lk} = \dfrac{\varphi_{Lk+1} - \varphi_{Lk}}{T_s} \end{cases} \tag{1.10}$$

Scriviamo ora:

$$\begin{cases} v_k = r\ \dfrac{\omega_{Rk} + \omega_{Lk}}{2} \\ \Omega_k = r\ \dfrac{\omega_{Rk} - \omega_{Lk}}{2l} \end{cases} \tag{1.11}$$

in cui v_k ed Ω_k rappresentano, rispettivamente, la velocità istantanea lineare e quella angolare. Possiamo così esprimere il modello 1.10 nella forma compatta:

$$\begin{cases} x_{ak+1} = x_{ak} + T_s\ v_k\ \cos(\vartheta_k) \\ y_{ak+1} = y_{ak} + T_s\ v_k\ \sin(\vartheta_k) \\ \vartheta_{k+1} = \vartheta_k + T_s\ \Omega_k \end{cases} \tag{1.12}$$

Ponendo inoltre $\Delta\varphi_k = \varphi_{k+1} - \varphi_k$, con φ_R e φ_L i rispettivi angoli spazzati dalle due ruote, si può anche scrivere la seguente relazione:

$$\begin{cases} T_s\ v_k = r\ \dfrac{\Delta\varphi_{Rk} + \Delta\varphi_{Lk}}{2} \\ T_s\ \Omega_k = r\ \dfrac{\Delta\varphi_{Rk} - \Delta\varphi_{Lk}}{2l} \end{cases} \tag{1.13}$$

1.3 Legge di controllo

Analizzando il modello 1.12 ci rendiamo conto che, data la terna di condizioni iniziali $[x_0\ y_0\ \vartheta_0]^T$, il sistema può essere completamente descritto a partire dalle velocità istantanee 1.11. Quindi, volendo sintetizzare una strategia di controllo che faccia asservire al sistema una traiettoria desiderata, basterà agire su v_k ed Ω_k.

Una volta ottenuti ${v_k}^*$ ed ${\Omega_k}^*$, ossia le velocità, istante per istante, che mi permettono di asservire la traiettoria target, posso risalire agli andamenti di φ_R e φ_L che sono le effettive grandezze che posso controllare utilizzando i motori e gli encoder. Basta risolvere il sistema 1.13 rispetto gli incrementi $\Delta\varphi_R$ e $\Delta\varphi_L$ ottenendo:

$$\begin{cases} \Delta\varphi_{Rk} = (v_k + l\ \Omega_k)\ \dfrac{T_s}{r} \\ \Delta\varphi_{Lk} = (v_k - l\ \Omega_k)\ \dfrac{T_s}{r} \end{cases} \tag{1.14}$$

che infine può essere riscritto come:

$$\begin{cases} \varphi_{Rk+1} = \varphi_{Rk} + (v_k + l\ \Omega_k)\ \dfrac{T_s}{r} \\ \varphi_{Lk+1} = \varphi_{Lk} + (v_k - l\ \Omega_k)\ \dfrac{T_s}{r} \end{cases} \tag{1.15}$$

Così, a partire dalla coppia di condizioni iniziali $[\varphi_{R0} \;\; \varphi_{L0}]^T$, possiamo risalire alle due funzioni target φ^*_{Rk} e φ^*_{Lk}.

Ricapitolando, una volta scelta una traiettoria $T_k = [x_{ak} \;\; y_{ak} \;\; \vartheta_k]^T$, $k \in \{0, 1, ..., N\}$ (con T_k campionamento di una funzione $T : \mathbb{R} \to \mathbb{R}^3$, di cui si presuppone la continuità ed una certa regolarità), trovo una successione di velocità ${v_k}^*$ ed ${\Omega_k}^*$ soluzione del sistema 1.12 (per T_k fissato). Una volta ottenute le velocità posso risalire a φ^*_{Rk} e φ^*_{Lk}, ossia gli angoli che rispettivamente devono aver spazzato le due ruote in ogni istante k.

1.3.1 Controllo Motori DC

Impostato il problema matematico, bisogna ora introdurre i modelli dei motori DC collegati alle due ruote. In questo progetto si è scelto di identificare ogni motore come un sistema (discreto) del secondo ordine:

$$(a_1 \; z^2 + a_2 \; z + a_3) \; \varphi_k = b \; u_k \tag{1.16}$$

in cui φ_k rappresenta sempre l'angolo spazzato dall'asse del motore (o ugualmente dalla ruota innestata ad esso), u_k la tensione in ingresso ed infine z^n è l'operatore di anticipo di n passi.

Per identificare i parametri incogniti, a partire dagli ingressi forniti e dagli angoli misurati, verrà utilizzato un modello ARMAX, ossia l'equazione 1.16, appena introdotta, a cui si aggiunge una componente che modellizza, in un certo modo, l'errore di processo e di misura. In formule:

$$A(z) \; \varphi_k = B(z) \; u_k + C(z) \; \eta_k \tag{1.17}$$

con η_k rumore bianco.

Una volta identificati i parametri incogniti si potrà utilizzare il modello 1.16 per il calcolo di una successione $\{u^*_0, u^*_1, ..., u^*_k\}$ degli ingressi che mi permettano di ottenere $\{\varphi^*_2, \varphi^*_3, ..., \varphi^*_{k+2}\}$ (in cui la mancanza di φ ad istanti antecedenti a 2, indica l'impossibilità di influenzare lo stato iniziale della ruota, come d'altronde ci si aspetta). In particolare, utilizzando un algoritmo di controllo inverso, si può scrivere:

$$u^*_k = \frac{a_1 \; \varphi^*_{k+2} + a_2 \; \varphi^*_{k+1} + a_3 \; \varphi^*_k}{b} \tag{1.18}$$

in cui si noti come per valori limitati dell'angolo φ^*_k, si ottengano valori limitati dell'ingresso u^*_k.

Il solo controllo appena esposto non è in realtà di utilizzo pratico. Infatti, per funzionare correttamente, presupporrebbe che il sistema identificato ed il sistema reale siano identici, senza alcun errore di processo. Per questo motivo, oltre alla componente in feedforward espressa dalla 1.18, è necessario aggiungere una componente in feedback. La scelta più semplice, da un punto di vista implementativo, è un controllore PID (Proporzionale-Integrale-Derivativo) sull'errore d'inseguimento, ossia tra φ^*_k target e la $\overline{\varphi}_k$ misurata dal sensore. Ponendo $e_k = \varphi^*_k - \overline{\varphi}_k$, una semplice legge di controllo PID può essere ottenuta scrivendo:

$$\overline{u}_k = u^p_k + u^i_k + u^d_k \tag{1.19}$$

con

$$\begin{cases} u_k^p = K_p \; e_k \\ u_k^i = u_{k\text{-}1}^i + K_i \; T_s \; e_k \\ u_k^d = \dfrac{K_d}{T_s} \; (e_k \text{ - } e_{k\text{-}1}) \end{cases}$$

Otteniamo così il segnale di controllo complessivo come:

$$u_k = \overline{u}_k + u_k^* \tag{1.20}$$

in cui la componente in feedforward, u^*, attua un'azione di controllo in anticipo, partendo dalla traiettoria desiderata e dalla conoscenza matematica (seppur parziale) del sistema, mentre quella in feedback, $\overline{u}$, ha il compito di riportare (se necessario) il sistema vicino al target sfruttando le misure ottenute dai sensori. Seppur il problema dell'asservimento possa essere risolto utilizzando il solo contributo del PID, la strategia esposta permette di ottenere un risultato ottimale, con il pregio di una più semplice operazione di taratura del PID.

Capitolo 2

Realizzazione Progetto

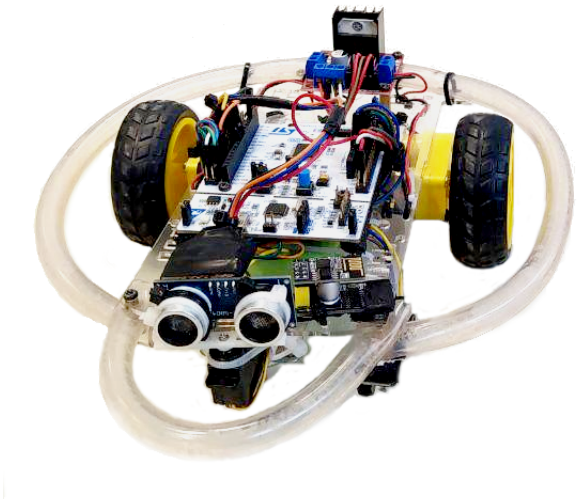

Figura 2.1: Foto progetto finito

2.1 Implementazione Hardware

Per la realizzazione differential drive del telaio del è stato utilizzato un telaio per smart car acquistato online, su cui sono stati installati la scheda STM32F446RE ed i vari attuatori e sensori. Vengono ora descritti i principali moduli utilizzati.

2.1.1 Stadio di Alimentazione

Per l'alimentazione del dispositivo, con lo scopo di renderlo autonomo da cablaggi, è stata utilizzata una batteria ricaricabile Ni-Mh da $7.2V$, mostrata in figura 2.2.

Figura 2.2: Batteria

La batteria è stata collegata alla scheda del microcontrollore (d'ora in poi chiamata MCU, MicroController Unit) tramite il pin V_{in}, da cui è possibile alimentare il dispositivo con una tensione di $7 \sim 12V$. Per fare in modo che il pin d'ingresso V_{in} sia abilitato bisogna fare attenzione ad inserire il jumper JP5 della scheda in modalità E5V.
Infine, in parallelo alla MCU, è stato collegato il driver dei motori DC che verrà descritto nella sezione successiva.

2.1.2 Attuatori

Per permettere al differential drive di muoversi sono stati utilizzati due motori DC con integrati encoder ad effetto Hall, nello specifico il loro modello è FIT0458, in figura 2.3. Sono dei motori con una fattore di riduzione 120:1 e tensione di alimentazione, nominale, di $3 \sim 7.5V$. Per poterli pilotare, è stato utilizzato un driver con integrato il chip della ST L298N. L'utilizzo di tale dispositivo si rende necessario poiché la MCU non può alimentare direttamente i motori DC, non essendo progettata per erogare la potenza necessaria. Il driver invece, alimentato da una sorgente esterna alla MCU, può essere controllato tramite i sui pin d'ingresso da segnali digitali, attivando o meno la tensione della batteria sui motori.

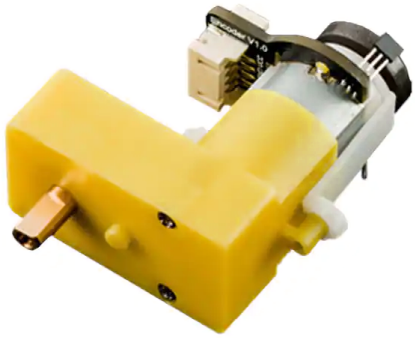

Figura 2.3: Motore DC

Per quanto riguarda ulteriori attuatori, è stato utilizzato un piccolo servo motore, con lo scopo di variare la direzione di un sensore di distanza ad ultrasuoni, il tutto posizionato sulla parte anteriore della macchina. In questo modo, si permette al dispositivo di poter scansionare possibili ostacoli, presenti in diversi punti dello spazio.

Figura 2.4: Servo motore

2.1.3 Sensori

Per la misura degli angoli spazzati dalle singole ruote, sono stati utilizzati degli encoder incrementali ad effetto Hall con segnali A B in quadratura. Come accennato precedentemente, essi sono integrati nei motori DC FIT0458 in figura 2.3. In particolare, il singolo encoder presenta una risoluzione di 8 impulsi per giro completo del rotore. Considerando il fattore di riduzione della trasmissione di 120:1, si ottiene una risoluzione di 960 impulsi ogni giro completo della ruota. Inoltre, se nei singoli impulsi vengono utilizzati sia i fronti di discesa che di salita, per incrementare (o decrementare) il contatore connesso all'encoder, si ottiene una risoluzione raddoppiata. O, ancora, contando per mezzo delle transizioni di entrambi i canali dell'encoder, otteniamo infine una risoluzione quadruplicata. In sintesi, gli encoder utilizzati consentono di avere una risoluzione a partire da un minimo di 960 stati (quasi 10 bit), ad un massimo 3840 stati (quasi 12 bit) ogni giro completo, ottenendo così il minimo angolo osservabile che va da $360°/960 \approx 0.4°$ a $360°/3840 \approx 0.1°$, che per gli scopi del progetto risulta più che sufficiente.

Un altro importante sensore utilizzato è quello ad ultrasuoni. Esso viene utilizzato per misurare la distanza tra il robot e possibili ostacoli lungo il tragitto, di modo da poter agire per evitarli. In particolare, il modulo utilizzato è il comune HC-SR04, mostrato in figura 2.5. Il funzionamento del dispositivo verrà descritto in modo dettagliato successivamente, nel paragrafo che espone le librerie realizzate per il firmware.

Figura 2.5: Sensore ad ultrasuoni

L'ultimo sensore descritto qui è quello ad infrarossi (in figura 2.6), utilizzato anch'esso per la rilevazione di possibili ostacoli. Anche se può sembrare ridondante impiegare un ulteriore sensore per il medesimo scopo, l'utilità di quest'ultimo sta nella tecnologia differente. Esso, infatti, è in grado di rilevare alcuni ostacoli che il sensore ad ultrasuoni non potrebbe rilevare in modo efficiente (per esempio materiali più "morbidi"), aggiungendo così informazioni più dettagliate riguardo lo spazio circostante la macchina.

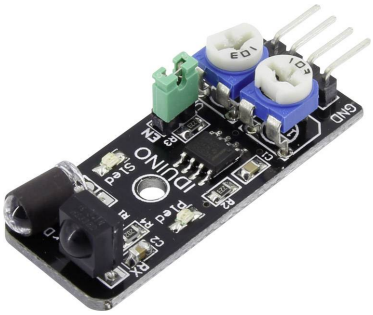

Figura 2.6: Sensore ad infrarossi

2.1.4 Comunicazione

Per rendere il robot capace di comunicare con altri dispositivi, è stato installato un piccolo componente WiFi, il modulo ESP8266-01S in figura 2.7. Per utilizzarlo come ponte da seriale (lato MCU) a WiFi (comunicazione esterna), il modulo è stato flashato con il firmware ESP-Link. Questo firmware, una volta installato utilizzando l'adattore USB, da la possibilta di configurare il modulo WiFi da remoto. Infatti, dopo essersi collegati all'access point (AP) fornito dalla scheda, basta semplicemente accedere tramite browser all'indirizzo IP del dispositivo, in genere, di default, http://192.168.4.1. A questo indirizzo verrà aperta una semplice pagina web, contenuta nel modulo, che permette di configurare i principali parametri della scheda. Per usare invece il ponte Seriale-WiFi, una volta collegati all'AP, basta utilizzare l'indirizzo IP del dispositivo sulla porta 23. Quindi (prendendo come base l'IP di default) leggendo da o scrivendo all'indirizzo http://192.168.4.1:23, tramite protocollo TCP-IP, automaticamente si ottiene una lettura o scrittura sui pin della seriale del modulo ESP. In questo modo, collegando la seriale della MCU con la seriale del modulo WiFi si ottiene un semplice ponte che permette la comunicazione wireless della scheda di sviluppo.

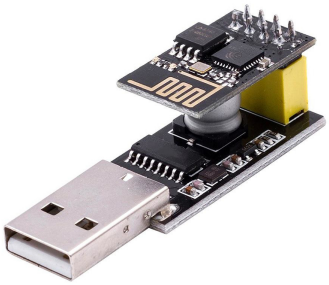

Figura 2.7: Modulo WiFi con adattatore USB

2.2 Implementazione Firmware

L'implementazione del firmware è stata svolta interamente utilizzando il software STM32CubeIDE, basato sull'ambiente di sviluppo Eclipse. Tra le pricipali librerie utilizzate ovviamente c'è la HAL (Hardware Abstraction Layer), che implementa un'importante interfaccia per la gestione della scheda, e la CMSIS-RTOS (Common Microcontroller Software Interface Standard - Real-Time Operating System), che fornisce le funzionalità di un sistema operativo e quindi l'utilizzo di vari thread tramite uno scheduler. Un passaggio basilare nella realizzazione del firmware, è stata la configurazione del dispositivo tramite l'interfaccia grafica di CubeMX, di cui se ne riportata un frammento nella figura 2.8. Tramite questo tool si semplifica di molto l'operazione di settaggio della MCU, necessaria anche per il corretto funzionamento delle librerie realizzate. Infatti, all'interno dei metodi, le periferiche vengono considerate già configurate in modo adeguato per fornire le risorse necessarie.

2.2.1 Principali Librerie Realizzate

Per interfacciarsi con i componenti descritti nel paragrafo 2.1 sono state create diverse librerie. Esse forniscono un insieme di metodi che rendono lo sviluppo del main (o dei main threads, in caso di utilizzo di RTOS) più semplice e leggibile. Inoltre, ciò, svincola il blocco di codice principale dal doversi occupare direttamente del funzionamento dei dispositivi.

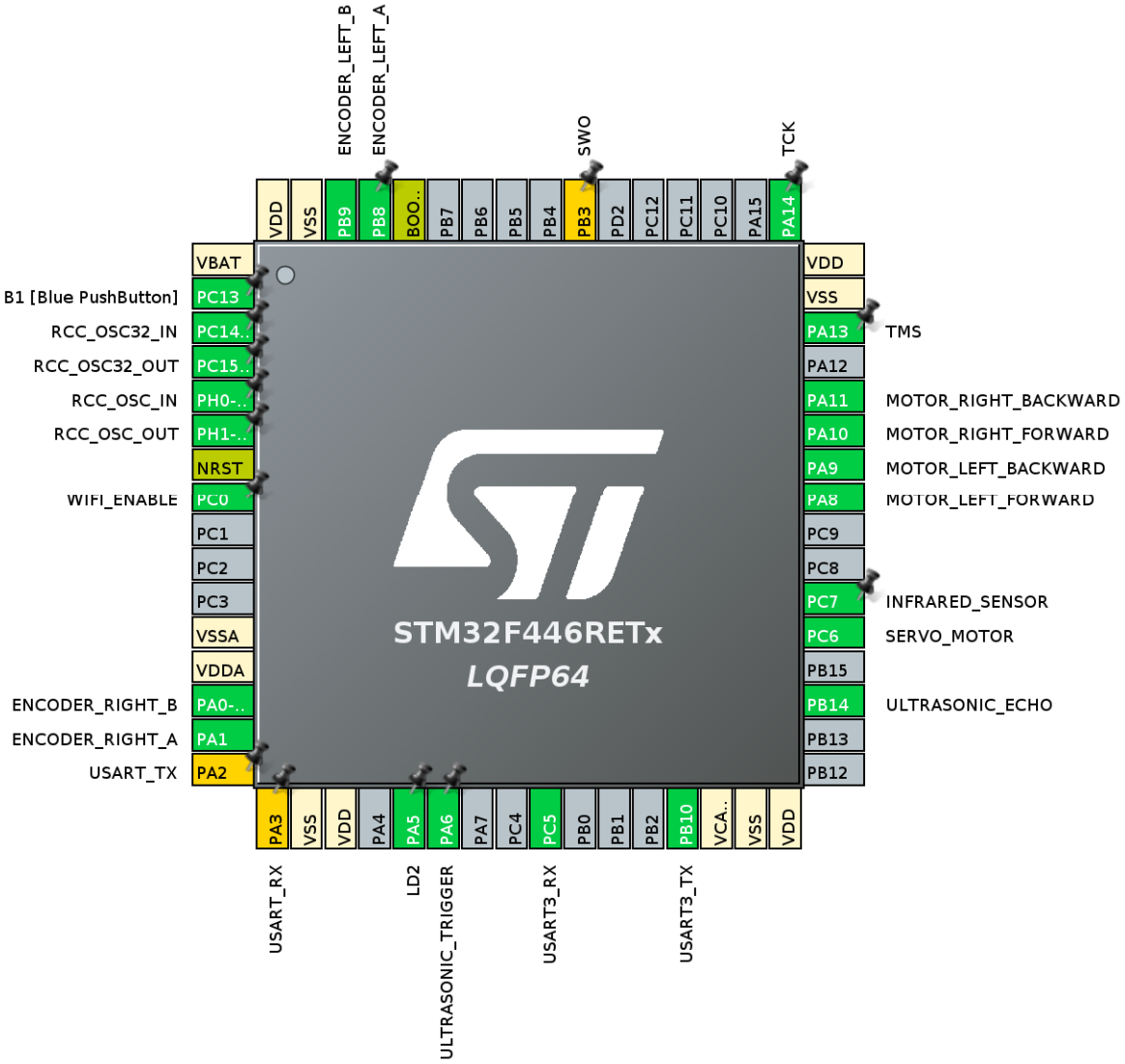

Figura 2.8: Pinout MCU, interfaccia CubeMX

Di seguito vengono descritte le implementazioni delle principali librerie che, nel file system dei progetti, sono contenute nelle cartelle `my_lib`.

2.2.2 Differential Drive

Questa è la libreria fondamentale, in cui sono state implementate le funzionalità dei motori DC, degli encoder incrementali e quindi il controllo che permette di asservire una traiettoria predefinita. Inoltre, al suo interno, viene registrato uno storico dell'evoluzione dello stato della macchina, utile per la successiva analisi dei dati. Per l'utilizzo più efficace della risorsa, è stata definita la seguente struttura:

```
/* ---------------------------------------------------------------------------- */

typedef struct _DIFFDRIVE_HandleTypeDef{
    DIFFDRIVE_InitTypeDef           Init;
    DIFFDRIVE_MOTOR_TypeDef         motor[2];
    DIFFDRIVE_ENCODER_TypeDef       encoder[2];
    DIFFDRIVE_CONTROL_TypeDef       control[2];
    DIFFDRIVE_TIMING_TypeDef        timing;
    DIFFDRIVE_STATE_TypeDef         state;
    DIFFDRIVE_HISTORY_TypeDef       history;
    DIFFDRIVE_Mechanical_Parameter mech_prmtr;
}DIFFDRIVE_HandleTypeDef;

/* ---------------------------------------------------------------------------- */
```

Come si può notare, il tipo definito, è formato a sua volta da sotto-strutture, permettendo così una maggiore leggibilità del codice. Le variabili principali sono: `motor`, `encoder`, `control` e `timing`, di cui bisogna fare attenzione ad alcuni handle che esse contengono, perché richiedono un'adeguato settaggio usando CubeMX. In particolare, in ogni elemento di `motor` (considerando che è un array di due elementi, ruota sinistra e destra) sono contenuti due timer che devono essere preconfigurati per generare segnali PWM. In questo modo, tramite il driver, un segnale permette al motore DC di girare in senso orario e l'altro ne permette il senso opposto. In ogni elemento di `encoder`, invece, è contenuto un timer che deve essere settato, appunto, in modalità Encoder. Anche in `timing`, che serve per gestire la temporizzazione, è contenuto un timer che deve essere pre-abilitato dal NVIC (Nested Vectored Interupt Controller), considerato che nei metodi della libreria verrà utilizzato in modalità interrupt. Per quanto riguarda la frequenza di quest'ultimo timer, viene automaticamente impostata dentro il metodo che inizializza la libreria, di modo da non dover essere costretti ad impostare `prescaler` e `period` da CubeMX. In particolare nel progetto è stata fissato un tempo di campionamento di $10ms$, ma può essere impostato a piacere. La variabile `control`, invece, contiene le strutture necessarie per gestire il controllo in feedforward ed in feedback descritto nella sezione 1.3.1, e non necessita di periferiche da configurare.
Di seguito viene riportato un estratto del file header in cui sono mostrati i prototipi delle funzioni implementate:

```
/* ---------------------------------------------------------------------------- */

void DIFFDRIVE_Init(DIFFDRIVE_HandleTypeDef*);
void DIFFDRIVE_DeInit(void);

void DIFFDRIVE_MOTOR_SetPWMs(float*);
void DIFFDRIVE_MOTOR_Stop(void);

void                    DIFFDRIVE_ENCODER_Reset(void);
DIFFDRIVE_StatusTypeDef DIFFDRIVE_ENCODER_CaptureMeasure(void);

void DIFFDRIVE_CONTROL_Start(void);
void DIFFDRIVE_CONTROL_Stop(void);
void DIFFDRIVE_CONTROL_Reset(void);
void DIFFDRIVE_CONTROL_WheelStep(float*);

void DIFFDRIVE_TIMING_Start(void);
void DIFFDRIVE_TIMING_Stop(void);
void DIFFDRIVE_TIMING_Wait(void);
void DIFFDRIVE_TIMING_PeriodElapsedCallback(TIM_HandleTypeDef*);
```

```
void DIFFDRIVE_STATE_Update(float*);

void DIFFDRIVE_HISTORY_Start(void);
void DIFFDRIVE_HISTORY_Stop(void);
void DIFFDRIVE_HISTORY_Update(float, float);
void DIFFDRIVE_HISTORY_Get_Init(void);
void DIFFDRIVE_HISTORY_Get_Sample(DIFFDRIVE_STATE_TypeDef*,
                                  DIFFDRIVE_STATE_TypeDef*);

void DIFFDRIVE_WheelSpeed2TrajectorySpeed(float*, float*, float*);
void DIFFDRIVE_TrajectorySpeed2WheelAngle(float*, float, float);

void DIFFDRIVE_TrackingStart(void);
void DIFFDRIVE_TrackingStop(void);
void DIFFDRIVE_TrackingStep(float, float);
void DIFFDRIVE_TrackingArray(int, float*, float*);
void DIFFDRIVE_InputArray(int, float**);

/* ------------------------------------------------------------------------- */
```

Anche in questo caso si nota la suddivisione dei metodi in diverse sezioni (le stesse che costituiscono l'HandleTypeDef della libreria) i cui scopi sono abbastanza intuibili a partire dal nome. Gli ultimi due gruppi, invece, non fanno parte di nessuna sotto-categoria e sono le funzioni che verranno utilizzate all'esterno della libreria. Di questi, il primo gruppo esegue i calcoli per passare dalle velocità della traiettoria ai rispettivi angoli spazzati dalle ruote, e viceversa (formule 1.13 e 1.15). Il secondo invece raggruppa le funzioni che implementano il controllo descritto nel paragrafo 1.3 e permettono il movimento della macchina. Nel dettaglio `DIFFDRIVE_TrackingStart(void)` è necessaria per inizializzare l'inseguimento di traiettoria, `DIFFDRIVE_TrackingStep(float v, float om)` attua un singolo passo di asservimento, mentre `DIFFDRIVE_TrackingArray(int N, float *v, float *om)` ne esegue N, infine `DIFFDRIVE_TrackingStop(void)` blocca il sistema una volta concluso il controllo.
La restante funzione `DIFFDRIVE_InputArray(int N, float **input)`, ricevendo un array multidimensionale (contente gli ingressi da fornire in modo indipendente alla ruota sinistra ed a quella destra), setta i segnali PWM ed inoltre si cura di creare uno storico delle grandezze del sistema nel tempo. Ciò si rivela necessario per il processo d'identificazione dei motori DC, come descritto nella formula 1.17.

2.2.3 Servo Motor

Un altro attuatore utilizzato è il servo motore. Per la gestione del dispositivo è stata creata una piccola libreria e di seguito ne vengono riportati l'HandleTypeDef e i prototipi dei metodi implementati:

```
/* ------------------------------------------------------------------------- */

typedef struct _SERVO_HandleTypeDef{
    SERVO_InitTypeDef  Init;
    TIM_HandleTypeDef* htim;
    uint32_t           pwm_channel;
    int                min_duty_beats;
    int                max_duty_beats;
}SERVO_HandleTypeDef;
```

```
void SERVO_Init(SERVO_HandleTypeDef*);
void SERVO_DeInit(SERVO_HandleTypeDef*);

void SERVO_SetBeats(SERVO_HandleTypeDef*, int);
void SERVO_SetDegree(SERVO_HandleTypeDef*, float);

/* ---------------------------------------------------------------------------- */
```

Anche in questo caso, il timer contenuto nel HandleTypeDef deve essere preconfigurato in modalità PWM, ma la frequenza verrà settata in modo automatico all'interno del metodo di Init. Delle funzioni mostrate, `SERVO_SetDegree(SERVO_HandleTypeDef* hservo, float deg)` è quella che verrà utilizzata all'interno del main file, il cui compito è quello di impostare il segnale PWM con il giusto duty cycle, conducendo così il motore nella posizione angolare desiderata.

2.2.4 Ultrasonic Sensor

Per utilizzare il sensore di distanza ad ultrasuoni HC-SR04 è stata realizzata una libreria apposita, `ultrasonic_sensor`. Viene qui presentata una parte del file header, contente HandleTypeDef e prototipi delle funzioni:

```
/* ---------------------------------------------------------------------------- */

typedef struct _ULTRASONIC_HandleTypeDef{
    ULTRASONIC_InitTypeDef           Init;
    TIM_HandleTypeDef*               htim;
    float                            distance;
    float*                           distance_history;
    uint32_t                         hist_idx;
    ULTRASONIC_IO_TypeDef            io;
    volatile ULTRASONIC_Flag_Typedef flag;
    uint8_t                          id;
}ULTRASONIC_HandleTypeDef;


ULTRASONIC_StatusTypeDef ULTRASONIC_Init(ULTRASONIC_HandleTypeDef*);
void                     ULTRASONIC_DeInit(void);

ULTRASONIC_StatusTypeDef ULTRASONIC_CaptureMeasure(ULTRASONIC_HandleTypeDef*);
void                     ULTRASONIC_Delay_us(ULTRASONIC_HandleTypeDef*, uint16_t);

void                     ULTRASONIC_IC_Callback(TIM_HandleTypeDef*);
void                     ULTRASONIC_Timeout_Callback(TIM_HandleTypeDef*);

/* ---------------------------------------------------------------------------- */
```

Dei metodi mostrati, quello più importante è `ULTRASONIC_CaptureMeasure`, il cui scopo è acquisire una misura e caricarla sul registro `distance` dell'handle. Nel dettaglio, all'interno del metodo vengono eseguiti i seguenti passaggi:

- Viene settato il pin TRIG con valore logico alto per almeno $10\mu s$, ciò avvia la scansione del sensore che genera 8 cicli di un'onda a $40KHz$.

- Si inizia a leggere il pin ECHO, che assume il valore logico alto non appena il trasmettitore cessa di generare il segnale a $40KHz$, ed assume valore logico basso non appena il ricevitore acquisisce il pacchetto d'onda riflessa dall'ostacolo.
- Misurando i secondi in cui il pin ECHO è rimasto con valore logico alto, si ottiene il tempo di volo dell'impulso che parte dal sensore, rimbalza e viene ricevuto indietro.
- Ottenuto il tempo di volo, tramite formula inversa (conoscendo la velocità del suono nell'aria), si ottiene infine la distanza percorsa dall'impulso.
- La distanza tra il sensore e l'ostacolo sarà la metà di quella percorsa dall'onda.

Nella figura 2.9 viene rappresentato graficamente il diagramma di timing del sensore che ne sintetizza il funzionamento. Riguardo la misura del tempo che il pin ECHO trascorre con valore logico alto, si è usato un meccanismo di input capture (IC) in modalità interrupt (IT), utilizzando le variabili appartenti all'handle della libreria e la funzione `HAL_TIM_IC_Start_IT(htim, io.echo_IC_tim_channel)`. Si noti infatti, tra i prototipi, la funzione `ULTRASONIC_IC_Callback` che, precedentemente linkata all'apposito registro del timer, viene invocata non appena si verifica un IC e setta la flag interna all'handle. Ovviamente, anche qui, ci sono alcune periferiche che bisogna far attenzione a preconfigurare. Il timer deve essere settato per la modalità IC ed abilitato dal NVIC per gli interrupt, inoltre deve essere abilitata l'opzione Register CallBack. Infine, un ulteriore periferica da configurare è il GPIO (General Purpose Input Output) dove verrà collegato il pin TRIG.

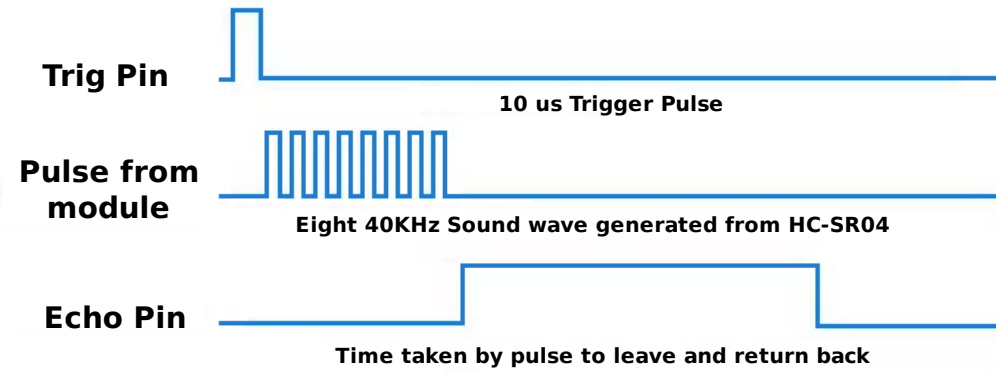

Figura 2.9: Diagramma di timing del sensore HC-SR04

2.2.5 Datastream

L'ultima importante libreria descritta qui è `datastream`, in cui sono stati realizzati i metodi necessari per la ricezione e la trasmissione di dati tramite comunicazione seriale UART (Universal Asynchronous Receiver-Transmitter). Nella versione attuale implementa solo le funzionalità base che sono state utilizzate al fine del progetto, tralasciando le svariate modalità e utilità che potrebbe offrire una risorsa che si occupa di comunicazione. Anche qui viene presentato un breve estratto del file header:

```
/* ------------------------------------------------------------------------- */

typedef struct _DATASTREAM_HandleTypeDef{
    DATASTREAM_InitTypeDef  Init;
    UART_HandleTypeDef*     huart;
    bool                    DMA_Rx_Enable;
    int32_t                 DMA_Rx_Stream_IRQn;
    volatile bool           RxCpltFlag;
    uint8_t                 id;
}DATASTREAM_HandleTypeDef;


DATASTREAM_StatusTypeDef DATASTREAM_Init(DATASTREAM_HandleTypeDef*);
void                     DATASTREAM_DeInit(void);


DATASTREAM_StatusTypeDef DATASTREAM_ReceiveDMA(DATASTREAM_HandleTypeDef*, uint8_t*,
                                               uint32_t);

void DATASTREAM_Print(DATASTREAM_HandleTypeDef*, DATASTREAM_PrintMode, int, ...);

void DATASTREAM_RxCpltCallback(UART_HandleTypeDef*);
void DATASTREAM_ErrorCallback (UART_HandleTypeDef*);

/* ------------------------------------------------------------------------- */
```

In questo caso, per far funzionare la libreria, deve essere preconfigurata la periferica UART, nel progetto utilizzata con una Baud Rate di $460800 Bits/s$ ed inoltre bisogna attivare il DMA (Direct Memory Access), utilizzato per la ricezione dei dati.
Le funzioni più importanti, invocate nel main, sono `DATASTREAM_ReceiveDMA` e `DATASTREAM_Print`. La prima necessita di tre argomenti in ingresso: un handle della libreria, un puntatore ad un buffer e la quantità di dati da ricevere. La funzione, una volta invocata, avvia una ricezione UART in modalità DMA. Una volta conclusa la ricezione, la variabile `RxCpltFlag` (contenuta nell'handle) viene aggiornata col valore `true`, per mezzo della funzione `DATASTREAM_RxCpltCallback`, precedentemente linkata all'apposito registro della variabile `huart`. Per accedere alla flag è stata anche definita una semplice macro:

```
#define __DATASTREAM_RX_DMA_CPLT(__HANDLE__) (__HANDLE__)->RxCpltFlag
```

con il fine di rendere il codice più leggibile dall'esterno.

La seconda funzione principale, invece, si occupa della trasmissione dei dati tramite UART, ma questa volta in maniera bloccante. In particolare, `DATASTREAM_Print`, è stata definita come funzione variadica (notare ... come ultimo argomento del prototipo della funzione), consentendo così di trasmettere un numero variabile di dati. L'ingresso `DATASTREAM_PrintMode` è, invece, un `enum` che permette di scegliere come formattare i dati. Le due modalità implementate sono int_csv e float_csv che permettono di trasmettere un certo numero di dati sotto forma di stringa, in cui ogni valore è separato da una virgola.

Capitolo 3

Modalità di Funzionamento

Grazie all'utilizzo delle librerie, descritte nel precedente capitolo, sono state realizzate diverse modalità di funzionamento, quindi diversi firmware. Nel workspace di STM32 sono presenti diversi progetti, i cui nomi sono `Identification_and_Trajectory_Tracking`, `Autonomous`, `Remote_Controller` e `Matlab_Controller`. I primi due verranno esposti in modo più dettagliato nei paragrafi successivi. Gli ultimi due invece verranno solo brevemente descritti a fine capitolo, per non appesantire la relazione.

3.1 Identificazione e Inseguimento Traiettoria

Questo firmware è utilizzato solo con lo scopo di identificare il modello e di testare le prestazioni del controllo implementato. Una volta raccolti, i dati possono essere trasferiti ad un sistema esterno tramite ponte UART-WiFi (utilizzando protocollo TCP-IP, lato wireless), implementato dal modulo ESP. In questo modo, utilizzando un PC con installato il software MATLAB, è possibile eseguire un'analisi dei dati acquisiti dalla scheda STM32. Utilizzando diverse `#define` e altre istruzioni del preprocessore (come `#ifdef`, `#ifndef`) è possibile modificare il comportamento del firmware, consentendo di utilizzarlo in due modi distinti:

- Evoluzione a partire da un segnale in ingresso, con raccolta dati per la successiva identificazione tramite ambiente MATLAB.
- Verifica su controllo e modello identificato, facendo seguire al sistema una traiettoria predefinita pre-caricata sulla memoria della MCU.

Per rendere il tutto più automatico sono stati creati diversi script MATLAB che permettono di generare alcuni header dentro il workspace di STM32CubeIDE. Essi verranno generati nella cartella `WORK_DIR/PROJ_DIR/Core/Inc/my_data` ed avranno un aspetto simile al seguente:

```
/*
 * diffdrive_parameters.h
 *
 *  Generated from MATLAB
 *  17-Feb-2022 00:04:54
 *
 */

#ifndef INC_MY_DATA_DIFFDRIVE_PARAMETERS_H_
#define INC_MY_DATA_DIFFDRIVE_PARAMETERS_H_


// Half the distance between wheels, in meters
#define WHEEL_DISTANCE 0.077500f

// Wheels radius in meters
#define WHEEL_RADIUS 0.034500f

// Radiant each seconds (rad/s)
```

```
#define WHEEL_MAX_SPEED 10.000000f

// Supply voltage
#define SUPPLY_VOLTAGE 7.500000f

// Minimum voltage
#define MOTOR_MIN_VOLTAGE 0.000000f

// Maximum voltage
#define MOTOR_MAX_VOLTAGE 7.500000f

// Control law time step
#define TIME_STEP 0.010000f


#endif /* INC_MY_DATA_DIFFDRIVE_PARAMETERS_H_ */
```

Nel file d'esempio, `diffdrive_parameters.h`, vengono mostrati i principali parametri fisici del differential drive. Generando automaticamente questo codice, dopo aver settato alcuni valori dentro uno script MATLAB, verranno aggiornati i file dei progetti STM32, rendendo più semplice la condivisione dei dati tra i due software.
Altri file generati automaticamente sono `idinput.h`, `control_parameters.h` e `reference.h`. Il primo contiene gli array (archiviati tramite defines) con i campioni di diversi segnali utilizzati per l'identificazione. Il secondo contiene i parametri dei modelli ARMA (AutoRegressive Moving Average) e quelli dei PID, per attuare il controllo esposto nella formula 1.20. Il terzo invece contiene degli array con una traiettoria target utilizzata per verificare la bontà del controllo attuato.

Per utilizzare il firmware nella versione che permette l'identificazione del sistema, viene inserita la seguente define nel file `main.h` del progetto:

```
#define IDENTIFICATION
```

In questo modo, nel file `main.c` verrà resa visibile al compilatore la seguente porzione di codice:

```
// Declaring array of float' pointers
float *input[2];

// Assigning values
input[left ]= SMOOTH_INPUT_L;
input[right]= SMOOTH_INPUT_R;

// Move differential drive
DIFFDRIVE_InputArray(N_SAMPLE_ID, input);
```

In cui `left` e `right` sono enum definiti in `differential_drive.h` mentre i segnali `SMOOTH_INPUT_L` e `SMOOTH_INPUT_R` sono tra gli array contenuti in `idinput.h` mediante defines. L'ultima istruzione, invece, contiene la funzione `DIFFDRIVE_InputArray` che, come esposto nel capitolo precedente, ha il compito di muovere i motori modulando il duty cycle dei segnali PWM, a partire dalla variabile `input`. Una volta fatto evolvere il sistema, il firmware permette di trasmettere i dati raccolti utilizzando la funzione `DIFFDRIVE_HISTORY_Get_Sample`, per leggere campione per campione lo storico della macchina, e `DATASTREAM_Print` per inviare i dati tramite ponte UART-WiFi.
Nella figura 3.1 viene riportato il risultato dell'identificazione del sistema, ottenendo la funzione di

trasferimento di ogni motore tramite il comando `armax` di MATLAB. Come si può notare, i valori ottenuti dalla simulazione dei modelli ARMA identificati e le misure effettive degli angoli, hanno un comportamento molto simile. Per verificare davvero la bontà di questo processo bisognerebbe confrontare sistema reale e simulato con vari ingressi di test. Ciò esula dallo scopo del progetto. Infatti, integrando nella legge di controllo un PID (come espresso nella sezione 1.3.1), si ottengono risultati ottimali senza il bisogno di una più raffinata modellizzazione del processo.

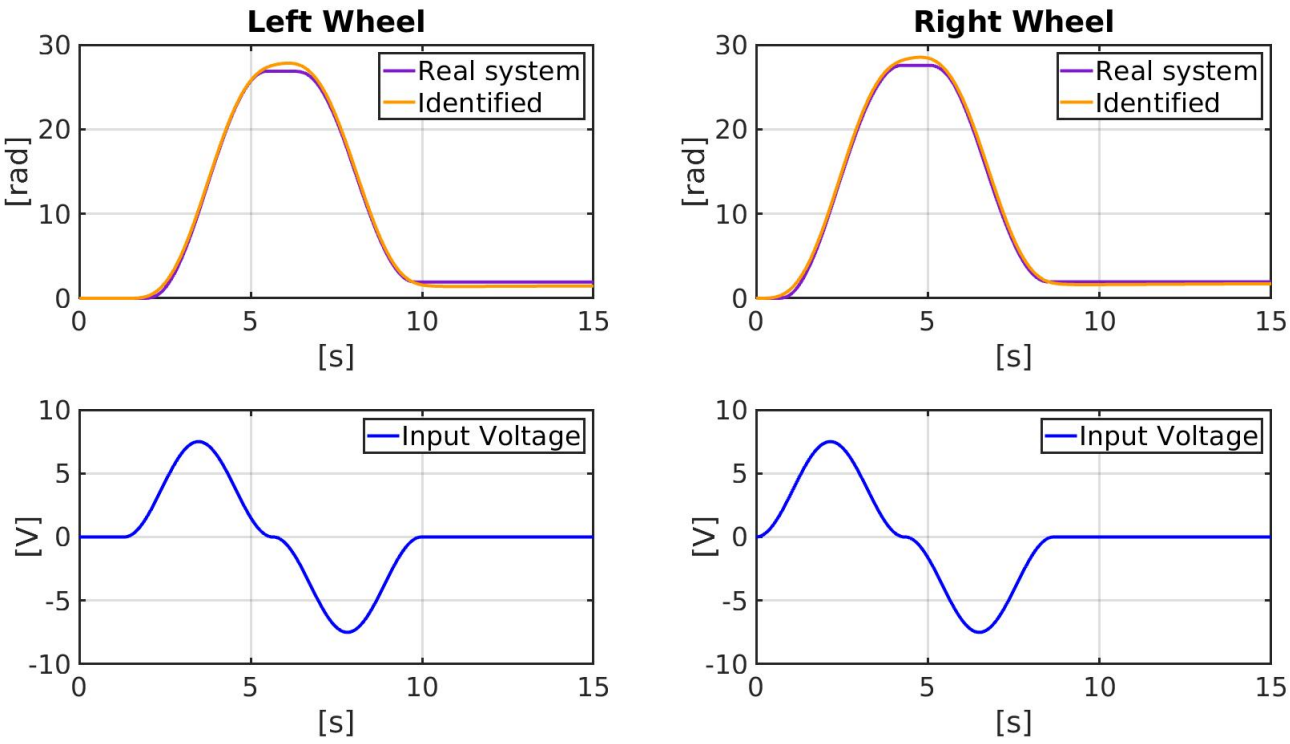

Figura 3.1: Risultati identificazione

Conclusa la fase di identificazione, si possono testare le prestazioni della legge di controllo generando una traiettoria target e archiviandola nel file `reference.h`.
Per utilizzare il firmware in questa modalità, viene inserita la seguente define:

```
#define TRAJECTORY_TRACKING
```

In questo caso invece, nel file `main.c`, verrà reso visibile al compilatore il seguente codice:

```
// Start control
DIFFDRIVE_TrackingStart();

// Tracking a target trajectory (Array of N sample)
DIFFDRIVE_TrackingArray(N_SAMPLE_REF, V_REF, OM_REF);

// Stop differential drive
DIFFDRIVE_TrackingStop();
```

In cui `V_REF` e `OM_REF` sono appunto le velocità istantanee che descrivono la traiettoria target (così come esposto nel paragrafo 1.3), contenute nel file `reference.h`.
Anche qui, una volta concluso il movimento della macchina sarà possibile trasmettere i dati raccolti esattamente come nella versione descritta precedentemente. Nella figura 3.2 vengono mostrati gli andamenti della traiettoria, dell'orientamento del differential drive (ossia ϑ) e delle velocità, confrontando target e misure.

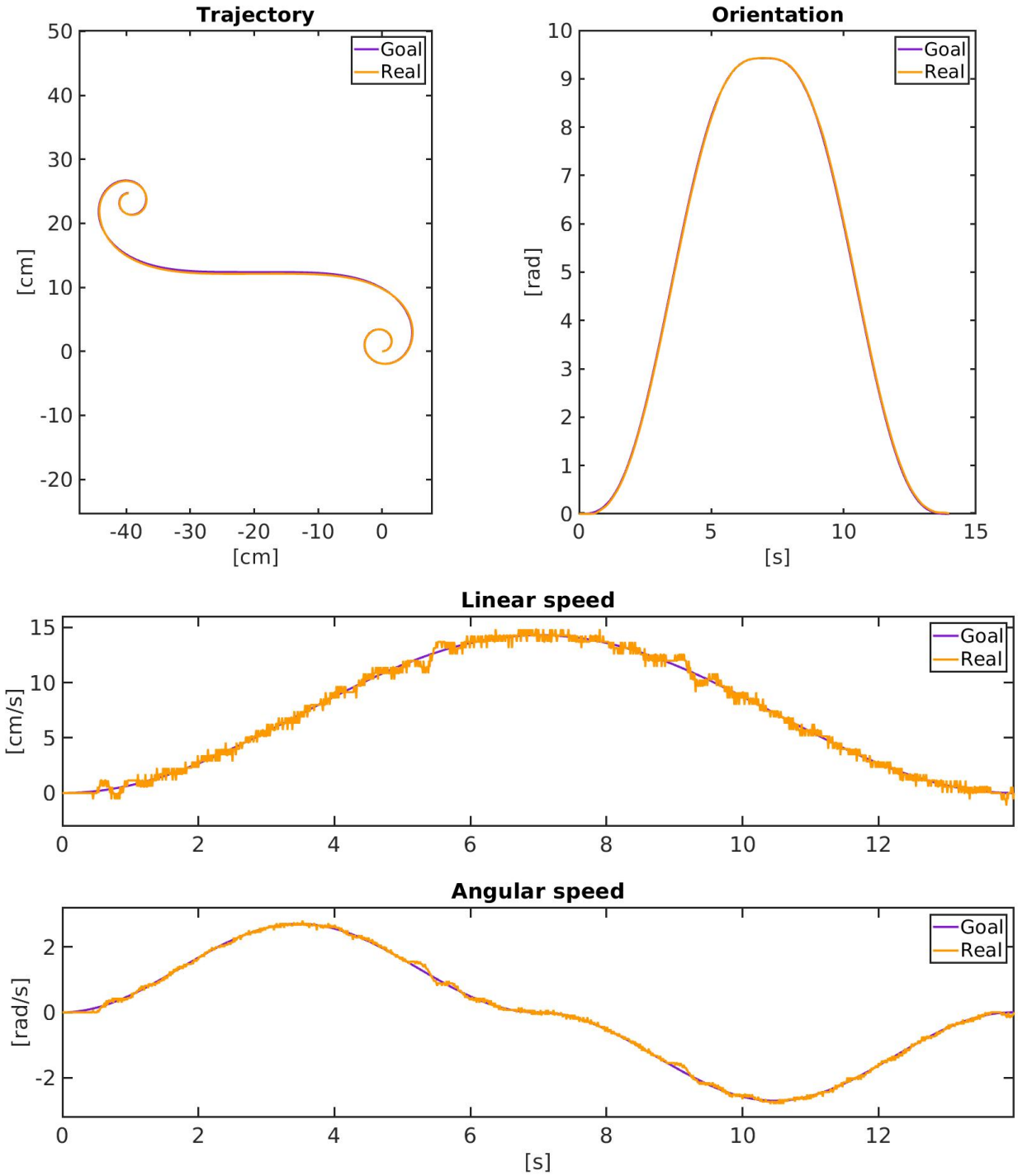

Figura 3.2: Test inseguimento di traiettoria

Come si può vedere il controllo è molto preciso sulla traiettoria e sull'orientamento. Un po' meno preciso si rivela sulle velocità, presentando un andamento a prima vista "rumoroso". Questo rumore apparente, è ereditato dalla risoluzione degli encoder incrementali, che portano ad una quantizzazione dei valori di velocità possibili. Inoltre ciò si rende visibile a causa dei piccoli valori degli incrementi presi in esame. Anche per questo motivo si è preferito attuare il controllo sulle variabili cumulative, ossia gli angoli spazzati dalle ruote, piuttosto che sulle velocità istante per istante.

3.2 Guida Autonoma

Un'altra modalità di funzionamento implementata è quella con guida autonoma. In questa versione il differential drive si muove evitando gli ostacoli circostanti e senza il bisogno di preimpostare una traiettoria. Essa, infatti, viene generata in tempo reale basandosi sui valori letti dal sensore ad ultrasuoni e da quello ad infrarossi. Per quanto riguarda la posizione fisica di quest'ultimi, il primo è stato montato nella parte anteriore della macchina, sopra un servo motore (si faccia riferimento alla foto 2.1), di modo da poter scansionare lo spazio circostante in diverse direzioni. Il secondo invece, montato sempre nella parte anteriore, è statico ed utilizzato solamente per rilevare ostacoli particolarmente vicini. Tornando alla generazione di traiettoria real time, è stata attuata una strategia abbastanza semplice, seguendo due linee guida:

- La velocità lineare sarà positiva e direttamente proporzionale alla distanza rilevata dai sensori. Al di sotto di una soglia di sicurezza preimpostata, invece, avrà segno negativo.
- La velocità angolare avrà segno positivo (rotazioni antiorarie) quando il sensore ad ultrasuoni sarà rivolto verso destra (dal punto di vista della macchina), viceversa avrà segno negativo quando sarà rivolto verso sinistra. Il modulo della velocità, invece, seguirà una legge inversamente proporzionale alla distanza rilevata.

In questa maniera la macchina sarà portata ad allontanarsi da ostacoli che si trovano ai lati, essendo costretta a cambiare direzione, mentre la velocità lineare sarà modulata in modo da rallentarla quando un ostacolo si avvicina.

Per la realizzazione di questo firmware è stata utilizzata la libreria freeRTOS, in modo da poter suddividere i diversi compiti che devono essere svolti. In particolare sono stati creati quattro diversi thread, di cui vengono mostrati i prototipi delle Entry Function nel codice seguente:

```
/* Differential Drive Task */
void MoveDiffDrive(void *argument);

/* Infrared Task */
void InfraredSensor(void *argument);

/* Tx Task */
void TxStream(void *argument);

/* Trajectory Generator Task */
void TrajectoryGenerator(void *argument);
```

Vengono ora descritte, sinteticamente, le implementazioni minimali dei quattro tasks.

■ Il thread `MoveDiffDrive` non fa altro che muovere la macchina, eseguendo in loop le seguenti istruzioni:

```
/* Local variables */
float v, om;

/* Filtering values generated from TrajectoryGenerator task */
MAF_Update(&hmaf_diffDrive_v,  linear);
MAF_Update(&hmaf_diffDrive_om, angular);

/* Extract filtered values */
v =  __MAF_GET_VAL(&hmaf_diffDrive_v);
om = __MAF_GET_VAL(&hmaf_diffDrive_om);

/* Make one step of trajectory tracking */
DIFFDRIVE_TrackingStep(v, om);
```

in cui **linear** ed **angular** sono le variabili globali delle velocità, condivise tra i threads, e MAF ed i sui handles globali (**hmaf**), implementano dei filtri a media mobile (Moving Average Filter).

■ Il thread `InfraredSensor` non fa altro che leggere il valore digitale sul pin del sensore ad infrarossi per poi passarlo in un filtro. In codici:

```
/* Local variable */
float sens;

/* Read current value on IR pin (0 is obstacle detected, 1 is free space)   */
sens = HAL_GPIO_ReadPin(INFRARED_SENSOR_GPIO_Port, INFRARED_SENSOR_Pin);

/* Using MAF filter to transform 0-1 digital state in values that span from */
/* 0 to 1 (like a normalized distance measure)                              */
MAF_Update(&hfilter_infrared, sens);
```

■ Il thread `TxStream`, invece, viene attivato solo nel caso in cui il pulsante blu della scheda viene premuto e, una volta sospesi gli altri threads, trasmette i dati raccolti utilizzando `DATASTREAM_Print`:

```
/* State variables */
DIFFDRIVE_STATE_TypeDef state_measure, state_target;

/* Number of saved samples */
int N_sample= hdiffdrive.history.sample_cnt;

/* Init History Get function */
DIFFDRIVE_HISTORY_Get_Init();

for(int k= 0; k < N_sample; k++){
    /* Get one state sample from history */
    DIFFDRIVE_HISTORY_Get_Sample(&state_measure, &state_target);

    /* Send pose sample to UART connected to ESP WiFi module */
    DATASTREAM_Print(
        &hstream, float_csv, 3, state_measure.pose.x_a,
        state_measure.pose.y_a, state_measure.pose.theta
    );
}
```

■ Infine, il thread `TrajectoryGenerator` implementa le due linee guida espresse all'inizio del paragrafo, e ne viene mostrata solo la porzione chiave dell'implementazione, per non appesantire troppo la trattazione:

```
/* Local variables */
float v, om;

/* Robot linear speed, direct proportional to distance and negative below  */
/* a distance threshold                                                    */
v = (distance - distance_threshold) * 4e-3;

/* Robot angular speed, inversely proportional to distance, with sign that */
/* depends on servo current position                                       */
if( deg_array[deg_idx] > deg_eq )
    om = -43.2/distance;
else
    om =  43.2/distance;

/* Updating filters */
MAF_Update(&hmaf_trajGen_v, v);
MAF_Update(&hmaf_trajGen_om, om);

/* Extracting filtered values (to have a smoother trajectory) */
linear =  __MAF_GET_VAL(&hmaf_trajGen_v);
angular = __MAF_GET_VAL(&hmaf_trajGen_om);
```

in cui `deg_eq` è la posizione di riferimento del servo motore, cioè quella in cui il sensore montato sopra punta in avanti. La variabile `distance`, invece, contiene la combinazione della distanza letta dal sensore ad ultrasuoni e da quello ad infrarossi. Infine `linear` ed `angular` sono le variabili globali condivise col thread `MoveDiffDrive`.

3.3 Controllo Remoto e Controllo da Matlab

Come scritto in precedenza, in questo brevissimo paragrafo vengono citati gli ultimi due progetti realizzati, `Remote_Controller` e `Matlab_Controller`.

Il firmware `Remote_Controller` permette di pilotare il differential drive utilizzando un joystick USB collegato ad un PC, oppure uno smartphone con installata una app per lo stream dei dati acquisiti dall'accelerometro. In entrambi i casi, basterà collegare il dispositivo al Access Point della scheda ESP ed inviare i dati tramite protocollo TCP-IP. In particolare i dati dovranno essere formattati rispettando le modalità implementate nel firmware.

Il progetto `Matlab_Controller` funziona in maniera simile. L'unica differenza è che vengono periodicamente inviate dalla MCU le misure della posa del robot, di modo da poter temporizzare il trasmettitore per inviare i dati con la giusta cadenza. Inoltre, questo feedback, permette di poter visualizzare in tempo reale l'andamento della macchina, utilizzando un'animazione realizzata su MATLAB.

Elenco delle figure