UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica e dell'Automazione

Regolazione e Controllo di un Sistema di Frenatura per Test di Motori di Autotrazione

Regulation and Control of a Braking System for Automotive Engine Testing

Candidato:
Matteo Gregorini

Relatore:
Professore Andrea Bonci

Anno Accademico 2023-2024

Università Politecnica delle Marche
Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica e dell'Automazione
Via Brecce Bianche – 60131 Ancona (AN), Italy

A chi mi vuole bene

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

La seguente tesi si propone di esplorare approfonditamente il tema della ricerca del miglior metodo di controllo possibile per la regolazione e il controllo di un sistema di frenatura per test di motori di autotrazione, in collaborazione con l'azienda *Soft-Engine*. L'attività verte sulla progettazione coordinata di un banco di prova che simuli in maniera più accurata possibile quelli che sono i banchi prova potenza prodotti dall'azienda stessa. L'obiettivo principe della collaborazione è lo studio e lo sviluppo di un algoritmo di controllo della frenatura che consenta di valutare in modo efficace e preciso le prestazioni dei motori testati. Per l'implementazione dell'algoritmo, e quindi per il controllo del sistema, ci si è serviti di un microcontrollore.

Il lavoro viene presentato in macro-capitoli che offrono una panoramica completa delle attività svolte, fungendo sia da riferimento per l'approfondimento dei temi trattati sia da base per l'azienda ed eventuali futuri ricercatori per portare avanti il progetto.

Nel Capitolo 1, ci si concentra sulla progettazione e l'allestimento del banco di prova, presentando il sistema e analizzando i singoli componenti hardware e i loro collegamenti.

Il Capitolo 2, invece, è incentrato sullo studio del metodo di controllo migliore per questa applicazione. Vengono vagliate diverse opzioni e si approfondisce quella ritenuta più valida.

Si prosegue con il Capitolo 3, che mostra l'implementazione dell'algoritmo di controllo scelto sul microcontrollore, con relativi codici e spiegazioni.

Con il Capitolo 4, infine, si propongono e si analizzano i test effettuati sul banco, concludendo con l'enfatizzazione dei risultati ottenuti e uno sguardo ai possibili sviluppi futuri per il progetto.

L'obiettivo finale di questo elaborato è fornire una soluzione affidabile ed efficace per la regolazione e il controllo del sistema di frenatura di banchi prova per test di motori di autotrazione, contribuendo al miglioramento dei processi di sviluppo e ottimizzazione dei motori. Inoltre, il proposito è che i risultati ottenuti possano fornire un apporto all'avanzamento degli studi in questo ambito.

Capitolo 1

Hardware

In questo capitolo vengono approfonditi la progettazione e lo sviluppo del banco prova in scala utilizzato come simulazione dei banchi prova potenza per test di motori di autotrazione, prodotti dall'azienda *Soft-Engine*. L'importanza di un ambiente simulativo accurato e affidabile è data dalla possibilità di replicare le condizioni di test reali ma in un ambiente controllato e sicuro. Lo schema di principio dell'intero sistema è riportato in Figura 1.1.

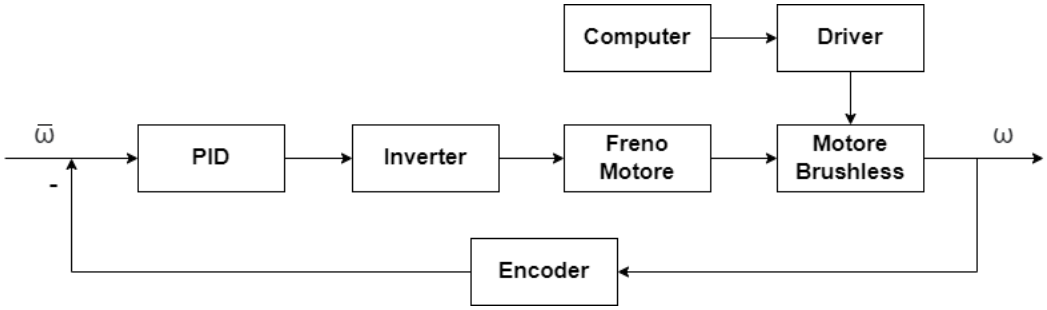


Figure 1.1: *Schema di Principio*

Il sistema banco prova reale-auto da testare, oggetto dello studio in quanto interesse primario dell'azienda, può essere schematizzato in tre componenti principali: motore dell'automobile, freno motore del banco, encoder di lettura della velocità dei rulli del banco. Al fine di progettare il sistema di prova, è pertanto necessario individuare dei componenti appropriati che permettano una simulazione semplificata ma sufficientemente fedele della situazione reale. Dopo aver condotto uno studio dettagliato in collaborazione con l'azienda, si è ritenuto che le scelte più adatte siano un motore AC brushless per emulare il motore dell'auto, un motore asincrono trifase come freno motore e un encoder rotativo per misurare la velocità di rotazione.

I due motori sono alimentati dai rispettivi inverter, secondo dei collegamenti che vengono mostrati in questo capitolo. Gli alberi dei due motori sono collegati tra loro tramite una cinghia di trasmissione dentata dal rapporto di 1:4. Le velocità massime di rotazione sono infatti 6000RPM per il motore brushless e 1500RPM per il motore trifase. All'albero del motore brushless, inoltre, è attaccato l'encoder per l'acquisizione degli RPM per il feedback necessario alla retroazione. Come precedentemente anticipato, l'intero controllo del sistema di frenatura è governato

da un microcontrollore, che pertanto gestirà l'alimentazione dell'inverter del motore trifase. La descrizione dell'algoritmo implementato è però oggetto del Capitolo 3. Qui di seguito, invece, vengono approfonditi i componenti citati. L'intero banco di prova, allestito dall'azienda, è riportato in Figura 1.2.

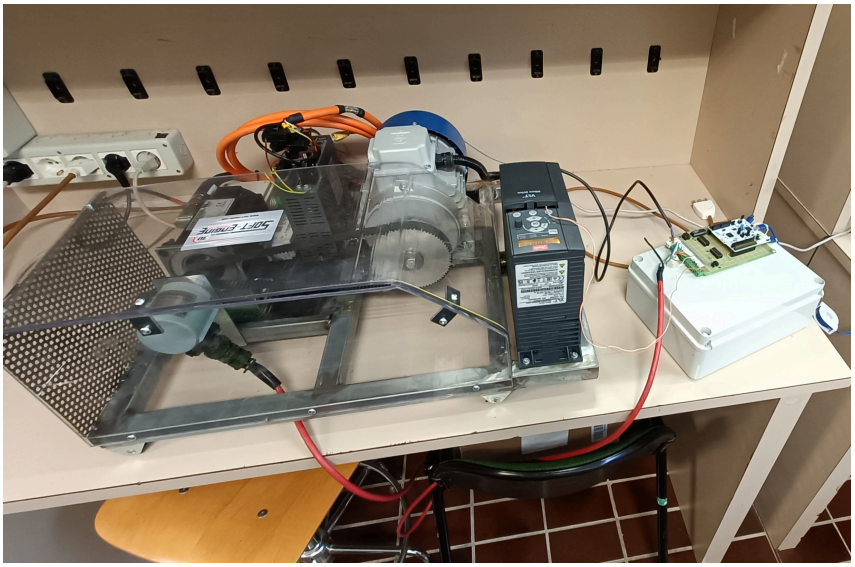


Figure 1.2: *Banco prova*

1.1 Motore AC Brushless

All'interno del banco di prova simulativo, il motore AC brushless ha il ruolo di emulare quello che sul banco di prova reale è il motore di un'automobile. La necessità è che il motore simuli quindi profili di curve di accelerazione o decelerazione di motori di autotrazione, per essere poi frenato esternamente dal motore trifase. Il motore AC brushless in questione, prodotto dalla *Kollmorgen*, è del modello *AKM42H-ANDNCA00* (Figura 1.3).

Figure 1.3: *Motore AKM42*

È un motore a 5 coppie polari, con una velocità massima di rotazione di 6000RPM capace di supportare una corrente massima di 6Arms. La sua alimentazione e il suo controllo avvengono entrambe mediante un unico connettore, collegato ad un driver anch'esso della *Kollmorgen*, modello *AKD-P00606-NBCC-I000* (Figura 1.4). A seguire l'approfondimento dei collegamenti e delle alimentazioni, aventi come riferimento il lavoro di tesi [1], condotto sui medesimi hardware *Kollmorgen*.

Figure 1.4: *Driver AKD*

1.1.1 Schema dei Collegamenti

Il sistema motore-driver richiede una doppia alimentazione. Una prima, visibile in Figura 1.5 (blocchetto bianco), genera la tensione di 24V necessaria ad alimentare il driver. Una seconda, sempre presente in Figura 1.5 (scatoletta grigia), necessaria affinché il driver possa alimentare il motore. Come anticipato, la connessione driver-motore avviene tramite un unico connettore (cavo arancione).

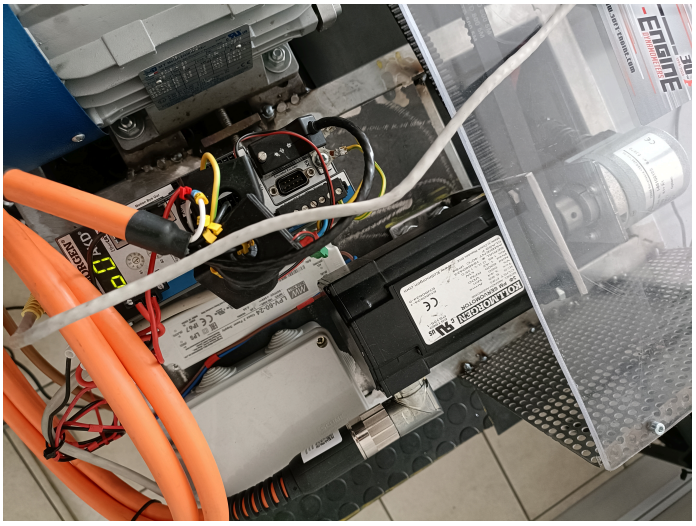


Figure 1.5: *Collegamenti driver-motore-alimentazione*

Il collegamento driver-computer, necessario per la trasmissione dei segnali d'onda per il movimento del motore, avviene invece attraverso un cavo Ethernet da inserire nella porta X11 del driver (Figura 1.6). La generazione dei segnali di controllo del motore avviene attraverso il software *WorkBench* fornito dalla *Kollmorgen*, di cui si parlerà nel prossimo sotto-capitolo.

Figure 1.6: *Porta Ethernet driver*

Di seguito, lo schema dei collegamenti del sistema driver-motore (Figura 1.7).

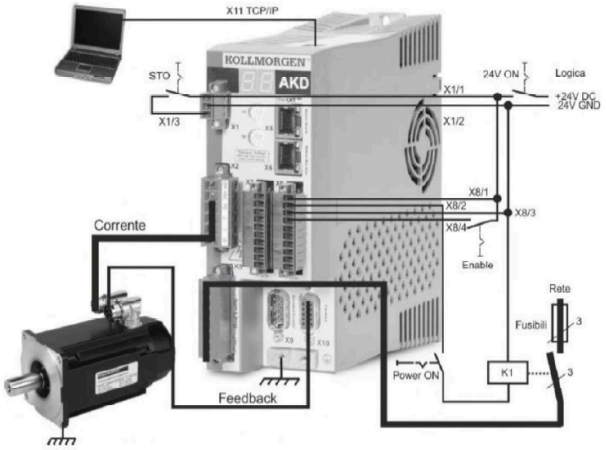


Figure 1.7: *Schema di collegamento sistema driver-motore*

1.1.2 Software WorkBench

Il software *WorkBench* è il programma dedicato all'interfacciamento con i driver, messo a disposizione dalla *Kollmorgen*. Dopo aver installato il software sul proprio computer, è necessario collegare quest'ultimo al driver tramite cavo Ethernet come

descritto in precedenza. La versione del *WorkBench* utilizzata per questo lavoro di tesi è la 2.14.0.9656.

Al fine di un corretto accoppiamento tra il driver e il computer, occorre configurare opportunamente l'indirizzo IP sia del driver che del computer stesso. In caso contrario, il computer non sarà in grado di riconoscere il driver. In entrambi i casi, è necessario assegnare ai dispositivi un indirizzo IP statico. Lato driver, per far ciò, è necessario ruotare i selettori mostrati in Figura 1.8.

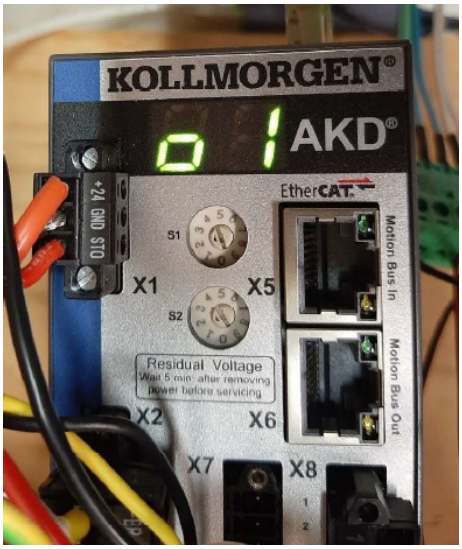


Figure 1.8: *Selezionatori indirizzo IP driver*

È possibile selezionare qualunque valore di S1 e S2, per generare l'indirizzo IP statico 192.168.0.S1S2. In questo caso, si è optato per i valori S1=2 e S2=2, ottenendo così l'indirizzo IP 192.168.0.22.

Lato computer, invece, le impostazioni dell'indirizzo IP sono riportate come in Figura 1.9.

Figure 1.9: *Impostazioni indirizzo IP computer*

Ora, avviato il software *WorkBench*, è sufficiente creare un nuovo progetto e selezione il driver come dispositivo che, se gli indirizzi IP sono stati impostati correttamente, verrà automaticamente riconosciuto dal software. L'interfaccia offerta dal *WorkBench* dispone di una serie di funzionalità che consentono la totale gestione del motore brushless tramite il driver, alla base di tutti i test che verranno presentati.

Tra queste funzionalità, la prima di nostro interesse riguarda il sistema di controllo del motore integrato al driver. Il motore *Kollmorgen* è infatti dotato di un sistema di feedback, che attua un controllo in velocità del motore attraverso l'utilizzo di un sensore di posizione. Essendo l'intero focus della tesi sviluppare un sistema di controllo di frenatura del motore attraverso un secondo motore esterno, è necessario disattivare il sistema di controllo integrato al fine di non falsare i test che seguiranno. Ricordiamo infatti che il motore brushless ha il ruolo di simulare il motore di un'automobile in accelerazione, che deve lasciarsi però frenare da un carico esterno, esercitato in questo caso dal motore trifase che vedremo nel Capitolo 1.2, che funge da freno del sistema simulando i freni del banco reale. Se il controllo interno non fosse disattivato, ad eventuali sollecitazioni esterne il motore risponderebbe correggendo la sua velocità ignorando ogni variazione o, eventualmente, andando in blocco qualora il carico esercitato fosse troppo elevato. Il software mette a disposizione la possibilità di controllare il motore in posizione o in corrente. Dalla Figura 1.10 è possibile notare come il controllo in posizione utilizzi il sistema di feedback sopracitato.

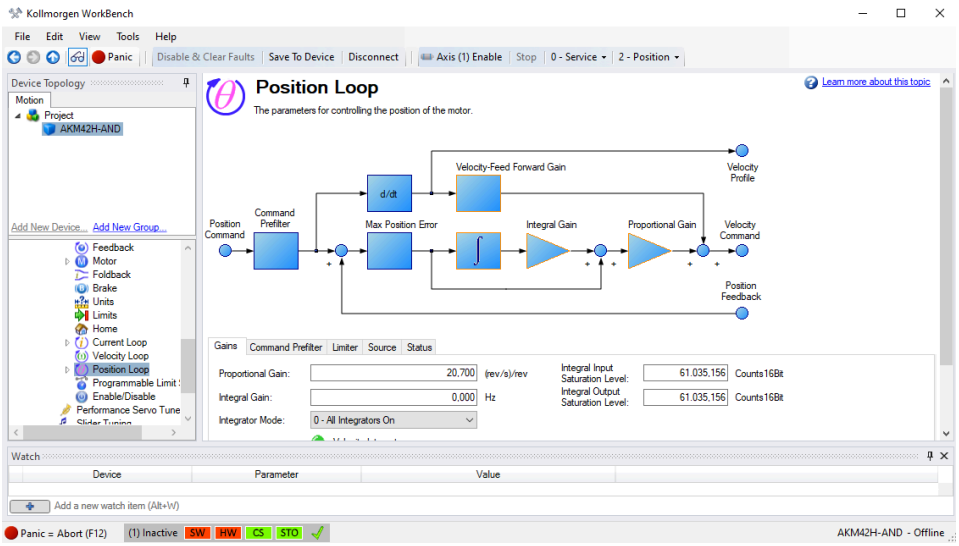


Figure 1.10: *Position Loop*

Pertanto, questa opzione, per quanto più precisa, è da scartare. L'unica alternativa possibile che non prevede il feedback in posizione, come visibile in Figura 1.11, è il controllo in corrente. Questa impostazione soddisfa perciò le esigenze richieste, permettendo la gestione della velocità del motore ma al contempo la possibilità di una frenatura esterna e lo svolgimento, quindi, dei test secondo quanto necessario. Il resto delle impostazioni è sicché basato su questa scelta.

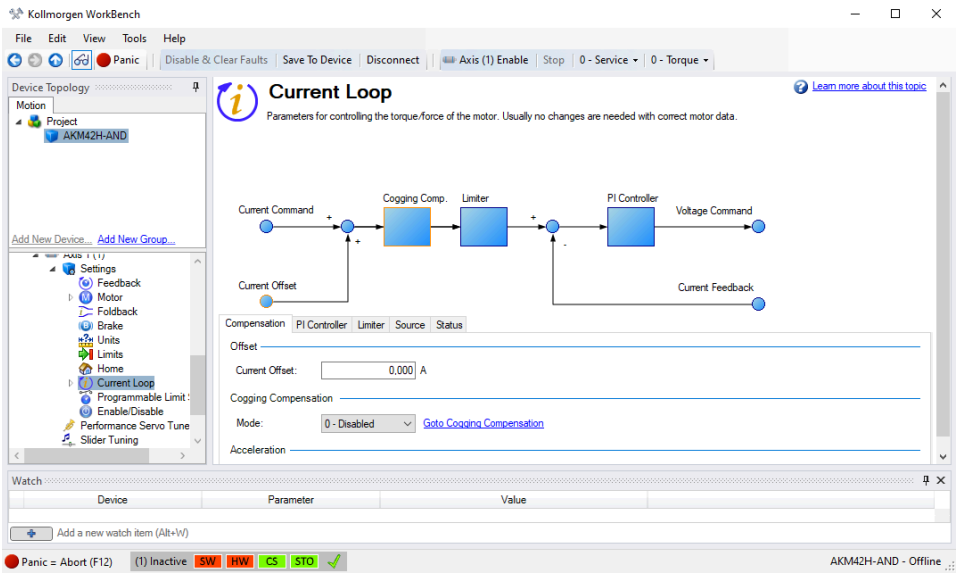


Figure 1.11: *Current Loop*

La funzionalità che più ci interessa è quella di generazione del segnale di controllo del motore *Kollmorgen*. Questa possibilità è contenuta nell'impostazione Service Motion (Figura 1.12). Come anticipato e come visibile dall'immagine, è possibile impostare il valore di corrente desiderato, al quale corrisponderà una certa velocità di rotazione del motore. Il valore massimo di corrente impostabile è 6Arms, ma come si vedrà per i test verranno utilizzati valori ben minori la soglia massima, in quanto già a 1.5Arms il motore raggiunge la sua velocità massima di 6000RPM.

Impostare una specifica corrente non consente di avere un controllo preciso sulla velocità di rotazione, in quanto a parità di corrente impostata ma a misurazioni differenti, il motore raggiunge velocità leggermente differenti a regime. Essendo però il focus delle misurazioni il controllo in velocità esterno, questo fattore non interferisce con i test in quanto non è rilevante quale velocità raggiunga il motore a regime nella situazione in cui non è controllato esternamente.

Il segnale che è possibile fornire al motore tramite questa funzionalità è un'onda quadra, il cui valore di picco è variabile in tempo reale modificando la corrente fornita. Il profilo della curva di accelerazione del motore, alimentato da questo segnale, è sufficientemente simulativo della curva di accelerazione dei motori di autotrazione testati nei banchi reali. Pertanto, come vedremo nel Capitolo 4 dedicato ai test effettuati, è sufficiente variare questo parametro per simulare le curve di accelerazione di interesse.

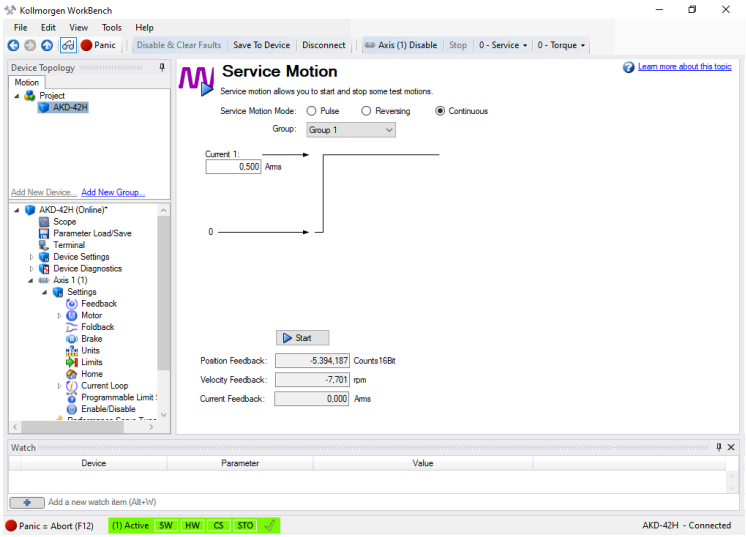


Figure 1.12: *Service Motion*

Come ultima funzionalità del *WorkBench* significativa ai fini delle misurazioni, vi è la possibilità di acquisizione delle curve di velocità e di corrente del motore tramite lo strumento *Scope*, visibile in Figura 1.13. Tutti i grafici presenti nel capitolo dedicato ai test, sono stati registrati tramite questa funzione.

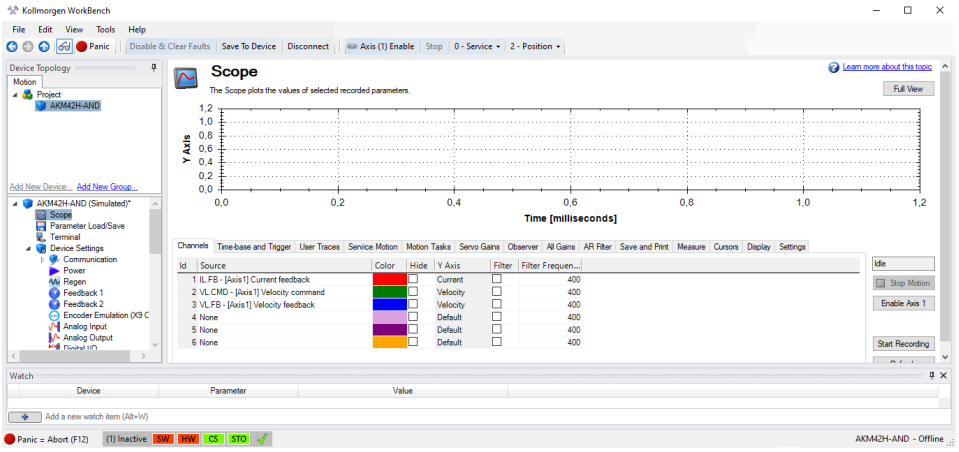


Figure 1.13: *Scope*

1.2 Motore Asincrono Trifase

All'interno del banco di prova simulativo, il motore asincrono trifase ha il ruolo di freno motore, emulando l'azione che sul banco di prova reale ha il motore di frenatura a correnti parassite. Il motore asincrono in questione è del modello *TN90-B/4-B3* (Figura 1.14).

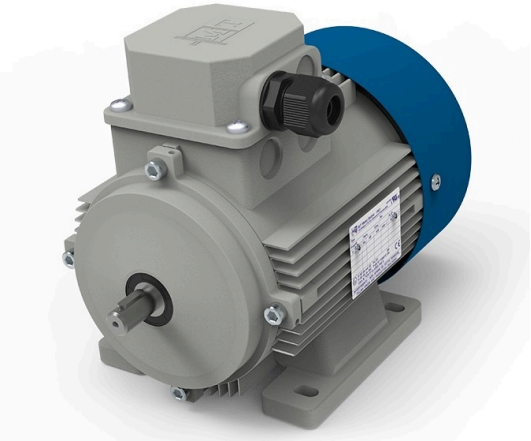


Figure 1.14: *Motore TN90-B4-B3*

È un motore a 4 poli, con una velocità massima di rotazione di 1500RPM a una frequenza di 50Hz. Il suo controllo e la sua alimentazione sono gestiti dal microcontrollore *STM32 NUCLEO-G474RE* (Figura 1.15).

Figure 1.15: *Microcontrollore STM32 NUCLEO-G474RE*

Necessitando però il motore di un'alimentazione in corrente alternata, che il microcontrollore non è in grado di generare attraverso le sue periferiche, questo richiede il supporto di un inverter che converta un determinato voltaggio tra 0V-3.3V in entrata, generabile tramite le periferiche del micro, in corrente alternata in uscita verso il motore. La velocità di rotazione del motore varia in maniera proporzionale in questo range, dove 0V corrispondono a 0RPM e 3.3V a 1500RPM. L'inverter scelto per l'adempimento di questo compito è il *VLT FC-051*, della *Danfoss* (Figura 1.16).

A seguire l'approfondimento dei collegamenti e delle alimentazioni di questa parte di sistema.

Figure 1.16: *Inverter VLT FC-051*

1.2.1 Schema dei Collegamenti

L'inverter utilizzato richiede una doppia alimentazione. Una prima, fornita dalla spina di collegamento alla rete, necessaria ad alimentare l'inverter stesso. Una seconda, in ingresso dal microcontrollore, necessaria ad alimentare il motore trifase dopo essere stata opportunamente convertita in corrente alternata. La connessione inverter-motore avviene tramite un unico connettore (cavo nero), visibile in Figura 1.17.

Figure 1.17: *Collegamento inverter-motore*

I pin di ingresso dell'inverter per la tensione 0V-3.3V, generata dal microcontrollore, sono il 53 (cavo marrone) e il 55 (cavo azzurro, ground), visibili in Figura 1.18. Affinché la corrente alternata, convertita a partire da questa tensione, sia trasmessa

al motore, e cioè affinché il motore sia alimentato, il contatto tra i pin 12 e 18 deve essere chiuso. Pertanto, è stato posto un ponticello tra i due pin, in dettaglio in Figura 1.18.

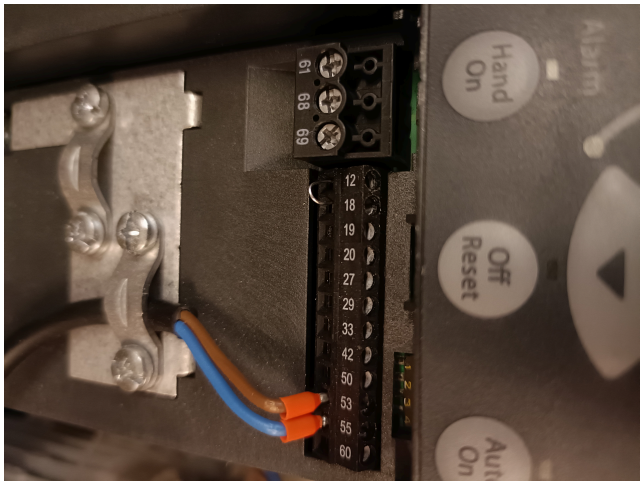


Figure 1.18: *Pin inverter*

Come anticipato, la velocità di rotazione del motore varia proporzionalmente in questo intervallo, dove 0V corrispondono a 0RPM e 3.3V a 1500RPM. Essendo gli 0V sotto la soglia minima rilevata dall'inverter, quando il microcontrollore fornisce tale voltaggio l'inverter non trasmette alimentazione al motore trifase. In altre parole, in questa situazione, è come se il contatto tra i pin 12 e 18 dell'inverter fosse aperto. Questa funzionalità è necessaria in quanto in alcune fasi il freno motore non deve esercitare nessun tipo di azione frenante sul motore brushless. Nello specifico, ogni qualvolta ci si trova sotto la soglia del setpoint di RPM impostato, che il motore *AKM* non deve superare, il motore trifase non deve intervenire. Pertanto, in queste situazioni, il microcontrollore fornirà un voltaggio pari a 0V non alimentando il motore. L'approfondimento della logica di controllo è però oggetto del Capitolo 2 e del Capitolo 3.

La periferica del microcontrollore adibita alla generazione del voltaggio 0V-3.3V è il DAC (Digital-to-Analog Converter), ovvero il convertitore digitale-analogico. Il pin di uscita della scheda è il pin A4, collegato tramite la millefori al morsetto a due vie bianco visibile in Figura 1.19. I cavi ad esso collegati sono infatti quelli del pin 53 (cavo marrone) e 55 (cavo azzurro, ground) dell'inverter, presentati in precedenza.

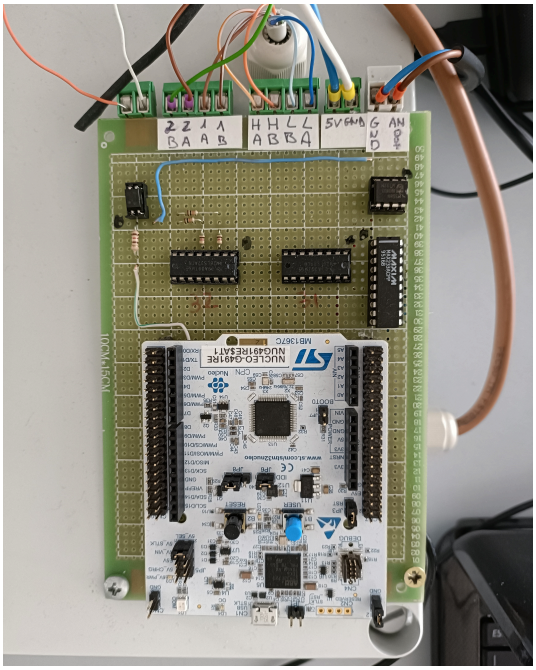


Figure 1.19: *Collegamenti microcontrollore*

Al fine di poter attuare un controllo in retroazione, è necessario un feedback sulla variabile di processo che si intende controllare. La variabile in questione è la velocità di rotazione del motore, misurata in RPM. Pertanto, allo scopo di acquisire la misurazione di tale parametro, si è optato per l'utilizzo di un encoder, modello *IH-63-I* (Figura 1.20). Questo è attaccato all'albero del motore brushless. È un encoder rotativo, con una risoluzione di 1000 impulsi per giro, sufficiente per le successive sperimentazioni. L'alimentazione richiesta è di 5V, fornita direttamente dal microcontrollore. Sia la trasmissione di dati che l'alimentazione avvengono mediante un unico connettore (cavo rosso), visibile in Figura 1.20, che si divide in 6 cavi adibiti alle varie funzioni. Il cavo marrone e il cavo verde, visibili in Figura 1.19, trasmettono l'informazione degli RPM del motore dall'encoder alla scheda mediante le porte 2A e 2B del morsetto quattro vie verde, collegate tramite la millefori al pin A1 del micro. Il cavo azzurro e il cavo bianco (ground), sempre visibili in Figura 1.19, sono collegati all'alimentazione da 5V fornita direttamente dalla scheda tramite il morsetto due vie verde. I restanti due cavi di cui si compone il connettore dell'encoder, non visibili in figura, non vengono utilizzati in quanto adibiti alla trasmissione dell'informazione del verso di rotazione dell'encoder, non necessario ai fini dei test.

Essendo il microcontrollore e la scheda millefori su cui è montata già predisposte per applicazioni future, quali l'utilizzo della scheda per il controllo del motore a correnti parassite del banco di prova reale, il resto dei cavi e dei circuiti presenti in Figura 1.19 non verranno presi in esame in quanto per l'appunto adibiti ad altre applicazioni.

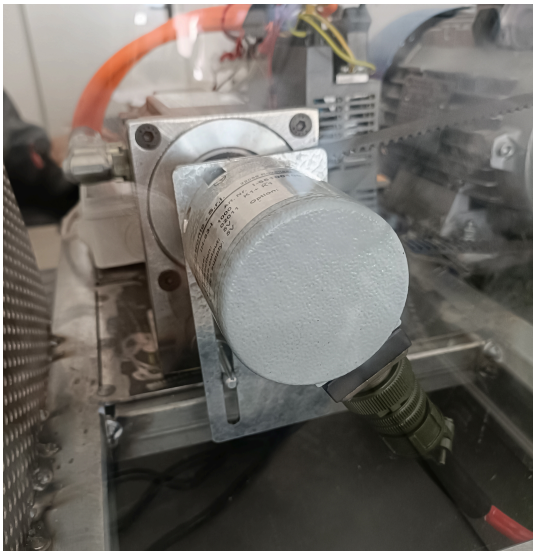


Figure 1.20: *Encoder IH-63-I*

Di seguito, lo schema dei collegamenti del sistema microcontrollore-inverter-motore-encoder (Figura 1.21).

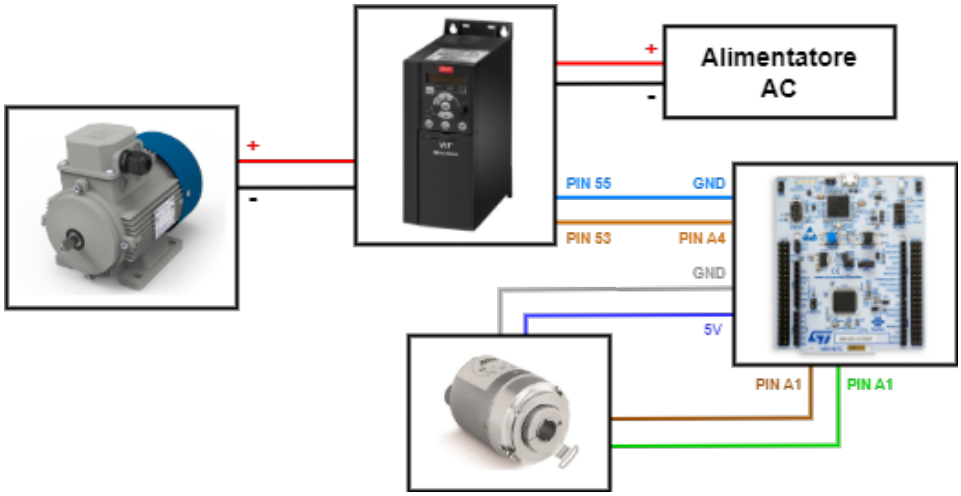


Figure 1.21: *Schema di collegamento sistema micro-inverter-motore-encoder*

Capitolo 2

Controllo

Il focus di questo lavoro di tesi è la ricerca, lo studio e la scelta del miglior sistema di controllo possibile per il sistema di frenatura per banchi prova potenza progettati dall'azienda *Soft-Engine*. Si è perciò proceduto con la ricerca in letteratura di situazioni il più analoghe possibile, imbattendosi quasi esclusivamente in due sistemi di controllo per applicazioni di questo tipo: il controllo fuzzy e il controllo PID.

Il controllo fuzzy è una tecnica che utilizza la logica fuzzy per gestire sistemi complessi e caratterizzati da incertezza. Questo metodo si basa su una serie di regole linguistiche che traducono conoscenze qualitative in azioni di controllo quantitative. È particolarmente utile in situazioni dove è difficile definire con precisione un modello matematico del sistema. Tuttavia, nel caso del controllo del sistema di frenatura di un banco di prova, il controllo fuzzy presenta alcune limitazioni. Innanzitutto, la progettazione di un controllo fuzzy per un sistema come questo richiede una definizione precisa delle regole fuzzy e dei set fuzzy, che può risultare complessa e laboriosa. Inoltre, la natura imprecisa del controllo fuzzy potrebbe compromettere la stabilità e la precisione del sistema, rischio inaccettabile per applicazioni dove la sicurezza e l'affidabilità sono critiche.

Il controllo PID, di contro, è una soluzione ampiamente riconosciuta nel mondo dell'industria per la sua semplicità, efficacia ed affidabilità. Questo tipo di controllo è infatti facilmente implementabile e i suoi parametri sintonizzabili senza difficoltà, offrendo un controllo preciso e stabile, essenziale per un sistema di frenatura. Per queste ragioni, si è deciso di procedere con il controllo PID in accordo con l'azienda. Di seguito, viene effettuata una summa di quello che è stato lo studio teorico di questo tipo di controllo, avvenuta principalmente sui testi [2, 3] e sulle slide del corso universitario di Laboratorio di Automazione, teoria che è poi alla base dello sviluppo dell'algoritmo di controllo presentato nel Capitolo 3.

2.1 PID Standard

Il controllo proporzionale-integrale-derivativo (PID) è un metodo di controllo ad anello chiuso, basato su feedback. Il principio fondante di ogni sistema di controllo in retroazione consiste nel calcolare costantemente l'errore, come la differenza tra il valore desiderato (setpoint) e la variabile di processo attuale da controllare, e nel

determinare il valore della variabile di controllo che permetta al sistema di comportarsi come desiderato. In un controllore PID, l'ingresso del sistema è determinato dalla somma di tre componenti: il termine proporzionale (P), che è proporzionale all'errore corrente; il termine integrale (I), che è proporzionale all'integrale del segnale di errore; e il termine derivativo (D), che è proporzionale alla derivata del segnale di errore. In Figura 2.1 la rappresentazione del controllo ad anello chiuso tramite PID.

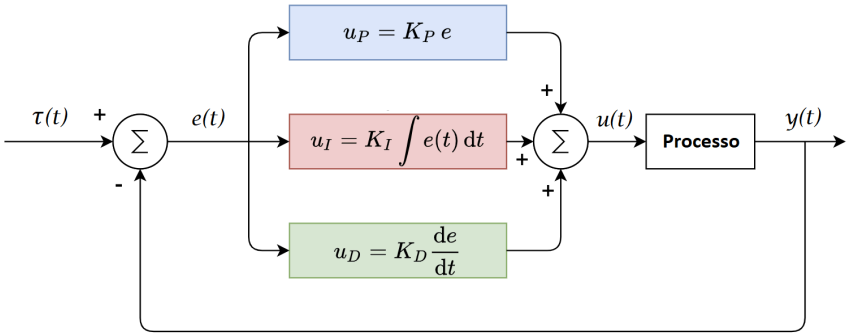


Figure 2.1: *Schema controllo PID anello chiuso*

2.1.1 Termine Proporzionale

Il termine proporzionale dipende esclusivamente dal valore dell'errore corrente (e), che è la differenza tra il setpoint attuale (r) e la variabile di processo attuale (y). La variabile di controllo per questa componente (up) viene calcolata come una semplice amplificazione dell'errore con un guadagno, noto come guadagno proporzionale K_p, il quale è il parametro di regolazione dell'azione proporzionale. La relazione tra la variabile di controllo proporzionale e l'errore corrente è data da

$$u_p(t) = K_p e(t) + uO_p \tag{2.1}$$

In generale, aumentare il guadagno proporzionale incrementa la velocità di risposta del sistema di controllo. Tuttavia, se il guadagno è eccessivamente alto, la variabile di processo può iniziare a oscillare, portando potenzialmente all'instabilità del sistema. Gli effetti del solo termine proporzionale sono visibili in Figura 2.2. Inoltre, l'uso esclusivo del controllo proporzionale tende a lasciare un offset, ossia un errore residuo tra il setpoint e il valore di processo, poiché questo tipo di regolatore necessita di un errore per generare una risposta di uscita. Questo offset può essere eliminato aggiungendo un termine integrale, che compensa l'errore residuo integrando l'errore nel tempo per produrre una componente integrale (I) nell'uscita del regolatore. L'azione integrale, quindi, può essere vista come un'opzione di regolazione per ridurre l'errore di offset.

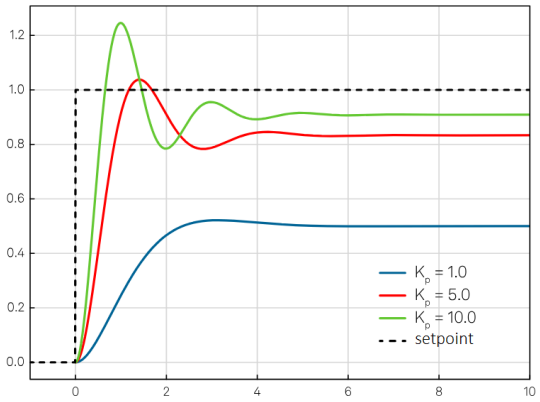


Figure 2.2: *Effetti termine proporzionale*

2.1.2 Termine Integrale

Il termine integrale tiene conto dei valori passati dell'errore integrando il segnale di errore nel tempo. La variabile di controllo per questa componente (ui) viene calcolata moltiplicando questo integrale per un guadagno, chiamato guadagno integrale K_i, che è il parametro di regolazione dell'azione integrale. La relazione tra la variabile di controllo integrale e l'errore corrente è data da

$$u_i(t) = K_i \int_0^t e(\tau)\, d\tau \tag{2.2}$$

Questa formula mostra che la risposta integrale aumenterà continuamente nel tempo, a meno che l'errore non sia pari a zero, e ciò permette di eliminare l'errore di offset causato dall'azione proporzionale. Quando l'errore è eliminato, il termine integrale smette di crescere. In questo modo, mentre l'effetto proporzionale diminuisce con la riduzione dell'errore, viene compensato dalla crescita dell'effetto integrale.

Tuttavia, un aspetto negativo dell'azione integrale è che un guadagno integrale elevato, pur riducendo il tempo di risposta, può far oscillare il valore attuale intorno al setpoint e portare all'instabilità del sistema. Questo perché, anche in assenza di errore in un dato momento, l'integratore potrebbe continuare a risentire degli errori precedenti, proseguendo con la correzione fino a superare il setpoint desiderato. Gli effetti sul sistema dell'aggiunta del termine integrale sono visibili in Figura 2.3.

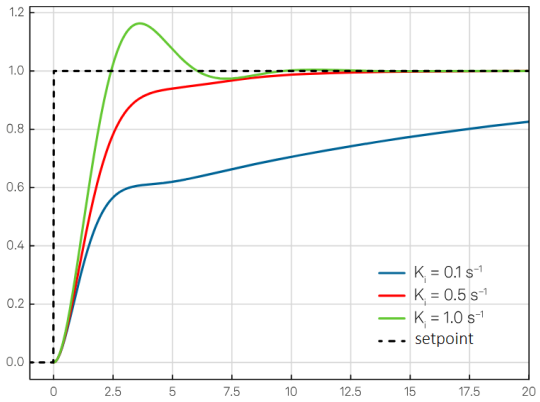


Figure 2.3: *Effetti termine integrale*

2.1.3 Termine Derivativo

Il termine derivativo differisce dagli altri poiché non considera l'entità dell'errore presente o passato, ma il suo tasso di variazione. La risposta derivativa è proporzionale al tasso di variazione dell'errore e viene regolata attraverso un guadagno derivativo K_d, che rappresenta il parametro di regolazione dell'azione derivativa. La relazione tra la variabile di controllo derivativa e l'errore corrente è data da

$$u_d(t) = K_d \frac{d}{dt} e(t) \tag{2.3}$$

Lo scopo di questa azione non è ridurre l'errore a zero, bensì il suo tasso, appiattendo la sua traiettoria in una linea orizzontale. Ciò smorza lo sforzo di controllo e riduce l'overshoot, migliorando generalmente la stabilità del sistema. Questi effetti sono visibili in Figura 2.4. Tuttavia, un'azione derivativa eccessiva può destabilizzare il sistema, impedendo al processo di adattarsi correttamente ai cambiamenti.

L'uso del termine derivativo può presentare problemi, poiché è molto sensibile al rumore nella misurazione della variabile di processo, che viene amplificato all'uscita del regolatore. In alcuni casi, l'utilizzo di un filtro appropriato sulla variabile di processo può risolvere il problema, ma introduce ulteriori complicazioni, poiché è necessario bilanciare accuratamente l'azione derivativa con il filtraggio.

In definitiva, l'uso di questo termine può essere vantaggioso, ma deve essere applicato nelle giuste situazioni e quantità. L'ipotesi di partenza dovrebbe essere quella di verificare se il processo funziona adeguatamente con il solo controllo PI, poiché una regolazione attenta dei fattori P e I può già produrre buoni risultati in termini di riduzione dell'overshoot. L'aggiunta dell'azione derivativa deve essere fatta con cautela e con un opportuno filtraggio.

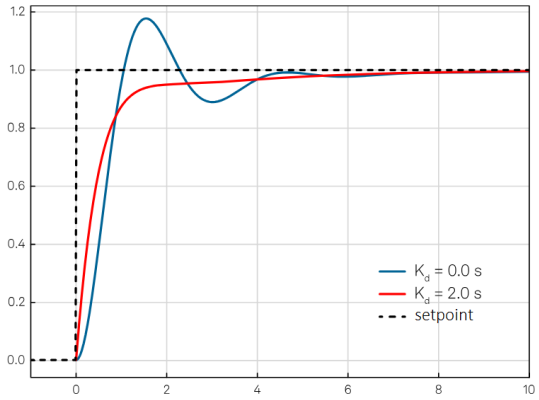


Figure 2.4: *Effetti termine derivativo*

2.2 Strutture PID

La combinazione delle azioni proporzionale, integrale e derivata può avvenire in diverse forme, matematicamente equivalenti ma con caratteristiche differenti. Le due più comunemente utilizzate sono la struttura parallela e la struttura seriale.

Le due strutture si differenziano solo per i parametri di regolazione e, di conseguenza, negli effetti della regolazione dei singoli coefficienti. Nello specifico

- Forma parallela: disaccoppiamento completo delle azioni proporzionale, integrale e derivata.
- Forma seriale: influenza del guadagno K sull'azione proporzionale, integrale e derivativa, che implica un'azione omogeneamente più aggressiva all'aumentare di K; effetto sia della costante integrale che di quella derivativa sulla costante proporzionale.

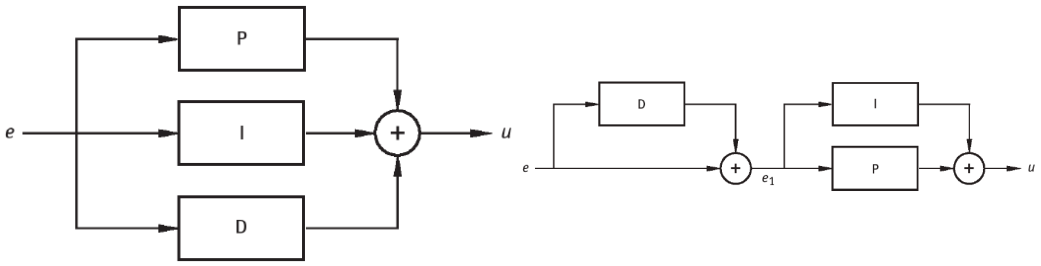
Si noti che, anche se il disaccoppiamento può sembrare sempre l'opzione più vantaggiosa, in quanto significa che ogni aggiustamento effettuato sul regolatore influisce su un solo aspetto della sua azione, a volte è meglio che il parametro del guadagno influisca su tutte e tre le azioni di controllo. In Figura 2.5 la rappresentazione grafica delle due forme.

2.3 PID Avanzato

Nei capitoli precedente è stata descritta la forma base del PID, ma esistono molte varianti che possono essere implementate per risolvere alcune problematiche che si manifestano nell'utilizzo di questa forma. Alcune di esse sono discusse di seguito.

2.3.1 Filtro Derivativo

L'utilizzo del termine derivativo, come anticipato, può comportare diverse problematiche. Nello specifico, l'ingresso di riferimento può avere angoli acuti, e questo



(a) *Struttura parallela* **(b)** *Struttura seriale*

Figure 2.5: *Strutture PID*

porta ad una derivata di riferimento molto grande e ad un ingresso di controllo conseguentemente grande se si utilizza un termine derivativo puro. Inoltre, questo termine è soggetto alla problematica del rumore, perché la derivata amplifica all'uscita del controllore il rumore sulla misura della variabile di processo. Questi due effetti sono visibili nei grafici di Figura 2.6.

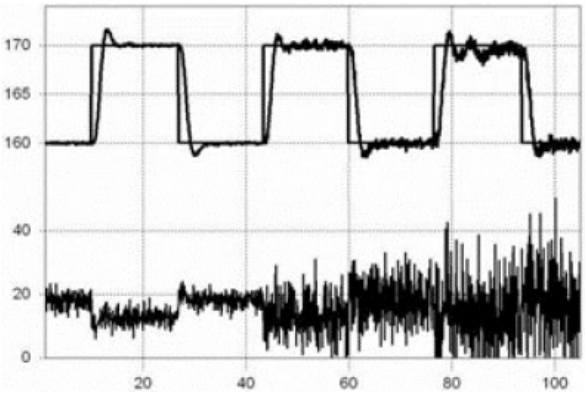


Figure 2.6: *Problematiche introduzione termine derivativo*

Queste ragioni portano spesso a complicazioni nelle applicazioni pratiche, e quindi la soluzione più immediata nei sistemi di controllo industriali attuali è tipicamente quella di disattivare il termine derivativo.

In alternativa, se si vuole mantenere l'uso di questo il termine, il primo problema può essere facilmente risolto modificando l'azione derivativa in modo che consideri la derivata della variabile di processo anziché la derivata dell'errore. In molte applicazioni il riferimento è costante, quindi questa implementazione è ragionevole.

Il secondo problema può essere invece risolto inserendo un filtro del primo ordine sul termine derivativo e regolando il suo polo in modo che non si verifichi il chattering dovuto al rumore, poiché questo attenua il rumore ad alta frequenza. Questo introduce un nuovo parametro di regolazione, che determina la frequenza di taglio del filtro, chiamata coefficiente del filtro derivativo.

2.3.2 Anti-Windup

Il windup sull'integrale è un problema dei controllori PID con azione integrale che si verifica a causa della saturazione degli attuatori. Ogni attuatore fisico è soggetto a saturazione a causa dei suoi limiti di funzionamento massimo e minimo. Se il controllore comanda all'attuatore di produrre un'uscita superiore o inferiore al suo limite, l'attuatore si satura in corrispondenza del suo limite superiore o inferiore, pilotando il sistema con un ingresso di controllo diverso dal valore della variabile controllata richiesto dal PID come visibile in Figura 2.7. Quando ciò accade, l'anello di retroazione si interrompe perché l'uscita del controllore non può più influenzare la variabile controllata. Questa condizione causa un tempo di salita maggiore e quindi un maggiore accumulo di errori nell'integratore rispetto a quelli senza saturazione. Questo accumulo di errori porta a un ampio overshoot e a un tempo di risposta più lungo, con un ritardo tra la variazione del setpoint e la risposta del sistema.

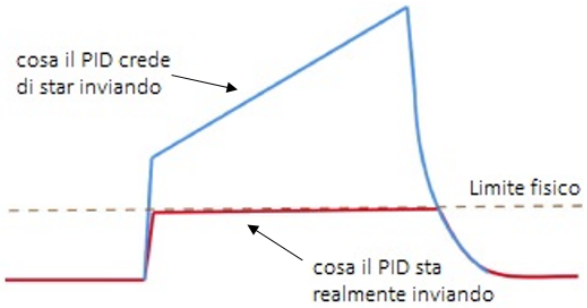


Figure 2.7: *Effetto windup*

Esistono diverse soluzioni possibili per questo problema, modificando l'azione integrale, ma i due schemi anti-windup più utilizzati sono i seguenti

- Clamping: molto semplicemente, ogni volta che si verifica una saturazione, l'integratore viene spento.
- Back-Calculation: si fornisce un percorso di retroazione supplementare intorno all'integratore, che diventa attivo e contribuisce a stabilizzare l'integratore solo quando l'anello di retroazione principale è aperto per la saturazione. In questo modo, l'errore non viene ignorato del tutto, ma il suo effetto può essere eliminato. Nello specifico, il back-calculation utilizza la differenza tra l'uscita del regolatore satura e non satura come valore di errore aggiuntivo e opposto, che riduce l'integratore impedendo il windup. Questo errore ha un proprio guadagno, detto guadagno di back-calculation K_t, che è un nuovo parametro di regolazione.

In Figura 2.8, sono comparati gli effetti dei due metodi anti-windup sulla risposta del sistema.

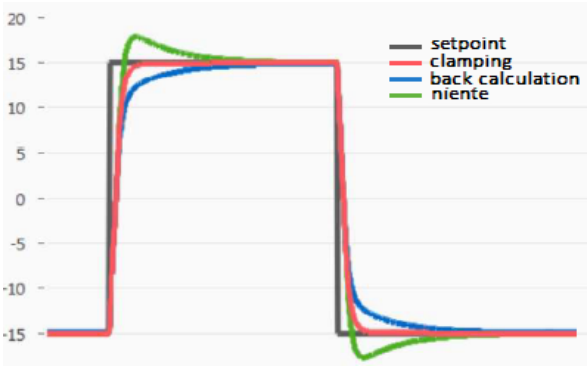


Figure 2.8: *Effetto anti-windup*

2.3.3 Gain Scheduling

Finora abbiamo ipotizzato che il processo da controllare sia lineare, implicando che la risposta del processo alle variazioni dell'uscita del controllore sia uniforme per tutti i punti operativi. L'utilizzo delle stesse impostazioni del controllore per l'intero intervallo operativo è generalmente efficace quando il processo è lineare. Tuttavia, nella pratica, i sistemi di controllo presentano sempre delle non linearità. In molti casi, queste non linearità possono essere trascurate senza compromettere significativamente le prestazioni del sistema. In situazioni in cui sono richieste prestazioni elevate, però, le non linearità possono causare problemi che devono essere affrontati.

Per ovviare a questo problema, si utilizza la tecnica del gain scheduling. Invece di mantenere fissi i parametri K_p, K_i e K_d del PID, questi vengono adattati dinamicamente in funzione di una variabile di scheduling, che rappresenta le condizioni operative del sistema, come ad esempio la velocità di rotazione di un motore o direttamente l'errore misurato. Si supponga di desiderare un controllo robusto per la maggioranza dell'intervallo di funzionamento, mentre si vuole un controllo aggressivo quando la variabile di scheduling è in prossimità del limite alto o basso. Oppure, più comunemente, che si necessiti di migliorare le prestazioni del controllo richiedendo una risposta più aggressiva in presenza di errori grandi e più morbida in presenza di errori piccoli.

Dopo aver individuato la variabile operativa di nostro interesse, identificandola come variabile di scheduling, è necessario costruire una tabella con diversi parametri del controllore. Ad ogni fascia di valori della variabile, è associato un diverso set di parametri PID. Si tratta, questo, di un lavoro che richiede molto tempo e risorse. Considerate, però, la robustezza, la flessibilità e il generale miglioramento delle prestazioni che ne conseguono, è una tecnica che giustifica lo sforzo della complessa fase di scheduling e della precisa taratura iniziale.

2.4 Regolazione Parametri

Un aspetto cruciale per il funzionamento ottimale del controllore PID, consiste nella regolazione (tuning) dei suoi parametri, ovvero dei guadagni proporzionale, integrale e derivativo. Per far ciò, si può applicare un approccio matematicamente più fondato o uno più qualitativo. I metodi di regolazione più sistematici, si dividono in due grandi categorie

- Metodi ad Anello Aperto: si basano sulla stima dei parametri del modello scelto per descrivere il processo, che tipicamente derivano da test ad anello aperto. Alcuni esempi sono: metodo *Cohen-Coon*, metodo *Ziegler-Nichols* ad anello aperto, criteri di errore minimo.
- Metodi ad Anello Chiuso: non richiedono una stima parametrica e i parametri PID sono calcolati in base al risultato di un test ad anello chiuso. Alcuni esempi sono: metodo *Tyreus-Luyben*, metodo *Ziegler-Nichols* ad anello chiuso.

Tuttavia, l'applicazione di questi metodi risulta complicata in quanto sono necessarie la conoscenza del modello di processo e una stima della sua approssimazione. L'alternativa è quella di utilizzare un metodo euristico, in cui si seguono regole generali per ottenere risultati approssimativi o qualitativi. Nello specifico, un metodo universalmente riconosciuto e applicato è il cosiddetto trial-and-error. I principali vantaggi di questo metodo sono la semplicità nell'ottenere risultati ragionevoli, l'approccio intuitivo e la necessità di una conoscenza minima del processo. D'altra parte, suoi limiti sono la necessità di diverso tempo per ottenere buone prestazioni, perché si tratta di una regolazione iterativa, e la non garanzia del raggiungimento di una soluzione robusta e stabile. Di seguito, viene approfondito proprio questo metodo poiché valutato il più appropriato al netto di vantaggi e svantaggi.

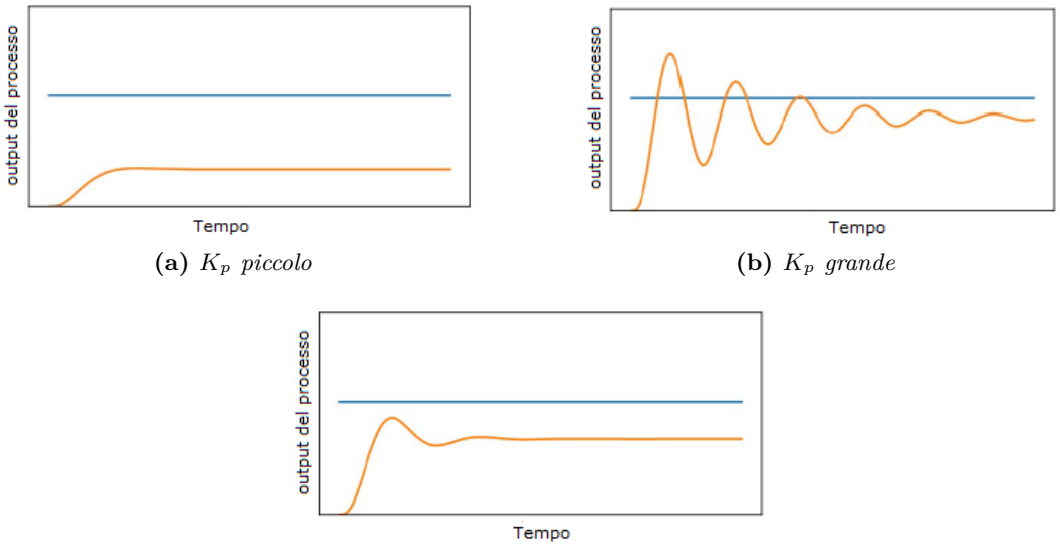
2.4.1 Trial-and-Error

Il principio alla base della tecnica di regolazione trial-and-error, è la possibilità di mettere a punto un controllore PID regolando in modo incrementale l'aggressività delle singole azioni del controllore, partendo da proporzionale, integrale e derivato, fino a raggiungere il comportamento desiderato. Le fasi da seguire sono pertanto le seguenti.

Regolare l'azione proporzionale pura (P): disattivare le azioni I e D e regolare il guadagno proporzionale K_p dando ripetuti salti al setpoint e osservando la risposta del ciclo chiuso.

- Iniziare con un K_p basso. Un buon punto di partenza può essere considerare in quale ordine di grandezza le variazioni della variabile manipolata provocano una variazione della variabile controllata e prendere una piccola frazione di questo valore.

- Ripetere l'esperimento di risposta a gradino con un aumento graduale del guadagno. Aumentando il K_p, la risposta diventa sempre più veloce e l'errore di offset diminuisce.
- Continuare fino a raggiungere il punto in cui la risposta del sistema oscilla chiaramente e diventa addirittura instabile se il guadagno viene aumentato ulteriormente. Una buona impostazione è quella in cui si verifica un notevole overshoot, che però si attenua rapidamente.



(a) K_p *piccolo*

(b) K_p *grande*

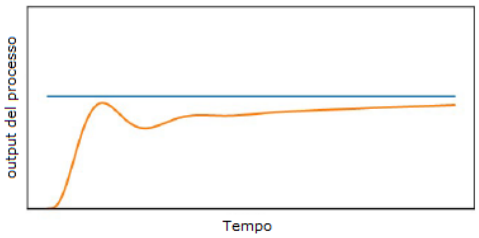
(c) K_p *ideale*

Figure 2.9: *Regolazione parametro proporzionale*

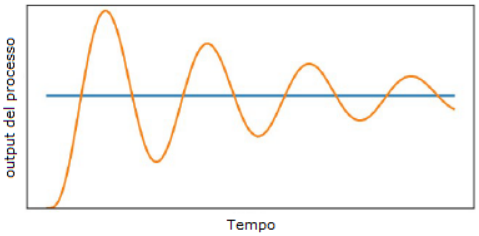
Aggiungere la componente integrale (I): mantenendo il K_p finale ottenuto dal passo precedente, regolare il guadagno integrale K_i dando ripetuti salti al setpoint e osservando la risposta del ciclo chiuso.

- Iniziare con un K_i piccolo. Un buon punto di partenza può essere l'inverso della stima della costante di tempo del sistema controllato.
- Analogamente all'impostazione della componente P, ripetere l'esperimento del salto di setpoint con un aumento graduale di K_i. Come per l'azione proporzionale, un'azione integrale troppo aggressiva porta ad oscillazioni indesiderate o addirittura all'instabilità.
- Continuare fino a raggiungere il punto in cui la risposta del sistema raggiunge il setpoint desiderato alla velocità desiderata.

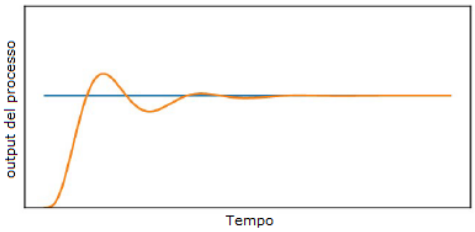
Si fa presente che per molte applicazioni un regolatore PI ben regolato è perfettamente sufficiente. Se si è già soddisfatti delle prestazioni del regolatore, ci si può fermare senza aggiungere la componente derivativa.



(a) K_i *piccolo*



(b) K_i *grande*

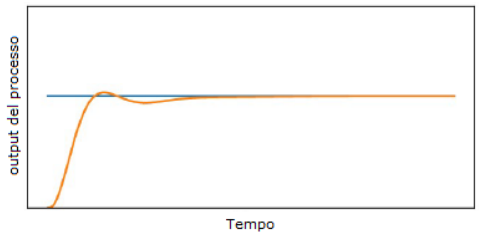


(c) K_i *ideale*

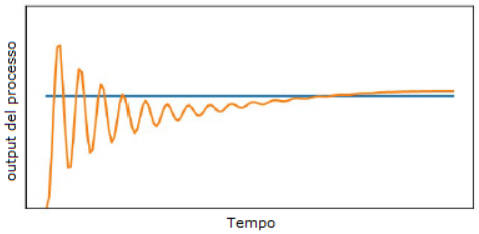
Figure 2.10: *Regolazione parametro integrale*

Aggiungere la componente derivativa (D): se si vuole aggiungere l'azione derivativa per migliorare la risposta del sistema riducendo l'overshoot e le oscillazioni, ripetere la procedura dei due passi precedenti. Un buon punto di partenza per il K_d può essere un decimo del valore K_i precedentemente selezionato.

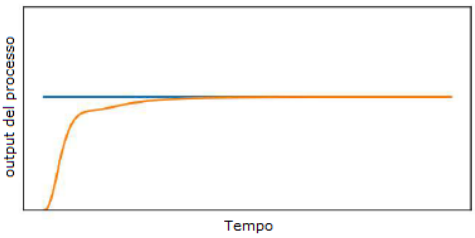
Si noti che la componente D consente anche di selezionare componenti P più grandi senza che il sistema inizi a oscillare. Questo significa che ora è possibile regolare nuovamente il guadagno proporzionale.



(a) K_d *piccolo*



(b) K_d *grande*



(c) K_d *ideale*

Figure 2.11: *Regolazione parametro derivativo*

2.4.2 Tabella di Gain Scheduling

Dopo aver selezionato un set di parametri PID seguendo il metodo descritto precedentemente, procedere ad effettuare altri test in cui questi parametri rimangono fissi mentre cambiano altre variabili. Si supponga, ad esempio, di trovarsi nella situazione oggetto di questa tesi, e di utilizzare lo stesso set di parametri PID ma a velocità prossime ai limiti inferiore e superiore del motore. Qualora eseguendo questi test il comportamento del controllore PID non fosse più ottimale, significherebbe di essere di fronte ad un sistema non lineare. Pertanto, è necessario utilizzare la tecnica del gain scheduling.

L'applicazione di tale metodo necessita la creazione di una tabella dedicata in cui, ad ogni fascia di valori della variabile di scheduling, è associato un diverso set di parametri PID. Al fine di individuare quali siano i range della variabile in cui suddividere la tabella, procedere come segue.

- Testare il proprio sistema a diversi valori della variabile di scheduling, inizialmente molto distanti tra loro, e verificare per quali valori i parametri PID precedentemente regolati non producono la risposta desiderata. Tenere nota di questi valori.
- Per ogni valore segnato, procedere nuovamente al tuning dei parametri PID, secondo il metodo di trial-and-error. Si consiglia di partire direttamente dallo step del K_i e, solo se non si ottengono risultati, modificare anche il K_p.
- Ottenute delle macro-fasce, ripetere l'operazione quante volte si ritiene necessario al fine di aumentare il numero di fasce della tabella e di conseguenza di migliorare la risposta del sistema. Si noti che più fasce vengono create, più il controllore PID sarà ottimale.

La creazione della tabella di gain scheduling, nonostante sia un'operazione iterativa e quindi piuttosto lunga, tanto più quanto si vuole ottimizzare la risposta del sistema, è un passaggio cruciale in presenza di sistemi non lineari. Di seguito è riportato un esempio di tabella di gain scheduling.

Table 2.1: *Esempio Gain Scheduling*

fascia	K_p	K_i	K_d
velocità <= 500RPM	0.1	0.02	0.005
500RPM < velocità < 2000RPM	0.12	0.01	0.003
velocità >= 2000RPM	0.09	0.01	0.007

Capitolo 3

Software

L'intero sistema di controllo del banco di prova, come anticipato, è gestito da un microcontrollore modello *STM32 NUCLEO-G474RE*. In questo capitolo viene descritto il processo di implementazione della logica di controllo sul microcontrollore, tramite software *STM32CubeIDE* versione 1.14.1, con relativi codici ed algoritmi. Dopo aver presentato il diagramma di flusso dell'intero codice, ne vengono approfondite delle parti con l'aggiunta di spiegazioni relative alle periferiche utilizzate e alla loro abilitazione.

3.1 main.c

Il source file main.c rappresenta il fulcro dell'intero programma. La prima operazione del codice consiste nella definizione della struttura catena di controllo e nella dichiarazione di variabili globali private utilizzate per la gestione e l'abilitazione delle periferiche hardware, come segue.

```
// Definizione della struttura catena di controllo
typedef struct {
    encoder_t encoder;
    pid_controller_t pid;
    float rpm_setpoint;
    float pending_voltage;
} control_chain_t;
```

Listing 3.1: "Definizione control chain"

```
// Variabile per la gestione del convertitore DAC
DAC_HandleTypeDef hdac1;

// Variabile per la gestione del timer 2
TIM_HandleTypeDef htim2;
```

Listing 3.2: "Dichiarazione periferiche"

La configurazione delle specifiche delle periferiche DAC e timer è trattata nei capitoli successivi.

Il main.c continua chiamando la funzione *User_init*. Questa ha il compito di inizializzare tutte le periferiche hardware (DAC, encoder, timer) e configurare i

regolatori PID in modo che il sistema sia pronto per eseguire il controllo del banco prova. Al fine di snellire il corpo del main.c, e per rendere più agevoli le modifiche, si fa uso di questa funzione invece di inserirvi direttamente le inizializzazioni. Per lo stesso principio, all'interno del corpo del while del main.c viene richiamata la funzione *User_loop*. Questa funzione implementa un controllo PID con gain scheduling per gestire il sistema di frenatura del motore, assicurandosi che non superi una velocità massima predefinita.

Nello specifico, legge gli RPM attuali del motore tramite un encoder utilizzando la funzione *encoder_rpm* e calcola l'errore come differenza tra il setpoint di RPM e gli RPM misurati. Se sia il setpoint che l'errore sono positivi o zero, il controllo del motore avviene applicando un gain scheduling utilizzando come viariabile di scheduling l'errore. A seconda del range dell'errore, si utilizzano parametri del PID variabili per calcolare la tensione *pending_voltage*, che viene poi impostata all'uscita del DAC. Nello specifico, se l'errore è minore o uguale a 1000, si utilizza il PID della catena 0, se l'errore è minore o uguale a 2000, si utilizza il PID della catena 1 e se l'errore è maggiore di 2000, si utilizza il PID della catena 2. Se, invece, l'errore è negativo, il controllo viene disattivato impostando la tensione all'uscita DAC a 0V di modo da disattivare l'azione frenante del motore togliendogli l'alimentazione. Questo perché, nella fase in cui la velocità del motore che simula l'automobile si trova sotto il setpoint, il freno motore non deve intervenire in quanto non si necessita di nessun tipo di frenatura.

Il diagramma di flusso del codice appena descritto è riportato in Figura 3.1.

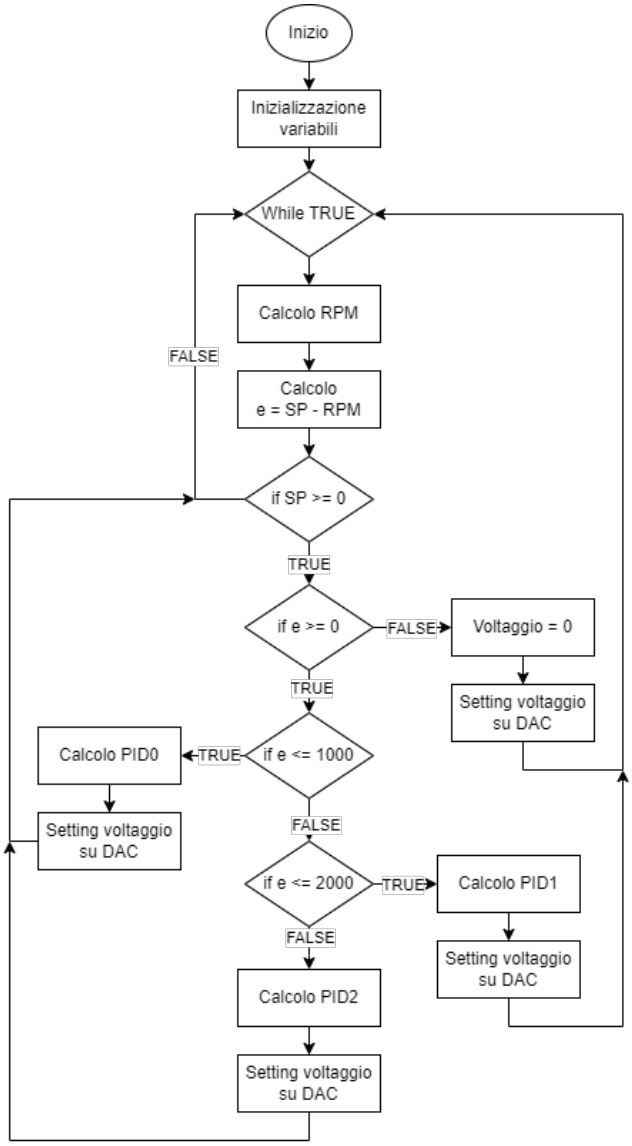


Figure 3.1: *Diagramma di flusso main.c*

3.1.1 User_init

La funzione *User_init* inizializza diverse componenti cruciali per il funzionamento del sistema. Il codice viene riportato di seguito.

```
1  void User_init() {
2
3      //inizializzazione del canale DAC
4      HAL_DAC_Start(&hdac1, DAC_CHANNEL_1);
5
6      //inizializzazione della struttura dati chains a zero
7      memset(chains, 0, sizeof(chains));
8
9      //inizializzazione setpoint e pending voltage
10     chains[0].rpm_setpoint = 0; //setpoint di partenza desiderato
```

```
11      chains[0].pending_voltage = 0; //pending voltage di partenza
12
13      //inizializzazione dell'encoder
14      //TIM_CHANNEL_3   un placeholder, in quanto non viene utilizzato
            il verso di rotazione
15      encoder_init(&chains[0].encoder, 1000, &htim2, TIM_CHANNEL_2,
            TIM_CHANNEL_3);  //1000 sono gli impulsi dell'encoder usato
16      encoder_start(&chains[0].encoder);
17
18      //inizializzazione dei diversi PID per applicare il gain
            scheduling
19
20      //inizializzazione del PID 0, con determinati parametri
21      PID_init(&chains[0].pid, 0.1, 3.3, 0.1); //Tc, u_max, u_min
22      PID_tune(&chains[0].pid, 0.001, 0.01, 0); //Kp, Ki, Kd
23
24      //inizializzazione del PID 1, con parametri diversi
25      PID_init(&chains[1].pid, 0.1, 3.3, 0.1); //Tc, u_max, u_min
26      PID_tune(&chains[1].pid, 0.001, 0.005, 0); //Kp, Ki, Kd
27
28      //inizializzazione del PID 2, con parametri diversi
29      PID_init(&chains[2].pid, 0.1, 3.3, 0.1); //Tc, u_max, u_min
30      PID_tune(&chains[2].pid, 0.001, 0.003, 0); //Kp, Ki, Kd
31  }
```

Listing 3.3: "*User_init*"

Nello specifico, come prima cosa avvia il convertitore digitale-analogico DAC sul canale *DAC_CHANNEL_1*. Questo è necessario per generare un segnale analogico a partire da un valore digitale.

Successivamente, imposta il setpoint desiderato di RPM, e il voltaggio da applicare all'uscita DAC. Questi valori iniziali determinano l'obiettivo di velocità del motore e il valore di tensione da applicare per raggiungere questo obiettivo.

Si prosegue inizializzando e avviando l'encoder, utilizzato per misurare la velocità del motore in RPM. Questa chiamata di funzione configura l'encoder con 1000 impulsi per rotazione, utilizzando il timer htim2 sui canali *TIM_CHANNEL_2* e *TIM_CHANNEL_3*.

Infine, si inizializzano e si assegnano i parametri di tre PID differenti. I valori di *Tc* (tempo di campionamento), *u_max* (output massimo) e *u_min* (output minimo) sono comuni a tutti e tre i PID. Si noti che l'output minimo non è 0 bensì 0.1 in quanto se si trasmettessero 0V al motore questo non sarebbe più alimentato, mentre il controllore penserebbe di star applicando la massima frenatura. I guadagni K_p, K_i e K_d sono invece diversi per ognuno dei PID. Disporre di PID con parametri differenti è essenziale al fine di applicare il gain scheduling.

3.1.2 User_loop

Come anticipato, la funzione *User_loop* implementa il ciclo di controllo che legge la velocità attuale del motore tramite l'encoder, calcola l'errore rispetto a un setpoint desiderato, e poi utilizza un regolatore PID per determinare la tensione necessaria da inviare al motore per correggere tale errore. Il codice applica la tecnica di gain scheduling, per adattare dinamicamente i parametri del PID in base all'entità dell'errore. Di seguito, viene riportato il codice.

```
1  void User_loop() {
2
3      // Limitatore del setpoint alla velocit  massima di rotazione del
           motore
4      if (chains[0].rpm_setpoint > 6000){
5              chains[0].rpm_setpoint = 6000;
6             }
7
8      // Valore di volt massimi che il DAC pu   generare
9      float dac_max_voltage = 3.3;
10
11     // Lettura RPM tramite encoder
12     float rpm = Encoder_rpm(&chains[0].encoder);
13
14     // Calcolo dell'errore
15     float error = chains[0].rpm_setpoint - rpm;
16
17     // Verifica presenza setpoint
18     if (chains[0].rpm_setpoint >= 0) {
19         // Valutazione errore positivo o negativo
20         if (error >= 0) {
21             // Applicazione gain scheduling, fascia 0-1000
22             if (error <= 1000) {
23                 // Calcolo del pending_voltage tramite la funzione
                       PID_compute
24                 chains[0].pending_voltage = PID_compute(&chains[0].pid
                       , chains[0].rpm_setpoint, rpm);
25                 // Impostazione del voltaggio calcolato all'uscita DAC
26                 Set_dac_value(pending_voltage, dac_max_voltage, &hdac1
                       , DAC_CHANNEL_1);
27                 // Applicazione gain scheduling, fascia 1000-2000
28             } else if (error <= 2000){
29                 // Calcolo del pending_voltage tramite la funzione
                       PID_compute
30                 chains[0].pending_voltage = PID_compute(&chains[1].pid
                       , chains[0].rpm_setpoint, rpm);
31                 // Impostazione del voltaggio calcolato all'uscita DAC
32                 Set_dac_value(pending_voltage, dac_max_voltage, &hdac1
                       , DAC_CHANNEL_1);
33                 // Applicazione gain scheduling, fascia >2000
34             } else {
```

```
35                  // Calcolo del pending_voltage tramite la funzione
                        PID_compute
36                  chains[0].pending_voltage = PID_compute(&chains[2].pid
                        , chains[0].rpm_setpoint, rpm);
37                  // Impostazione del voltaggio calcolato all'uscita DAC
38                  Set_dac_value(pending_voltage, dac_max_voltage, &hdac1
                        , DAC_CHANNEL_1);
39              }
40          } else {
41              // Se l'errore   negativo, disattivazione del motore
                    ponendo pending_voltage a 0
42              chains[0].pending_voltage = 0;
43              // Impostazione del voltaggio all'uscita DAC
44              Set_dac_value(pending_voltage, dac_max_voltage, &hdac1,
                    DAC_CHANNEL_1);
45          }
46      }
47      // Attesa di 100 ms prima della ripetizione del ciclo
48      HAL_Delay(100);
49  }
```

Listing 3.4: "*User_loop*"

Il codice fa uso di una funzione, chiamata *Set_dac_value*, che, dati in ingresso il valore di *pending_voltage* calcolato con la funzione *PID_compute*, il massimo voltaggio erogabile dalla periferica DAC e i parametri della periferica stessa, converte la tensione e imposta il valore calcolato direttamente sull'uscita digitale-to-analogico. Il codice della funzione è il seguente.

```
1 void Set_dac_value(float pending_voltage, float dac_max_voltage,
      DAC_HandleTypeDef* hdac, uint32_t dac_channel) {
2     // Conversione della tensione in un valore digitale (assumendo VDD
          = 3.3V and 12-bit DAC)
3     uint16_t dac_value = (uint16_t)((pending_voltage / dac_max_voltage
          ) * 4095);
4     // Impostastione del valore sul DAC
5     HAL_DAC_SetValue(hdac, dac_channel, DAC_ALIGN_12B_R, dac_value);
6 }
```

Listing 3.5: "*Set_dac_value*"

Come prima operazione, converte la tensione analogica fornita da *PID_compute* in un valore digitale che il DAC può comprendere. Poiché si sta utilizzando un DAC a 12 bit, il valore digitale massimo è 4095. La formula successiva calcola la proporzione del valore digitale rispetto alla tensione massima che il DAC può generare. Questa proporzione viene poi scalata fino a 4095 per ottenere il valore digitale corrispondente. Il valore viene quindi impostato nella periferica DAC di riferimento che lo converte in analogico. La porta DAC del microcontrollore utilizzata per questa funzione è la *DAC1_OUT1*. Per configurarla, basta accedere allo strumento dedicato, visibile in Figura 3.2.

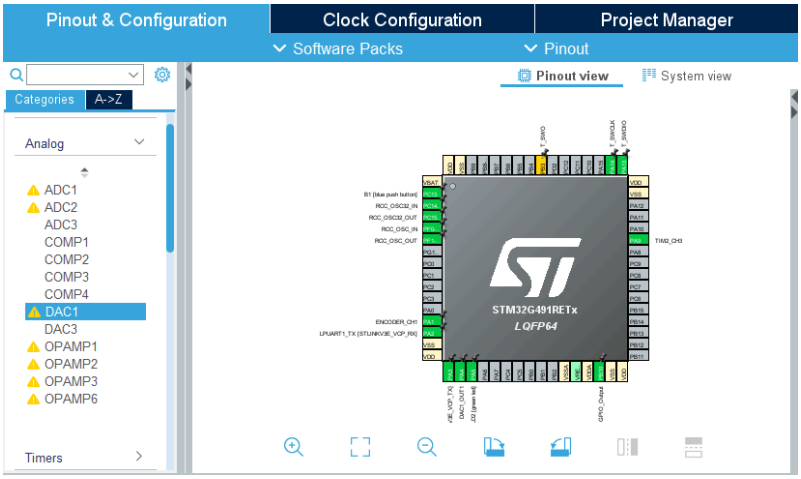


Figure 3.2: *Strumento gestione pin*

Il terminale scelto come uscita DAC è il pin A4, visibile in Figura 3.3. Le impostazioni sono quelle di default.

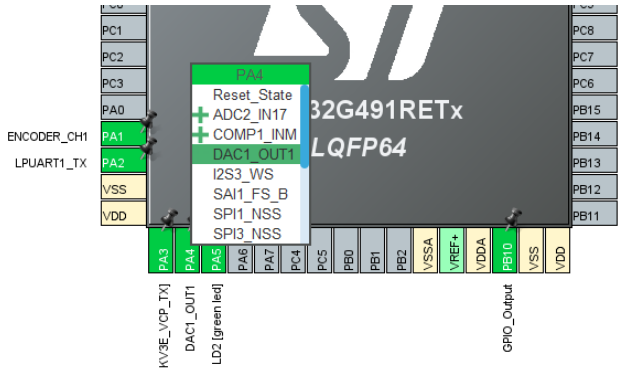


Figure 3.3: *PinA4 DAC1_OUT1*

3.2 pid.c

Questo source file definisce le funzioni necessarie all'inizializzazione e al tuning del controllore PID, e al calcolo della variabile di controllo da applicare al sistema. È pertanto punto cardine della tesi in quanto sintetizza i concetti studiati finora traducendoli e implementandoli in forma di algoritmo, adibito all'intero controllo del sistema di frenatura. Il codice presentato ha applicazione universale e, perciò, è possibile migrarlo su altri microcontrollori o altri software se necessario. Il diagramma di flusso dell'algoritmo di controllo PID è riportato di seguito in Figura 3.4.

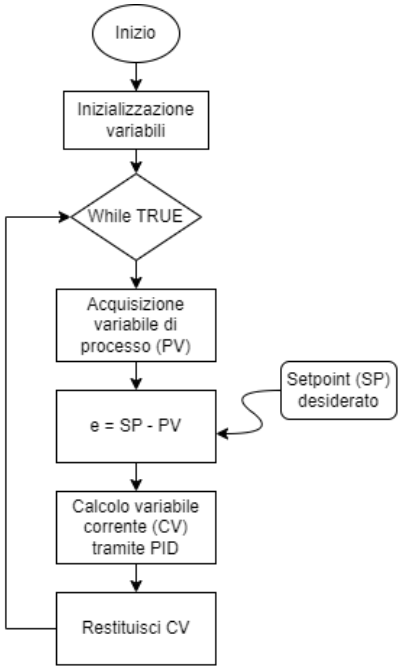


Figure 3.4: *Diagramma di flusso pid.c*

3.2.1 PID_init e PID_tune

Le funzioni *PID_init* e *PID_tune* sono deputate all'inizializzazione dei valori caratterizzanti il controllore PID, opportunamente valutati e regolati. Queste funzioni configurano il controllore PID per operare con i parametri desiderati prima di utilizzarlo nel ciclo di controllo.

La funzione *PID_init* inizializza i parametri di base del PID, quali il tempo di campionamento e i limiti dell'uscita. Il suo codice è riportato di seguito.

```
1 void PID_init(pid_controller_t *pid, float Tc, float u_max, float
      u_min){
2     pid->Tc = Tc; // Imposta il periodo di campionamento
3     pid->u_max = u_max; // Imposta il limite superiore output PID
4     pid->u_min = u_min; // Imposta il limite inferiore output PID
5 }
```

Listing 3.6: "*PID_init*"

La funzione *PID_tune* imposta i parametri di tuning del PID, quali i guadagni proporzionale, integrale e derivativo. Il suo codice è riportato di seguito.

```
1 void PID_tune(pid_controller_t *pid, float Kp, float Ki, float Kd){
2     pid->Kp = Kp; // Imposta il guadagno proporzionale
3     pid->Ki = Ki; // Imposta il guadagno integrale
4     pid->Kd = Kd; // Imposta il guadagno derivativo
5 }
```

Listing 3.7: "*PID_tune*"

3.2.2 PID_compute

La funzione *PID_compute* implementa il controllore PID. Ha il compito di regolare una variabile di processo (PV) da controllare per mantenerla vicino a un valore desiderato, chiamato setpoint (SP). Essa utilizza tre componenti: proporzionale, integrale e derivativo. All'intervallo di campionamento desiderato, calcola l'errore come la differenza tra il setpoint e il valore misurato. Il termine proporzionale (P) aiuta a ridurre rapidamente la differenza. Il termine integrale (I) corregge eventuali errori residui persistenti. Il termine derivativo (D) anticipa cambiamenti rapidi e smorza la risposta del sistema. La somma di questi tre termini produce la variabile di controllo (CV) in uscita dal controllore PID. Successivamente, la funzione verifica se questa uscita supera i limiti predefiniti, adattandola se necessario per evitare sovrasaturazioni. Infine, la funzione restituisce l'uscita regolata, che verrà applicata al sistema controllandolo, mantenendo così la variabile di processo vicina al setpoint in modo quanto più stabile ed efficace possibile. Di seguito, il codice della funzione opportunamente commentato.

```
1  float PID_compute(pid_controller_t *pid, float setpoint, float
       measured){
2
3      // Variabili statiche per mantenere il valore tra le chiamate
           della funzione
4      static float old_error = 0, Iterm = 0;
5
6      // Calcolo dell'errore tra il setpoint e il valore misurato
7      float error = setpoint - measured;
8
9      // Calcolo della componente proporzionale (P)
10     float Pterm = pid->Kp * error;
11
12     // Calcolo della nuova componente integrale (I)
13     // Viene usato old_error per evitare l'accumulo eccessivo di
           errore attuale
14     float newIterm = Iterm + (pid->Ki) * pid->Tc * old_error;
15
16     // Calcolo della componente derivativa (D)
17     // Basata sulla differenza tra l'errore attuale e l'errore
           precedente
18     float Dterm = (pid->Kd/pid->Tc) * (error - old_error);
19
20     // Aggiornmento dell'errore precedente con l'errore attuale per la
            prossima iterazione
21     old_error = error;
22
23     // Somma delle tre componenti per ottenere l'uscita del PID
24     float u = Pterm + newIterm + Dterm;
25
26     // Controllo della saturazione dell'uscita PID
27     // Se l'uscita supera il limite massimo o minimo, viene limitata
```

```
28      if (u > pid->u_max){
29          // Saturazione al limite superiore
30          u = pid->u_max;
31      } else if (u < pid->u_min){
32          // Saturazione al limite inferiore
33          u = pid->u_min;
34      } else {
35          // Se l'uscita non supera i limiti, aggiorna il termine
                integrale
36          // Blocco anti-windup, previene l'accumulo eccessivo nel
                termine integrale
37          Iterm = newIterm;
38      }
39
40      // Restituzione del valore di uscita del PID
41      return u;
42  }
```

Listing 3.8: "*PID_compute*"

Si noti che la struttura del PID scelta è quella in forma parallela (righe 10-24), in quanto si è ritenuto più vantaggioso il disaccoppiamento totale delle azioni proporzionale, integrale e derivata. Inoltre, si è optato per l'utilizzo del metodo di Eulero in avanti per il termine integrale (riga 14) e per la differenza all'indietro per il termine derivativo (riga 18). Infine, si noti che il sistema anti-windup implementato è il clamping (riga 37), e che non è presente nessun filtro derivativo in quanto, come in molte altre applicazioni di controllo industriale, si è deciso di non utilizzare il termine derivativo essendo la risposta con il solo controllo PI già sufficientemente soddisfacente. Ciò rende di conseguenza inutile un filtro su di esso.

3.3 encoder.c

Questo source file definisce le funzioni necessarie all'inizializzazione e allo start dell'encoder, e all'acquisizione degli RPM del motore attraverso l'encoder. A questo scopo, è necessario configurare un timer del microcontrollore in modalità encoder. Il timer scelto è il *TIM2_CH2*, le cui impostazioni sono riportate in Figura 3.5.

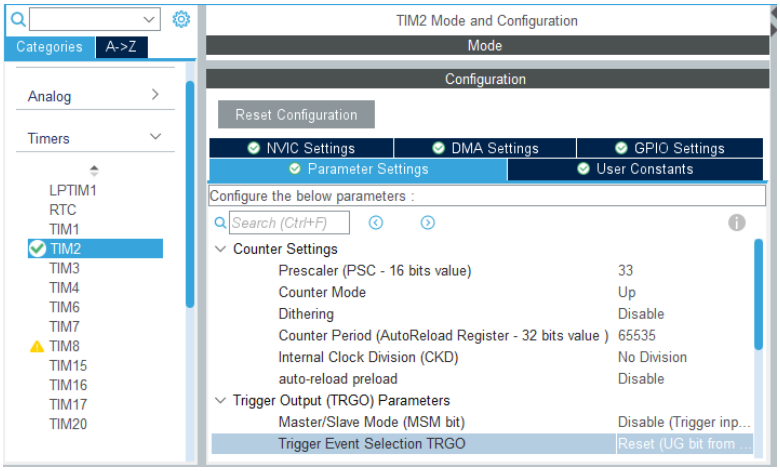


Figure 3.5: *Parametri timer TIM2_CH2*

Il terminale scelto per ricevere i dati in ingresso dall'encoder è il pin A1, visibile in Figura 3.6.

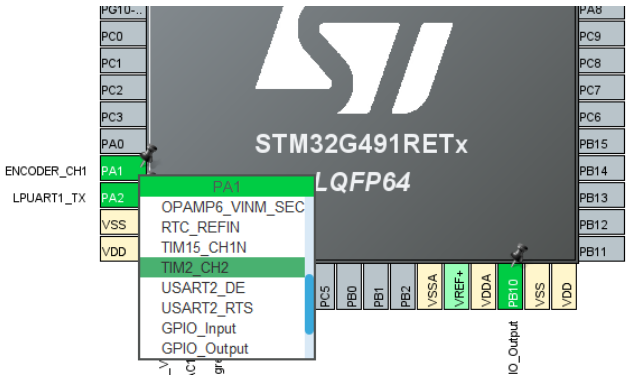


Figure 3.6: *PinA1 TIM2_CH2*

3.3.1 Encoder_init

Questa funzione è essenziale per preparare l'encoder alla lettura dei segnali e per la sua configurazione. Questo processo include l'inizializzazione dei parametri come il numero di passi fisici dell'encoder, il timer utilizzato per leggere i segnali, e i canali di input per i segnali A e B dell'encoder. Viene calcolata anche una soglia di periodo per rilevare se l'encoder sta girando a velocità prossima allo zero RPM. La funzione registra le callback necessarie per gestire le interruzioni generate dal timer associato all'encoder e inizialmente arresta l'encoder per prepararlo all'operazione successiva. Il codice della funzione è riportato di seguito.

```
1 void Encoder_init(encoder_t *enc, uint16_t steps, TIM_HandleTypeDef *
    timer, uint32_t chanA, uint32_t chanB) {
2     // Pulizia della struttura encoder_t
3     memset(enc, 0, sizeof(encoder_t));
```

```
4
5     // Assegnazione dei parametri di inizializzazione all'encoder
6     enc->timer = timer;              // Handle del timer associato all'
          encoder
7     enc->channelA = chanA;           // Canale A di input capture per
          il segnale dell'encoder
8     enc->channelB = -1;              // Canale B, non attualmente
          utilizzato
9     enc->steps = steps;              // Numero di passi fisici dell'
          encoder
10    enc->zero_rpm_period_thrs = ZERO_RPM_THRESHOLD / ((
          __HAL_TIM_GET_AUTORELOAD(timer) + 1) / (float)
          get_timer_counter_frequency(timer));
11                                     // Calcolo della soglia di periodo
                                           per rilevare uno zero RPM
12    enc->rpm = 0;                    // Inizializzazione della velocit
           di rotazione a zero
13
14    // Registrazione dell'encoder per la gestione interna
15    register_encoder(enc);
16
17    // Registrazione delle callback per le interruzioni del timer
18    HAL_TIM_RegisterCallback(timer, HAL_TIM_IC_CAPTURE_CB_ID,
          on_timer_input_capture);
19    HAL_TIM_RegisterCallback(timer, HAL_TIM_PERIOD_ELAPSED_CB_ID,
          on_timer_period_elapsed);
20
21    // Arresto iniziale dell'encoder
22    Encoder_stop(enc);
23 }
```

Listing 3.9: "*Encoder_init*"

3.3.2 Encoder_start e Encoder_stop

La funzione *Encoder_start* prepara l'encoder per iniziare la lettura dei segnali, inizializzando le variabili interne e avviando le operazioni di cattura degli eventi e del timer. Il suo codice è riportato di seguito.

```
1 void Encoder_start(encoder_t *enc) {
2     enc->period = 0;
3     enc->previous.capture = 0;
4     enc->previous.period = 0;
5     enc->last.capture = 0;
6     enc->last.period = 0;
7     enc->rpm = 0;
8
9     HAL_TIM_IC_Start_IT(enc->timer, enc->channelA);
10    HAL_TIM_Base_Start_IT(enc->timer);
11 }
```

Listing 3.10: "*Encoder_start*"

La funzione *Encoder_stop*, invece, ferma tutte le operazioni dell'encoder e reimposta le configurazioni necessarie per una futura inizializzazione. Il suo codice è riportato di seguito.

```
void Encoder_stop(encoder_t *enc) {
    HAL_TIM_Base_Stop_IT(enc->timer);
    HAL_TIM_IC_Stop_IT(enc->timer, enc->channelA);
    __HAL_TIM_SET_COUNTER(enc->timer, 0);
}
```

Listing 3.11: "*Encoder_stop*"

3.3.3 Encoder_rpm

La funzione *Encoder_rpm* calcola e restituisce le rotazioni dell'encoder in RPM, basandosi sui parametri e sulle misurazioni prese durante il suo funzionamento. Il suo ruolo è cruciale per monitorare e regolare dinamicamente la velocità di rotazione del motore, assicurando che il sistema risponda in modo appropriato alle variazioni nelle misurazioni dell'encoder. Il codice della funzione è riportato di seguito.

```
float Encoder_rpm(encoder_t *enc) {
    // Ottienimento della frequenza del contatore del timer associato
        all'encoder
    uint32_t count_freq = get_timer_counter_frequency(enc->timer);

    // Calcolo della frequenza di trigger dell'encoder
    uint32_t trigger_freq = enc->steps >> (__HAL_TIM_GET_ICPRESCALER(
        enc->timer, enc->channelA) >> 2);

    // Salvataggio e disabilitazione temporaneamente degli interrupt
        del timer
    uint32_t backup_ier = enc->timer->Instance->DIER;
    enc->timer->Instance->DIER = 0;

    // Calcolo della differenza in periodi completi
    int period_diff = 0;
    if (enc->previous.period > enc->last.period) {
        period_diff = 0xffffffff - enc->previous.period + enc->last.
            period;
    } else {
        period_diff = enc->last.period - enc->previous.period;
    }

    // Calcolo della differenza nei contatori catturati
    int32_t diff = (__HAL_TIM_GET_AUTORELOAD(enc->timer) + 1) *
        period_diff;
    diff += (int32_t) enc->last.capture - (int32_t) enc->previous.
        capture;

    // Calcolo degli RPM solo se la differenza   diversa da zero
    if (diff != 0) {
```

```
26          enc->rpm = 60.0 * ((float) count_freq / (float) (trigger_freq
                * diff));
27      } else {
28          enc->rpm = 0.0f; // Se diff   zero, gli RPM sono nulli
29      }
30
31      // Verifica se l'encoder si   fermato
32      if (enc->period - enc->last.period > enc->zero_rpm_period_thrs) {
33          enc->rpm = 0.0f; // Se l'encoder   fermo, imposta gli RPM a
                zero
34          // Disabilitazione temporaneamente del callback di scadenza
                del periodo
35          backup_ier &= ~TIM_DIER_UIE;
36      }
37
38      // Ripristino dello stato degli interrupt del timer
39      enc->timer->Instance->DIER = backup_ier;
40
41      // Restituzione del valore calcolato degli RPM
42      return enc->rpm;
43  }
```

Listing 3.12: "*Encoder_rpm*"

Capitolo 4

Test e Risultati

Questo capitolo finale riporta tutta la serie di test effettuati sul banco di prova, che hanno come obiettivo principale la verifica dell'efficacia dell'algoritmo di controllo della frenatura sviluppato e implementato sul microcontrollore. Il passo successivo è quello di regolare i parametri del PID in maniera iterativa, fino ad ottenere dei valori che garantiscano delle performance che soddisfino in primis gli standard dell'azienda *Soft-Engine*. I test, infatti, sono supervisionati dall'azienda e testano man mano funzionalità e prestazioni di loro interesse.

Per ogni prova, viene fornita una descrizione dettagliata dell'oggetto del test, seguita da una tabella che riporta i parametri significativi della singola sperimentazione. In seguito, viene presentato il grafico dei risultati prodotti dal test, registrati tramite lo strumento *Scope* del software *WorkBench*, accompagnato da un'analisi approfondita dei risultati stessi.

In tutti i test, se non differentemente specificato, il motore *Kollmorgen*, che deve simulare un'autovettura, è alimentato tramite un segnale di corrente di 1.000Arms, fornito attraverso il software *WorkBench* come visto nel Capitolo 1.1.2. Se non frenato esternamente, il motore raggiungerebbe così una velocità che si aggira attorno ai 4000RPM. Effettuare test a velocità superiori non è stato ritenuto necessario né opportuno per questioni di sicurezza e stabilità del banco prova.

4.1 Test 1

Il primo test effettuato consiste nell'utilizzo del PID non tanto come sistema di frenatura, quanto come regolatore di velocità, che cioè controlla il motore sia in fase di accelerazione che di decelerazione, stabilizzandolo alla velocità desiderata contrastando eventuali variazioni date da carichi esterni. L'obiettivo è quello di verificare il corretto funzionamento del controllore PID, prima di attivare variazioni e limitazioni sulla sua azione che non renderebbero chiara la fonte di eventuali problematiche. Si è applicato pertanto il metodo trial-and-error per il tuning dei parametri. Si è partiti disattivando tutti i parametri PID tranne il K_p, e si è osservata la risposta impostato un certo setpoint.

Table 4.1: *Test 1a*

setpoint	K_p	K_i	K_d
1000RPM	0.0005	0.0	0.0

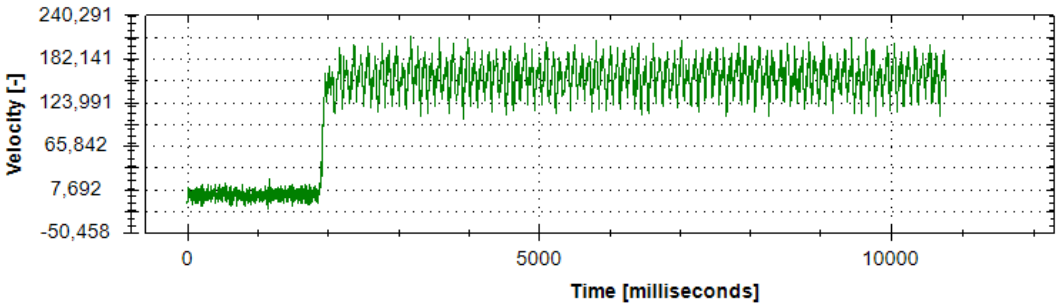


Figure 4.1: *Test 1a*

Si osservi che non solo la risposta oscilla molto, ma si è anche molto distanti dal valore di setpoint. L'offset è altresì molto elevato, e ciò implica che il K_p è troppo basso. Si prova ora pertanto ad aumentare il K_p.

Table 4.2: *Test 1b*

setpoint	K_p	K_i	K_d
1000RPM	0.0015	0.0	0.0

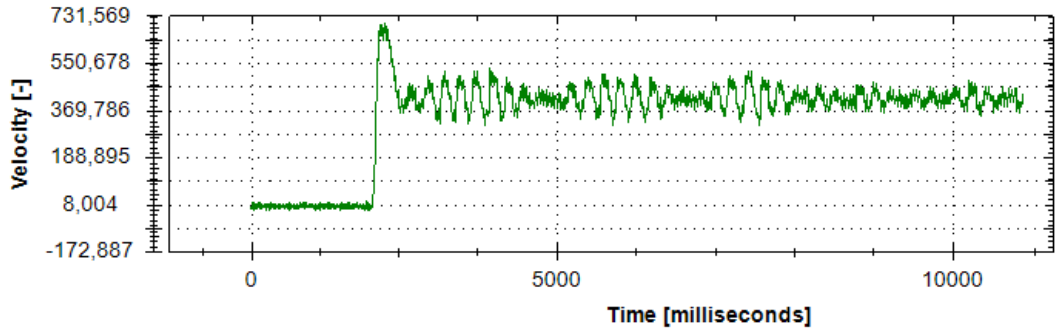


Figure 4.2: *Test 1b*

Si noti che la risposta oscilla molto, ma si è avvicinata molto al setpoint. L'offset, d'altronde, non è mai eliminabile con la sola azione proporzionale. Questo grafico descrive perfettamente la situazione di K_p troppo alto. Pertanto il K_p ideale è compreso tra i due estremi. Nel test che segue si prova a trovare questo valore.

Table 4.3: *Test 1c*

setpoint	K_p	K_i	K_d
1000RPM	0.001	0.0	0.0

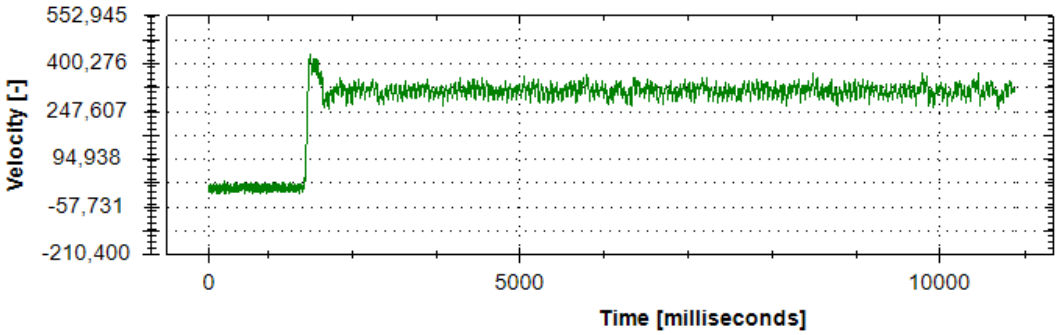


Figure 4.3: *Test 1c*

La risposta del sistema è ottima. Pertando il K_p ideale, a queste condizioni, è 0.001. Continuando l'applicazione del trial-and-error, si aggiunge ora il guadagno integrale K_i.

Table 4.4: *Test 1d*

setpoint	K_p	K_i	K_d
1200RPM	0.001	0.0005	0.0

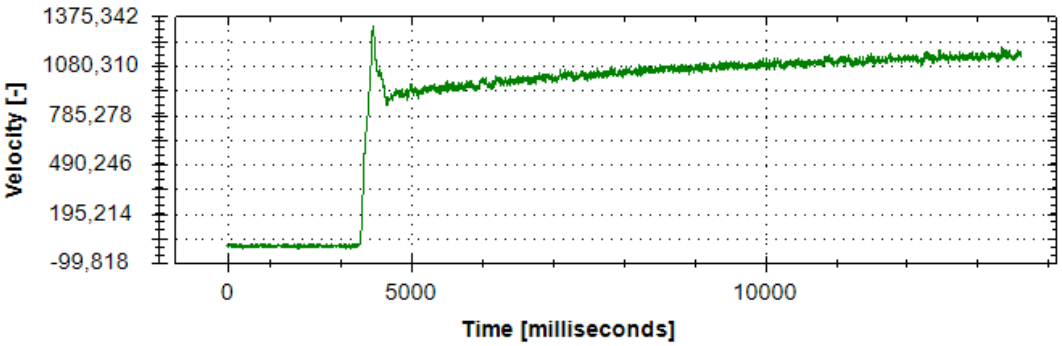


Figure 4.4: *Test 1d*

Questo valore di K_i è troppo basso. Si prova perciò ad aumentarlo.

Table 4.5: *Test 1e*

setpoint	K_p	K_i	K_d
1200RPM	0.001	0.0012	0.0

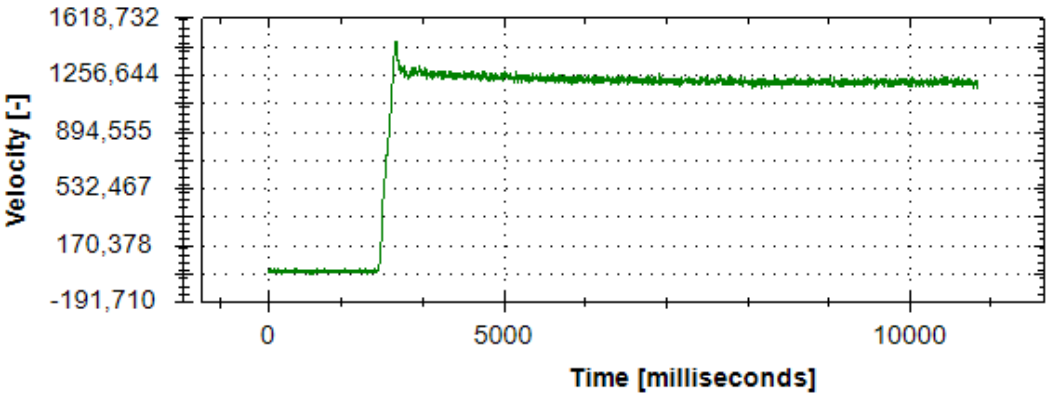


Figure 4.5: *Test 1e*

La risposta è soddisfacente. Trattandosi questo di un test preliminare, si è deciso di fermarsi a questi risultati e procedere ad uno step successivo. Si noti che, prevedendo l'alta probabilità di non applicare la componente derivativa, si è rimandata la sperimentazione di tale parametro.

4.2 Test 2

Terminate le sperimentazioni sul controllo in velocità, si è apportata una modifica software che attiva il controllo PID solamente a valori di errore positivo, e cioè, solo quando ci si trova sotto la soglia del setpoint. Questa è l'effettiva applicazione del sistema di controllo che interessa all'azienda, ovvero un'azione di sola frenatura del motore. Il PID entrerà pertanto in funzione una volta che il motore supererà il setpoint impostato. Fino a quel momento, accelererà privo da carichi esterni. Da qui a seguire, questa modifica software sarà sempre attiva.

Come base di partenza per testare il nostro sistema di frenatura PID, sono i parametri fino ad ora raccolti.

Table 4.6: *Test 2a*

setpoint	K_p	K_i	K_d
1200RPM	0.001	0.0012	0.0

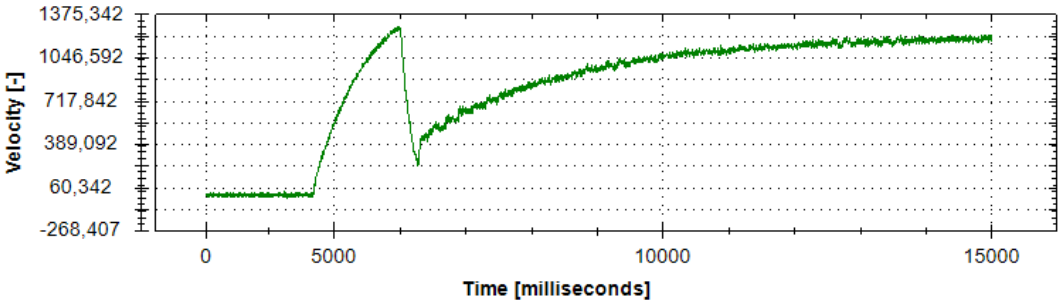


Figure 4.6: *Test 2a*

Le osservazioni da fare sono due. La prima, fondamentale, è che all'attivazione del controllo PID, e cioè superato il setpoint, la velocità del motore subisce un brusco picco verso il basso, per poi da lì risalire. Dopo innumerevoli sperimentazioni e tentativi, non riuscendo a correggere questa problematica, in accordo con l'azienda si è accettato di convivere con questo difetto hardware per tutto il proseguo dei test. Ciò è infatti dovuto all'iniziale innesto del motore tramite inverter, a cui, come detto, non si è riusciti a trovare soluzione. D'altro canto, il problema si verifica sollo alla prima attivazione del controllo PID, e, pertanto, non inficia la bontà dei test.

La seconda osservazione è che i parametri del PID precedentemente scelti, validi in quelle condizioni, non producono una risposta altrettanto ottimale in questa situazione. Si è proseguito, pertanto, a ricalcolare tali parametri. Nello specifico, si è osservato che il problema risiedesse nel guadagno integrale e, perciò, si è partiti modificando quello.

Table 4.7: *Test 2b*

setpoint	K_p	K_i	K_d
1200RPM	0.001	0.015	0.0

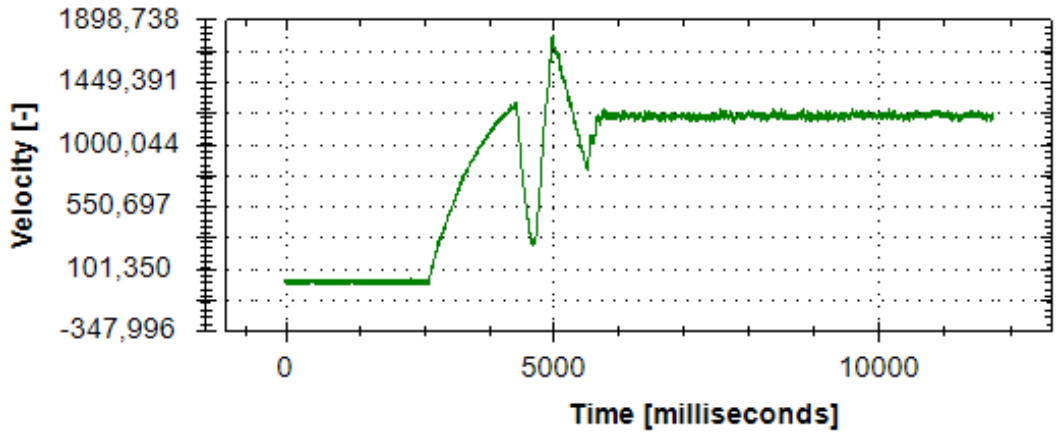


Figure 4.7: *Test 2b*

Questo valore del K_i è invece troppo alto, generando un picco in alto che supera di molto il setpoint. Il K_i ideale è perciò compreso in questo range, e si prova ad individuare questo valore nel seguente test.

Table 4.8: *Test 2c*

setpoint	K_p	K_i	K_d
1200RPM	0.001	0.01	0.0

Figure 4.8: *Test 2c*

La risposta del sistema è ottima. Come pronosticato, l'aggiunta della componente derivativa sembra del tutto superflua, considerando che i vantaggi apportati non giustificherebbero gli svantaggi.

4.3 Test 3

Si effettua ora lo stesso test, mantenendo invariati i valori K del PID, ma alzando solamente il setpoint.

Table 4.9: *Test 3a*

setpoint	K_p	K_i	K_d
3000RPM	0.001	0.01	0.0

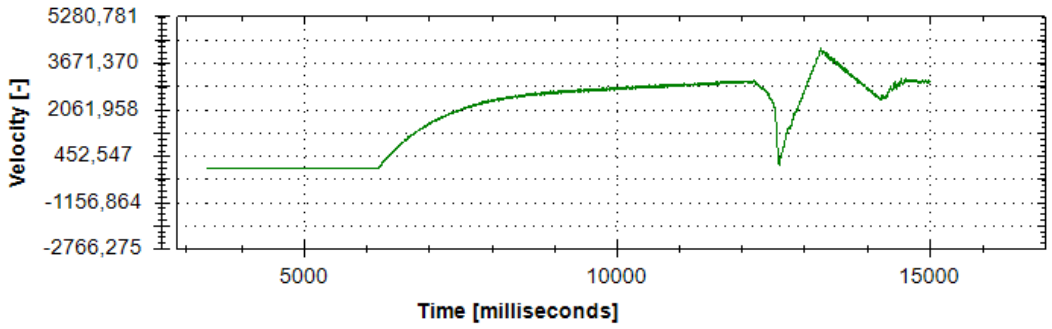


Figure 4.9: *Test 3a*

La risposta non è più soddisfacente, in quanto il K_i risulta essere troppo grande. Pertanto, si procede a variare per tentativi il coefficiente fino a ottenere una risposta che si reputi adeguata.

Table 4.10: *Test 3b*

setpoint	K_p	K_i	K_d
3000RPM	0.001	0.007	0.0

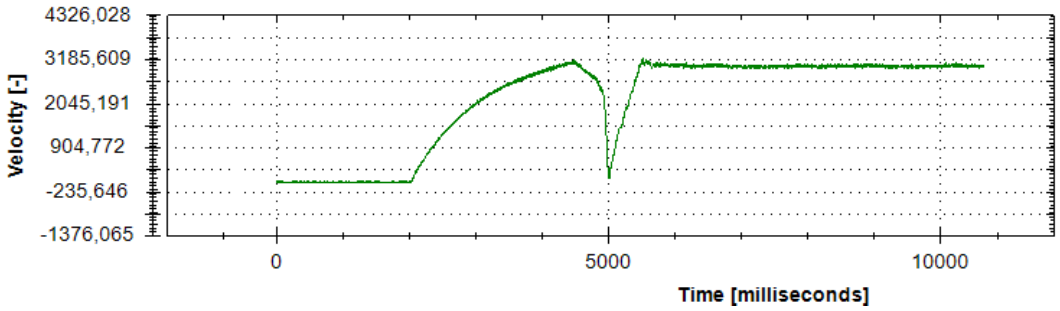


Figure 4.10: *Test 3b*

Per questi valori, la risposta del sistema è adeguata. Un'importante osservazione che ne consegue è quindi che, a differenti valori della velocità, gli stessi parametri PID non garantiscono un controllo ugualmente ottimale. Pertanto, si è in presenza di un sistema non lineare. Per risolvere la non-linearità, è necessario applicare la tecnica del gain scheduling, individuando il corretto valore di K_i al variare del range di velocità. Prima di procedere in questo senso, però, vengono fatti altri test per approfondire meglio il comportamento del sistema.

4.4 Test 4

Finora, abbiamo effettuato le prove mantenendo sempre costante il setpoint una volta impostato dal principio. La necessità di *Soft-Engine*, però, è quella di cambiare

setpoint in corso di svolgimento dei test sulle vetture. Pertanto, è necessario studiare come si comporta il sistema in risposta alla variazione del setpoint, nello specifico in questo test all'aumento graduale del setpoint. Il valore di riferimento viene quindi impostato ad un valore iniziale, per essere poi gradualmente modificato con uno scalino costante a sistema in evoluzione.

Table 4.11: *Test 4a*

setpoint	Δ	K_p	K_i	K_d
1000RPM	+500RPM	0.001	0.01	0.0

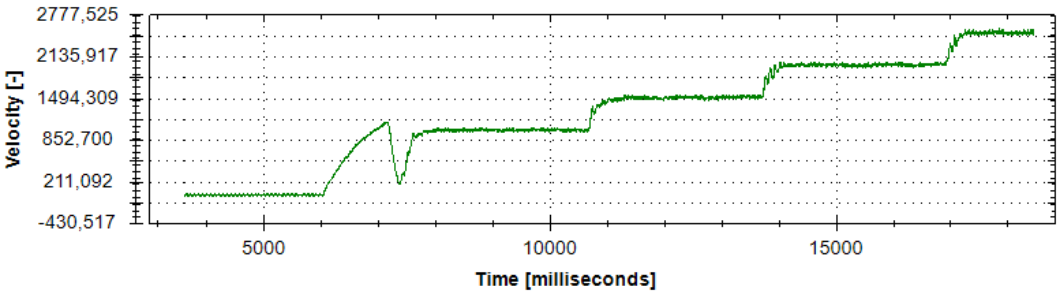


Figure 4.11: *Test 4a*

Il grafico mostra un'ottima risposta da parte del controllore PID. Si noti però che, a diverse velocità, la risposta del PID è la medesima, nonostante non sia stato ancora effettuato il gain scheduling.

Si procede quindi ad effettuare lo stesso test, ma con decremento dal setpoint iniziale anziché incremento.

Table 4.12: *Test 4b*

setpoint	Δ	K_p	K_i	K_d
3000RPM	-500RPM	0.001	0.01	0.0

Figure 4.12: *Test 4b*

Si osservi come, all'attivazione del PID a 3000 rpm, la risposta non è ottima, riconfermando quanto visto nel Test 3, ma in contraddizione con quanto osservato nel test di cui sopra. Questo fa pensare che non sia tanto la velocità a cui si opera il problema, quanto la variazione tra la velocità corrente e il setpoint desiderato. È necessario pertanto eseguire altri test per individuare quale sia la variabile più appropriata per l'applicazione dello scheduling.

4.5 Test 5

Per approfondire la designazione della variabile di scheduling, si effettua un test per verificare come il controllore PID risponde ad una variazione del setpoint molto brusca. Pertanto, si applica un delta molto grande tra i due setpoint.

Table 4.13: *Test 5a*

setpoint	Δ	K_p	K_i	K_d
500RPM	+3000RPM	0.001	0.01	0.0

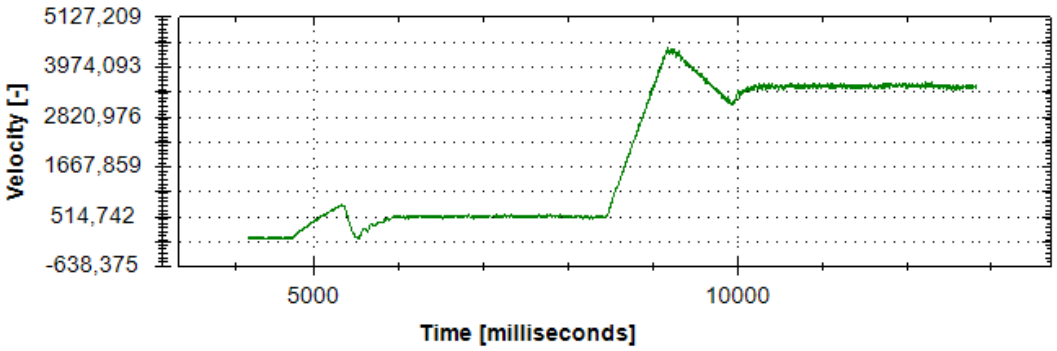


Figure 4.13: *Test 5a*

Come facilmente ipotizzabile, con questo valore di K_i la risposta non è ottimale. Si prova quindi ad utilizzare altri parametri.

Table 4.14: *Test 5b*

setpoint	Δ	K_p	K_i	K_d
500RPM	+3000RPM	0.001	0.005	0.0

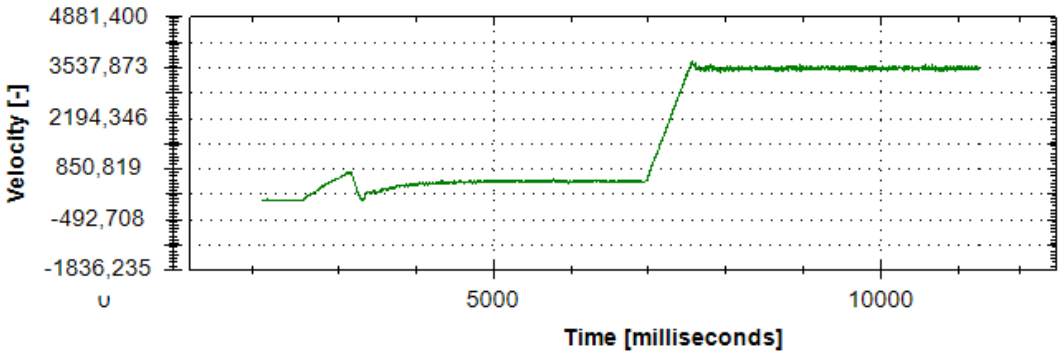


Figure 4.14: *Test 5b*

Per questi valori, la risposta del sistema è soddisfacente. Perciò, per grandi variazioni del setpoint, l'azione di controllo deve essere più morbida con un K_i più basso. Si può pertanto concludere che la variabile di scheduling migliore da utilizzare non è tanto la velocità di rotazione del motore, quanto la differenza tra la velocità attuale e il setpoint. Questa variabile non è altro che l'errore su cui già si basa l'intero calcolo della variabile di controllo tramite PID. Sembrerebbe, inoltre, che l'unico parametro che deve essere modificato tra un delta e l'altro è il K_i.

4.6 Test 6

In questa serie finale di test, si intende applicare il metodo di gain scheduling. Individuata nell'errore tra velocità corrente e setpoint la variabile di scheduling, è ora necessario definire le fasce per le quali utilizzare parametri PID differenti. Inoltre, per ogni fascia, è necessario calcolare i valori dei suddetti parametri.

Per far ciò, si inizia facendo cambiare il setpoint in maniera non progressiva, aumentando e diminuendo il suo valore con delta più o meno grandi. Il tutto, inizialmente, senza creare uno scheduling ma utilizzando un set di valori fissi per i parametri del PID.

Table 4.15: *Test 6a*

salto	setpoint	K_p	K_i	K_d
1	500RPM	0.001	0.01	0
2	3500RPM	0.001	0.01	0
3	700RPM	0.001	0.01	0
4	2700RPM	0.001	0.01	0
5	1000RPM	0.001	0.01	0
6	2500RPM	0.001	0.01	0
7	500RPM	0.001	0.01	0

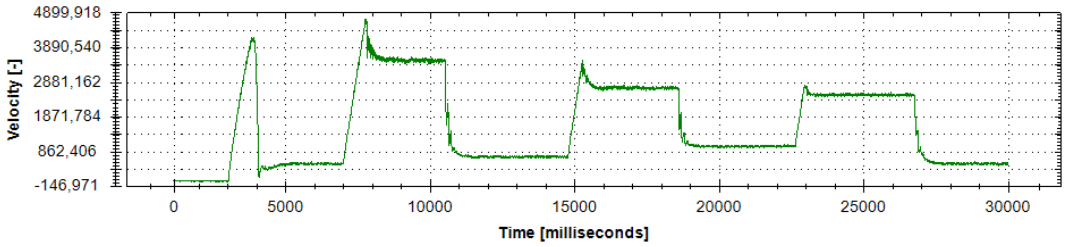


Figure 4.15: *Test 6a*

Oltre ad aver confermato l'ipotesi per cui mantenere un K_i fisso non garantisce la stessa risposta per differenti delta del setpoint, questo test è utile per individuare già in maniera abbastanza esaustiva le fasce del gain schedulig. Infatti, a diversi l'overshoot corrispondono diverse fasce. In questo grafico si può infatti notare come i tre overshoot in salita siano distinti tra loro. Questo ha portato all'idea di creare tre fasce per la nostra tabella di scheduling. Nel test che segue, mantenendo invariati i setpoint, si procede a definire le fasce per la prima volta.

Table 4.16: *Test 6b*

fascia	K_p	K_i	K_d
$0 < e < 1000$	0.001	0.01	0
$1000 < e < 2000$	0.001	0.007	0
$2000 < e < 4000$	0.001	0.003	0

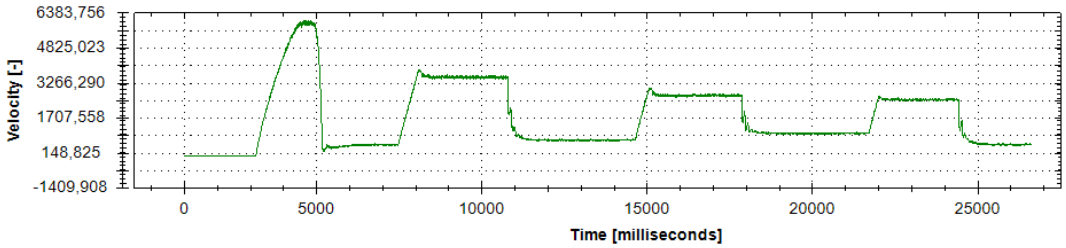


Figure 4.16: *Test 6b*

Si noti come gli overshoot siano notevolmente diminuiti, avendo già ottenuto una buona risposta complessiva. Si effettuano altri test per ottimizzare ancora di più il controllo.

Table 4.17: *Test 6c*

fascia	K_p	K_i	K_d
$0 < e < 1000$	0.001	0.01	0
$1000 < e < 2000$	0.001	0.005	0
$2000 < e < 4000$	0.001	0.003	0

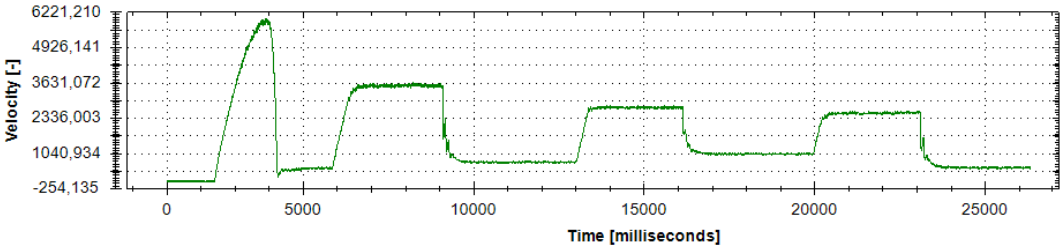


Figure 4.17: *Test 6c*

Per questi valori, la risposta del sistema è eccellente. Non sono infatti più presenti overshoot significativi.

Nonostante il lavoro di ottimizzazione possa proseguire, ad esempio aumentando il numero di fasce, in accordo con l'azienda si è deciso di sospendere qui le sperimentazioni perché ritenute già soddisfacenti. Infatti, se da un lato i test fino ad ora condotti sono la prova che questo sistema di controllo PID è efficiente in questo tipo di situazioni piuttosto fedeli alla realtà, dall'altro si sta parlando pur sempre di simulazioni. Pertanto, progredire ulteriormente con l'ottimizzazione non apporterebbe vantaggi significativi. Cambiando hardware e passando ai banchi prova potenza prodotti dall'azienda, infatti, tutto il lavoro di tuning e scheduling andrà rifatto da zero. Questi test e più in generale il presente elaborato, tuttavia, rappresentano una solida base di partenza per questa operazione.

Conclusioni e Sviluppi Futuri

Il proposito di questo lavoro di tesi era quello di regolare e controllare un sistema di frenatura per test di motori di autotrazione su banchi prova potenza, progettati dall'azienda *Soft-Engine*.

Come primo passo, è stato necessario progettare un banco prova in scala che simulasse in maniera più accurata possibile quelli che sono i banchi prova potenza prodotti dall'azienda stessa. Questo ha implicato la creazione di un ambiente di test controllato e sicuro, capace di replicare le condizioni reali di funzionamento dei motori, affidandosi all'esperienza pluriennale di *Soft-Engine* nel settore per coordinarne l'allestimento.

Parallelamente, si è condotto uno studio mirato sulle varie metodologie di controllo disponibili per sistemi di frenatura, optando per approfondire il funzionamento e l'applicazione del controllo PID (Proporzionale-Integrale-Derivativo). Questa scelta è stata motivata dalla sua comprovata efficacia nel mantenimento della stabilità e nella precisione di controllo necessarie per i test sui motori di autotrazione. Le alternative vagliate, di contro, non sono state ritenute adatte a questo genere di processo.

Si è quindi proceduto all'implementazione del suddetto algoritmo di controllo sul microcontrollore preposto alla gestione dell'intera logica di frenatura. Fin dal principio, sono state utilizzate le nozioni acquisite durante la fase di ricerca per implementare un controllo che vantasse funzionalità avanzate quali l'anti-windup e il gain scheduling. Tali caratteristiche, sono state successivamente oggetto dei test sul banco prova per validarne l'efficacia.

In particolare, l'introduzione del gain scheduling si è dimostrata cruciale per adattare dinamicamente i parametri del PID e far fronte alla non linearità del sistema. Questa tecnica ha contribuito significativamente a migliorare la risposta del sistema in situazioni di variazione rapida dei requisiti operativi, e rappresenta uno dei risultati più rilevanti di questo elaborato.

Tuttavia, lo svolgimento dei test ha evidenziato dei limiti hardware del banco prova che, nonostante non abbiano compromesso i risultati ottenuti, hanno limitato le possibilità di sperimentazione. Tra questi, il più significativo è un netto calo di velocità all'accensione del motore tramite inverter, che nonostante i numerosi tentativi non è stato possibile risolvere. Pertanto, uno dei primi sviluppi possibili per questo progetto potrebbe essere il miglioramento dell'hardware che costituisce il banco di prova, sostituendo alcune componenti, quali ad esempio l'inverter, con versioni più all'avanguardia e prestanti.

In alternativa, avendo l'algoritmo risposto positivamente ai vari test, sarebbe possibile attuare il controllo direttamente sui banchi prova potenza costruiti dall'azienda. Per far ciò, sarebbe solamente necessario riscrivere la parte di codice relativa alla gestione dell'output della variabile di controllo, in quanto i banchi prova reali utilizzano freni a correnti parassite come sistema di frenatura. Questi, richiedono un segnale PWM, che il microcontrollore dovrebbe generare in sostituzione all'uscita

DAC. La scheda predispone già di tutta l'elettronica necessaria per questa applicazione, pertanto l'adattamento sarebbe relativamente semplice e non richiederebbe modifiche strutturali significative. Ciò nonostante, un banco prova simulativo sarebbe comunque essenziale per la maggiore praticità e sicurezza di svolgimento dei test, perciò questa non potrebbe comunque essere l'unica soluzione intrapresa.

Lato software, invece, i possibili sviluppi futuri sono molteplici. In primo luogo, sarebbe possibile aumentare ulteriormente il numero delle fasce del gain scheduling per rendere il controllo ancora più adattivo. Tuttavia, in questa fase del progetto, questa miglioria risulterebbe superflua in quanto cambiando hardware tutto il lavoro di tuning e scheduling sarebbe da impostare nuovamente dal principio. Avrebbe perciò senso farlo qualora ci si trovasse a lavorare con i banchi prova definitivi, e non quelli simulativi.

Altre possibili aree di ricerca potrebbero essere quella dell'aggiunta della componente derivativa, che come abbiamo visto è disattivata poiché le complicazioni superano i benefici, oppure quella dell'implementazione di tecniche più avanzate come il controllo predittivo. Ciò nonostante, i risultati maturati possono ritenersi già soddisfacenti per quella che è l'applicazione richiesta.

La tesi viene quindi conclusa con la consapevolezza di aver portato a termine l'obiettivo inizialmente prefissato, e con la speranza che le conclusioni raggiunte possano costituire una base solida per i successivi sviluppi del progetto e offrire validi spunti per future ricerche in questo ambito.

Bibliografia

[1] Maria Cristina Giannini. Prototipazione di un dispositivo embedded per il rilevamento guasti tramite mcsa in condizioni non stazionarie su dispositivi azionati da motori elettrici. Master's thesis, UNIVPM, 2020.

[2] Liuping Wang. *PID Control System Design and Automatic Tuning using MATLAB/Simulink*, volume 1. IEEE Press, 2019.

[3] Tore Hägglund. *Process Control in Practice*, volume 1. De Gruyter, 2023.