Università di Roma Sapienza

Facoltà di Scienze Matematiche Fisiche e Naturali

Dipartimento di Informatica

Emanuele Cimò

Remote Controlled Balancing Robot

Relatore: Prof. Andrea Sterbini

gnegativ@gmail.com

Indice

Introduzione

CAPITOLO 1

Principi fisici

In questo capitolo verranno brevemente descritti, a livello teorico, i principi fisici che sono alla base di questo progetto di Tesi, così come di alcuni dispositivi utilizzati per la sua realizzazione.

In particolare, verrà descritto il modello del pendolo inverso e le equazioni fondamentali che descrivono il giroscopio e l'accelerometro. Inoltre, verrà riportata una breve descrizione della realizzazione pratica di questi strumenti di misura.

1.1 Il pendolo inverso

Un *pendolo semplice* è costituito da un filo inestensibile a cui è appeso un punto materiale di massa m, libero di oscillare attorno a un punto fisso detto polo; la componente della forza peso lungo il filo contro bilancia la tensione del filo stesso, mentre la componente della forza peso perpendicolare al filo è la forza di richiamo che produce il moto oscillatorio del pendolo.

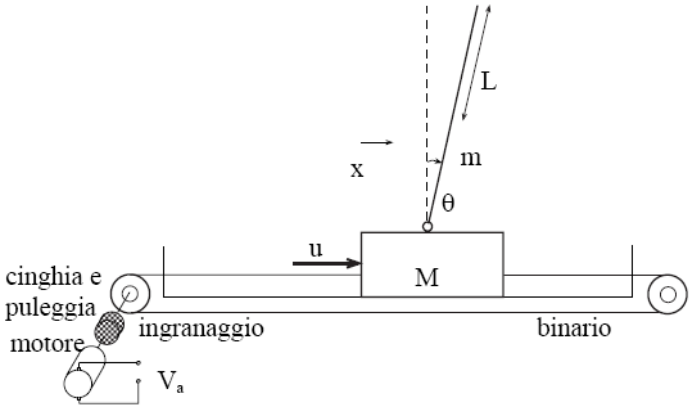

Figura 1.1: Schematizzazione del sistema pendolo inverso.

Il *pendolo inverso* è un pendolo semplice rovesciato, rigido e privo di punto fisso, esso è costituito da un carrellino che trasporta un'asta rigida libera di ruotare intorno

ad un giunto. La parte più bassa, quindi il carrello, può muoversi per bilanciare le oscillazioni della parte più alta e garantire così l'equilibrio. Una schematizzazione del sistema Pendolo inverso è riportata in Figura 1.1.

Il comportamento dell'insieme asta–carrello può essere descritto da un modello dinamico, le cui variabili di stato sono la posizione e la velocità di entrambi i componenti: l'asta e il carrello.

L'unico ingresso è costituito dalla spinta orizzontale, u, applicata al carrello tramite il motore, mentre l'uscita è la posizione angolare dell'asta.

Le forze che agiscono verticalmente vengono compensate dalla reazione del terreno, mentre le forze che agiscono orizzontalmente sono date dall'equazione:

$$u = M\ddot{y} + ml\ddot{\theta} \tag{1.1}$$

dove M è la massa del carrello, m è la massa dell'asta ed l è la sua lunghezza. Nell'equazione 1.1, dove abbiamo assunto $M \gg m$ ed $lsin\theta \simeq l\theta$, i due termini a secondo membro sono dovuti alle inerzie delle due masse.

La somma dei momenti intorno al giunto è data dalla relazione:

$$0 = ml^2\ddot{\theta} + m\ddot{y}l - mgl\theta \tag{1.2}$$

dove il primo è il momento d'inerzia della massa m, il secondo è il momento generato dalla forza applicata u, ed il terzo dalla forza peso.

Assumendo come variabile di stato: $\overline{x} = [x_1, x_2, x_3, x_4]$ tale che $x_1 = y$, $x_2 = \dot{y}$, $x_3 = \theta$ e $x_4 = \dot{\theta}$, sostituendo nelle relazioni 1.1 e 1.2, si ottiene,per il pendolo inverso, il seguente sistema di equazioni di stato:

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= -\tfrac{mg}{M-m}x_3 + \tfrac{M}{M-m}u \\ \dot{x}_3 &= x_4 \\ \dot{x}_4 &= \tfrac{Mg}{l(M-m)}x_3 + \tfrac{M}{l((M-m)}u \end{aligned} \tag{1.3}$$

Il sistema descritto nell'equazione 1.3 è dunque un sistema non lineare autonomo del tipo:

$$\ddot{x}(t) = f[x(t), u(t)] \tag{1.4}$$

Il punto $\overline{x}_0 = [0, 0, 0, 0]$ è un punto di equilibrio per il sistema 1.3, in corrispondenza di un ingresso identicamente nullo u(t)=0, ed è proprio questo il punto di lavoro in cui si cerca di far funzionare il sistema.

Come è noto, tale sistema costituisce un importante benchmark per il test di metodologie di controllo. Il problema di controllo si riconduce dunque, a volere stabilizzare, nel punto di equilibrio alto, la posizione di un'asta vincolata ad un carrello libero di traslare lungo una guida, agendo sui motori che controllano lo stesso.

1.2 Il giroscopio

Il giroscopio è un qualunque sistema fisico dotato di una simmetria di rotazione intorno a un asse. Con il termine di giroscopio si è soliti indicare un corpo montato su sospensione cardanica in modo da poter ruotare in qualunque direzione.

Le caratteristiche fondamentali di un sistema di questo tipo sono due. La prima è l'elevata inerzia, ovvero la permanenza dell'asse di rotazione dovuta alla conservazione del momento angolare. In altre parole se un giroscopio è installato su una sospensione cardanica che permette alla ruota di orientarsi liberamente nelle tre direzioni dello spazio, il suo asse si manterrà orientato nella stessa direzione anche se il supporto cambia orientamento.

La seconda è la precessione, ovvero la tendenza dell'asse di rotazione a disporsi ad angolo retto rispetto al piano individuato dall'asse stesso e da una qualsiasi forza ad esso applicata (come ad esempio la forza peso), e che consiste sostanzialmente in un lento moto conico dell'asse.

Queste due proprietà sono comuni a qualunque corpo in rotazione intorno a un asse di simmetria, compresa la Terra.

Prima di riportare l'equazione fondamentale che descrive un qualunque giroscopio, accenneremo brevemente ad alcune caratteristiche proprie del moto di rotazione di un corpo rigido.

Nella trattazione del moto dei corpi rigidi si può andare dal caso più semplice che è quello di moto vincolato, in cui l'asse di rotazione è fisso, al moto non vincolato attorno ad un asse coincidente con un asse centrale d'inerzia, al caso di un corpo che ruota attorno ad un asse centrale d'inerzia, ma è soggetto ad un momento di forza (e l'asse mostra un moto di precessione attorno ad una direzione fissa) fino al caso più generale in cui il corpo è posto inizialmente in rotazione attorno ad un asse diverso da un asse centrale d'inerzia, ed il moto si svolge sotto l'azione di un sistema di forze qualunque.

Nei casi più generali la soluzione delle equazioni del moto può risultare molto complicata, per il nostro scopo sarà sufficiente la descrizione di casi particolari nei quali la trattazione risulta semplificata.

1.2.1 La conservazione del momento angolare

Per un corpo rigido, non puntiforme, il momento angolare totale è la somma dei singoli momenti angolari che, di norma, non sono paralleli all'asse di rotazione: di conseguenza in generale anche il momento angolare totale non è parallelo all'asse di rotazione.

Ad ogni modo, per ogni corpo esiste un'asse di rotazione tale che il momento angolare totale risulta parallelo a tale asse. In particolare, qualsiasi sia la forma del corpo, dato un asse di rotazione è possibile trovare un Sistema di Riferimento (quindi tre direzioni, fra di loro perpendicolari) in cui il momento angolare totale del corpo risulta parallelo a tale asse.

Gli assi di tale Sistema di Riferimento sono chiamati *assi principali di inerzia* ed i corrispondenti momenti di inerzia sono detti *momenti principali di inerzia*. In pratica, gli assi principali di inerzia costituiscono un sistema di riferimento solidale col corpo ed, in generale, ruotano rispetto all'osservatore. Nel caso in cui il corpo sia dotato di qualche tipo di simmetria, gli assi principali di inerzia coincidono con questi assi di simmetria.

Nel caso in cui il corpo ruoti attorno ad un asse principale di inerzia, il momento angolare totale è parallelo alla *velocità angolare*, ω, e può essere descritto dalla seguente relazione vettoriale:

$$\overrightarrow{L} = I\overrightarrow{\omega} \tag{1.5}$$

dove I è il *momento d'inerzia* del corpo.

Nel caso più generale, in cui il corpo rigido ruota intorno ad un asse qualsiasi, il momento angolare totale può essere sempre espresso rispetto agli assi principali di inerzia, secondo la relazione:

$$\overrightarrow{L} = I_1\omega_x\widehat{u_x} + I_2\omega_y\widehat{u_y} + I_3\omega_z\widehat{u_z} \tag{1.6}$$

dove $(\widehat{u_x}, \widehat{u_y}, \widehat{u_z})$ sono i versori degli assi principali d'inerzia, che ovviamente non sono fissi nello spazio, ma ruotano con il corpo stesso, essendo ad esso solidali, e $(\omega_x, \omega_y, \omega_z)$ sono le componenti della velocità angolare lungo tali versori.

$\overrightarrow{L}$ ed $\overrightarrow{\omega}$ hanno direzioni diverse, ma I_1, I_2 e I_3, essendo riferite agli assi principali d'inerzia, sono quantità fisse che possono essere calcolate per ogni corpo.

L'equazione fondamentale che descrive un qualunque giroscopio, che ruota intorno ad un suo asse principale di inerzia, è:

$$\tau = \frac{d\overrightarrow{L}}{dt} = \frac{d(I\overrightarrow{\omega})}{dt} = I\frac{d\omega}{dt} \tag{1.7}$$

dove τ è il momento torcente e $d\omega/dt$ è l'accelerazione angolare

Dall'equazione 1.3 deriva che applicando un momento torcente τ perpendicolarmente all'asse di rotazione, quindi perpendicolare ad $\overrightarrow{L}$, si sviluppa una forza perpendicolare sia a τ sia ad $\overrightarrow{L}$. Il moto che ne deriva è detto *precessione* e la sua velocità angolare ω_P è data dalla relazione:

$$\tau = \overrightarrow{\omega_P} \times \overrightarrow{L} \tag{1.8}$$

In generale, un giroscopio è costituito da una ruota libera di ruotare attorno ad un asse, passante per un punto fisso, posto a distanza h dal centro di massa della ruota. L'asse di rotazione coincide con uno degli assi principali d'inerzia del giroscopio stesso.

Il giroscopio ha simmetria cilindrica rispetto ad un'asse scelto. Supponendo, senza perdere in generalità che tale asse di simmetria sia l'asse z, i momenti d'inerzia rispetto agli assi x ed y coincidono mentre quello rispetto all'asse z è diverso. I_z viene detto *momento assiale*, mentre $I_x = I_y = I_{xy}$ viene detto *momento equatoriale*.

1.2.2 Realizzazioni pratiche

Nella pratica le tipologie di giroscopio possono essere diverse.

Di seguito, per ognuna, viene riportata una breve descrizione del principio di funzionamento.

Giroscopio meccanico

Il più classico dei giroscopi è quello a funzionamento meccanico. In questo tipo di giroscopio una massa metallica rotante, sfruttando l'effetto giroscopico, rivela la rotazione del corpo lungo il suo asse da stabilizzare.

Più in dettaglio, un giroscopio meccanico è costituito da due masse rotanti montate agli estremi dell'albero di un motore elettrico; il motore è vincolato ad un perno e può girare su un asse perpendicolare al proprio, essendo mantenuto in posizione centrale da due molle. Quando il corpo ruota il sistema motore–masse rotanti si inclina vincendo la forza di reazione delle molle.

Sotto al motore è posta un sensore ad *effetto hall* che comunica all'elettronica dell'apparecchio l'inclinazione del motore stesso. Tale inclinazione è proporzionale alla velocità con la quale il corpo sta ruotando.

Giroscopio piezoelettrico

Un'altra categorie è quella dei giroscopi piezoelettrici. Il loro funzionamento è simile a quello dei giroscopi meccanici, il vantaggio è che la velocità di rotazione è letta da un sensore piezoelettrico allo stato solido.

Il sensore è costituito da tre lamine di ceramica piezoelettrica montate su un prisma a base triangolare equilatera. Una di queste lamine viene fatta vibrare ad una frequenza prestabilita. Se il prisma è in quiete (non ruota) le altre due lamine ricevono la stessa vibrazione, generando quindi due segnali uguali ed opposti che si annullano. Se il prisma ruota le due lamine che fungono da sensori ricevono la vibrazione in modo diverso e la somma dei segnali generati dalle due lamine genera, a sua volta, un segnale positivo o negativo a seconda del verso di rotazione.

L'elettronica del giroscopio elabora poi il segnale ricevuto dal prisma come nel caso del giroscopio meccanico.

I principali vantaggi dei giroscopi piezoelettrici sono la loro sensibilita' e velocita' di risposta, di molto superiore a quella dei giroscopi meccanici, tuttavia i giroscopi piezoelettrici sono sensibili a variazioni di temperatura che provocano uno spostamento del loro punto neutro, fenomeno conosciuto come *drift*.

Giroscopio siliconico

Il giroscopio siliconico, così come quello piezoelettrico, basa il suo principio di funzionamento sulla risonanza.

Due strutture polisiliconiche sono innestate su una base fissata al circuito. Le particelle di silicone sono mosse da una corrente elettrostatica che induce una vibrazione su di esse. Quando la base ruota intorno al suo asse le particelle si spostano in maniera proporzionale al grado di rotazione alla quale sono sottoposte. Questo spostamento è misurato utilizzando una struttura capacitiva che è in grado di produrre una differenza di potenziale proporzionale allo spostamento.

Come i giroscopi piezoelettrici, anche quelli siliconici sono sensibili a variazioni di temperatura, e quindi soggetti al fenomeno del drift.

1.3 L'accelerometro

L'accelerometro è uno strumento di misura in grado di rivelare l'accelerazione di un corpo.

Il principio di funzionamento si basa sulla capacità dello strumento di misurare l'inerzia di un corpo, di massa nota, quando questo viene sottoposto ad un'accelerazione. Solitamente, la massa è sospesa attraverso un elemento elastico alla struttura del dispositivo. In seguito ad una accelerazione la massa tenderà a spostarsi dalla posizione di riposo di una quantità che risulta proporzionale all'accelerazione subita.

Misurando lo spostamento del corpo è quindi possibile risalire al valore della sua accelerazione.

Vediamo di seguito, alcuni concetti utili per capirne il funzionamento.

1.3.1 Inerzia di un corpo

In prima approssimazione, possiamo definire l'inerzia di un corpo come la 'capacità' dello stesso di opporsi a variazioni del suo stato di moto.

Il *principio di inerzia* (primo principio della dinamica) stabilisce che un corpo di massa m, permane nel suo stato di quiete o di moto rettilineo uniforme a meno che non intervenga una forza, F, esterna a modificare tale stato.

In tal caso il corpo verrà accelerato con un accelerazione direttamente proporzionale alla forza applicata, secondo la nota legge:

$$F = m\ddot{x} \tag{1.9}$$

dove il coefficiente di proporzionalità è proprio la massa, m, del corpo, mentre $\ddot{x}$, che indica la derivata seconda rispetto al tempo della sua posizione, è ovviamente l'accelerazione dello stesso.

Per lo scopo di questo lavoro di Tesi è anche utile ricordare, brevemente, la definizione di *momento di inerzia* di un corpo ed alcune considerazioni ad esso legate.

Dato un punto materiale di massa m, il suo momento di inerzia rispetto ad un asse qualunque, posto a distanza r rispetto al punto stesso, è dato dalla relazione:

$$i = mr^2 \tag{1.10}$$

da cui possiamo vedere che la 'resistenza' alla variazione di stato è direttamente proporzionale sia alla massa del punto sia alla sua distanza dall'asse.

Nel caso di un sistema di n punti materiali, ciascuno di massa m_i e posto a distanza r_i dall'asse scelto, il momento d'inerzia diventa:

$$I = \sum\nolimits_{i=1,n} m_i {r_i}^2 \tag{1.11}$$

Infine nel caso di un corpo rigido, considerando quest'ultimo come un sistema continuo ed indicando con ρ la densità di tale sistema, la sommatoria è sostituita da un integrale, ottenendo così la relazione:

$$I = \int_V \rho r^2 dv \tag{1.12}$$

dove dv è l'elemento di volume.

Tale relazione può essere semplificata, considerando la massa totale del corpo, $M = \int_V \rho dv$, come interamente concentrata nel suo baricentro, il quale dista r dall'asse rispetto al quale si vuole calcolare il momento d'inerzia. In tali condizioni, possiamo tornare ad una formulazione del tutto analoga a quella di un punto materiale, che però costituisce un'approssimazione.

Sempre in relazione all'argomento di questa Tesi, è utile sottolineare che il momento di inerzia è spesso utilizzato per descrivere la dinamica dei corpi in rotazione attorno ad un asse, ed il suo valore fornisce, come è stato già accennato, una misura dell'inerzia del corpo rispetto alle variazioni del suo stato di moto, in questo caso, rotatorio.

Inoltre, tale grandezza tiene conto di come è distribuita la massa del corpo attorno all'asse di rotazione. Consideriamo, ad esempio due corpi, A e B, aventi la stessa massa (distribuita in modo uniforme), e la stessa forma, ma tali che il corpo A ha dimensioni doppie rispetto al corpo B. Se volessimo fare ruotare entrambe i corpi intorno ad uno stesso asse (come ad esempio il loro baricentro), sarebbe più difficili accelerare (cioè cambiare lo stato di moto) il corpo A. Questo avviene perchè la sua massa è, nel complesso distribuita più distante dall'asse di rotazione.

Da questo esempio è anche possibile capire come la relazione 1.10 costituisca un'approssimazione nel caso di corpi rigidi, infatti utilizzando tale relazione, nel caso appena descritto, il momento d'inerzia dei corpi A e B sarebbe lo stesso, mentre, evidentemente il corpo A ha un momento d'inerzia maggiore rispetto al corpo B.

1.3.2 Realizzazioni pratiche

Sostanzialmente le diverse tipologie di accelerometro differiscono per il tipo di materiale utilizzato per l'elemento elastico, il tipo di massa e la modalità di misurazione del movimento della stessa.

Accelerometro piezoelettrico

L'accelerometro piezoelettrico, sfrutta, per la rilevazione dello spostamento della massa, il segnale elettrico generato da un cristallo piezoelettrico quando quest'ultimo è sottoposto ad una compressione.

In questi accelerometri la massa viene sospesa su un cristallo piezoelettrico, che, in questo caso, costituisce sia il sensore, che l'elemento elastico. In presenza di un'accelerazione la massa (che presenta una certa inerzia) comprime il cristallo, il quale genera un segnale elettrico proporzionale alla compressione.

Generalmente gli accelerometri piezoelettrici utilizzando un cristallo, sono caratterizzati da una sensibilità relativamente bassa, ma possono rivelare accelerazioni anche molto elevate. Il difetto principale è che questi accelerometri non sono in grado di rivelare accelerazioni statiche. Infatti se la compressione sul cristallo permane, il segnale generato tende a dissiparsi dopo un breve periodo.

Accelerometro capacitivo

L'accelerometro capacitivo, come principio per la rilevazione dello spostamento della massa, sfrutta la variazione della capacità elettrica di un condensatore al variare della distanza tra le sue armature.

La massa infatti viene realizzata con materiale conduttivo e costituisce un'armatura del condensatore, mentre l'altra armatura è attaccata alla struttura fissa del dispositivo, nell'immediata prossimità della massa. L'armatura mobile, viene sospesa su un elemento elastico relativamente rigido (solitamente una membrana siliconica) ed un apposito circuito rileva la capacità del condensatore così realizzato, generando un segnale elettrico proporzionale alla posizione della massa.

CAPITOLO 2

Hardware e protocolli

2.1 Lego Mindstorms NXT

Il Mindstorms Lego Next (NXT), prodotto dalla Lego a partire dall'agosto 2006, è un kit composto da un brick programmabile, alcuni sensori per l'interfacciamento, tre motori con encoder e numerosi mattoncini Lego.

In seguito alla decisione, da parte della Lego, di rendere Open Source le specifiche hardware ed i sorgenti del firmware sono nati numerosi sistemi operativi per il Mindstorms, che permettono la scrittura di software sviluppati utilizzando diversi linguaggi di programmazione tra cui C e Java. Tra le varie implementazioni, quelle che hanno ottenuto maggior successo sono Lego Java Operating System (leJOS), che permette lo sviluppo di software mediante Java, e RobotC che invece utilizza il linguaggio C. Per entrambe le implementazioni è necessaria l'installazione di un firmware modificato all'interno dell'NXT.

Nel caso di leJOS, che è il sistema operativo utilizzato in questo lavoro di tesi, all'interno del firmware modificato risiede una Java Virtual Machine.

Di seguito verranno riportare le caratteristiche tecniche del brick programmabile.

Caratteristiche tecniche

Le caratteristiche del brick sono le seguenti:

- Processore a 32 bit Atmel AT91SAM7S256 (classe ARM7) a 48 MHz
- Coprocessore 8 bit Atmel ATmega48 (classe AVR: è un Reduced Instruction Set Computer (RISC) a 8 bit) a 8 MHz, con 4k di memoria flash e 512 byte di RAM
- 256 KB di memoria flash
- 64 KB di RAM
- Interfaccia bluetooth v2.0+EDR (chipset CSR BlueCore 4 version 2, con clock a 26 MHz, con firmware stack Bluelab 3.2) velocità teorica massima 0,46 Mbit/sec
- Display Liquid Crystal Display (LCD) grafico bianco e nero a matrice di punti da 100Œ64 pixel

- Speaker mono 8 bit fino a 16 KHz
- Tastierino con quattro pulsanti
- 4 porte di input che supportano la comunicazione I2C a bassa velocità, ed il campionamento a 10 bit di un segnale analogico tramite il coprocessore Atmel
- 3 porte di output per il pilotaggio dei motori tramite un segnale Pulse Width Modulation (PWM).
- Sviluppo software con LabVIEW di National Instruments

2.1.1 leJOS

Giovanni Marcianò Uso didattico della Robotica.

leJOS è un sistema operativo Open Source, arrivato alla versione 0.7, che sostituisce il firmware dell'NXT con una Java Virtual Machine.

leJOS permette di eseguire un codice Java all'interno dell'NXT, implementando molte delle caratteristiche del linguaggio Java, tra cui, i thread, gli array multidimensionali, le operazioni in virgola mobile, la ricorsione, ed modello ad eventi. Inoltre leJOS è in grado di fornisce alcune **API** per la gestione dell'hardware dell'NXT.

leJOS è praticamente una evoluzione della TinyVM, che è la Virtual Machine utilizzata per l'RCX, il predecessore dell'NXT.

leJOS presenta un ambiente completamente *Object Oriented*, le cui caratteristiche principali sono:

- Ereditarietà: Qualunque oggetto può essere modellato a partire da un *super–oggetto* precedentemente modellato. E' possibile quindi ridefinire solamente i metodi caratteristici del nuovo oggetto, mentre il funzionamento dei metodi non specificati rimane invariato.
- Incapsulamento: E' possibile accedere al contenuto di un oggetto attraverso un suo metodo che è definito pubblico e per questo accessibile dall'esterno.
- Polimorfismo: Se viene chiamato un metodo di un oggetto padre e l'istanza è una sua specializzazione (ereditata), allora il metodo da eseguire sarà quello dell'oggetto figlio.
- Numeri in virgola mobile: leJOS permette di utilizzare numeri in virgola mobile e quindi di eseguire calcoli di funzioni trigonometriche, fondamentali per alcuni algoritmi di controllo.
- Thread: Una delle parti più importanti di leJOS è il supporto alla programmazione concorrente. La Virtual Machine gestisce i thread secondo uno scheduling di tipo preemptive. Anche se il numero massimo di thread che possono essere creati è 255, data la limitata quantità di memoria disponibile sull' NXT, è sconsigliato superare i 10 thread.
- Array Multidimensionali: leJOS aggiunge il supporto per array multidimensionali con dimensione anche maggiore di 255 elementi.

- Modello ad eventi: leJOS è in grado di usare il modello ad eventi standard di Java, che include listeners e sorgenti di eventi.

- Eccezioni: leJOS include il supporto alle eccezioni ed alla ricorsione con un numero di annidamenti limitato. La mancanza di un Garbage Collector, nelle versioni fino alla 0.6 era una grande limitazione. Infatti, dal momento che non esiste un costrutto specifico per distruggere oggetti in Java, la memoria dell' NXT si esauriva rapidamente se non si prestava particolare attenzione al numero di oggetti utilizzati. Questa limitazione forzava la programmazione ad avere uno stile più procedurale che orientato agli oggetti. Ad ogni modo, la nuova versione versione 0.7 utilizzata in questo lavoro di tesi comprende una prima implementazione del Garbage Collector.

2.2 Protocolli di comunicazione

2.2.1 Protocollo PPM

Ad oggi, il più diffuso protocollo utilizzato per la comunicazione tra trasmittente e ricevente nei modelli radiocomandati, è il Pulse Position Modulation (PPM).

Questo protocollo di comunicazione si basa sull'utilizzo di un'onda quadra ad ampiezza fissa di circa 5 volt e duty cycle variabile tra 1 ms e 2 ms. Il periodo tra un impulso ed il successivo è fissato a circa 20 ms. Quest'ultimo, in realtà non è un valore critico e può anche scendere fino a 15 ms, oppure salire fino a 40 ms, a seconda delle specifiche di trasmissione e del numero di canali utilizzati.

Il motivo principale di un così lungo periodo tra un impulso ed il successivo è chiarito pensando che l'utilizzo tipico di un radiocomando per modellismo coinvolge la trasmissione di più canali contemporaneamente. Infatti, tramite l'utilizzo di un multiplexer all'interno della radio, è possibile spostare temporalmente i singoli impulsi e concentrare, in un tempo di 20 ms, fino ad un massimo teorico di 10 canali.

Il valore del duty cycle di ogni impulso invece è quello che determina la posizione (ad esempio di un servocomando) ed è centrato sul valore di 1.5 ms. Pertanto, tipicamente, valori compresi tra 1 ms e 1.5 ms determineranno uno spostamento antiorario, mentre valori compresi tra 1.5 ms e 2 ms determineranno uno spostamento nel verso orario.

Quando il segnale arriva alla ricevente, questa si occupa di smistare i diversi canali attraverso un de–multiplexer, e di inviarli ai rispettivi apparati collegati. La velocità di comunicazione tra i diversi apparati è ovviamente vincolata dal tempo che intercorre tra un segnale e il successivo. In determinati casi però, alcuni dispositivi parzialmente indipendenti dal comando ricevuto (come ad esempio il giroscopio) possono utilizzare una versione del PPM più veloce di quella standard per comunicare con il loro servo dedicato. Questa nuova specifica riduce il tempo tra due impulsi successivi a soli 3 ms circa, migliorando notevolmente la risposta dei servocomandi.

Condizione necessaria per l'utilizzo di questa nuova versione è che il dispositivo che risponde al comando deve sia stato appositamente progettato per gestire questo tipo di segnale. Un servo standard sarebbe infatti inutilizzabile dopo poco tempo a causa dell'eccessivo carico di lavoro richiesto per lo spostamento continuo tra una posizione e l'altra. Questa nuova versione del protocollo di comunicazione PPM è utilizzata principalmente con lo scopo di mantenere fissa la posizione della coda di un elicottero Radio Controllato (RC), unitamente all'utilizzo di servocomandi digitali, i quali, grazie

alla maggiore precisione e velocità di risposta, sono in grado di sfruttare appieno questa nuova caratteristica.

In Figura

2.2.2 Protocollo I^2C

Il protocollo Inter–Integrated–Circuit (I^2C) è uno standard ideato dalla Philips, nel 1980, per superare le difficoltà inerenti l'utilizzo di bus paralleli per la comunicazione tra un'unità di controllo e le varie periferiche.

In un bus parallelo l'informazione viaggia attraverso molti fili. Fin quando una sola periferica è collegata al microcontroller, i problemi legati alla presenza di molte linee possono essere tenuti sotto controllo. Qualora però le periferiche dovessero essere più di una, far giungere il bus ad ogni periferica potrebbe diventare un problema.

Questo inconveniente viene interamente superato dal bus I^2C, in quanto questo permette la comunicazione tra diverse periferiche mediante due sole linee.

L'I^2C bus è infatti uno standard seriale che permette di collegare sullo stesso bus un numero elevato di periferiche, ognuna individuata da un proprio indirizzo.

Un notevole vantaggio nei dispositivi che fanno uso del bus I^2C è quello della loro semplicità d'uso. Infatti tutte le regole del protocollo che bisogna rispettare per una corretta comunicazione vengono solitamente gestite a livello hardware.

La prima versione del bus I^2C permetteva di trasmettere fino a 100Kbit/s, questa velocità è stata portata a 400Kbit/s nelle modifiche apportate nel 1992, e successivamente incrementata fino a 3.4Mbit/s.

Specifiche elettriche del bus I^2C

Come è già stato accennato, il bus I^2C è un bus seriale che permette la comunicazione mediante due sole linee, più una linea di massa.

Le due linee utilizzate per la comunicazione sono denominate Serial Data (SDA) ed Serial Clock (SCL) e sono entrambe bidirezionali. La prima è utilizzata per il transito dei dati, che sono in formato ad 8 bit, mentre la seconda è utilizzata per trasmettere il segnale di clock, necessario per la sincronizzazione della trasmissione.

Le linee SDA e SCL devono essere implementate per mezzo di uscite *open drain* oppure *open collector*. Questa caratteristica è particolarmente importante qualora si voglia implementare il protocollo I^2C interamente via software. Tale caratteristica rende necessaria una resistenza di *pull–up*, ovvero di una resistenza collegata tra la linea ed il polo positivo, per ogni linea. Valori tipici per le resistenze di pull up sono compresi tra 2KOhm e 10KOhm.

L'inserimento della resistenza di pull–up implica che se le linee SDA e SCL non sono utilizzate, si trovano ad un livello alto del segnale digitale.

Se è vero che il bus I^2C permette la connessione di più periferiche su uno stesso bus, è anche vero che lo stesso permette la comunicazione tra due soli dispositivi alla volta. Il dispositivo che trasmette le informazioni viene detto *trasmettitore*, il dispositivo che le riceve viene detto *ricevitore*. Entrambe le posizioni non sono fisse, ovvero, un trasmettitore può anche divenire ricevitore in una differente fase della trasmissione dati. Al contrario, in ogni comunicazione i ruoli del *Master* e dello *Slave* una volta fissati rimangono tali durante tutta la comunicazione. Il Master è il dispositivo che

inizia e termina la comunicazione, lo Slave può solo ricevere o trasmettere informazioni su richiesta del Master.

Non tutti i dispositivi possono essere dei Master del bus I^2C. Per esempio una memoria per il mantenimento dei dati non sarà un Master del bus, mentre è ragionevole supporre che un microcontroller lo possa essere. Su uno stesso bus è inoltre possibile la presenza di più Master, ma solo uno alla volta ricoprirà questo ruolo. Se per esempio due microcontroller iniziano una comunicazione, anche se potenzialmente potrebbero essere ambedue dei Master, solo uno lo sarà, in particolare il Master sarà quello che ha dato inizio alla comunicazione, di conseguenza l'altro sarà uno Slave.

Ogni periferica inserita nel bus I^2C possiede un indirizzo che la individua sul bus in modo univoco. Questo indirizzo può essere fissato dal produttore in sede di fabbricazione o parzialmente fissato dal progettista. L'indirizzo è costituito da 7 bit nelle versioni standard o da 10 bit nelle versioni estese. Nel caso di indirizzamento a 7 bit si avrebbe potenzialmente la possibilità di indirizzare 128 periferiche mentre nel caso di 10 bit si avrebbe la possibilità di indirizzare fino a 1024 periferiche.

Comunicazione sul bus I^2C

Vediamo ora, nella pratica, come avviene una comunicazione su un bus I^2C. Ricordiamo ancora che solo le periferiche Master possono avviare una comunicazione.

Le fasi della comunicazione sono le seguenti:

1) Il Master controlla che le linee SDA e SCL non siano già impegnate in un'altra comunicazione, ovvero che siano entrambe poste ad livello alto del segnale digitale. Se il bus è libero, il Master invia il messaggio che fa capire alle altre periferiche che il bus è ora occupato. Le altre periferiche si mettono in ascolto per comprendere con chi il Master ha intenzione di comunicare.

2) Il Master provvede all'invio del segnale di sincronizzazione, sulla linea SCL, che sarà rappresentato da un onda quadra, non necessariamente periodica.

3) Il Master invia l'indirizzo della periferica con la quale vuole parlare.

4) Il Master indica se la comunicazione che vuole intraprendere verso la periferica è di lettura o scrittura.

5) Il Master attende la risposta da parte della periferica che nella chiamata ha riconosciuto il suo indirizzo. Se nessuna periferica risponde il Master libera il bus.

6) Dopo l'avvenuto riconoscimento della periferica, il Master inizia lo scambio dei dati. Lo scambio avviene inviando pacchetti di 8 bit. Ad ogni pacchetto è necessario attendere il segnale che avvisa dell'avvenuta ricezione.

7) Quando la trasmissione è terminata il Master libera il bus inviando un segnale di stop.

Vediamo ora, in dettaglio, come vengono realmente ottenute le varie fasi della comunicazione.

- **Fase 1–2:** L'hardware del microcontroller, dedicato alla gestione dell'interfaccia I^2C, controlla le linee SDA e SCL per un tempo superiore al massimo periodo di trasmissione consentito dall'hardware. Se il bus risulta libero, il Master invia la sequenza di Start, che consiste nel portare la linea SDA a livello basso quando SCL è a livello alto. Dopo l'invio della sequenza di Start, il bus è considerato occupato.

- **Fase 3:** Dopo la transizione della linea SDA da alto a basso, il Master invia il segnale di sincronizzazione per le altre periferiche. A differenza della sequenza di Start e di Stop la linea SDA assume un valore valido solo se la linea SCL è a livello basso. Questo vuol dire che non sono ammesse transizioni di livello della linea SDA durante il livello alto della linea SCL, se non da parte del Master per inviare un nuovo Start o uno Stop.

- **Fase 4–5:** Come detto il formato dell'indirizzo può essere sia a 7 bit che a 10 bit, per semplicità si considera prima il formato a 7 bit. I 7 bit dell'indirizzo vengono inviati dal bit più significativo al bit meno significativo. In coda a questo indirizzo viene aggiunto un bit per segnalare se il Master vuole intraprendere, con la periferica individuata da tale indirizzo, una comunicazione di scrittura o di lettura. In particolare se tale bit è 0 vuol dire che il Master vuole scrivere sulla periferica, se il bit è 1 vuol dire che il Master vuole leggere della periferica.

- **Fase 6:** L'invio dell'indirizzo a 7 bit e della modalità del colloquio (lettura/scrittura), avviene grazie ad otto transizioni, da livello alto a livello basso, della linea SCL. Al nono impulso della linea SCL il Master si aspetta una risposta di un bit da parte della periferica che ha chiamato. La risposta della periferica chiamata consiste nel mantenere a livello basso la linea SDA, per la durata di un ciclo SCL. Cioè il Master attende l'Acknowledge da parte della periferica chiamata. Una sola periferica risponderà alla chiamata del Master. Qualora la periferica non sia presente il Master libera il bus permettendo ad eventuali altri Master di prenderne il controllo.

- **Fase 7:** Dopo l'avvenuto riconoscimento, avviene lo scambio dei dati verso la periferica, nel caso di scrittura, o dalla periferica al Master, in caso di lettura. In una comunicazione si possono avere sia fasi di scrittura sia fasi di lettura. Ad ogni invio di un byte sarà necessario l'*Acknowledge* del byte inviato o ricevuto. In particolare quando il Master invia un byte allo Slave si aspetta, dopo l'ottavo bit, un bit basso sulla linea SDA. Se lo Slave ha inviato un byte al Master, si aspetta che quest'ultimo gli invii un bit alto dopo di risposta. La mancanza dell'Acknowledge determina un errore di comunicazione.

- **Fase 8:** Quando la comunicazione è terminata, il Master libera il bus, inviando la sequenza di Stop. Questa consiste nella transizione dal livello basso ad alto della linea SDA, quando la linea SCL è alta. Se il Master deve effettuare un'altra comunicazione con un altro Slave, piuttosto che liberare il bus e rischiare di perdere il diritto del controllo, può inviare una nuova sequenza di Start.

CAPITOLO 3

Applicazioni sperimentali: Un'interfaccia per giroscopi RC

Lo scopo di questa parte del progetto era la creazione di un'interfaccia per l'NXT, per mezzo della quale fosse possibile raggiungere lo scopo prefissato, cioè il mantenimento dell'equilibrio del robot, attraverso un qualunque giroscopio progettato per l'utilizzo in sistemi RC.

In questa parte del progetto è stato utilizzato un giroscopio Futaba GY401 per la misurazione dell'angolo di tilt del robot.

Stando alle specifiche hardware del Lego NXT su ogni porta di ingresso ed ogni porta d'uscita è presente una tensione che può essere prelevata per alimentare dispositivi e sensori esterni. Questa tensione ha un valore di 4.3 Volt, ed è condivisa tra tutte le porte. Questa tensione è prelevabile direttamente attraverso i piedini 2–3, per quanto riguarda la massa (GND) e attraverso il piedino numero 4 per quanto riguarda il segnale positivo.

La massima corrente prelevabile deve essere inferiore a 180 mA, infatti nel caso si eccedesse questo valore l'NXT effettuerebbe un reset. Già dai primi tentativi è risultato chiaro che la corrente prelevabile da ogni porta non era sufficiente ad alimentare il GY401. Questo determinava una mancanza di stabilità in tensione durante la fase di inizializzazione del giroscopio GY401 che a sua volta impediva allo stesso di attivarsi.

Per ovviare a questo problema è stata utilizzata una fonte di alimentazione esterna per il sensore, stabilizzata a 5 Volt.

3.1 Interfacciamento del giroscopio

3.1.1 Giroscopio Futaba GY401

Il giroscopio GY401 è prodotto dalla giapponese Futaba per essere utilizzato nella stabilizzazione della coda negli elicotteri radiocomandati. Tale dispositivo è dotato di funzionalità avanzate, come il supporto per la comunicazione PPM veloce nell'utilizzo con servocomandi digitali (vedi paragrafo 2.2.1), il controllo remoto della sensibilità, e il controllo remoto della funzioalità Angular Velocity Control System (AVCS) (heading hold).

La funzione AVCS, assolutamente indispensabile nelle manovre 3D di un elicottero RC, può essere vista come l'integrale nel tempo della funzione dello spostamento della

Figura 3.1: Il giroscopio Futaba GY401

coda. In altri termini questa funzione memorizza istante per istante la posizione attuale della coda e cerca di ripristinarla quando questa subisce uno spostamento non voluto, imputabile a fattori esterni come ad esempio atmosferici quali il vento.

D'altra parte, per come è progettata, questa funzione non è sensibile a piccolissime variazioni di movimento della coda, dovute sia ad una instabilità intrinseca di questi piccoli elicotteri sia a vibrazioni causate da lievi imprecisioni meccaniche. Inoltre se uno spostamento supera il range di sensibilità del giroscopio, la vecchia posizione viene semplicemente abbandonata e sostituita da quella appena raggiunta.

All'interno del giroscopio alloggia un sensore CRS03 prodotto dalla Silicon Sensing Systems ed assemblato attraverso processi produttivi Micro Electronic Mechanical Systems (MEMS). Tale sensore è a singolo asse e produce in uscita un segnale analogico, nel range (0.5–4.5) Volt, proporzionale alla velocità angolare rilevata. Le caratteristiche principali del sensore sono evidenziate in Figura

Una caratteristica di questo sensore è l'elevato range di rivelazione, è in grado infatti di misurare velocità angolari fino a $(\pm 573°/s)$. Tale caratteristica però, unitamente al rumore massimo generato, con valore di 15 mV, potrebbe determinare la mancata capacità di rivelazione di piccole variazioni di velocità angolare. Questo è in particolar modo vero se non si presta attenzione nella progettazione dell'alimentazione del sensore. Il consumo di corrente del sensore a livello di regime è di 35 mA, ma sale a quasi 180 mA durante la sua inizializzazione. Questa elevata corrente in fase di inizializzazione è necessaria affinchè il giroscopio restituisca i dati in maniera corretta. Tra le altre caratteristiche di questo sensore vale la pena sottolineare la ridotta sensibilità a variazioni di temperatura ed un drift minimo e predicibile nel tempo. E' comunque doveroso sottolineare che non tutti i *datasheet* trovati riportano gli stessi valori, forse a causa di numerose versioni del sensore aventi lo stesso nome.

Alle spalle del CRS03 si trova un microcontroller prodotto dalla Hitachi, in particolare l'H8/3294 costituito da 80 pin e che dispone di 32 k–byte di memoria ROM e 1 k–byte di memoria RAM. Questo dispositivo è in grado di supportare una velocità massima di 16 MHz, ma per limitare il consumo generale del giroscopio, utilizza usa una frequenza di lavoro pari a 8 MHz. Questo microcontroller campiona il segnale del sensore tramite un convertitore analogico/digitale (A/D) integrato a 10 bit, lo converte in un segnale PPM ed implementa la funzione AVCS per il blocco di coda. Infine miscela adeguatamente il segnale elaborato con quello pervenuto dalla ricevente e lo

trasmette al suo servo.

E' importante sottolineare che il GY401 non sfrutta il PPM ricevuto in ingresso dalla ricevente come portante per generare il PPM in uscita, ma ne crea uno completamente indipendente da quello ricevuto. L'utilizzo di un PPM proprio fa si che anche quando il segnale in ingresso è instabile, soprattutto per quanto riguarda il tempo che intercorre tra un impulso ed il successivo, quello in uscita prodotto dal giroscopio è invece sempre perfetto. Questo garantisce un pilotaggio del servocomando davvero ottimale.

Infine ricordiamo la necessità di avere i due segnali in ingresso, cioè la regolazione della sensibilità e l'attivazione o meno dell'AVCS, già presenti prima dell'accensione del giroscopio, altrimenti alcune funzionalità dello stesso non verranno attivate.

3.1.2 Regolatore di tensione

Allo scopo di alimentare il giroscopio Futaba GY401 con una tensione stabile di 5 Volt (vedi introduzione 3) è stato costruito un piccolo regolatore di tensione stabilizzata sfruttando l'integrato regolatore 7805, unitamente ai suoi due necessari condensatori ceramici e ad un condensatore elettrolitico utilizzato per una rudimentale stabilizzazione del voltaggio ed eliminazione del segnale PWM presente sull'uscita della porta numero 3 del robot NXT.

In questo modo si ha a disposizione una tensione di 5 Volt e una corrente 1.5 Ampere teorici, valori che rendono possibile la corretta alimentazione del giroscopio. La tensione prelevata dal regolatore si è dimostrata stabile, malgrado una discreta quantità di dissipazione di energia sotto forma di calore. In Figura

3.1.3 Generatori di segnale PPM

Come spiegato nel paragrafo 3.1.1, relativo al giroscopio Futaba GY401, per corretta inizializzazione del dispositivo sono necessari 2 segnali PPM, uno per la regolazione della sensibilità del giroscopio e l'altro per l'attivazione della modalità AVCS.

Con questo scopo, sono stati realizzati due generatori di segnale ciascuno dei quali utilizza un integrato NE555 unitamente ad un inverter 7404. Tramite il potenziometro di ciascun generatore di segnale è possibile variare i parametri per il corretto utilizzo del giroscopio. In Figura 3.2 è riportato lo schema elettrico dei due generatori.

3.1.4 Microcontroller 16F876

Il modello 16F876 è un microcontroller a 28 pin prodotto dalla Microchip.

Al suo interno contiene un processore RISC ad 8 bit in grado di utilizzare un massimo di 35 istruzioni. La maggior parte di esse è eseguita in un singolo ciclo macchina.

La velocità del Programmable Interface Controller (PIC) può essere impostata al massimo a 20 MHz, e può essere variata utilizzando quarzi di diverse frequenze.

Il dispositivo può essere alimentato con tensioni che partono da 2 Volt fino ad un massimo di 5.5 Volt ed il tipico consumo di corrente è di 0.6 mA quando il microcontroller opera ad una frequenza di 4 MHz.

Il modello 16F876 dispone di 8 Kbytes di memoria Flash, (368 x 8) bytes di memoria RAM e (256 x 8) bytes di memoria Electronically Erasable Programmable Read Only

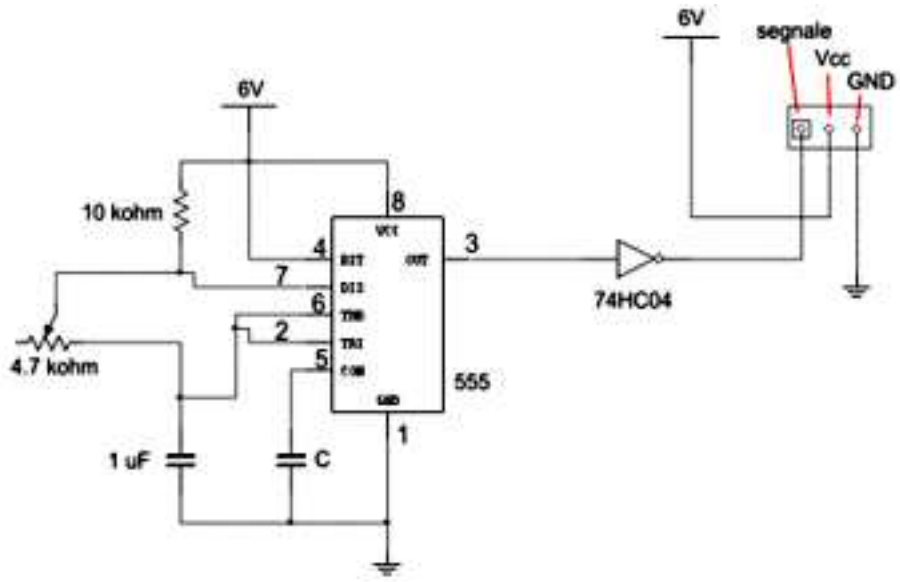

Figura 3.2: Schema elettrico del generatore di segnale PPM

Memory (EEPROM). Tra le sue caratteristiche ricordiamo la capacità di gestire in hardware una serie di Interrupt (interrogabili anche in polling dall'utente) derivanti da 14 differenti eventi possibili ed inoltre la possibilità di implementare funzioni di Power On Reset (POR), Power Up Timer (PWRT), Oscillator Start Up Timer (OST) e Watchdog Timer (WDT).

Le caratteristiche delle sue periferiche interne sono le seguenti:

- Timer0: 8-bit timer/counter con 8-bit prescaler
- Timer1: 16-bit timer/counter con prescaler che può essere incrementato anche durante la funzione sleep attraverso un quarzo esterno.
- Timer2: 8-bit timer/counter con un 8-bit period register, prescaler e postscaler
- Due moduli Capture, Compare, PWM con le seguenti caratteristiche: Capture a 16-bit con risoluzione massima di 12.5 nS, Compare a 16-bit con risoluzione massima di 200 nS e PWM con risoluzione massima di 10-bit.
- Multipli convertitori Analog-to-Digital a 10-bit.
- Synchronous Serial Port (SSP) con SPI. (Master mode) e funzionalità I^2C. (Master/Slave)
- Universal Synchronous Asynchronous Receiver Transmitter (USART/SCI) con indirizzamento a 9-bit
- Parallel Slave Port (PSP) a 8-bit, con RD esterno e controllo WR e CS (solo versione 40/44 pin)
- Controllo e individuazione di Brown-out per la funzionalità Brown-out Reset (BOR)

Esistono numerose revisioni del PIC 16F876, ognuna delle quali corregge alcuni bug riscontrati durante l'utilizzo. La difficoltà principale nasce nel capire quale revisione si stà attualmente utilizzando, dal momento che nessuna indicazione è riportata sul package.

Per la programmazione dell'integrato è stata utilizzata una piattaforma hardware di sviluppo prodotta dalla Microsystems Engineering (MSE), nello specifico la scheda $\mu PIC'Trainer$. In realtà questa piattaforma non stata progettata per l'utilizzo del modello 16F876 essendo stata prodotta alcuni anni prima dell'integrato. E' stato quindi necessario ricorrere ad una modifica hardware per permettere l'alloggiamento fisico del 16F876 e per utilizzare la funzionalità Low Voltage Programming (LVP), caratteristica di questo ed altri microcontroller.

Il microcontroller è stato programmato utilizzando il linguaggio C attraverso l'ambiente di sviluppo HI-TIDE, prodotto dalla HI-TECH ed utilizzato nella sua versione free. Questa versione utilizza il compilatore HI-TECH C PRO, anch'esso sviluppato dalla HI-TECH ed utilizzato in versione not optimized/free.

Il codice sviluppato è stato successivamente caricato sul microcontroller attraverso l'uso del software IC–Prog.

3.1.5 Modulo Capture del PIC 16F876

In questo paragrafo verranno descritti i metodi utilizzati per realizzare l'interfacciamento del giroscopio con l'NXT.

Ricordiamo (vedi paragrafo 3.1.1) che il giroscopio produce in uscita un segnale PPM con duty cycle variabile, di durata compresa tra 1 ms e 2 ms circa, che identifica la posizione del servo necessaria a mantenere fisso l'asse di rotazione del giroscopio. Dal momento che l'ampiezza di questo segnale non varia nel tempo la sua trasformazione in segnale analogico (campionamento) è alquanto laboriosa e imprecisa. Questo campionamento seguito da una nuova conversione in segnale digitale porta ad un notevole degrado del segnale con una inaccettabile perdita di precisione. Inoltre dal momento che il coprocessore interno dell' NXT ha una frequenza di campionamento di 333 Hz, che corrisponde a 3 campionamenti ogni millisecondo circa, non è possibile campionare direttamente il segnale del giroscopio in quanto tale frequenza sarebbe insufficiente a ricostruire il segnale in modo corretto, dal momento che questo ha una durata massima di 2 ms. Infatti per ottenere un segnale accettabile è necessaria una frequenza di campionamento minima di 10000 Hz che corrisponde ad un campionamento ogni 0.1 ms con il quale è possibile distinguere almeno 10 differenti posizioni del servo.

Per questi motivi si è deciso di utilizzare per la decodifica e conversione del segnale PPM del giroscopio un dispositivo esterno. Tale dispositivo è il già descritto microcontroller 16F876 (paragrafo 3.1.4. In particolare per l'acquisizione del segnale, è stata utilizzata la modalità Capture del suo modulo Capture, Compare, PWM (CCP). Il microcontroller è stato configurato per lavorare ad una frequenza di 4 MHz, effettuando quindi un ciclo macchina ogni microsecondo.

Il Timer1, che è un timer composto da 2 registri da 8 bit ciascuno, è stato utilizzato in modalità Counter, quindi ogni istruzione eseguita dal PIC incrementerà il suo valore di una unità. In particolare l'incremento avviene ogni microsecondo ed in ogni momento è possibile leggere dal Timer1 un valore complessivo di 16 bit.

La funzione Capture può essere utilizzata in diverse modalità. In questo lavoro di tesi sono state utilizzate due modalità in particolare. La cattura di un evento *rising*

edge (passaggio da 0 Volt a 5 Volt del segnale PPM) e quella di un evento *falling edge* (passaggio da 5 Volt a 0 Volt del segnale PPM).

Nel dettaglio il principio di funzionamento utilizzato dalla modalità Capture può essere schematizzato nel seguente modo: Il modulo Capture è settato in modalità Capture every rising edge, questo significa che non appena viene rivelato un fronte di salita del segnale il modulo genera un interrupt. In seguito il valore del Timer1 è salvato in una variabile temporanea ed il modulo Capture viene riconfigurato in modalità Capture every falling edge. Tale modalità genera un secondo interrupt non appena il segnale di ingresso cambia stato. Il nuovo valore del Timer1 viene salvato in una nuova variabile temporanea e viene calcolata la differenza con il precedente valore del Timer1. Il risultato è il numero di cicli macchina, ovvero di microsecondi, che il PIC ha eseguito tra il fronte di salita del segnale ed il successivo fronte di discesa dello stesso.

Infine si riconfigura il modulo Capture in modalità Capture every rising edge, si resetta il Timer1 ed il ciclo può riprendere dall'inizio.

Ovviamente, per un corretto funzionamento è necessario che il tempo di *overflow* del Timer1 sia maggiore della massima durata possibile del segnale PPM.

L'algoritmo di cattura del segnale PPM può essere consultato in appendice B.

3.1.6 Modulo I^2C del PIC 16F876

Il trasferimento del segnale PPM dall microcontroller all'NXT è stata realizzata implementando una comunicazione I^2C all'interno del PIC.

Dal momento che l'NXT gestisce le comunicazioni I^2C esclusivamente come Master il microcontroller è stato configurato come Slave. Il PIC è in grado di implementare i segnali di Start, Stop ed i vari Ack in hardware senza necessità di intervento esterno da parte del programmatore. Al contrario è necessario eseguire il controllo del *flag* relativo all'interrupt generato da un evento I^2C.

Come accennato nel paragrafo 2.2.2, riguardante questo protocollo di comunicazione, il flag da controllare presenta sostanzialmente 6 possibilità:

1) MASTER WRITE - LAST BYTE WAS ADDRESS. Il Master desidera comunicare ed all'interno della trasmissione è inserito l'indirizzo del destinatario. In questo caso è sufficiente effettuare una lettura del buffer per prevenire l'overflow.

2) MASTER WRITE - LAST BYTE WAS DATA. Il Master invia i dati al destinatario, il cui indirizzo non è presente in quanto già precedentemente inviato. Anche in questo caso è sufficiente effettuare una lettura del buffer per prevenire l'overflow.

3) MASTER READ - LAST BYTE WAS ADDRESS. Il Master desidera leggere un dato ed all'interno della trasmissione è inserito l'indirizzo dello Slave. In questo caso è necessario riempire il buffer con il dato da inviare. La comunicazione non ripartirà finchè il buffer non sarà riempito.

4) MASTER READ - LAST BYTE WAS DATA. Il Master desidera leggere dati. Anche in questo caso è necessario riempire il buffer con il dato da spedire, altrimenti la comunicazione non ripartirà.

5) NACK FROM MASTER / INITIALIZE I^2C. Il Master comunica che la trasmissione può essere chiusa. In questo caso si riconfigura il modulo I^2C alle condizioni iniziali.

6) I^2C ERROR. In caso di errori è necessario resettare il modulo disabilitandolo e successivamente riabilitandolo.

Numerosi test hanno dimostrato una comunicazione tra il PIC 16F876 e l'NXT abbastanza robusta, con sporadici errori di comunicazione sicuramente imputabili all'implementazione non standard del protocollo I^2C del robot che è gestita completamente via software.

Dalle specifiche dell'implementazione del protocollo di comunicazione I^2C del sistema leJOS infatti si evince che la modalità del *Clock Stretching* non è del tutto implementata e questo può portare ad errori di comunicazione in quanto lo Slave non ha la totale libertà di mettere in pausa la comunicazione. Infatti, ci sono solo due momenti in cui lo Slave può far attendere il Master prima di una sua risposta. In particolare nei casi 3 e 4 sopracitati, lo Slave può usufruire di un tempo indefinito per procurare ed elaborare il dato da spedire. Qualunque altra interruzione, anche dell'ordine di microsecondi, ha dimostrato di far perdere la sincronia con il clock di comunicazione generato dal Master, causando un errore di comunicazione.

Il valore da spedire in una qualunque comunicazione tra Master e Slave è un numero contenuto in un registro da 16 bit derivato dall'unione dei due registri ad 8 bit del Timer1. Questo valore è idealmente compreso tra 1000 e 2000 cicli macchina circa, con un valore neutro pari a 1500 cicli macchina. Dal momento che la comunicazione I^2C permette la spedizione di un Byte o di suoi multipli è stato necessario spedire ogni singola informazione attraverso due messaggi.

Inoltre poichè la comunicazione I^2C dell'NXT è fissata a 9600 bit al secondo, ovvero 9,6 bit ogni ms, per ogni singola informazione del giroscopio saranno necessari quasi 2 millisecondi per la sua spedizione al robot. Questo in realtà non è un problema dal momento che anche utilizzando la versione PPM veloce del giroscopio, questo non sarà in grado di fornire nuove informazioni prima di 3 ms circa.

3.1.7 Lettura del giroscopio tramite NXT

Per leggere il valore inviato dal giroscopio sono state utilizzate le Application Programming Interface (API) rese disponibili dal leJOS.

In particolare è stato prima di tutto istanziato un oggetto di tipo I^2C, ne è stato settato l'indirizzo, che nel nostro caso è stato impostato a 0x20 in esadecimale, sono state effettuate due letture in successione, il byte della prima ricezione è stato spostato di 8 bit a sinistra e successivamente le due trasmissioni sono state ricomposte tramite un operazione di OR sui bit. Il codice necessario per un'operazione di lettura è riportato di seguito:

```
sensor.setAddress(0x20);
byte[] buf=new byte[2];
sensor.getData(0x13, buf, 2);
int value=((buf[0]<<8 | buf[1]& 0xFF));
```

3.2 Risultati ottenuti

Il sistema così ottenuto ha dimostrato una serie di limitazioni.

In primo luogo l'impossibilità di stabilire una corrispondenza esatta tra i valori restituiti dal giroscopio e la reale velocità angolare, in gradi/s con la quale il robot si muoveva. In altre parole non è stato possibile, a causa della mancanza di una strumentazione di misura precisa ed affidabile, ricreare una scala in gradi/s corrispondente al range di valori compresi tra 1000 e 2000 restituiti dal sistema giroscopio–PIC.

In secondo luogo, il giroscopio Futaba GY401 ha dimostrato di avere una differente sensibilità nelle due direzioni di rotazione. Infatti le due scale di misurazione (oraria e antioraria) infatti sono uguali solo se il punto neutro, in posizione di riposo, divide in maniera esattamente identica l'intero intervallo di misura. Se invece il punto neutro si sposta da un lato allora in quella direzione il giroscopio risulterà meno sensibile. Il punto neutro è regolato tramite il generatore PPM come descritto nel paragrafo 3.1.3, si è quindi provato a cercare tale punto utilizzando il potenziometro del generatore. Riscontrando differenze, seppur minime, tra le due scale si è dedotto che non è possibile trovare tale punto con esattezza. Infatti anche se la distanza dal punto neutro ideale è minima, la stessa comporta un errore non trascurabile nella differente sensibilità di rivelazione tra la parte oraria e quella antioraria.

Questa differenza risulta evidente soprattutto per piccoli angoli di rotazione. Nella pratica è infatti impossibile ruotare il giroscopio nella direzione più sensibile, seppur con un movimento minimo, senza che lo strumento rilevi una qualche misurazione. E' invece possibile ruotarlo nella direzione della parte meno sensibile notando una frequente mancata misurazione.

Per questi motivi, il sistema GY401–PIC è stato abbandonato e per superare queste problematiche si è deciso di seguire un'altra strada che verrà descritta nei prossimi paragrafi.

3.3 Interfacciamento diretto con il sensore giroscopico

E' stato realizzato un sistema alternativo in cui il sensore CRS03 (vedi paragrafo 3.1.1) contenuto all'interno del giroscopio è stato interfacciato direttamente con l'NXT al quale è stato affidato il compito di campionare il segnale analogico in uscita dal sensore per mezzo del convertitore A/D a 10 bit del Lego.

Per prelevare il segnale analogico è stato sufficiente dividere i due circuiti interni del giroscopio, quello del sensore e quello di gestione del segnale. Questi due circuiti infatti sono collegati tra loro soltanto da tre pin, due dei quali portano l'alimentazione al sensore, mentre un terzo ne preleva la velocità angolare.

L'alimentazione del sensore è stata collegata direttamente al regolatore di tensione, ed il pin che restituisce l'informazione della velocità angolare è stato collegato al piedino numero 1 di una porta di ingresso che è collegato al convertitore A/D del coprocessore ATmega48 del robot.

Con questo tipo di interfacciamento i due generatori di segnale PPM diventano superflui, in quanto il segnale prelevato è assolutamente indipendente dal circuito di gestione del giroscopio.

3.3.1 Lettura del giroscopio tramite NXT

In questa nuova implementazione del sistema, per effettuare una lettura della velocità angolare è necessario campionare il segnale analogico attraverso il convertitore a 10 bit integrato nell'NXT.

Poichè il sensore ha un range di misurazione pari a $\pm 573°/s$ e che gli estremi di tale intervallo corrispondono a 0.5 Volt e 4.5 Volt, per trasformare il segnale campionato nella corrispondente velocità angolare è necessario prima risalire al valore di Vo tramite la relazione:

$$\text{Vo} = (u/1023) * 5$$

dove u è il valore restituito dal convertitore A/D. Successivamente si può risalire al valore della velocità angolare, $\dot{\theta}$, tramite la relazione:

$$\dot{\theta} = (\text{Vo-2.5})/0.00349$$

In questo modo una tensione di 0.5 Volt indica una velocità angolare di $-573°/s$, mentre una tensione di 4.5 Volt indica una velocità angolare di $+573°/s$. Il punto neutro sarà a circa 2.5 Volt ed indicherà ovviamente una velocità angolare pari a 0.

3.4 Risultati ottenuti

Il sistema alternativo in cui il sensore del giroscopio GY401 è interfacciato direttamente all'NXT che si occupa di campionare il segnale proveniente dal sensore stesso ha permesso di superare il problema relativo alla differenza di misurazione tra la scala oraria e quella antioraria.

Infatti, avendo eliminato il circuito di gestione del sensore non è più stato necessario individuare il corretto punto neutro, essendo questo fornito direttamente dal sensore.

Tuttavia, tale sistema è afflitto da un'altra tipologia di problema. A causa dell'elevato range di funzionamento del sensore ed avendo a disposizione un intervallo di 4 Volt in uscita, una rotazione di un grado corrisponderà ad una variazione di 3.49 mV.

Dal momento che il convertitore A/D del coprocessore ha una risoluzione di 10 bit, e può lavorare tra 0 Volt e 5 Volt, si deduce che è possibile misurare variazioni minime di 4.8 mV e di conseguenza, in teoria, si dispone di una precisione pari a 1.4 gradi/s circa.

Questo valore è notevolmente ridotto se si considera il rumore, che stando alle specifiche può arrivare fino a 15 mV; questo significa che potremmo non essere in grado di rivelare variazioni anche 4 gradi.

Questa imprecisione è assolutamente accettabile per lo scopo per il quale il giroscopio è stato progettato, e cioè di stabilizzare la coda di piccoli elicotteri RC, in quanto variazioni nell'ordine di qualche grado sono costantemente presenti a causa di piccole vibrazioni o flussi d'aria generati dal movimento del rotore principale.

Per il nostro scopo invece non essere in grado di rivelare una variazione di qualche grado comprometterebbe l'intero progetto, ed è per questo che l'idea di utilizzare il giroscopio GY401 è stata abbandonata.

CAPITOLO 4

Applicazioni sperimentali: Una piattaforma inerziale

In questo capitolo verrà descritto ... **DA COMPLETARE**

4.1 Materiali e metodi

4.1.1 Analog Devices ADXRS150

Il sensore giroscopico ADXRS150, ad un asse, prodotto dalla Analog Devices, è un dispositivo integrato completo capace di rilevare velocità angolari comprese in un range di ($\pm 150°/s$).

Questo giroscopio produce una variazione di tensione ai capi delle sue uscite proporzionale alla velocità angolare. Tale tensione, come per il CRS03, può variare tra 0.5 Volt e 4.5 Volt.

Si tratta di un dispositivo costruito con tecnologia Silicon Micro Machine (SMM), costituito da strutture polisiliconiche sensibili ed, in questo caso, fornito su una basetta prodotta dalla Spark Fun nella quale sono presenti i resistori ed i condensatori necessari per il corretto funzionamento.

Tra le connessioni disponibili è presente anche una uscita analogica il cui segnale varia in funzione della temperatura; tale uscita può essere utilizzata per un'accurata calibrazione del drift del giroscopio. La variazione del segnale in funzione della temperatura è tipicamente di 8.4 $mV/°C$.

Il sensore è dotato anche di un ingresso per l'aggiustamento del punto neutro (che tipicamente è intorno 2.5 Volt) e di due piedini attraverso i quali è possibile eseguire il *self–test* del sensore.

Inoltre tramite una resistenza esterna il range di funzionamento del sensore può essere incrementato fino ad una maggiorazione del 50% del suo valore di base, che però comporta una diminuzione della sensibilità ed un aumento del drift.

La corrente necessaria al funzionamento è decisamente inferiore, se confrontata con quella del CRS03, e si attesta sui 6 mA in condizioni di normale funzionamento. Tuttavia anche utilizzando questo sensore è necessario prestare particolare attenzione alla tensione con la quale questo viene alimentato per evitare che alte frequenze in ingresso sporchino il segnale.

In figura

4.1.2 MindSensors ACCL–Nx–V1

L'accelerometro a 3 assi, è un prodotto, distribuito dalla MindSensors, destinato ad essere utilizzato con il Robot NXT prodotto dalla Lego.

Per il trasferimento dati utilizza il protocollo di trasmissione I^2C nella sua prima versione, per questo motivo la velocità di trasferimento è al massimo di 9600 bit/s.

Il sensore utilizzato in questo dispositivo è l' ADXL-330, prodotto dalla Analog Devices che, a seconda dell'impostazione selezionata, è in grado di misurare su ciascun asse accelerazioni massime di $(\pm 2.5g)$,$(\pm 3.3g)$ oppure $(\pm 6.7g)$, dove g indica l'accelerazione gravitazionale.

Trattandosi di un sensore accelerometrico di tipo capacitivo, ed è in grado di percepire accelerazioni sia statiche gravitazionali, sia accelerazioni dinamiche dovute ad esempio a vibrazioni e movimenti. Queste differenti misurazioni sono rese disponibili attraverso la comunicazione I^2C mediante la lettura dei diversi registri del dispositivo. Infatti, questo accelerometro è dotato anche di un microcontroller PIC 16F819, prodotto dalla Microchip, che campiona il segnale del sensore accelerometrico e implementa una comunicazione I^2C di tipo Slave.

Inoltre l'accelerometro prevede anche una procedura di calibrazione manuale per ciascun asse ed una impostazione numerica dell'indirizzo I^2C.

In figura

4.1.3 Il controller

Ogni volta che un dispositivo deve mantenere costante un determinato valore, ad esempio una velocità o una temperatura serve un controller che corregga eventuali ed inevitabili errori rispetto al valore di consegna.

Un controller cercherà di correggere l'errore che avviene tra l'effettiva misurazione di un parametro ed il valore desiderato dello stesso eseguendo azioni correttive e valutandone il risultato.

Uno schema di funzionamento di un generico controller è illustrato in Figura 4.1.

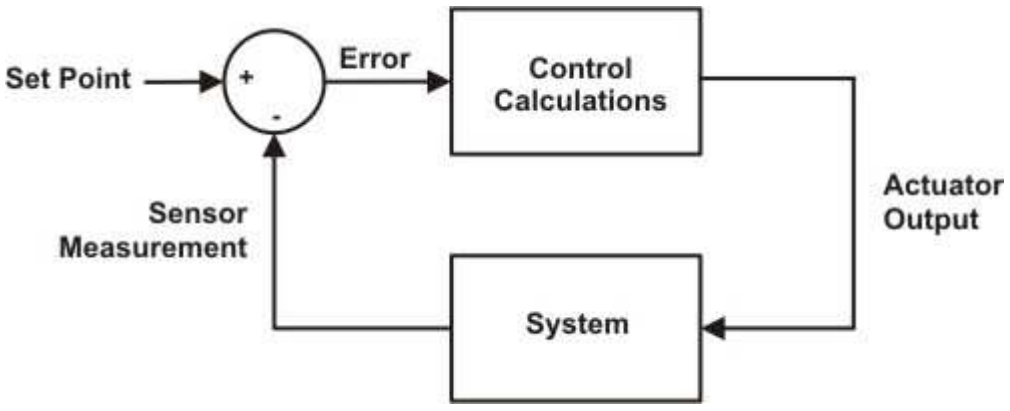

Figura 4.1: Controller generico

Lo schema descrive un processo che si ripete nel tempo in maniera indefinita. La misurazione di un sensore è confrontata con il valore desiderato, chiamato *Set point*, e viene determinato l'eventuale errore. Il controller effettua alcune operazioni per determinare il corretto output da inviare ad un generico attuatore, il quale avrà

un certo effetto sul sistema. A questo punto il sensore viene nuovamente interrogato per verificare i benefici della correzione ed il ciclo ricomincia.

Ci sono numerosi tipi di controllori che utilizzano differenti metodi per il calcolo dell'output. Uno dei più conosciuti ed utilizzati è proprio il Proportional Integral Derivative (PID).

Il PID è un metodo di controllo basato su un meccanismo di *feedback* ed è largamente utilizzato nei sistemi di controllo industriali.

L'algoritmo di un PID utilizza principalmente tre parametri: Il parametro Proporzionale, che determina la reazione necessaria per l'errore corrente, il parametro Integrale, che determina la correzione in base alla somma degli errori precedenti, ed il parametro Derivativo che determina la correzione proporzionalmente alla velocità con la quale l'errore cambia nel tempo. La somma pesata di questi tre parametri restituisce il valore necessario per la correzione dell'errore.

Di seguito le tre tipologie di correzione verranno brevemente illustrate.

Proportional

Il controllo proporzionale cerca di correggere il valore corrente dell'errore semplicemente moltiplicandolo per una data costante Kp. Ne deriva una reazione all'errore che è proporzionale allo stesso.

In Figura 4.2 è riportato uno schema del principio di funzionamento della parte proporzionale del PID.

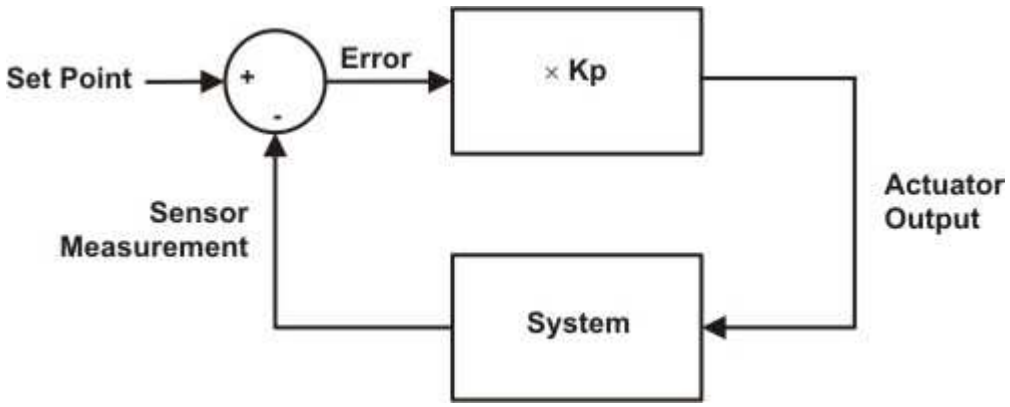

Figura 4.2: Correzione proporzionale

Vediamo ora, riportiamo in in Figura 4.3, un esempio pratico della correzione della parte proporzionale di un PID con un Kp=10. Nel grafico è mostrato l'andamento dell'errore rispetto al tempo, ed in particolare è evidenziata la correzione effettuata all'istante 2, con errore 3 e la correzione effettuata all'istante 7, con errore -2.

Il calcolo del fattore proporzionale è quindi:

$$P = Kp * error$$

Integral

Il problema principale dell'utilizzo della sola parte proporzionale è che essa non è in grado di correggere continue tendenze di crescita dell'errore, così è necessario introdurre una correzione integrale, la quale sarà anch'essa moltiplicata per una sua costante Ki.

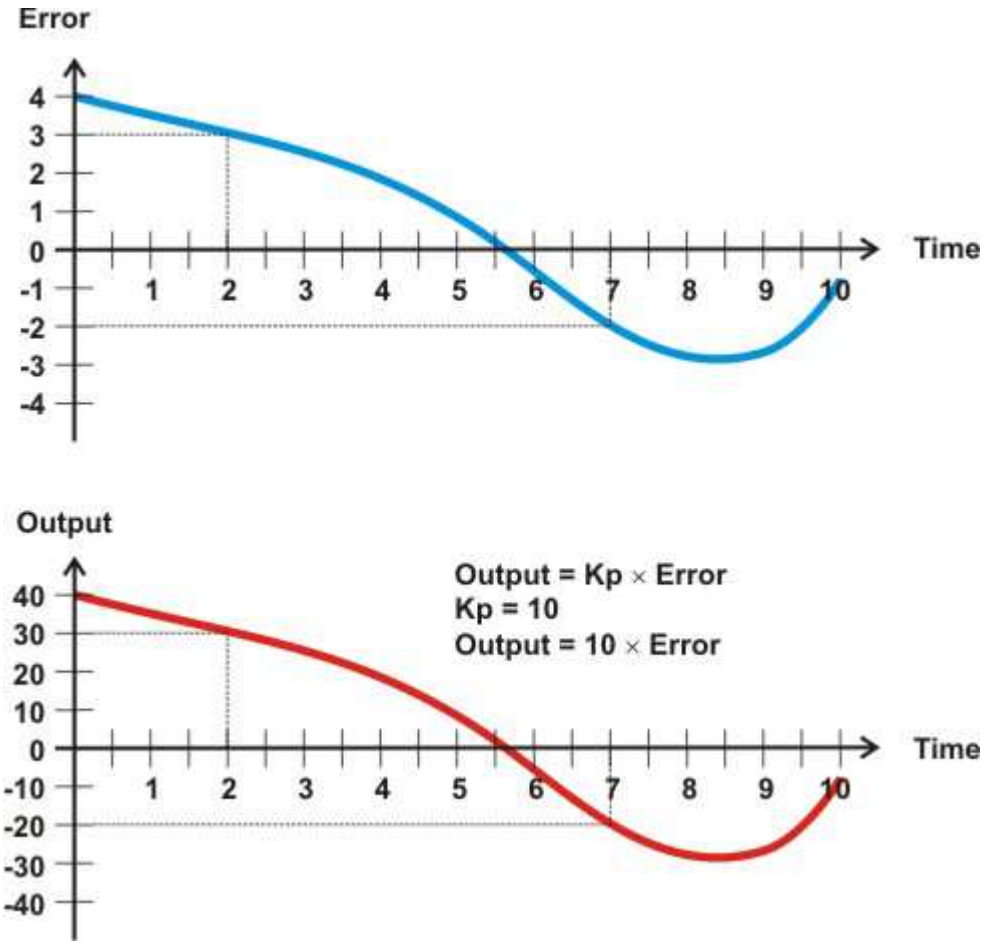

Figura 4.3: Andamento dell'errore e sua correzione proporzionale

Uno schema logico del funzionamento della parte integrale del PID è illustrato in Figura 4.4.

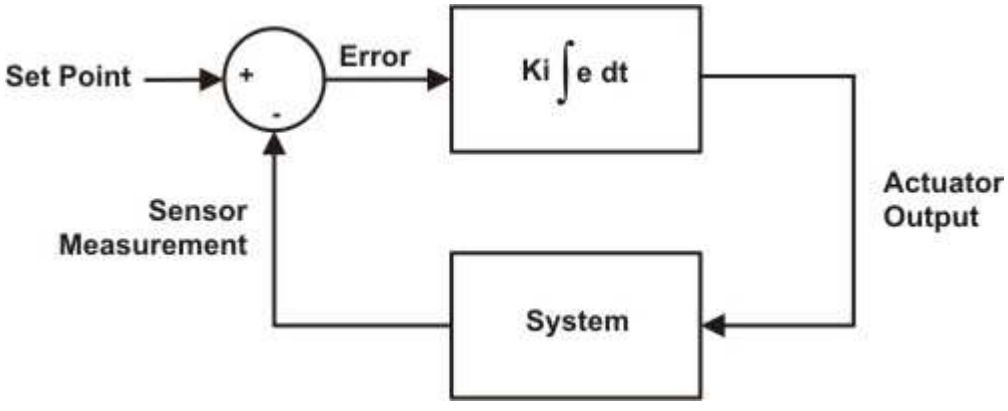

Figura 4.4: Correzione integrale

L'integrale misura l'area sottesa dalla curva rappresentante l'andamento dell'errore nel tempo, e l'asse dei tempi. Se nel tempo l'errore non torna mai a zero, l'area sottesa dal suo andamento temporale cresce in maniera indefinita, producendo una reazione all'errore proporzionale alla stessa e quindi sempre maggiore. In altre parole

il controller cerca di risolvere la lontananza dal set point effettuando una correzione crescente.

In Figura 4.5 è riportato un esempio di correzione integrale con Ki=10 ed errore fermo ad 1. Come si vede con il passare del tempo la correzione dell'errore tenderà a diventare sempre più grande in quanto l'area sottesa dalla curva rappresentante l'andamento dell'errore continua a crescere.

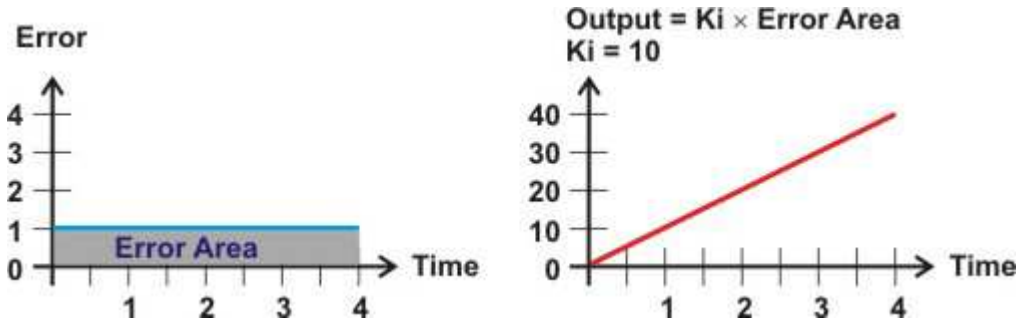

Figura 4.5: Andamento dell'errore e sua correzione integrale

Il modo più semplice per calcolare la correzione della parte integrale è attraverso il metodo di integrazione numerica. Dividendo l'area in piccoli intervalli di uguale durata, è possibile ottenere l'area dei singoli rettangoli moltiplicando il valore dell'errore istantaneo per la durata dell'intervallo. La somma di tutte le aree determinerà il valore cercato.

Per semplificare i calcoli è bene notare che è possibile considerare ciascun intervallo di tempo pari a 1, infatti dal momento che il valore dell'integrale andrà moltiplicato per una costante Ki, questa assunzione non influirà sul valore ottenuto.

Il calcolo dell'integrale risulta quindi essere:

$$\text{error(Accumulator)} = \text{error(Accumulator)} + \text{error(Current)}$$
$$\text{I} = \text{Ki} * \text{error(Accumulator)}$$

Derivative

Ci sono una serie di cambiamenti di errore che ne la parte proporzionale, ne quella integrale sono in grado di gestire correttamente. Si tratta di variazioni rapide, derivanti da fattori esterni al sistema, che producono altrettanto rapidi cambiamenti di errore.

Il compito della parte derivativa nel PID è proprio quello di opporsi a variazioni repentine del sistema reagendo in maniera altrettanto rapida.

In Figura 4.6 è riportato, in forma di schema, l'utilizzo della parte derivativa di un PID.

Anche in questo caso l'errore viene campionato periodicamente, ed il valore della derivata dell'errore rispetto al tempo (de/dt) è calcolato come la differenza tra due errori successivi divisa per la differenza tra gli istanti di tempo nei quali questi due errori sono stati misurati. Come già accennato per la parte integrale, per semplificare i calcoli è ovviamente possibile considerare l'intervallo di tempo di ogni misurazione uguale ad 1 variando opportunamente la costante Kd in modo tale da compensare questa scelta. In questo caso il calcolo della derivata si traduce in una semplice sottrazione:

$$\text{error(Delta)} = \text{error(Current)} - \text{error(Previous)}$$
$$\text{d} = \text{Kd} * \text{error(delta)}$$

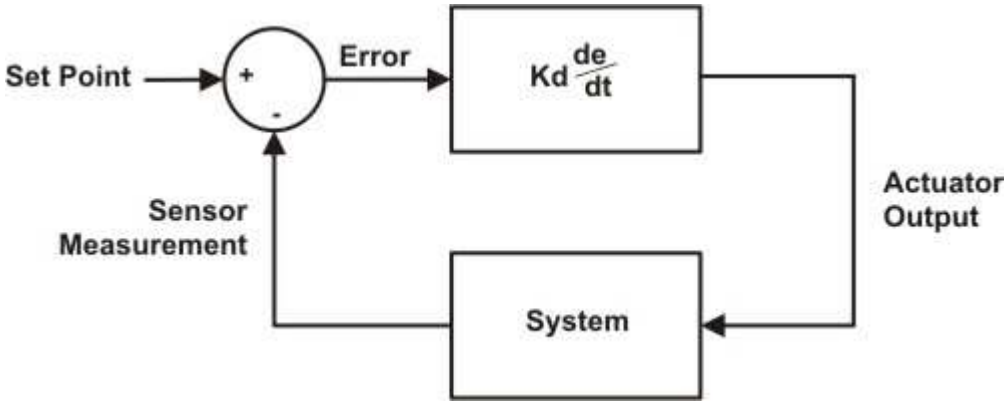

Figura 4.6: Correzione derivativa

Caratteristica fondamentale per il corretto funzionamento della parte derivativa è però che il segnale sia campionato ad una velocità superiore a quella con cui l'errore varia.

In Figura 4.7 è riportato l'andamento dell' errore e il corrispondente valore calcolato dalla derivata per ogni intervallo di tempo.

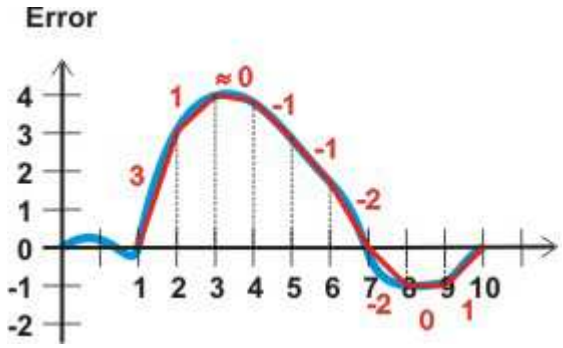

Figura 4.7: Andamento dell'errore e sua correzione derivativa

4.1.4 PID

Un controller PID utilizza tutte e tre le parti sopra analizzate per la correzione dell'errore. Il contributo di ciascuna delle tre potrà essere variato agendo sulle costanti Kp, Ki e Kd producendo differenti reazioni del sistema.

In Figura 4.8 è riportato lo schema di utilizzo contemporaneo della parte proporzionale, integrativa e derivativa di un controller PID.

4.1.5 Cascaded PID

In ultima analisi prendiamo in considerazione l'utilizzo di più PID in configurazione a cascata.

In questo caso la correzione prodotta dal primo PID viene utilizzata come set point di riferimento per il secondo controller. E' anche possibile usare un secondo parametro che sommato al valore calcolato dal primo PID fornisce il valore di riferimento per il secondo PID.

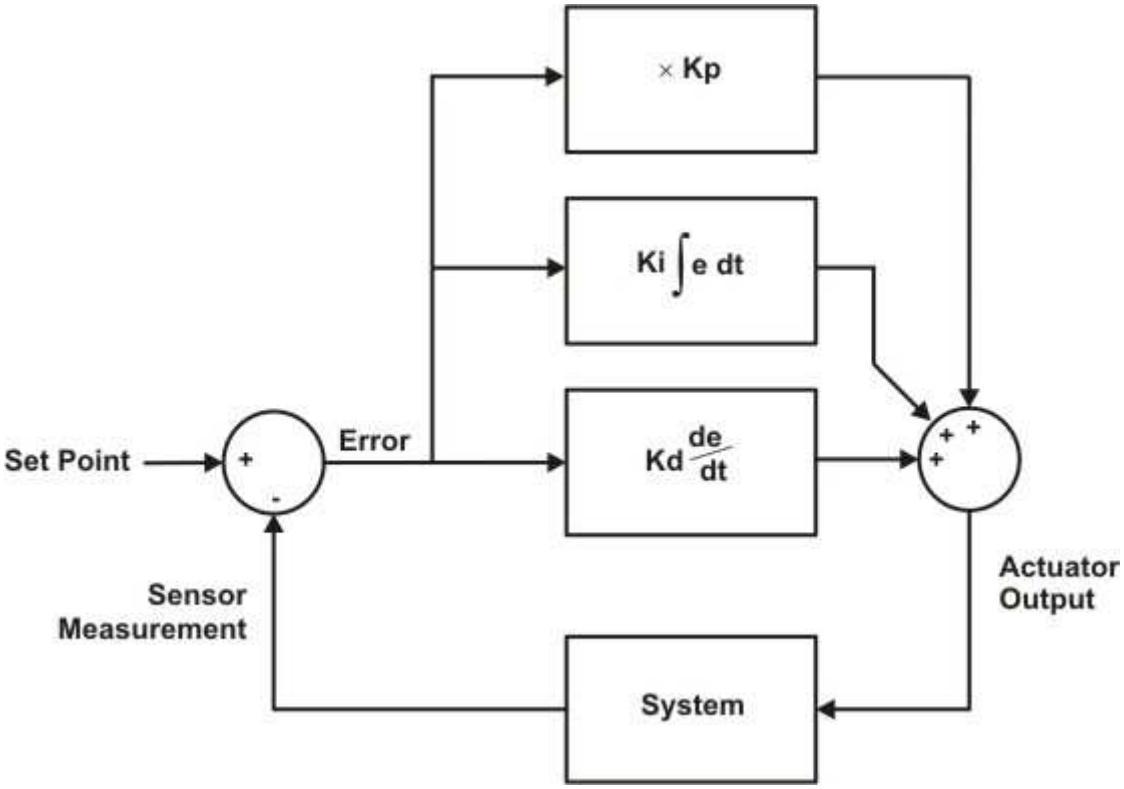

Figura 4.8: Utilizzo congiunto delle tre parti

Solitamente il controllo a cascata è utilizzato quando ci sono più parametri che influenzano il funzionamento di un sistema, come ad esempio posizione e velocità di un dispositivo, e l'utilizzo di questa configurazione comporta risultati dinamici migliori.

Se è vero che l'utilizzo di PID multipli in cascata è più difficile da configurare, in quanto i parametri considerati sono in numero maggiore, è altrettanto vero che questo approccio dimostra una maggior capacità di correzione degli errori in situazioni difficili, dove eventualmente, un unico PID non riesce ad intervenire correttamente.

4.1.6 Wii Remote Controller

Il Wii Remote Controller (Wiimote) è il dispositivo standard utilizzato per l'interazione dell'utente con la console Wii prodotta da Nintendo nel 2005.

La principale innovazione del Wiimote consiste nell'avere incorporati al suo interno una serie di strumenti che permettono un maggior coinvolgimento del giocatore attraverso la possibilità di interagire e manipolare dinamicamente oggetti presenti sullo schermo.

I sensori utilizzati all'interno del Wiimote sono due. Il primo è un accelerometro a tre assi, prodotto dalla Analog Devices. Nella maggior parte dei controller viene utilizzato il modello ADXR330, in grado di misurare accelerazioni pari a $\pm 3g$, mentre la comunicazione tra l'accelerometro ed il pad avviene internamente sfruttando il protocollo SPI nativo dell'ADXR330.

Il secondo è una telecamera Video Graphics Adapter (VGA) in tecnologia Complementary Metal Oxide Semiconductor (CMOS), che sfrutta il protocollo di comunicazione I^2C. Unitamente all'utilizzo di una barra a infrarossi posizionata in prossimità

del televisore, tale telecamera permette di stimare la rotazione del Wiimote sull'asse orizzontale fornendo un riferimento posizionale rispetto all'ambiente di gioco attraverso il metodo di triangolazione. La barra infrarossi infatti, è costituita da 10 led, 5 per ogni lato, aventi angolazioni leggermente differenti. Per questo l'utilizzo della sensor bar permette anche di stimare in maniera abbastanza precisa la distanza del Wiimote dal televisore.

Inoltre il dispositivo è dotato di: una memoria EEPROM da 16 Kbytes, di cui 6 KBytes possono essere utilizzate dall'utente per memorizzare informazioni, un piccolo altoparlante da 2.1 cm in grado di riprodurre suoni in formato 4-bit ADPCM, un micromotore utilizzato per produrre la vibrazione ed infine un integrato BCM2042, prodotto dalla Broadcom, usato per la connessione del dispositivo con la console Wii.

In particolare questo integrato instaura una connessione di tipo Bluetooth utilizzando il protocollo human interface device (HID) sviluppato dalla Microsoft, quindi è possibile una comunicazione diretta tra il Wiimote ed un PC che supporta il suddetto protocollo.

Il Wiimote ha una porta di espansione che permette l'utilizzo di accessori opzionali per un coinvolgimento sempre maggiore. Tra questi il *Nunchuk* è sicuramente il più utilizzato e permette di avere a disposizione una piccola cloche analogica ed un accelerometro aggiuntivo. Il protocollo di comunicazione utilizzato tra il Wiimote e le sue periferiche esterne è di tipo I^2C, in particolare la versione a 400 KHz che utilizza come indirizzo 0x52 in esadecimale. In Figura 4.9 è mostrato il Wiimote modello RVL–003.

Figura 4.9: Il Wii remote controller

4.1.7 Il Kalman Filter

Il Kalman Filter è un algoritmo ricorsivo che stima lo stato di un sistema dinamico.

Questo algoritmo è, per diversi aspetti, uno strumento potente, è infatti in grado di stimare lo stato passato, presente e futuro di un sistema anche quando non si dispone

di una conoscenza precisa del modello che lo descrive. L'algoritmo utilizza tutti i risultati di misura disponibili, senza tenere conto della loro precisione, per stimare il valore corrente di una variabile d'interesse attraverso la conoscenza del sistema e degli strumenti di misura, l'andamento statistico del rumore associato al sistema, l'errore sulle misurazioni, l'incertezza sul modello dinamico e qualunque informazione disponibile sulle condizioni iniziali della variabile d'interesse. Il Kalman Filter combina tutti i dati di misura disponibili, con la conoscenza a priori dello stato del sistema e dei dispositivi di misura per produrre una stima della variabile desiderata in cui l'errore statistico è minimizzato

Il Filtro risolve il problema generale della stima dello stato $x_k \in R^n$ di un sistema sottoposto a controllo, descritto come un sistema dinamico lineare discretizzato nel dominio del tempo:

$$x_k = Ax_{k-1} + Bu_{k-1} + w_{k-1} \tag{4.1}$$

attraverso una misura sperimentale di tale stato:

$$z_k = Hx_k + \nu_k \tag{4.2}$$

dove:

- A è una matrice, $n \times n$, di transizione di stato, dallo stato x_{k-1} al successivo stato x_k.
- B è una matrice, $n \times l$, che rappresenta il modello di controllo applicato al vettore di controllo $u \in R^l$.
- w_k è il rumore attribuito al processo di stima , avente una distribuzione normale con media nulla e covarianza Q.
- H è una matrice, $n \times m$, che descrive la relazione esistente tra la misura sperimentale dello stato z_k e la sua stima x_k.
- ν_k è il rumore attribuito alla misura, avente una distribuzione normale con media nulla e covarianza R.

In pratica lo stato di un sistema è rappresentato da un vettore di n numeri reali. Ad ogni incremento discreto del tempo, un operatore lineare (la matrice A) è applicato allo stato corrente, k-1, per generare il nuovo stato, k, aggiungendo rumore (w_k) ed eventualmente informazioni sul controllo del sistema (la matrice B).

In seguito un secondo operatore lineare aggiunge ulteriore rumore per generare la stima 'corretta' dello stato.

Poichè il filtro di Kalman basa la stima dello stato di un sistema sulla ricorsione, per avere una stima dello stato corrente è necessario conoscere solamente la stima dello stato precedente ed una misura dello stato corrente. Di seguito, vedremo, in breve, i passaggi matematici utilizzati dal filtro a tal fine.

Con la notazione $\hat{x}_{n|m}$, indicheremo la stima dello stato x in dato istante n (nel quale è avvenuta la misura) valida fino all'istante successivo m. Per ogni passo ricorsivo, il filtro descrive il sistema attraverso due variabili:

- $\hat{x}_{k|k}$ che rappresenta la stima del sistema all'istante k, data dall'osservazione all'istante k

- $P_{k|K}$ che rappresenta la matrice di covarianza dell'errore, in altre parole una misura dell'accuratezza della stima dello stato

Le operazioni eseguite dal filtro ricorsivo, possono essere divise in due *step* fondamentali: il primo è la **previsione**; il secondo l'**aggiornamento** (vedi Figura **??**).

La fase di previsione usa la stima dello stato precedente per produrre una stima dello stato corrente, secondo le relazioni:

$$\hat{x}_{k|k-1} = A\hat{x}_{k-1|k-1} + Bu_{k-1} \tag{4.3}$$

$$P_{k|k-1} = AP_{k-1|k-1}A^T + Q \tag{4.4}$$

Nella fase di aggiornamento, l'informazione sullo stato corrente proveniente dalla misura, z_k, è utilizzata per migliorare la stima data dalla relazione 4.3, ottenendo una stima più accurata. Viene, quindi, calcolata l'*innovazione*, cioè la differenza tra il valore appena stimato e la misura, mediante le relazioni:

$$\nu_k = z_k - H\hat{x}_{k|k-1} \tag{4.5}$$

$$S_k = HP_{k|k-1}H^T + R \tag{4.6}$$

Il guadagno del filtro:

$$K_k = P_{k|k-1}H^T {S_k}^{-1} \tag{4.7}$$

ed infine la stima dello stato corrente e l'errore su tale stima, vengono aggiornati, secondo le relazioni:

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k\nu_k \tag{4.8}$$

$$P_{k|k} = (I - K_kH)P_{k|k-1} \tag{4.9}$$

Nel paragrafo seguente, daremo un'idea di come il Kalman Filter sia stato adattato per gli scopi di questo lavoro di Tesi.

4.1.8 Applicazioni del Kalman Filter

Nel nostro caso il Kalman Filter è utilizzato per ottenere una stima dell'angolo θ di tilt del Robot attraverso la conoscenza di due misure, la velocità angolare, $\frac{d\theta}{dt}$, data dal giroscopio e l'accelerazione, $\frac{d^2\theta}{dt^2}$ data dall'accelerometro.

Il dato del giroscopio, verrà utilizzato per la previsione del modello, secondo l'equazione lineare:

$$x_{k+1} = Ax_k + Bu_k \tag{4.10}$$

che, nel caso specifico, diventa:

$$\begin{pmatrix} \theta \\ bias \end{pmatrix}_{k+1} = \begin{pmatrix} 1 & -dt \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \theta \\ bias \end{pmatrix}_k + \begin{pmatrix} dt \\ 0 \end{pmatrix} u_k \tag{4.11}$$

dove:

- la variabile u_k rappresenta l'input di ogni passo ricorsivo, cioè il dato proveniente dal giroscopio
- θ è la variabile di cui desideriamo avere una stima
- bias è il valore dell'offset del giroscopio
- le matrici A e B sono scelte in modo che il modello stimi la variabile θ attraverso l'integrazione della velocità angolare misurata dal giroscopio.

Eseguendo infatti il prodotto matriciale (righe x colonne) otteniamo:

$$\theta_{k+1} = \theta_k - biasdt + u_k dt \tag{4.12}$$

che con un semplice passaggio diventa:

$$\theta_{k+1} = \theta_k + (u_k - bias)dt \tag{4.13}$$

Come è possibile notare, nel modello per la stima dell'angolo θ l'offset del giroscopio (bias) rimane costante durante tutti i passi ricorsivi del modello, in realtà il valore di questo bias è caratterizzato da un drift che rende la nostra stima imprecisa. Il Kalman Filter utilizza l'informazione proveniente dall'accelerometro per aggiornare/migliorare la stima di θ data dalla prima fase di previsione.

Indichiamo con y il dato proveniente dall'accelerometro, e vediamo gli step eseguiti dal Kalman Filter nella fase di aggiornamento. Viene calcolata l 'innovazione, cioè la differenza tra il valore θ della stima ed il valore y dato dalla misura, secondo la relazione:

$$Err = y - C\theta \tag{4.14}$$

Indicando con P la covarianza della stima (che ricordiamo fornisce una misura dell'affidabilità della stessa), viene calcolata l'innovazione sulla covarianza, secondo la relazione:

$$s = CPC^T + S_z \tag{4.15}$$

ed il guadagno del filtro:

$$K = APC^T s^{-1} \tag{4.16}$$

Infine la stima dell'angolo e la relativa covarianza vengono aggiornate:

$$\theta = \theta + KErr \tag{4.17}$$

$$P = APA^T - KCPA^T + S_w \tag{4.18}$$

La matrice C è data dalla relazione:

$$\theta = C\theta$$

quindi nel nostro caso C = (1 0). S_z è la covarianza sulla misura dell'accelerometro, quindi nel nostro caso è l'oscillazione della misura intorno al valore dato, mentre S_w è una matrice 2x2 di covarianza della stima. **DA COMPLETARE**

4.1.9 Utilizzo della piattaforma inerziale

Nel nostro progetto l'angolo Θ corrispondente allo scostamento dell'NXT dal punto di equilibrio viene calcolato con l'ausilio del Kalman filter per ottenere una piattaforma di misurazione inerziale.

L'uso di questo algoritmo nasce dall'esigenza di utilizzare le informazioni fornite dal giroscopio e dall'accelerometro in maniera congiunta, limitando per ognuna delle due componenti, l'errore fornito dal sensore insieme alla reale misurazione. Un giroscopio infatti, restituisce la velocità angolare misurata molto velocemente e la sua misurazione non è influenzata dalle accelerazioni causate dal movimento del robot, ma è soggetto a fenomeni come il drift che con il passare del tempo rendono necessaria una nuova calibrazione del punto neutro. L'accelerometro dal canto suo, è più lento nel funzionamento, ma produce una misurazione più precisa e veritiera sull'angolo di tilt. Inoltre quest'ultimo non è soggetto al drift e quindi in via teorica non necessita di una ricalibrazione durante l'utilizzo. Al contrario però l'informazione dell'accelerometro non può essere utilizzata da sola per il calcolo dell'angolo di inclinazione in quanto il sensore non è in grado di distinguere l'accelerazione gravitazionale da una qualunque altra alla quale è sottoposto, come ad esempio quella prodotta dai motori per spostare il robot.

Detto questo quindi, è bene precisare che è possibile mantenere in equilibrio un robot usando solo un giroscopio, ma non è possibile farlo usando solo un accelerometro.

Nel caso di utilizzo del solo giroscopio probabilmente, se non viene accuratamente gestito il drift, l'angolo di tilt utilizzato per la correzione diverrà presto errato, rendendo in breve tempo impossibile la correzione dell'errore. Nello specifico di questo progetto, le informazioni ricevute dal giroscopio e dall'accelerometro vengono passate al Kalman filter, il quale utilizza un grado di fiducia per ognuno di essi, impostato in fase di calibrazione.

Come precedente specificato, per ogni intervallo di tempo, l'informazione ottenuta dal giroscopio sarà la prima ad influenzare l'angolo di output restituito dal Kalman filter. Passata una certa quantità di tempo di un singolo intervallo però, la fiducia data al giroscopio tenderà a diminuire, mentre quella dell'accelerometro aumenterà. Per questo motivo l'angolo theta comincerà a convergere, in una parte proporzionale al grado di fiducia impostato, al valore restituito dall'accelerometro. Questo processo si ripeterà per ogni intervallo di tempo nel quale è invocato l'utilizzo del Kalman filter. **DA VERIFICARE**

4.2 Controllo del bilanciamento

Per riuscire a mantenere in equilibrio l'NXT abbiamo usato una piattaforma inerziale costituita dai due sensori sopra descritti: Il giroscopio ADXRS150 e l'accelerometro ACCL–Nx–V1. L'informazione dei due sensori viene fusa attraverso l'uso del Kalman filter che provvede a restituire l'angolo θ di allontanamento dalla posizione voluta di equilibrio. In prima istanza il programma di gestione a bordo dell'NXT provvede a disabilitare il sistema di regolazione dei motori integrato in leJOS. Questo infatti è risultato troppo lento per riuscire a gestire le rapide correzioni necessarie per il mantenimento della posizione verticale del robot. In più viene liberata una certa quantità di risorse in seguito alla chiusura dei thread di gestione dei motori. L'intervallo di tempo utilizzato per le correzioni è stato impostato a 18 mS. Questo periodo è risultato un buon compromesso tra l'utilizzo di alcune variabili di tipo float per avere una maggiore precisione ed il tempo necessario per garantire una pronta risposta dei motori ai cambiamenti dell'angolo θ. L'informazione del giroscopio viene campionata direttamente attraverso il convertitore A/D a 10 bit integrato nell'NXT, e il valore restituito viene trasformato nella reale velocità angolare attraverso la seguente formula, dove u è il valore restituito dal convertitore:

$$\text{Vo} = (\text{u}/1023)*5$$

$$\dot{\theta} = (\text{Vo-}2.5)/0.0133$$

Ricordando che il convertitore integrato ha una risoluzione pari a circa 4.88 mV per valore e che il giroscopio ha una risoluzione di circa 12.5 mV/grado al secondo, si conviene che utilizzando questo tipo di sensore si avrà la possibilità di misurare anche velocità angolari nell'ordine di 0.39 gradi al secondo. Il rumore intrinseco del sensore ha dimostrato di mantenersi su livelli davvero molto bassi, andando ad influenzare la misurazione in maniera del tutto trascurabile.

L'informazione dell'accelerometro invece viene recuperato tramite due letture di tipo I^2C, una per il valore dell'accelerazione sull'asse x, e l'altra su quello dell'asse z. Le informazioni relative ai due assi permettono il calcolo di tilt effettuato attraverso la

funzione atan2 facente parte della libreria Math di Java. Questo metodo è stato necessario dal momento che l'accelerometro non è fissato sul punto di rotazione del robot, quindi durante una rotazione dello stesso è presente anche una parte di accelerazione sull'asse verticale. A questo punto il Kalman filter provvede a fondere i valori dei due sensori restituendo secondo il metodo descritto sopra, l'angolo θ.

Altre due informazioni però sono essenziali per il bilanciamento del robot: La sua posizione rispetto al terreno e la sua velocità. La prima si ottiene utilizzando il tachimetro integrato nei motori, e la seconda semplicemente derivando la prima informazione in un intervallo di tempo. Ciò corrisponde in pratica ad effettuare una differenza di due posizioni relative a due intervalli di tempo differenti.

A questo punto tutte le informazioni necessarie sono disponibili, ed è quindi possibile inviarle ai due PID utilizzati per mantenere l'equilibrio del robot. Il primo PID ha il compito di correggere eventuali errori sulla velocità rilevata dell'NXT. Il set point utilizzato per questo PID è lo 0 per un completo stazionamento del robot durante il bilanciamento. Il valore di correzione prodotto dal primo PID viene quindi utilizzato congiuntamente all'angolo θ calcolato dal Kalman, come set point per il secondo PID, il quale provvederà a calcolare la potenza necessaria da dare ai motori per stabilizzare il robot. **DA VERIFICARE**

4.3 Controllo remoto del Robot attraverso il Wiimote

Allo scopo di permettere lo spostamento del robot da remoto è stato utilizzato il Wiimote descritto nel paragrafo 4.1.6. Questo dispositivo però non è in grado di comunicare direttamente con l'NXT, in quanto quest'ultimo non supporta il protocollo HID che è invece il motodo utilizzato dal Wiimote nella trasmissione dati. Per questo motivo è stato necessario interporre tra i due dispositivi un computer, con lo scopo di catturare i segnali del pad e ritrasmetterli al robot tramite una comunicazione bluetooth seriale. Il software sviluppato per il computer utilizza il linguaggio Java, e risulta quindi adatto per la maggior parte dei sistemi operativi. La sua implementazione utilizza le librerie wiiuse, sviluppate in C e disponibili con licenza Open Source, tramite un *wrapper* Java, il wiiuseJ, che tramite l'utilizzo di alcune API permette di accedere a quasi tutti gli eventi generati dal Wiimote ed il loro utilizzo in applicazioni Java. In questo progetto di Tesi, vengono utilizzati solo gli eventi relativi alla posizione del pad, ed in particolare le accelerazioni rivelate sull'asse di pitch e sull'asse di roll che servono rispettivamente a determinare l'avanzamento e la rotazione del robot.

La rotazione dell'NXT avviene attraverso l'invio, di segno opposto per i due motori, del valore di rotazione generato dal Wiimote ai motori del robot. Il fatto di ruotare gli stessi nelle due direzioni, ma in maniera opposta, non influisce sul mantenimento del bilanciamento.

Per quanto riguarda l'avanzamento invece, il valore catturato dal Wiimote sull'asse di pitch viene sommato all'errore di velocità utilizzato come set point del primo PID.

Mentre la rotazione del robot non implica problemi si stabilità, l'avanzamento è più critico e durante i test effettuati ha dimostrato di non gradire repentini cambi di velocità. **DA VERIFICARE**

4.4 Risultati e discussione

Questa implementazione ha dimostrato di poter mantenere in equilibrio l'NXT su qualunque superficie e per un tempo indefinito attraverso una precisa taratura delle costanti moltiplicative del PID. Queste costanti sono state trovate in maniera sperimentale, attraverso il metodo di tipo trial and error e sono state impostate come segue: xxx,yyy,zzz,kkk,ppp,qqq.

Per velocizzare le operazioni di taratura, attraverso il software che permette la comunicazione del Wiimote con l'NXT, e che come ricordiamo deve essere eseguito su un computer, è stato possibile inviare valori numerici al robot per la modifica in tempo reale delle sei costanti relative ai due PID utilizzati. In generale la taratura è avvenuta attraverso i seguenti passi: Per prima cosa si è cercata la costante per la parte proporzionale del PID, quindi dopo aver disabilitato le parti integrativa e derivativa si è aumentato il valore di Kp finchè il sistema non ha cominciato ad oscillare. A questo punto tramite la parte derivativa si è trovata la costante che inibisse le oscillazioni procurate dalla parte proporzionale. Infine si è trovata la parte integrale eventualmente ritoccando quella derivativa.

I fattori che influenzano la calibrazione delle costanti sono numerosi; In primo luogo la superficie sulla quale il robot si trova che causa un cambiamento sull'attrito volvente e questo va ad influire sulla potenza necessaria per mantenere in piedi il robot. Altro fattore importante è lo stato di carica delle batterie. Man mano che queste si scaricano la potenza necessaria aumenta, e così anche le costanti del PID.

Una futura implementazione dovrebbe tenere in considerazione anche questo aspetto, cercando in maniera sperimentale un fattore per compensare la scarica non lineare delle batterie.

Una versione futura poi, potrebbe utilizzare il controllo di temperatura integrato nel giroscopio, essendo questa altra causa di cambiamenti nella misurazione dello stesso e di conseguenza nella stabilità dell'NXT.

Senza ombra di dubbio comunque i motori hanno dimostrato di essere il punto debole della catena, non tanto per la ridotta velocità che non permette il recupero dell'equilibrio una volta sorpassato un certo grado di inclinazione, ma piuttosto per la scarsa precisione dimostrata e per il grande gioco meccanico (backlash) presente all'interno degli ingranaggi. Questo alle volte infatti non permette al motore di fermarsi esattamente nella posizione voluta, causando una nuova correzione da effettuare. Questo difetto è ulteriormente amplificato dall'aver utilizzato ruote abbastanza grandi, ma che d'altro canto hanno dimostrato una maggiore velocità di correzione rispetto a quelle standard.

Ulteriore miglioramento potrebbe essere dato dall'utilizzo del Kalman filter all'interno di un microcontroller come il 16F876, per liberare risorse sull'NXT ed effettuare magari un maggior numero di calcoli in virgola mobile per ottenere una precisione ancora maggiore e di conseguenza una migliore stabilità. **DA VERIFICARE**

APPENDICE A

Acronimi

PPM Pulse Position Modulation
leJOS Lego Java Operating System
RC Radio Controllato
I^2C Inter–Integrated–Circuit
SDA Serial Data
SCL Serial Clock
MEMS Micro Electronic Mechanical Systems
PWM Pulse Width Modulation
LVP Low Voltage Programming
MSE Microsystems Engineering
CCP Capture, Compare, PWM
SMM Silicon Micro Machine
PID Proportional Integral Derivative
HID human interface device
API Application Programming Interface
RISC Reduced Instruction Set Computer
PIC Programmable Interface Controller
VGA Video Graphics Adapter
CMOS Complementary Metal Oxide Semiconductor
AVCS Angular Velocity Control System
NXT Lego Next
LCD Liquid Crystal Display
EEPROM Electronically Erasable Programmable Read Only Memory
POR Power On Reset
PWRT Power Up Timer
OST Oscillator Start Up Timer
WDT Watchdog Timer
Wiimote Wii Remote Controller

APPENDICE B

Codice

B.1 NXT: Codice del bilanciamento

```
public class GyroTest {

    // Update interval for the main control algorithm
    private static final int BalanceInterval = 18;
    // Scale factor used for PIDs
    private static final int PIDScale = 100;
    // Accelerometer offset
    private static final float AccOffset = 1.86f;
    // Accelerometer inaccuracy
    private static final float AccInaccuracy = 1.19f;
    // Fallen angle
    private static final int FallAngle = 300;
    // Constant to mantain motors in sync
    private static final int motorSync = 2;

    // Balancing PID constants
    static int kAng = 0;
    static int kAngRate =0;
    static int kIntAng =0;

    // Motion PID constants
    static int SkP =0;
    static int SkI =0;
    static int SkD =0;

    // Driving constants
    static int desiredSpeed=0;
    static int turn = 0;
```

```
public static void main(String[] args) throws Exception {

  // Balancing PID values
        int P=0;
        int I=0;
        int D=0;
        int iInst=0;

        // Motion control PID values
        int SP=0;
        int SI=0;
        int SD=0;
        int SInst=0;
        int SoldSpeed=0;

        // Driving values
        int motorOffset=0;
        int turnOffset=0;
        int prevPos = 0;
        int torque = 0;
        int prevTorque = 0;
        int lastTorque = 0;
        int SpeedError=0;
        int tune= 5;

        // Time values
        int dt;
        int OnemS = 1;
        boolean buttonDown = false;

        // General PIDs semaphores
        boolean go = false;

        // Motors planning
        Motor m1 = Motor.A;
        Motor m2 = Motor.B;
        m1.smoothAcceleration(false);
        m2.smoothAcceleration(false);
        m1.regulateSpeed(false);
        m2.regulateSpeed(false);
        m1.setPower(0);
        m2.setPower(0);
        m1.forward();
        m2.forward();
        m1.shutdown();
        m2.shutdown();
```

```
        m1.resetTachoCount();
        m2.resetTachoCount();

        // Start Kalman Filter
        TiltFilter filter = new TiltFilter();

        // Get startime
        int startTime = (int)System.currentTimeMillis() - BalanceInterval;

        // Get Gyro and Accelerometer
        GyroSensor gyro = new GyroSensor(SensorPort.S1);
        TiltSensor acc = new TiltSensor(SensorPort.S4);

        // Start Wiimote and PID values scanner
        new Scanner();

        Thread.sleep(100);

        // Ready to go
        Sound.twoBeeps();

while (!Button.ESCAPE.isPressed())
    {
         // Get Balancing time
dt = (int) System.currentTimeMillis() - startTime;
startTime += dt;

         // Get accelerometer x and z value in mg
// then calculate atan2 to obtain theta in radians
int accxVal = acc.getXAccel();
int acczVal = acc.getZAccel();
float accVal=((float) Math.atan2( -acczVal, accxVal ))*AccInaccuracy+AccOffset;

// Get gyro value in degree per sec
float gyroVal = -((((float)gyro.readValue())/1023*5f - 2.5f)/0.0133f);

         // Kalman expects rad/sec, so convert with 2*PI/360 * gyroVal
gyroVal=(float) Math.toRadians(gyroVal);

// Update kalman with gyro value and Dt in seconds
filter.state_update(gyroVal, (float)dt/1000);

// Update kalman with acc value
         filter.kalman_update(accVal);

         // Get Angle and Rate from Kalman filter in decimal degree to allow int math
         int angle = (int)((filter.get_kalman_angle())*10) - tune;
```

```
 int rate =(int)((filter.get_kalman_rate())*10);

 // Filter out gyro noise until 1.5 degree/sec
 if (rate >-15 && rate<15)
 rate=0;

 // Get actual NXT position and speed
int pos = ((m1.getTachoCount() + m2.getTachoCount())/2 );
int speed = pos - prevPos + desiredSpeed;
 prevPos = pos;

 // Find motor offset
 if(turn==0)
 motorOffset= m1.getTachoCount() - m2.getTachoCount()-turnOffset;
 else {
 motorOffset=0;
 turnOffset=m1.getTachoCount() - m2.getTachoCount();
 }

// Calculate PIDs value
if(go==true){

 // 1ř PID speed error control
 // P is current error (speed)
 // D is derivtive speed
 // I is sum of error of previous iterations
 SP=speed*SkP;
 SInst=speed*SkI;
 SI=SI+SInst;
 // Limit integral error
 if(speed==0)
 SI=0;
 SD=SkD*(speed-SoldSpeed);
 SoldSpeed=speed;
 SpeedError=(SP+SI+SD)/PIDScale;
 angle=angle-SpeedError;

 // 2ř PID angle and speed error control
 // P is current error (angle+speed)
 // D is actual angular speed (rate)
 // I is sum of error of previous iterations (intangle)
 P=kAng*angle;
 iInst=angle*kIntAng;
 D=kAngRate*rate;
 torque=(P+I+D)/PIDScale;

 // Limit power to 100 or accumulate intergral error
```

```
    if(torque>100)
    torque=100;
    else
    if (torque<-100)
    torque=-100;
    else
    I=I+iInst;
}

    // Turn off motors if NXT has fallen over
    if (angle > FallAngle || angle < -FallAngle)
        torque = 0;

    // Smooth torque from one power setting to the next
    torque = (torque*1 + prevTorque*3)/4;
    prevTorque = torque;

    // Limit backlash problem in the gear system
    if (torque != 0 && ((lastTorque ^ torque) & 0x80000000) != 0)
    {
        lastTorque = torque;
         int t = (torque < 0 ? -20 : 20);
        m1.setPower(-t);
        m2.setPower(-t);
        Thread.sleep(OnemS);
    }

    // Let PIDs run
    if (Button.ENTER.isPressed())
 go=true;


    // Tuning of rest angle
    if (Button.LEFT.isPressed())
    {

        if (!buttonDown){
        tune=tune-1;
        }
        buttonDown = true;
    }
    else if (Button.RIGHT.isPressed())
    {
        if (!buttonDown) {
        tune=tune+1;
        }
        buttonDown = true;
```

```
          }
          else
              buttonDown = false;

         // Balanace NXT
        m1.setPower(-torque-motorSync*motorOffset + turn);
         m2.setPower(-torque - turn);

         // Sleep for the remaining control interval
         Thread.sleep(startTime + BalanceInterval - (int)System.currentTimeMillis());
      }
}
}
```

B.2 NXT: Codice del Kalman Filter

```
public class TiltFilter
{

     *
     *  This file is part of the autopilot onboard code package.
     *
     *  Autopilot is free software; you can redistribute it and/or modify
     *  it under the terms of the GNU General Public License as published by
     *  the Free Software Foundation; either version 2 of the License, or
     *  (at your option) any later version.
     *
     *  Autopilot is distributed in the hope that it will be useful,
     *  but WITHOUT ANY WARRANTY; without even the implied warranty of
     *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
     *  GNU General Public License for more details.
     *
     *  You should have received a copy of the GNU General Public License
     *  along with Autopilot; if not, write to the Free Software
     *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
     *
     */


    /*
     * Our update rate.  This is how often our state is updated with
     * gyro rate measurements.  For now, we do it every time an
     * 8 bit counter running at CLK/1024 expires.  You will have to
     * change this value if you update at a different rate.
     */
//static const float dt = ( 1024.0 * 256.0 ) / 8000000.0;
```

```
//static const float dt = (1.0/50); // 50 hz
    /*
     * Our covariance matrix.  This is updated at every time step to
     * determine how well the sensors are tracking the actual state.
     */
     float[][] P = {
        {1, 0},
        {0, 1}};

    /*
     * Our two states, the angle and the gyro bias.  As a byproduct of computing
     * the angle, we also have an unbiased angular rate available.   These are
     * read-only to the user of the module.
     */
     float angle = 0.0f;
     float q_bias = 0.0f;
     float rate = 0.0f;
    /*
     * R represents the measurement covariance noise.  In this case,
     * it is a 1x1 matrix that says that we expect 0.3 rad jitter
     * from the accelerometer.
     */
     static final float R_angle = 0.3f;   //0.3f
    /*
     * Q is a 2x2 matrix that represents the process covariance noise.
     * In this case, it indicates how much we trust the acceleromter
     * relative to the gyros.
     */
     static final float Q_angle = 0.001f; //0.001f
     static final float Q_gyro = 0.003f;  //0.003f


    /*
     * state_update is called every dt with a biased gyro measurement
     * by the user of the module.  It updates the current angle and
     * rate estimate.
     *
     * The pitch gyro measurement should be scaled into real units, but
     * does not need any bias removal.  The filter will track the bias.
     *
     * Our state vector is:
     *
     * X = [ angle, gyro_bias ]
     *
     * It runs the state estimation forward via the state functions:
     *
     * Xdot = [ angle_dot, gyro_bias_dot ]
```

```
*
* angle_dot = gyro - gyro_bias
* gyro_bias_dot = 0
*
* And updates the covariance matrix via the function:
*
* Pdot = A*P + P*A' + Q
*
* A is the Jacobian of Xdot with respect to the states:
*
* A = [ d(angle_dot)/d(angle)     d(angle_dot)/d(gyro_bias) ]
*     [ d(gyro_bias_dot)/d(angle) d(gyro_bias_dot)/d(gyro_bias) ]
*
*   = [ 0 -1 ]
*     [ 0  0 ]
*
* Due to the small CPU available on the microcontroller, we've
* hand optimized the C code to only compute the terms that are
* explicitly non-zero, as well as expanded out the matrix math
* to be done in as few steps as possible.  This does make it harder
* to read, debug and extend, but also allows us to do this with
* very little CPU time.
*/
void state_update(
       final float q_m, /* Pitch gyro measurement */
       final float dt)
{
   /* Unbias our gyro */
   final float q = q_m - q_bias;

   /*
    * Compute the derivative of the covariance matrix
    *
    * Pdot = A*P + P*A' + Q
    *
    * We've hand computed the expansion of A = [ 0 -1, 0 0 ] multiplied
    * by P and P multiplied by A' = [ 0 0, -1, 0 ].  This is then added
    * to the diagonal elements of Q, which are Q_angle and Q_gyro.
    */
   final float[] Pdot = {
       Q_angle - P[0][1] - P[1][0], /* 0,0 */
       -P[1][1], /* 0,1 */
       -P[1][1], /* 1,0 */
       Q_gyro /* 1,1 */

   };
```

```
    /* Store our unbias gyro estimate */
    rate = q;

    /*
     * Update our angle estimate
     * angle += angle_dot * dt
     *       += (gyro - gyro_bias) * dt
     *       += q * dt
     */
    angle += q * dt;

    /* Update the covariance matrix */
    P[0][0] += Pdot[0] * dt;
    P[0][1] += Pdot[1] * dt;
    P[1][0] += Pdot[2] * dt;
    P[1][1] += Pdot[3] * dt;
}


/*
 * kalman_update is called by a user of the module when a new
 * accelerometer measurement is available.  ax_m and az_m do not
 * need to be scaled into actual units, but must be zeroed and have
 * the same scale.
 *
 * This does not need to be called every time step, but can be if
 * the accelerometer data are available at the same rate as the
 * rate gyro measurement.
 *
 * For a two-axis accelerometer mounted perpendicular to the rotation
 * axis, we can compute the angle for the full 360 degree rotation
 * with no linearization errors by using the arctangent of the two
 * readings.
 *
 * As commented in state_update, the math here is simplified to
 * make it possible to execute on a small microcontroller with no
 * floating point unit.  It will be hard to read the actual code and
 * see what is happening, which is why there is this extensive
 * comment block.
 *
 * The C matrix is a 1x2 (measurements x states) matrix that
 * is the Jacobian matrix of the measurement value with respect
 * to the states.  In this case, C is:
 *
 * C = [ d(angle_m)/d(angle)  d(angle_m)/d(gyro_bias) ]
 *   = [ 1 0 ]
 *
```

```
 * because the angle measurement directly corresponds to the angle
 * estimate and the angle measurement has no relation to the gyro
 * bias.
 */
void kalman_update(
        //const float ax_m,/* X acceleration */
        //const float az_m /* Z acceleration */
        final float angle_m /* angle in rad, determined from accel */)
{
    /* Compute our measured angle and the error in our estimate */
    //const float angle_m = atan2( -az_m, ax_m );
    //const float angle_m = atan2( ax_m, az_m );  //bk
    final float angle_err = angle_m - angle;

    /*
     * C_0 shows how the state measurement directly relates to
     * the state estimate.
     *
     * The C_1 shows that the state measurement does not relate
     * to the gyro bias estimate.  We don't actually use this, so
     * we comment it out.
     */
    final float C_0 = 1;
    /* const float C_1 = 0; */

    /*
     * PCt<2,1> = P<2,2> * C'<2,1>, which we use twice.  This makes
     * it worthwhile to precompute and store the two values.
     * Note that C[0,1] = C_1 is zero, so we do not compute that
     * term.
     */
    final float PCt_0 = C_0 * P[0][0]; /* + C_1 * P[0][1] = 0 */
    final float PCt_1 = C_0 * P[1][0]; /* + C_1 * P[1][1] = 0 */

    /*
     * Compute the error estimate.  From the Kalman filter paper:
     *
     * E = C P C' + R
     *
     * Dimensionally,
     *
     * E<1,1> = C<1,2> P<2,2> C'<2,1> + R<1,1>
     *
     * Again, note that C_1 is zero, so we do not compute the term.
     */
    final float E =
            R_angle + C_0 * PCt_0 /* + C_1 * PCt_1 = 0 */;
```

```
/*
 * Compute the Kalman filter gains.  From the Kalman paper:
 *
 * K = P C' inv(E)
 *
 * Dimensionally:
 *
 * K<2,1> = P<2,2> C'<2,1> inv(E)<1,1>
 *
 * Luckily, E is <1,1>, so the inverse of E is just 1/E.
 */
final float K_0 = PCt_0 / E;
final float K_1 = PCt_1 / E;

/*
 * Update covariance matrix.  Again, from the Kalman filter paper:
 *
 * P = P - K C P
 *
 * Dimensionally:
 *
 * P<2,2> -= K<2,1> C<1,2> P<2,2>
 *
 * We first compute t<1,2> = C P.  Note that:
 *
 * t[0,0] = C[0,0] * P[0,0] + C[0,1] * P[1,0]
 *
 * But, since C_1 is zero, we have:
 *
 * t[0,0] = C[0,0] * P[0,0] = PCt[0,0]
 *
 * This saves us a floating point multiply.
 */
final float t_0 = PCt_0; /* C_0 * P[0][0] + C_1 * P[1][0] */
final float t_1 = C_0 * P[0][1]; /* + C_1 * P[1][1]  = 0 */

P[0][0] -= K_0 * t_0;
P[0][1] -= K_0 * t_1;
P[1][0] -= K_1 * t_0;
P[1][1] -= K_1 * t_1;

/*
 * Update our state estimate.  Again, from the Kalman paper:
 *
 * X += K * err
 *
```

```
         * And, dimensionally,
         *
         * X<2> = X<2> + K<2,1> * err<1,1>
         *
         * err is a measurement of the difference in the measured state
         * and the estimate state.  In our case, it is just the difference
         * between the two accelerometer measured angle and our estimated
         * angle.
         */
        angle += K_0 * angle_err;
        q_bias += K_1 * angle_err;
    }

// in deg/sec
    float get_kalman_rate()
    {
        return 180.0f / 3.14159f * rate;
    }

    float get_kalman_gyro_bias()
    {
        return 180.0f / 3.14159f * q_bias;
    }

    float get_kalman_angle()
    {
        return 180.0f / 3.14159f * angle;
    }
}
```

B.3 NXT: Codice del thread di gestione dei comandi da eseguire

```
public class Scanner extends GyroTest  {

static BTConnection btc;
static DataInputStream dis;
private BT bt = new BT();

    public Scanner() {

     btc = Bluetooth.waitForConnection();
        dis = btc.openDataInputStream();
     bt.start();

    }

    private class BT extends Thread {
```

```
   public BT() {
      this.setDaemon(true);
   }

   public void run() {

    int n;
       int a=0;

      while (true){
       Sound.beep();
try {
while (a==0){
a =dis.available();
Thread.sleep(2);
Thread.yield();
}
n=dis.readInt();
if (n<1000){
if(n>=100)
desiredSpeed=n-150;
else
turn=n-50;
}
else
if (n>999 && n<=1999)
kAng=n-1000;
else
if (n>1999 && n<=2999)
kIntAng=n-2000;
else
if (n>2999 && n<=3999)
kAngRate=n-3000;
else
if (n>3999 && n<=4999)
SkP =n-4000;
else
if (n>4999 && n<=5999)
SkI =n-5000;
else
if (n>5999 && n<=6999)
SkD =n-6000;
else
    if (n==9999){
    dis.close();
          btc.close();
           System.exit(1);
```

```
         }

     } catch (Exception e) {
     System.exit(1);
     }
        }
        }
    }

}
```

B.4 PC: Codice per l'utilizzo del Wiimote

```
public class BTSend implements WiimoteListener {

public static DataOutputStream dos;
public static DataInputStream dis;
public static NXTComm nxtComm = null;
public static int Speed=0;
public static int Turn=0;
public static int n = 0;
public static int prevSpeed=0;
public static int prevTurn=0;


public void onButtonsEvent(WiimoteButtonsEvent arg0) {
        //System.out.println(arg0.toString());
        if (arg0.isButtonAPressed()){
         sendToNXT (9999);
         closeAllConnection ();

        }
    }

public void onMotionSensingEvent(MotionSensingEvent arg0) {

     Orientation orientation =arg0.getOrientation();
     Speed=(int)orientation.getPitch();
     Turn=(int)orientation.getRoll();



     if((Speed<=-45))
     Speed=146;
     else if((Speed>=-45)&&(Speed<-30))
     Speed=148;
     else if((Speed>=-30)&&(Speed<-10))
```

```
      Speed=149;
      else if ((Speed>=-10)&&(Speed<=10))
      Speed=150;
      else if((Speed>=10)&&(Speed<=30))
      Speed=151;
      else if((Speed>30)&&(Speed<=45))
      Speed=152;
      else if((Speed>45))
      Speed=154;



      if((Turn<=-40))
      Turn=60;
      else if((Turn>=-40)&&(Turn<-20))
      Turn=55;
      else if ((Turn>=-20)&&(Turn<=20))
      Turn=50;
      else if((Turn>=20)&&(Turn<=40))
      Turn=45;
      else if(Turn>40)
      Turn=40;
if(Speed!=prevSpeed)
     sendToNXT(Speed);
prevSpeed=Speed;
if(Turn!=prevTurn)
     sendToNXT(Turn);
prevTurn=Turn;

    }


public static void connectWiimote (){

Wiimote[] wiimotes = WiiUseApiManager.getWiimotes(1, true);
        if (wiimotes.length==0){
         System.out.println("NO Wiimote detected. Connect a Wiimote and retry");
         WiiUseApiManager.definitiveShutdown();
         System.exit(1);
        }
        else{
        System.out.println("Wiimote Found");
Wiimote wiimote = wiimotes[0];
try {
Thread.sleep(500);
} catch (InterruptedException e) {
}
```

```
wiimote.setTimeout((short)15, (short)15);
wiimote.activateMotionSensing();
wiimote.activateSmoothing();
wiimote.setAlphaSmoothingValue((float) 0.07);
wiimote.deactivateContinuous();
        wiimote.setOrientationThreshold(5);
        wiimote.addWiiMoteEventListeners(new BTSend());
        }

}

public static void connectNXT (){
String name="NXT";
String address="00:16:53:02:2E:80";


try {
nxtComm = NXTCommFactory.createNXTComm(NXTCommFactory.BLUETOOTH);
} catch (NXTCommException e) {
System.out.println("Failed to load Bluetooth driver");
System.exit(1);
}

NXTInfo[] nxtInfo = new NXTInfo[1];
nxtInfo[0] = new NXTInfo(name,address);
System.out.println("Connecting... ");

boolean opened = false;

try {
opened = nxtComm.open(nxtInfo[0]);
} catch (NXTCommException e) {
System.out.println("Exception from open");
}

if (!opened) {
System.out.println("Failed to open " + nxtInfo[0].name);
WiiUseApiManager.definitiveShutdown();
System.exit(1);
}

System.out.println("Connected to " + nxtInfo[0].btResourceString);

InputStream is = nxtComm.getInputStream();
OutputStream os = nxtComm.getOutputStream();

dos = new DataOutputStream(os);
```

```
dis = new DataInputStream(is);

}

public static void sendToNXT (int Value){


try {
Thread.sleep(50);
dos.writeInt(Value);
dos.flush();

} catch (Exception e) {
System.out.println("IO Exception writing bytes:");
System.out.println(e.getMessage());
}

}

public static void closeAllConnection (){
try {
     dis.close();
     dos.close();
     nxtComm.close();

     WiiUseApiManager.shutdown();
     WiiUseApiManager.definitiveShutdown();
     System.out.println("Bye... ");
     Thread.sleep(2000);
     System.exit(1);
     } catch (Exception ioe) {
     System.out.println("IOException closing connection:");
     System.out.println(ioe.getMessage());
     }

}

public static void main(String[] args) throws Exception {

connectNXT();
connectWiimote();
new PIDScanner();

}
}
```

B.5 PC: Codice del thread utilizzato per la taratura remota dei PID

```
import java.io.BufferedReader;
import java.io.InputStreamReader;



public class PIDScanner extends BTSend {
static BufferedReader d = new BufferedReader(new InputStreamReader(System.in));
static String s = null;
private BT bt = new BT();
    public PIDScanner() {
    bt.start();
    }

    private class BT extends Thread {
        public BT() {
            this.setPriority(NORM_PRIORITY-1);
            this.setDaemon(true);
        }

        public void run() {
         int P=0;
     int I=0;
     int D=0;
     int SkI=0;
     int SkP=0;
     int SkD=0;

         while(true){
     try {
s = d.readLine();

     n = Integer.parseInt(s);
     if (n>999 && n<=1999){
                 P=n-1000;
                 sendToNXT(n);
                }
                else
                if (n>1999 && n<=2999){
                 I=n-2000;
                 sendToNXT(n);
                }
                else
                if (n>2999 && n<=3999){
                    D=n-3000;
                    sendToNXT(n);
```

```
   }
   else
   if (n>3999 && n<=4999){
    SkP=n-4000;
    sendToNXT(n);
   }
   else
   if (n>4999 && n<=5999){
    SkI=n-5000;
    sendToNXT(n);
   }
   else
   if (n>5999 && n<=6999){
       SkD=n-6000;
       sendToNXT(n);
   }

else
if (n==9000){

 P=0;
    I=0;
    D=0;
    SkP=0;
    SkI=0;
    SkD=0;
    sendToNXT(50);
    Thread.sleep(10);
    sendToNXT(150);
    Thread.sleep(10);
    sendToNXT(1000);
    Thread.sleep(10);
    sendToNXT(2000);
    Thread.sleep(10);
    sendToNXT(3000);
    Thread.sleep(10);
    sendToNXT(4000);
    Thread.sleep(10);
    sendToNXT(5000);
    Thread.sleep(10);
    sendToNXT(6000);
    Thread.sleep(10);
  }
else
if(n==9999){
 sendToNXT(9999);
 closeAllConnection();
```

```
                Thread.sleep(100);
                break;

               }
     } catch (Exception e) {
     System.out.println("ERROR!! ");
}
     System.out.println("P " + P + " I " + I + " D " + D + " SkP " + SkP
     + " SkI " + SkI + " SkD " + SkD+"\n");
         }
         System.exit(1);
        }
        }
}
```

B.6 PIC: Porzione di codice per la cattura PPM

```
// Capture pulse from Gyro
void Capture (void) {

unsigned int start=0;
unsigned int finish=0;
int value=0;

// Reset Timer1 to prevent overflow
TMR1H=0;
TMR1L=0;
// Set rising edge capture
CCP1CON = 0b00000101;
// Clear CCP interupt from TMR1 flag
CCP1IF=0;
// Wait for rising edge flag
while (CCP1IF!=1)
continue;
// Save start time
start=((unsigned int)CCPR1H << 8) | CCPR1L;
// Clear CCP interupt from TMR1 flag
CCP1IF=0;
// Set falling edge capture
CCP1CON = 0b00000100;
// Clear CCP interupt from TMR1 flag
CCP1IF=0;
// Waiting for falling edge flag
while (CCP1IF!=1)
continue;
```

```
// Save finish time
finish=((unsigned int)CCPR1H << 8) | CCPR1L;
// Compute normalized period (-100/+100)
value=(int)(((finish-start)-1000)/5-88);
if (value>127)
value=127;
if (value<-127)
value=-127;
period=value;
// Clear CCP interupt from TMR1 flag
CCP1IF=0;
}
```

B.7 PIC: Porzione di codice per la gestione del protocollo I^2C

```
void interrupt my_isr(void){

unsigned char Tmp;

// I2C interrupt
if (SSPIF)
  {
// Hold clock low
CKP=0;
// Read the status register
Tmp = (SSPSTAT & 0b00101101);
// Case 1: MASTER WRITE - LAST BYTE WAS ADDRESS
if (Tmp == 0b00001001){
// Read the address to prevent overflow
  Tmp = SSPBUF;
}
// Case 2: MASTER WRITE - LAST BYTE WAS DATA
else if (Tmp == 0b00101001){
// Read the address to prevent overflow
Tmp = SSPBUF;
}
// Case 3: MASTER READ - LAST BYTE WAS ADDRESS
else if (Tmp == 0b00001100 ){
  // Fill buffer with first 8 bit data
SSPBUF=period;
}
// Case 4: MASTER READ - LAST BYTE WAS DATA
else if (Tmp == 0b00101100){
// Fill buffer with last 8 bit data
SSPBUF=period;
}
// Case 5: NACK FROM MASTER / INITIALIZE I2C
```

```
else if (Tmp == 0b00101000 || Tmp== 0b00101100){
// Config I2C with 7 bits without START/STOP interrupts
SSPCON= 0x36;
}
// Case 6: I2C ERROR
else {
// Something went wrong... resetting I2C module
// Disable I2C module
SSPCON=0x00;
  DelayUs(5);
  // Re-enable  module
  SSPCON= 0x36;
}
// Enable CLK
CKP=1;
// Clear flag
SSPIF=0;
}
}
```

Powered by LaTeX 2_ε