POLITECNICO DI BARI

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELL'INFORMAZIONE

CORSO DI LAUREA MAGISTRALE IN AUTOMAZIONE

ESAME DI MEASUREMENT AND DATA ACQUISITION SYSTEMS

Robot auto-bilanciante: modello, realizzazione e acquisizione dati

Domenico Bevilacqua

Anno Accademico 2018/2019

Sommario

CAPITOLO 1

1.1 Introduzione

I robot auto-bilancianti sono particolari automi che riescono a mantenere la posizione verticale e per farlo hanno bisogno di un certo numero di sensori per rilevare la loro posizione istante per istante, oltre a motori che possano reagire in modo da ristabilire l'equilibrio. Si tratta naturalmente di una posizione completamente instabile e proprio per questo è necessario creare un controllore efficiente.

In prima analisi la principale difficoltà di questi robot è quella di farli muovere e restare stabili, in piedi; in un primo tempo si cercò la soluzione più stabile, che consisteva nel farli muovere su tre o più ruote. Si passò poi a robot auto-bilancianti su due ruote, che conservano l'equilibrio sia in movimento che da fermi. Questo genere di robot è diventato celebre nella forma e nelle sembianze del Segway.

Mantenere l'equilibrio significa mantenerne il baricentro entro la base d'appoggio, cosa che può essere fatta controllando opportunamente i movimenti del robot. Si può ipotizzare che più il baricentro è alto, più la base di appoggio è stretta e meno stabile risulterà la struttura.

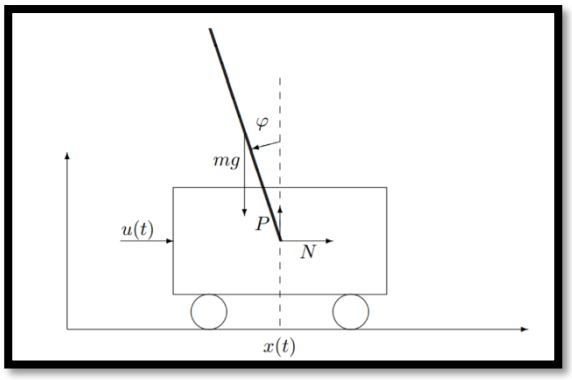
Per descrivere il sistema di controllo di un robot auto-bilanciante dobbiamo innanzitutto delineare un modello, e notiamo che si tratta sostanzialmente di un pendolo inverso. Da questa considerazione e dalle formule che legano il moto di un pendolo, possiamo stimare rapidamente la velocità del nostro sistema e quindi dedurre quanto dovrà essere veloce il controllo della stabilità che dovremmo adottare.

I robot auto-bilancianti su due ruote sono in grado di risolvere il problema di instabilità intrinseca del pendolo inverso utilizzando le due ruote per bilanciarsi attorno al punto di equilibrio. Avendo a che fare anche con disturbi esterni, questi sistemi non rimarranno immobili, ma subiranno sempre lievi oscillazioni. Per mantenere l'equilibrio sono state applicate diverse tecniche di controllo, come i classici algoritmi PID.

1.2 Modello di un pendolo inverso

Per modellare il sistema lo tratteremo innanzitutto come un sistema SISO dove l'ingresso è l'angolo del robot con la verticale e l'uscita la forza da esercitare con i motori.

Il modello del pendolo inverso è caratterizzato da un carrello sul quale è incernierata un'asta libera di muoversi, e il nostro scopo è di far scorrere il carrello affinché l'asta rimanga in equilibrio verticale.



Indichiamo rispettivamente con P(·) e N(·) la forza verticale e quella orizzontale esercitate dal carrello sull'asta, con L la distanza del baricentro dell'asta dalla cerniera e con I il momento di inerzia dell'asta rispetto al baricentro. Sia inoltre g l'accelerazione di gravità.

Il moto del baricentro dell'asta è descritto dalle equazioni:

$$N(t) = m\frac{d^2}{dt^2}(x(t) - L\,sin\varphi(t)) \tag{1}$$

$$P(t) - mg = m\frac{d^2}{dt^2}(L\,sin\varphi(t)) \tag{2}$$

Il moto di rotazione relativo al baricentro è dato dall'equazione:

$$LP(t)\,sin\varphi(t) + LN\,cos\varphi(t) = I\frac{d^2\varphi(t)}{dt^2} \tag{3}$$

Il moto del carrello è infine descritto dall'equazione:

$$M\frac{d^2x(t)}{dt^2} = -N(t) + u(t) - b\frac{dx(t)}{dt} \tag{4}$$

dove si è tenuto conto della presenza di un attrito viscoso di coefficiente b. Sostituendo le espressioni di $P(t)$ e $N(t)$ in (3) e (4), si perviene alla coppia di equazioni:

$$L\left[m\frac{d^2}{dt^2}(Lcos\varphi(t)) + mg\right]sin\varphi(t) + L\left[m\frac{d^2}{dt^2}(x(t) - L\,sin\varphi(t))\right]cos\varphi(t) = I\frac{d^2\varphi(t)}{dt^2} \tag{5}$$

$$M\frac{d^2x(t)}{dt^2} = -m\frac{d^2}{dt^2}(x(t) - L\,sin\varphi(t)) + u(t) - b\frac{dx(t)}{dt} \tag{6}$$

Assumendo $\varphi(t)$ molto piccolo, così che valgano le approssimazioni

$$sin\varphi(t) \approx \varphi(t), \quad cos\varphi(t) \approx 1$$

le (5) e (6) possono essere riscritte come segue:

$$(I + mL^2)\frac{d^2\varphi(t)}{dt^2} - Lmg\,\varphi(t) = Lm\frac{d^2x(t)}{dt^2} \quad (7)$$

$$(M + m)\frac{d^2x(t)}{dt^2} - mL\frac{d^2\varphi(t)}{dt^2} + b\frac{dx(t)}{dt} = u(t) \quad (8)$$

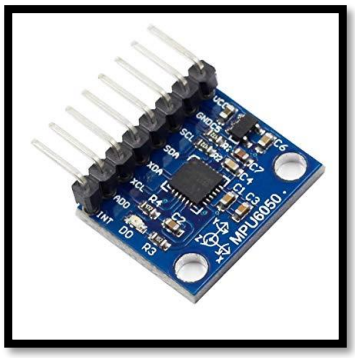
Le espressioni (7) e (8) sono le equazioni che governano il moto del pendolo inverso.

CAPITOLO 2

2.1 L'hardware

Per realizzare il robot autobilanciante abbiamo optato per un hardware semplice da reperire che non comportasse costi eccessivi. La logica di controllo è affidata al famoso microcontrollore ATmega328P, montato su una scheda Arduino Uno che ha svolto la funzione di principale piattaforma di sviluppo per il nostro progetto. Prima di addentrarci nell'analisi della logica software, è importante comunque soffermarci sull'hardware utilizzato per creare il progetto di cui segue un elenco saliente.

- Integrato InvenSense MPU-6050



E' il sensore che svolge il ruolo principale per il funzionamento del robot. L' *InvenSense MPU-6050* contiene, in un singolo integrato, un accelerometro MEMS a 3 assi ed un giroscopio MEMS a 3 assi, per un totale di 6 assi. Con lo giroscopio possiamo misurare l'accelerazione angolare di un corpo su di un proprio asse, mentre con l'accelerometro possiamo misurare l'accelerazione di un corpo lungo una direzione. È molto preciso, in quanto ha un convertitore AD (da analogico a digitale) da 16 bit per ogni canale. Perciò cattura i canali x, y e z contemporaneamente. Il sensore possiede un protocollo di comunicazione standard I^2C, che è stato appunto il canale da noi utilizzato per interfacciarlo con Arduino Uno.

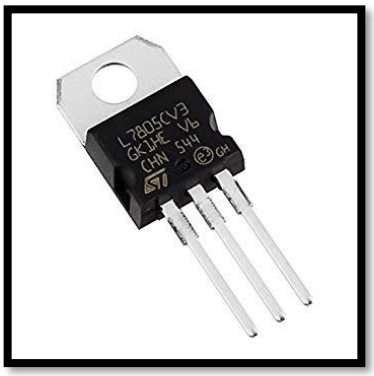
- Motori stepper bipolari 42Ncm

Ovviamente è stata importante anche la scelta dei motori da utilizzare; dopo vari tentativi la scelta è ricaduta su un paio di motori stepper bipolari a 1.5 A per fase. Essi lavorano a una tensione di 2,4 V e riescono a fornire una coppia di tenuta di 420 mN·m. La scelta di motori di tipo stepper è la più logica per la locomozione del robot, poiché il loro pilotaggio passo-passo permette un controllo abbastanza preciso da parte del microcontrollore sugli spostamenti retroattivi da effettuare.

- Driver A4988

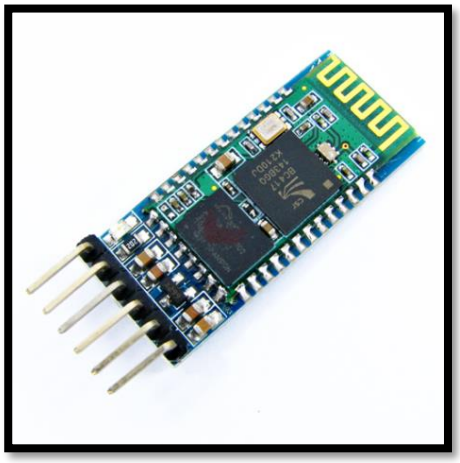
La scelta di motori stepper è, come già detto, la più ottimale nel controllo preciso dei movimenti del robot; tuttavia essa comporta la necessità di implementare un hardware dedicato esclusivamente dal pilotaggio degli stepper. Infatti sebbene Arduino sia già in grado di controllare questo tipo di motori, esso impegnerebbe il suo microcontrollore (che ovviamente non è multi-tasking) al solo pilotaggio di questo tipo di attuatori, tralasciando la logica di funzionamento in quel lasso di tempo. Anche se si parla di millesimi di secondo, sarebbe un deficit critico nel controllo in retroazione del robot. Per ovviare a questa problematica si utilizzano drivers specifici per gli stepper bipolari che permettono di effettuare uno step del motore ricevendo in ingresso un semplice impulso. Oltre a questo, l'eccezionale utilità di questi drivers risiede anche nella capacità di controllare la direzione di rotazione e soprattutto nel permettere un controllo microstepping, ovvero di far ruotare l'albero del motore anche di sotto-divisioni di step, fino a 1/16 di step. Nel nostro progetto si è optato per un controllo a 1/8 di step.

- Regolatore di tensione LM7805



L'alimentazione utilizzata fornisce un voltaggio superiore ai 5V con cui Arduino opera; per questo motivo è nata la necessita di limitare la tensione in ingresso ad Arduino, sebbene esso sia già preparato a questa eventualità con un regolatore già implementato sulla scheda stessa. Abbiamo deciso tuttavia, per garantire l'incolumità del microcontrollore di utilizzare un regolatore esterno che accetti in ingresso i 12,6V massimi della batteria e fornisca in uscita 5V costanti direttamente ad Arduino. Abbiamo optato per l'integrato LM7805 , che può ricevere in ingresso una tensione da un minimo di 7V a un massimo di 25V. A fronte dell'energia dissipata in calore da parte del regolare, abbiamo montato in aggiunta un dissipatore di calore.

- Modulo bluetooth HC-05



Il fine ultimo del progetto è quello di visualizzare i dati delle misurazioni e dello stato del robot senza ovviamente influenzarne il funzionamento. A tal fine è d'obbligo un sistema di comunicazioni tra Arduino e il computer che si basi su una tecnologia wireless, senza fili che sappur leggeri potrebbero inficiare l'operatività del robot. Abbiamo optato perciò a un modulo bluetooth che riesca a inviare i dati da leggere poi con un modulo gemello montato su un'altra scheda Arduino direttamente collegata tramite USB al computer. Il modulo Bluetooth HC-05 è uno dei moduli più popolari e poco costosi utilizzati per le comunicazioni RF. Il modulo ha una portata di 10 mt, si imposta facilmente tramite comandi AT ed è programmabile sia come master che come slave. Il modulo permette di trasformare una porta UART\USART più comunemente conosciuta come seriale in una porta Bluetooth, generalmente con profilo SPP(Serial Port Profile), diventando cosi una seriale over Bluetooth.

Questi i principali componenti hardware utilizzati per il progetto; ovviamente a essi sono stati aggiunti componenti minori: condensatori da 47 uF, posti in parallelo all'alimentazione in ingresso ai driver, proprio per preservare l'inculumità di essi ad eventuali picchi di tensione; resistenze da 3,3 KΩ, poste in serie tra le estremità della batteria per formare un partitore di tensione tramite il quale Arduino può leggere la tensione di alimentazione senza sovraccaricare i suoi pin analogici.

2.2 Il Software

2.2.1 Il Setup

Oltre alle varie misure da effettuare, il microcontrollore ha bisogno di fare affidamento ovviamente a molteplici variabili e costanti di vario tipo durante i suoi calcoli; sarebbe superfluo e prolisso elencarle tutte, per cui si darà per scontato nell'analisi del codice che esse siano state già dichiarate e eventualmente inizializzate. Omettiamo inoltre l'inizializzazione dei pin di lettura/scrittura.

Partiamo dunque con le prime righe di codice essenziali per il corretto funzionamento del robot; di essenziale importanza per la realizzazione di questo progetto è stato l'utilizzo dei registri dell'ATmega328P. Tra questi, il primo da impostare è quello riguarda la comunicazione I^2C, che Arduino utilizza (grazie alla libreria *Wire.h*) per comunicare con l'integrato MPU 6050. La frequenza di *full speed* del bus I^2C è di 400 KHz, impostabile su Arduino tramite il registro TWBR secondo la seguente semplice equazione:

$$TWBR = \frac{\frac{frequenza\ CPU}{frequenza\ bus} - 16}{2}$$

Tenendo sempre presente che il microprocessore di Arduino opera a 16 MHz, è facile calcolare che il registro deve essere impostato al valore 12 per ottenere una frequenza di 400 Khz. Oltre questo registro, abbiamo utilizzato anche un registro ausiliario che Arduino offre al fine di creare una sub-routine completamente indipendente dal loop principale; abbiamo sfruttato questa opportunità per inviare gli impulsi di comando ai drivers dei motori. E' un modo ingegnoso per dare ad Arduino capacità simili al multithreading. Per realizzare ciò, Arduino genera un interrupt che scatta per confronto di un contatore con un registro da noi assegnato; in corrispondenza dell'interrupt, Arduino esegue il codice della subroutine. Anche in questo caso i calcoli sono semplici (ricordiamo che Arduino opera a 16MHz e il contatore interessato è a 8 bit):

$$\frac{16\ MHz}{8\ bit} = 2MHz\ frequenza\ del\ contatore$$

$$numero\ di\ confronto = \frac{2MHz}{frequenza\ subroutine\ desiderata}$$

Volendo una frequenza di 50KHz (corrispondente a un periodo di 20 µs, sufficiente per l'invio degli impulsi ai drivers dei motori) il numero da impostare risulta 39 (il risultato è 40 ma ricordiamo che il contatore parte da 0). Il codice utilizzato per impostare quanto appena detto è

dunque il seguente:

```
Wire.begin();
TWBR = 12;
TCCR2A = 0;
TCCR2B = 0;
TIMSK2 |= (1 << OCIE2A);
TCCR2B |= (1 << CS21);
OCR2A = 39;
TCCR2A |= (1 << WGM21);
```

Passiamo ora alla configurazione del sensore giroscopio+accellerometro. Anche in questo caso si ricorre all'utilizzo di registri che abbiamo impostato tramite il bus I²C, più precisamente abbiamo utilizzato i registri:

- 0x6B -> PWR_MGMT_1: dato che all'avvio l'integrato parte in Sleep mode, settiamo questo registro per "svegliare" l'MPU 6050

- 0x1A -> CONFIG: imposta la frequenza del filtro *Digital Low Pass Filter* secondo la tabella:

DLPF_CFG	Accelerometer (F_s = 1kHz)		Gyroscope		
	Bandwidth (Hz)	Delay (ms)	Bandwidth (Hz)	Delay (ms)	Fs (kHz)
0	260	0	256	0.98	8
1	184	2.0	188	1.9	1
2	94	3.0	98	2.8	1
3	44	4.9	42	4.8	1
4	21	8.5	20	8.3	1
5	10	13.8	10	13.4	1
6	5	19.0	5	18.6	1
7	RESERVED		RESERVED		8

Volendo impostare una frequenza di 44Hz il valore da impostare è dunque 3 (0x03).

- 0x1B-> GYRO_CONFIG: imposta la scala di precisione del giroscopio, che va da ±250°/s fino ad arrivare anche a ±2000°/s. Ai nostri fini la scala di valore minore può andar più che bene, dunque abbiamo settato il registro a 0 (0x00).

- 0x1C->ACCEL_CONFING: imposta la scala di precisione dell'accelerometro, che può andare da un minimo di ±2g fino a un massimo di ±16g. Abbiamo scelto per le nostre misure una scala di ±4g, settando il registro a 1(0x01).

Il codice risultante è dunque:

```
Wire.beginTransmission(indirizzo_gyro);
Wire.write(0x6B);
Wire.write(0x00);
Wire.endTransmission();

Wire.beginTransmission(indirizzo_gyro);
Wire.write(0x1B);
Wire.write(0x00);
Wire.endTransmission();

Wire.beginTransmission(indirizzo_gyro);
Wire.write(0x1C);
Wire.write(0x08);
Wire.endTransmission();

Wire.beginTransmission(indirizzo_gyro);
Wire.write(0x1A);
Wire.write(0x03);
Wire.endTransmission();
```

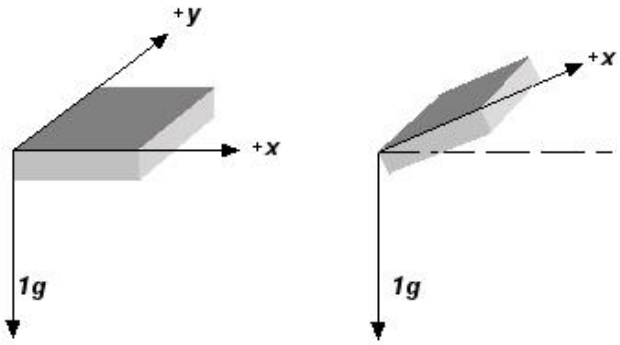
Ultima operazione effettuata nel setup iniziale è la calibrazione del giroscopio, ovvero nel trovare il valore medio (l'offset) fornito dal giroscopio quando esso è fermo per poi sottrarlo ad ogni lettura. Per fare ciò interroghiamo il registro corrispondente, ovvero 0x43->GYRO_XOUT, un numero adeguato di volte per poi fare la media di tutte le misurazioni ottenute mentre il robot è fermo:

```
for (int i = 0; i < 1000; i++)
{
  Wire.beginTransmission(indirizzo_gyro);
  Wire.write(0x43);
  Wire.endTransmission();
  Wire.requestFrom(indirizzo_gyro, 4);
  offset_pitch += Wire.read() << 8 | Wire.read();
  delayMicroseconds(1000);
}
offset_pitch /= 1000;
```

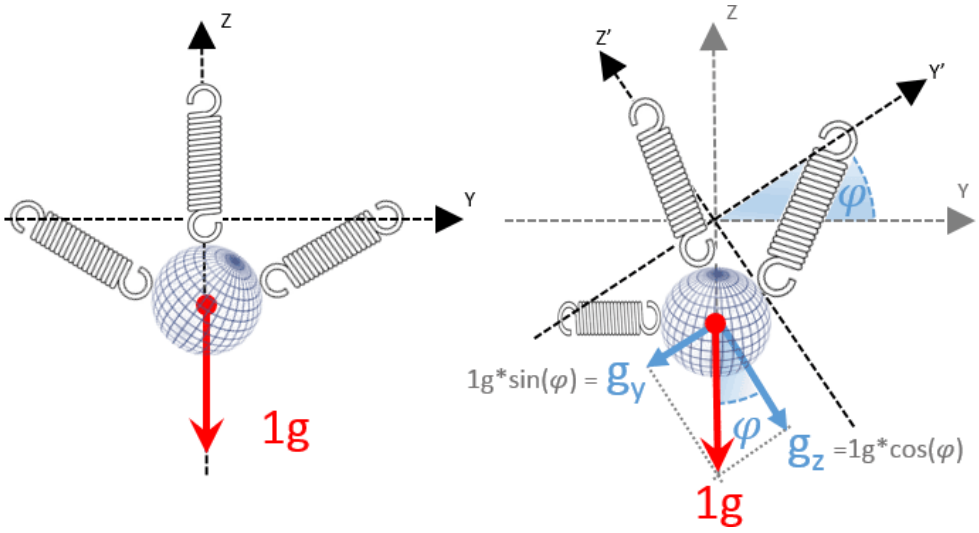
2.2.2 Misurazione di giroscopio e accelerometro

Entrando nel vivo delle misure effettuate dal robot, quelle di vitale importanza per ottenere l'angolazione del robot autobilanciante sono velocità angolare data dallo giroscopio e accelerazione lineare, data appunto dall'accelerometro.

Partiamo con la misurazione di quest'ultimo: per ottenere il valore rilevato dell'accelerometro dobbiamo semplicemente interrogare l'MPU 6050 all'indirizzo 0x3F-> ACCEL_YOUT; a questa misura il codice sottrae il corrispondete offset dell'accelerometro, ovvero il valore costante da esso fornito quando il robot si trova in posizione ferma eretta. Dalla misura ottenuta è necessario ricavare un angolo, operazione non banale; per questa finalità si sfruttano semplici basi teoriche di geometrica e trigonometria. Immaginiamo un corpo che si estende in tutte le tre direzioni, perfettamente orizzontale rispetto al suolo; su di esso agirà la forza peso del valore di 1g solo su una superficie.



Se tuttavia successivamente ruotiamo il corpo, la forza di gravità avrebbe componenti su due superfici stavolta, la cui somma sarebbe comunque 1g. Dal valore della componente gravitazionale che interessa una superficie dell'accelerometro è dunque possibile ricavare la misura dell'angolo con la semplice equazione:



$$\varphi = arcoseno(A_x)$$

Il codice che dunque fornisce l'angolo derivato dalla misura dell'accelerometro è il seguente:

```
//---------------------Calcolo Accellerometro------------------//
Wire.beginTransmission(indirizzo_gyro);
Wire.write(0x3F);
Wire.endTransmission();
Wire.requestFrom(indirizzo_gyro, 2);
acc_rawZ = Wire.read() << 8 | Wire.read();
acc_rawZ += offset_acc;
if (acc_rawZ > 8200)acc_rawZ = 8200;
if (acc_rawZ < -8200)acc_rawZ = -8200;
angolo_acc = -asin((float)acc_rawZ / 8200.0) * 57.296;
```

N.B. La misura dell'accelerometro viene divisa per una costante al fine di riportarla a 1g nel caso limite di 90° di inclinazione; si è moltiplicata la costante 57,296 per la conversione da radianti a gradi;

Per quanto riguarda la misura del giroscopio, dal punto di vista matematico essa fornisce direttamente una misura angolare a differenza dell'accelerometro. Più specificamente il giroscopio misura l'accelerazione angolare attorno a un asse specifico; dunque per estrarne un angolo è necessario semplicemente moltiplicare la misurazione per una costante di tempo che nel nostro caso sarà il periodo di esecuzione impiegato da ogni loop. Il registro dell'MPU 6050 da interrogare è 0x68->GYRO_XOUT:

```
//---------------------Calcolo Giroscopio----------------------//
Wire.beginTransmission(0x68);
Wire.write(0x43);
Wire.endTransmission(false);
Wire.requestFrom(0x68, 4, true);
gyr_rawX = Wire.read() << 8 | Wire.read();
gyr_rawY = Wire.read() << 8 | Wire.read();
gyr_rawY -= offset_pitch;
gyr_rawY = gyr_rawY / 131.0;
angolo_gyr = angolo_gyr + gyr_rawY * 0.004061;
```

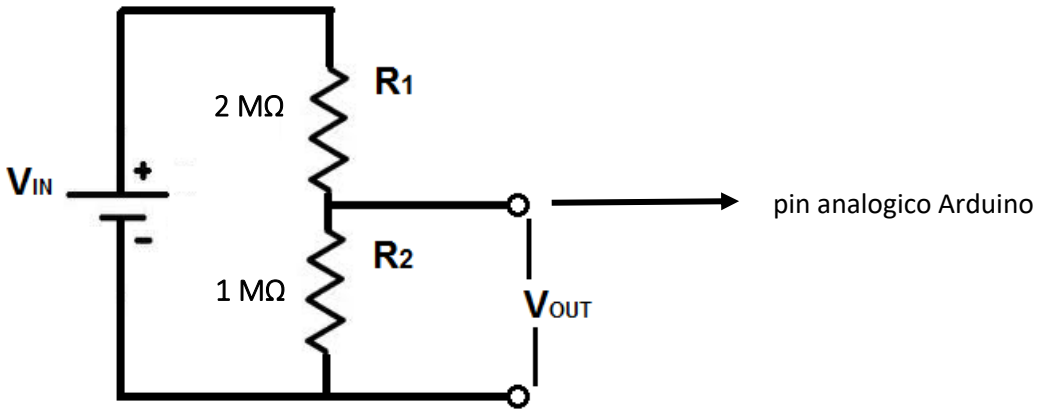
N.B. Ovviamente è imporante il valore iniziale di angolo assoluto a cui il giroscopio aggiunge di volta in volta la misura della velocità angolare misurata; essa viene semplicemente ricavata dall'accelerometro all'inizio.

Siamo infine interessati a combinare i risultati ottenuti dall'accelerometro e dal giroscopio per ottenere una misura finale dell'angolo di rollio del robot. Per ottenere questa esistono prevalentemente due scelte: l'implementazione di un filtro di Kalman, affidabile ma computazionalmente pesante, oppure l'utilizzo di un filtro complementare, meno preciso ma molto semplice. Abbiamo scelto di utilizzare un filtro complementare, che altro non fa che ottenere la misura dell'angolo finale dando pesi differenti alle due misurazioni da unire:

```
//---------Calcolo angolo filtro complementare (acc+gyr)--------//
angolo = 0.98 * angolo_gyr + 0.02 * angolo_acc;
if (angolo > 90) angolo = 90;
if (angolo < -90)angolo = -90;
```

2.2.3 Misurazione tensione della batteria

Come già accennato, per misurare la tensione fornita dalla batteria ad ogni loop abbiamo deciso di sfruttare un semplice partitore di tensione formato da due resistenze in serie rispettivamente da 1 e da 2 MΩ. Questo è necessario poiché Arduino non può accettare come ingresso ai suoi pin analogici una tensione superiore a 5V. I calcoli da effettuare per ottenere la misura della tensione sono dunque:



$$V_{out} = \frac{R_2}{R_1 + R_2} V_{in} = \frac{1}{3} V_{batteria}$$

La lettura va dunque moltiplicata per 3 per riottenere la tensione della batteria; va inoltre scalata secondo il valore massimo con cui Arduino indica 5V in ingresso, corrispondenti a un valore di 1023. La formula finale è dunque:

$$\frac{5}{1024} \cdot 3\ lettura = \frac{1}{68{,}2} lettura = V_{batteria}$$

```
//----------------------Lettura Batteria----------------------//
batteria = analogRead(BATT_PIN) / 68.27;
```

2.2.4 Calcolo PID

Per quanto riguarda l'implementazione del controllo PID ci siamo ricondotti alla famosa formula che comprende le tre costanti Kp, Ki e Kd per settare il contributo rispettivamente proporzionale, integrativo e derivativo dell'errore:

```
errore = angolo - pid_setpoint;

pid_integrale +=  errore;

pid_output = Kp * errore + Ki * pid_integrale + Kd * (errore - errore_prec);

errore_prec = errore;
```

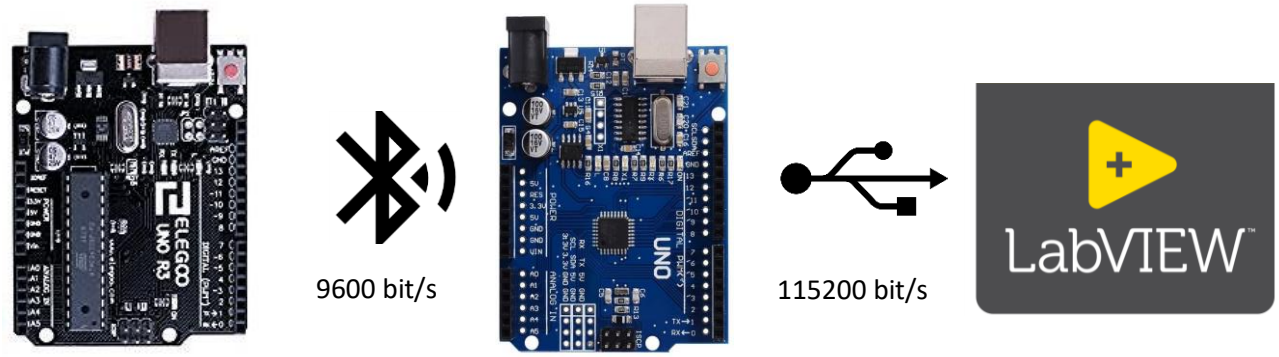
$$output = Kp \cdot e + Ki \cdot \int e\,dt + Kd \cdot \frac{de}{dt}$$

Da cui il codice:

N.B. Il contributo del derivativo avrebbe dovuto essere diviso per il lasso di tempo trascorso; dato che tuttavia la durata di ogni loop è costante (e piccola) è possibile trascurare l'operazione regolando di conseguenza la costante Kd.

2.2.5 Comunicazione

Arriviamo infine alla sezione del software che riguarda la comunicazione seriale del microprocessore; anzitutto precisiamo che il collegando il modulo bluetooth ai pin 0 e 1 (rispettivamente RX e TX della seriale) di Arduino, dal punto di vista del codice non cambia assolutamente nulla nel caso di comunicazione tramite cavo o wireless. La differenza risiede tuttavia nella velocità di trasmissione dei dati:



$$\frac{9600 + 115200}{8} = 15600\ bytes/s = \ 0{,}06\ ms\ per\ byte$$

Nonostante si parli di frazioni di millesimo di secondo, non bisogna trascurare questo calcolo per comprendere comunque i limiti di ricezione e soprattutto invio dei dati.

La ricezione è affidata a una sezione del codice abbastanza semplice, che si limita a riconoscere il primo byte in arrivo e ad assegnare conseguentemente i succesivi alla variabile appropriata; di grande utilità per questa finalità sono le funzioni parseFloat() e parseInt() che permettono di

ricavare un float o un int a partire dalla stringa ricevuta dalla seriale:

```
//----------------------Ricezione comandi----------------------//
if (Serial.available())
{
  byte_ricevuto = Serial.read();
  if (byte_ricevuto == 'p')
  {
    Kp = Serial.parseFloat();
  }
  if (byte_ricevuto == 'i')
  {
    Ki = Serial.parseFloat();
    pid_integrale = 0;
  }
  if (byte_ricevuto == 'd')
  {
    Kd = Serial.parseFloat();
  }
```

Oltre alle variabili PID, il robot può ricevere anche comandi per lo spegnimento e accensione dei motori, comandi per il pilotaggio manuale della direzione e anche la costante di campionamento del periodo per inviare i dati.

L'invio dei dati ha un codice leggermente più sofisticato; LabView infatti raccoglie un numero costante di bytes ad ogni ciclo, abbiamo dovuto trovare dunque un sistema che permetta di inviare un numero costante di bytes a prescindere dal numero della misura corrente da inviare. A tal proposito abbiamo utilizzato la peculiarità della funzione print() che, se sfruttata, restituisce il numero esatto di byte utilizzati per scrivere una determinata misura o carattere. Tramite essi, è possibile calcolare quanti byte rimangono per inviare un pacchetto di lunghezza costante, e "riempire i buchi" tra un valore e l'altro. Abbiamo impostato la lunghezza di ogni pacchetto a 5 bytes, comprendendo la lettera inziale di riconoscimento del tipo di dato:

```
if (contatore_loop == T_campionamento / 24)
{
  bytes_inviati = Serial.print("G");
  bytes_inviati += Serial.print((int)gyr_rawY);
  for (int i = 0; i < (5 - bytes_inviati); i++)
    Serial.print("#");
}
else if (contatore_loop == (T_campionamento / 24) * 2)
{
  bytes_inviati = Serial.print("A");
  bytes_inviati += Serial.print(round(angolo_acc * 10));
  for (int i = 0; i < (5 - bytes_inviati); i++)
    Serial.print("#");
}
else if (contatore_loop == (T_campionamento / 24) * 3)
{
  bytes_inviati = Serial.print("T");
  bytes_inviati += Serial.print(round(angolo * 10));
  for (int i = 0; i < (5 - bytes_inviati); i++)
    Serial.print("#");
    contatore_loop=0;
}
```

N.B I dati vengono inviati sequenzialmente a un certo numero di loop; dato che i dati da inviare sono in realtà 6 e ogni loop dura 4ms, il periodo di campionamento è diviso per 24.

N.B.B Abbiamo moltiplicato e arrotondato ogni variabile float per risparmiare un byte, ovvero quello del punto.

E' interessante, infine, calcolare il collo di bottiglia imposto dalla velocità massima di comunicazione trovato precedentemente:

$$T_{min} = \frac{l_{pacchetto}}{v_{max}} \cdot n_{pacchetti} = \ 1{,}8\ ms$$

La frequenza massima di invio dati del nostro sistema, dunque, è in definitiva circa 556Hz.

CAPITOLO 3

3.1 Programmazione in LabVIEW

Per l'acquisizione, la manipolazione e la visualizzazione dei dati si è utilizzato il software LabVIEW; in particolare, LabVIEW comunica con Arduino tramite comunicazione seriale al fine di ricevere e inviare dati a quest'ultimo.

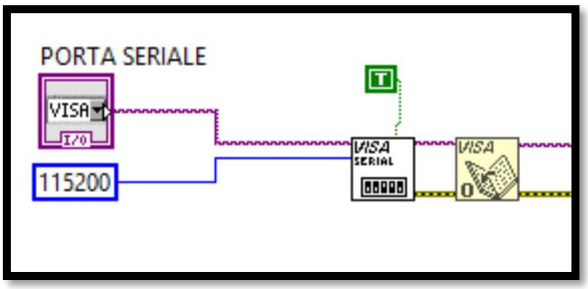
Il VI realizzato può essere schematizzato in 5 macro-sezioni:

1. Una prima sezione che contiene il blocco di configurazione della porta seriale, che deve contenere gli stessi parametri con cui è stata configurata la stessa sul microcontrollore, ovvero un baud-rate di 115200 e il blocco di apertura della comunicazione.
2. Una seconda sezione di durata pari a 1s in cui vengono inizializzati i vettori che terranno traccia dei vari dati.
3. Una terza sezione che si occupa della ricezione ed elaborazione dei dati provenienti da Arduino.
4. Una quarta sezione che prevede l'invio dei comandi ad Arduino.
5. Un'ultima sezione in cui viene creato un file di testo contenente i dati di interesse.

Di seguito verranno analizzate nel dettaglio queste 5 sezioni.

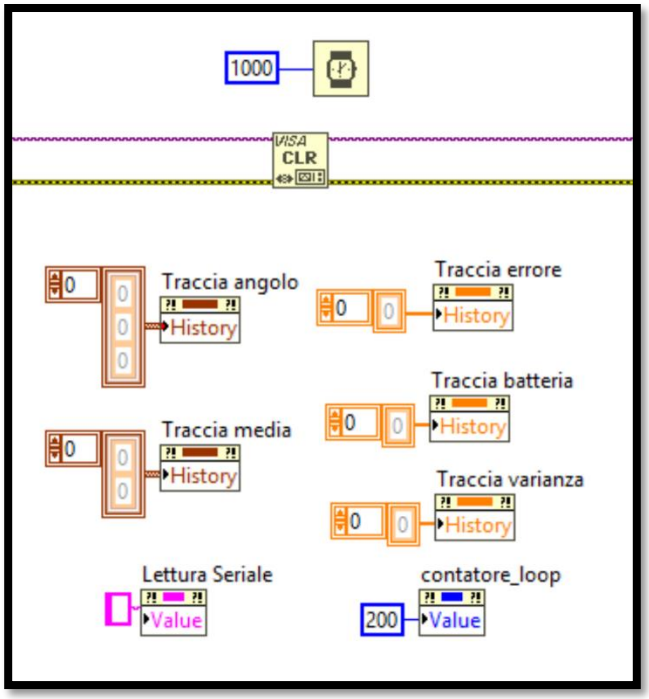
3.1.1 Prima sezione

Poiché LabVIEW comunica con Arduino tramite porta seriale, in primo luogo è necessario configurare la porta seriale facendo attenzione che contenga gli stessi parametri della porta seriale di Arduino. Una volta configurata, è possibile aprire la comunicazione col microcontrollore. E' necessario chiudere tale comunicazione a fine sessione di lavoro mediante il blocco *VISA Close Function*.



3.1.2 Seconda sezione

Prima che LabVIEW inizi a ricevere i dati da Arduino, è opportuno inizializzare i vettori che conterranno i dati elaborati e pulire il buffer di memoria (questa operazione viene effettuata sia prima che LabVIEW riceva i dati, sia dopo che LabVIEW ha inviato i dati ad Arduino).



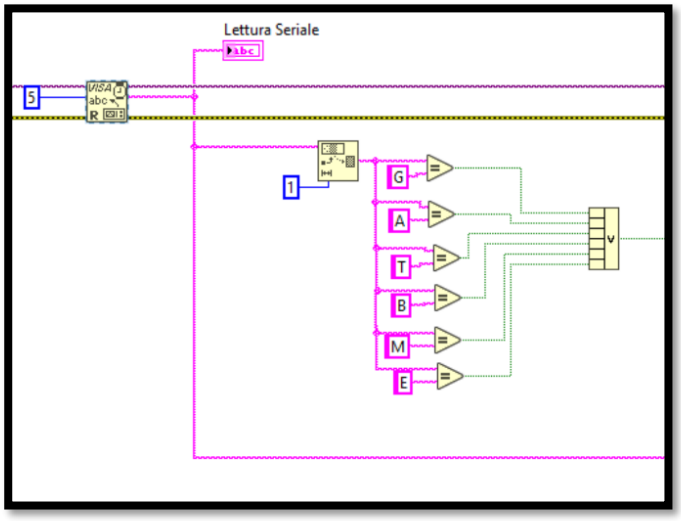
3.1.3 Terza sezione

In questa sezione, oltre a ricevere i dati provenienti da Arduino, elaborarli e li visualizzarli nel *Frontal Panel*, LabVIEW calcola anche i parametri statistici dell'errore.

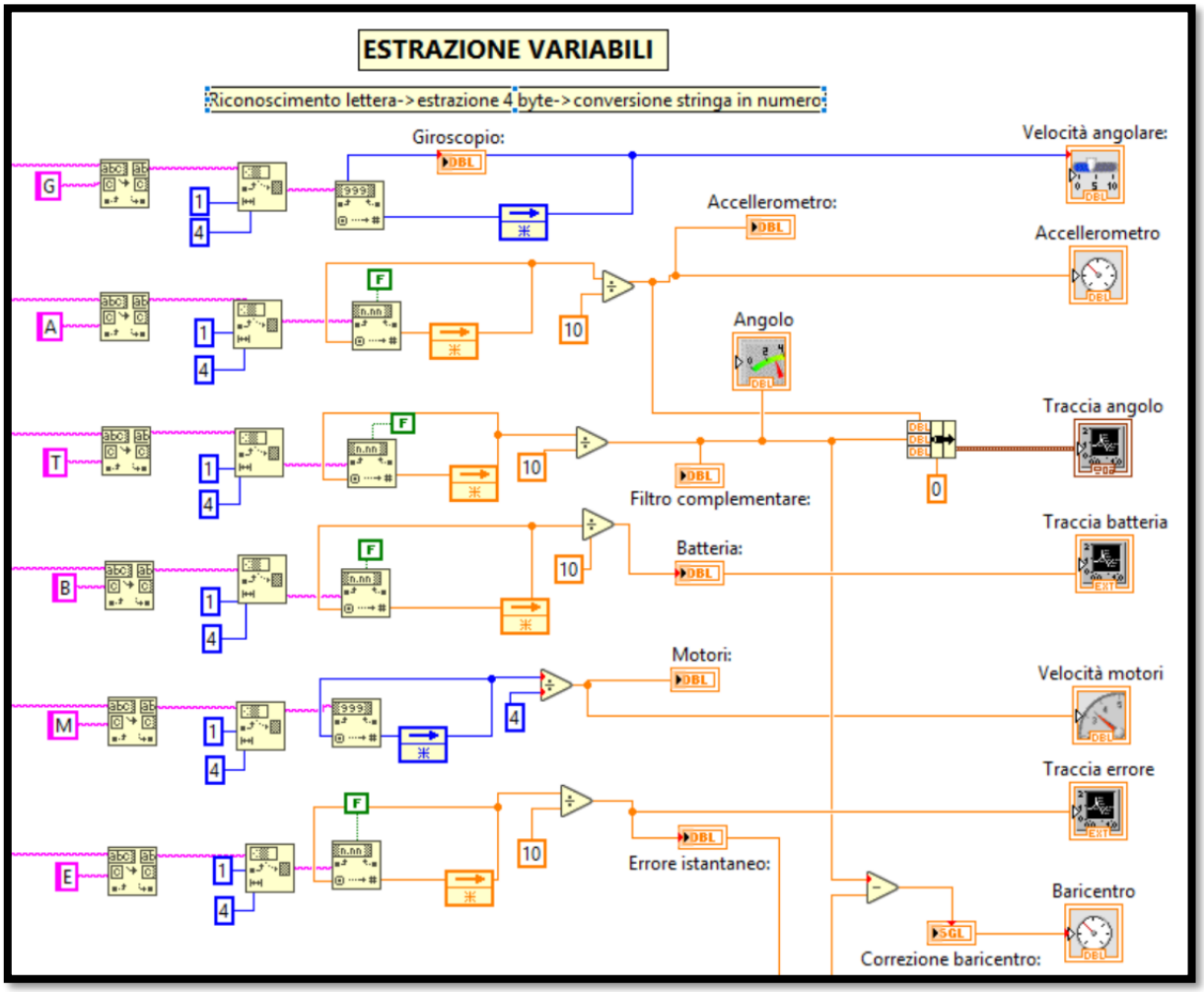
In primo luogo, vi è la lettura dei dati dalla seriale: per facilitare questa operazione, poiché i dati inviati sono molteplici (angolo dato dall'accelerometro, assetto, errore commesso, velocità dei motori e livello corrente della batteria), si è pensato di creare dei pacchetti di 5 bytes e il primo byte di ognuno identifica il tipo di informazione portato; una volta letto un pacchetto, si passa al confronto del primo byte.

Se questo non è uguale a nessuna delle lettere indicate significa che il pacchetto letto non contiene informazioni utili ed è necessario far in modo che LabVIEW si risincronizzi con i pacchetti inviati da Arduino. Per rendere possibile la sincronizzazione, legge un singolo byte prima di leggere un nuovo pacchetto; nel caso peggiore, ovvero nel caso in cui sia stato confrontato il secondo byte del pacchetto e non il primo, LabVIEW effettuerà altre 3 letture prima di leggere un pacchetto contenente informazioni utili. La logica implementata per lo

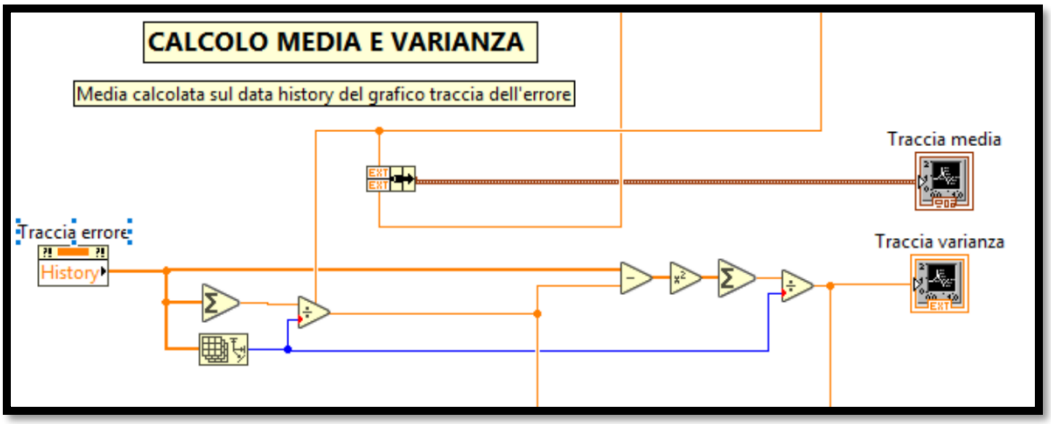
svolgimento di queste operazioni è mostrata nelle seguenti figure:



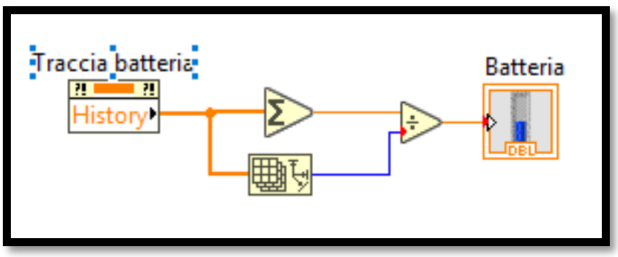
Nel caso in cui non sia necessaria la sincronizzazione, si passa all'elaborazione e visualizzazione dei dati.



Dopo aver creato due sub-stringhe, isolando il primo byte del pacchetto che si sa essere una lettera, i restanti 4 bytes vengono convertiti opportunamente da stringa in un numero. Il numero ottenuto viene diviso per 10 prima di essere visualizzato, perché precedentemente era stato moltiplicato per rendere tutti i valori degli interi e risparmiare un byte per la creazione dei pacchetti. Il solo valore che non viene diviso per 10 è il valore rappresentativo della velocità dei motori; esso viene diviso per 4 in modo da ottenere la velocità percentuale.



La media dell'errore (di conseguenza anche la varianza) è calcolata utilizzando i 100 valori del *data history* del grafico che traccia l'errore.



Un ultimo dato elaborato è il livello di batteria calcolato come media dei dati prelevati dal *data history* del grafico che tiene traccia del valore della batteria.

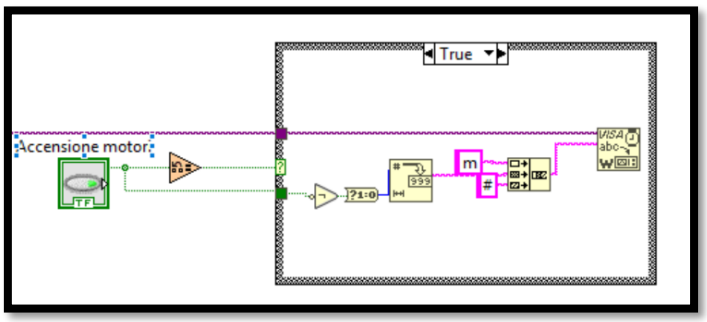
3.1.4 Quarta sezione

In questa macro-sezione del diagramma a blocchi LabVIEW invia dati, o meglio dire comandi, ad Arduino tramite ovviamente la comunicazione seriale.

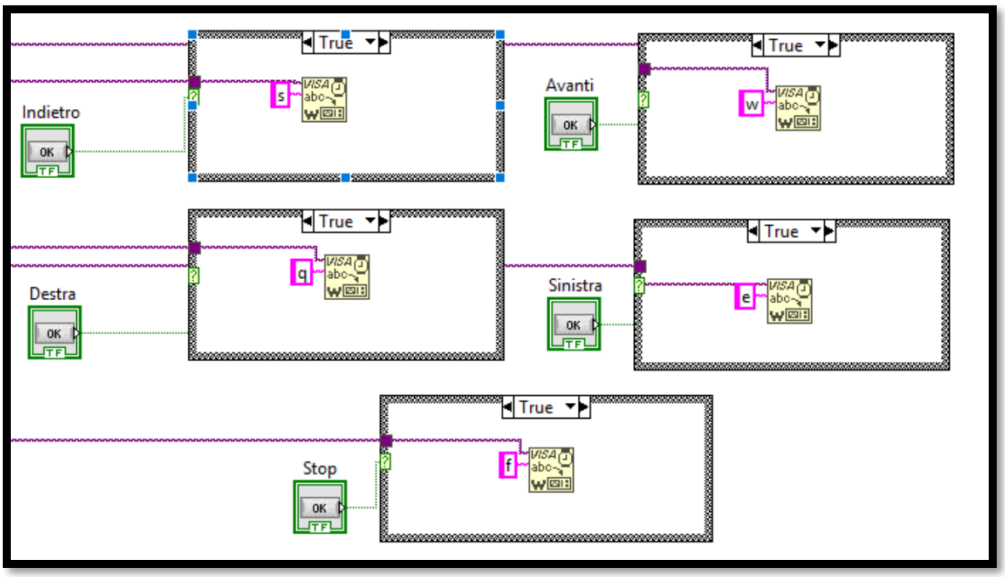
Quando dal *Frontal Panel* viene selezionato un comando, il *Case Structure* corrispondente passa a true e LabVIEW provvede all'invio dei dati.

I comandi che vengono inviati sono di 4 tipi:

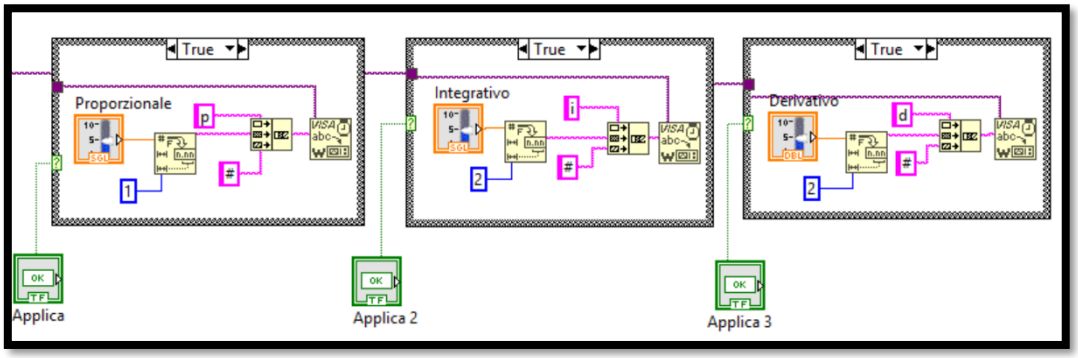
1. Accensione o spegnimento dei motori



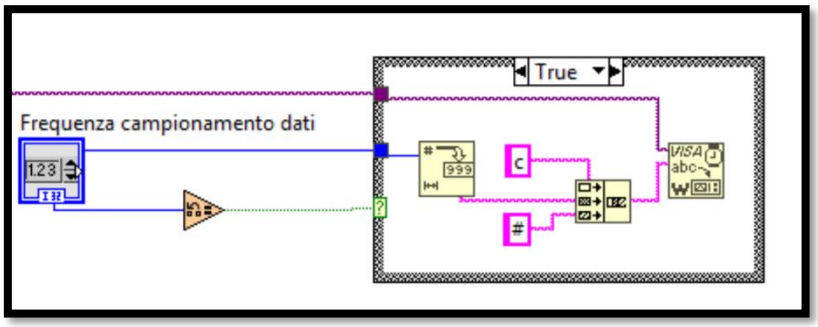
2. Direzione dei motori



3. Modifica dei parametri del controllore PID



4. Settaggio della frequenza di campionamento dati

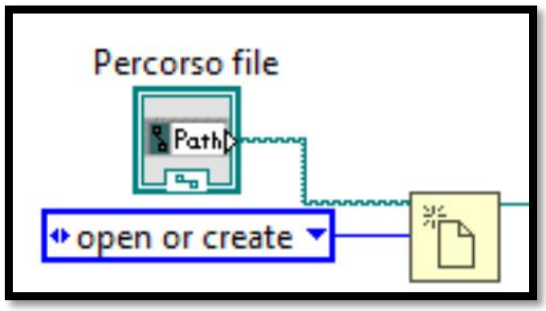


3.1.5 Quinta sezione

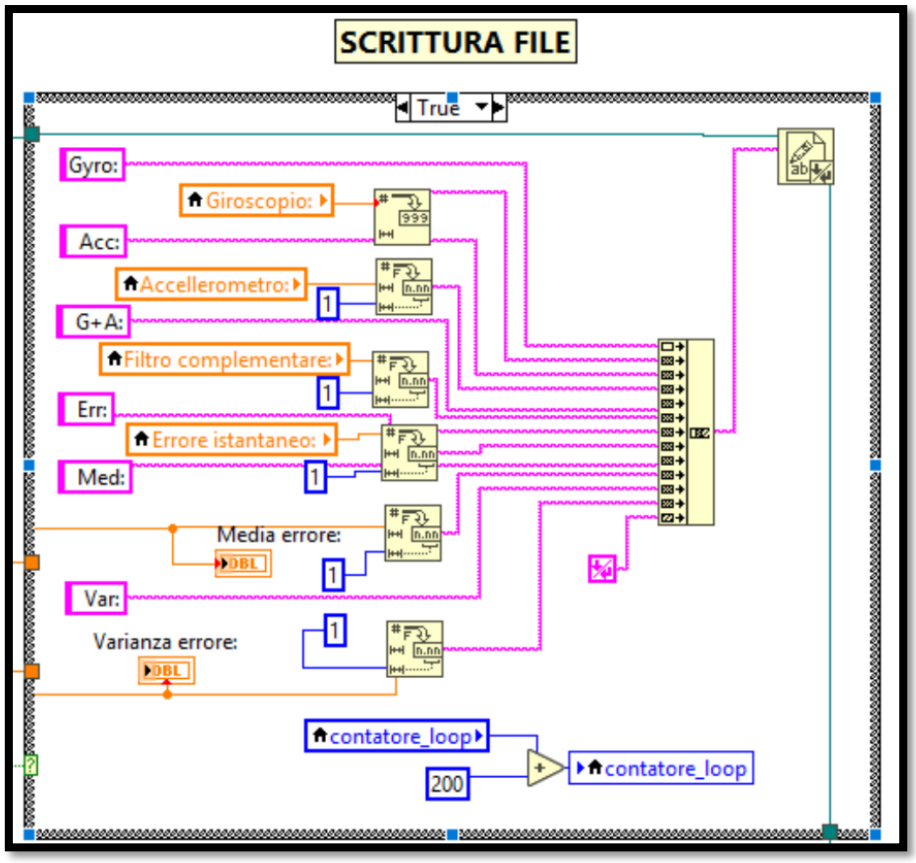
L'ultima fase del lavoro prevede la creazione di un file di testo in cui vengono salvati ogni 200 loops tutti i dati di interesse.

Per la creazione del file sono necessarie 3 fasi:

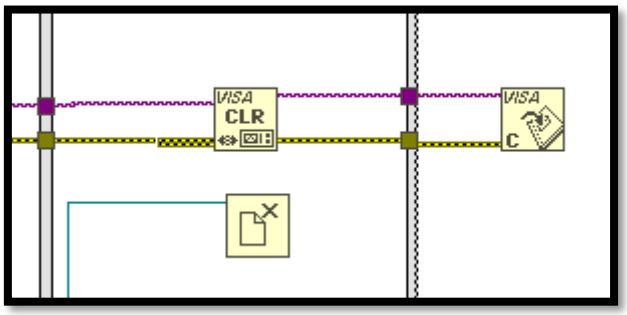
1. Creazione ed apertura del file: tramite apposito blocco, a cui bisogna passare il path del file da creare e il comando da eseguire, viene creato e aperto il file:



2. Scrittura del file: all'interno di un *Case Structure*, che diventa vero solo se il numero di loop è maggiore di 200, vengono scritti all'interno del file i dati, ovvero angolo dell'accelerometro, angolo del giroscopio, assetto, errore istantaneo, media e varianza dell'errore:



3. Chiusura del file: come per la comunicazione seriale, è necessario a fine sessione lavoro chiudere il file creato; si provvede anche a "pulire" il bus della seriale prima di chiudere anch'essa.



Conclusioni

Non possiamo che essere soddisfatti del risultato ottenuto. Nonostante il modello matematico sia abbastanza semplice, la realizzazione di un sistema così complesso non è affatto banale, soprattutto per quanto concerne il controllo su cui si basa tutto il funzionamento del pendolo inverso. La modifica anche minima di un solo parametro o l'aggiunta di una funzionalità comporta un comportamento diverso del robot, poiché ogni sotto-sistema lavora in simbiosi con gli altri. C'è quindi un "equilibrio" molto precario per quanto riguarda lo sviluppo del software e della logica di controllo; ciò è dovuto alle limitazioni hardware date da Arduino Uno che, seppur ha sorpreso noi stessi per le sue visibili e nascoste potenzialità, offre una potenza di calcolo limitata rispetto ad altri tipi di schede programmabili. Sotto questo punto di vista, il progetto si potrebbe sicuramente sviluppare ulteriormente se costruito intorno un microprocessore più potente, magari sempre della linea Arduino.

Questo discorso considera anche l'acquisizione e l'invio dei dati di misura, di cui questo progetto è in fin dei conti tema d'esame. Il collo di bottiglia è stato certamente l'abilità di calcolo disponibile, basti pensare al calcolo dell'angolo tramite accelerometro: esso è avuto tramite un arcoseno dell'accelerazione misura solo su un asse, mentre invece un risultato più preciso avrebbe richiesto l'arcotangente di misure su almeno due assi. Oltre al calcolatore, anche un sensore più sofisticato avrebbe potuto fare la differenza ovviamente, con magari letture di un numero maggiore di assi rispetto a sei. Anche questo, tuttavia, avrebbe comportato una potenza di elaborazione dati da parte del microcontrollore superiore a quella offerta dall'hardware utilizzato.

Considerando questo, è eccezionale ed esaltante poter vedere il risultato finale ottenuto dispositivi tra i più economici e di basso livello.