6

Software

6.1 Introducción y Estructura

En el presente apartado se hará un recorrido por el código utilizado en *Arduino Mega* para la configuración de los diferentes dispositivos electrónicos utilizados y el procesado para la obtención de los datos de interés por parte de los diferentes sensores. No se hará mención al código correspondiente al control del vehículo, pues esto se incluirá en la sección siguiente.

En la figura 6.1 se puede observar una captura del IDE (entorno de programación) de *Arduino*. Como ejemplo se tiene abierto un programa. Se puede observar que además aparecen otras dos pestañas al lado de la pestaña del archivo que se está visualizando. Estas otras dos pestañas corresponden a archivos que están asociados al archivo principal. Normalmente se trata de la continuación del mismo código del archivo principal, únicamente se escribe en otro archivo por cuestiones organizativas, pero no tiene otra razón práctica. Por otra parte, también aparecen como nuevas pestañas los archivos librerías creadas por el usuario (no incluidas en la carpeta de *Arduino*) y que están siendo usadas en el programa principal. Las pestañas que corresponden a librerías se identifican fácilmente porque además del nombre aparece la extensión .h del archivo librería.

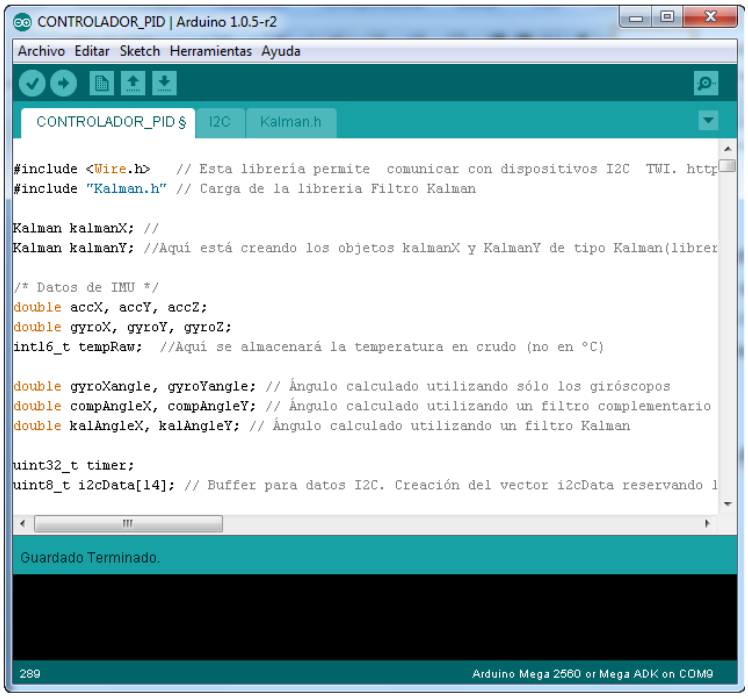

Figura 6.1: IDE de Arduino

Aunque no se detallará nada más del IDE de *Arduino*, se puede observar en la misma figura que en la parte superior incluye botones para la compilación y comprobación del código y para cargar el código en *Arduino* cuando se encuentra conectado mediante USB. Además de botones para crear una nueva pestaña (archivo asociado), y abrir nuevo archivo o guardar el actual.

Como ya se ha dicho, en esta memoria no se detallará el lenguaje de programación, aunque es análogo a otros muchos, con ligeras diferencias. Pero es conveniente puntualizar, al menos, para que aquellas personas que inicialmente no conocen nada del mismo puedan entender el código que en esta memoria se incluye, la estructura general que tienen todos los programas desarrollados en esta plataforma. En la figura 6.2 puede verse un ejemplo de esto. Se ha incluido un código de ejemplo sencillo que suele venir integrado como tutorial.

Archivo: Fade.ino

```
1  /*
2   Fade
3
4   This example shows how to fade a LED on pin 9
5   using the analogWrite() function.
6   This example code is in the public domain.
7   */
8
9  int led = 9;           // the pin that the LED is attached to
10 int brightness = 0;    // how bright the LED is
11 int fadeAmount = 5;    // how many points to fade the LED by
12
13 // the setup routine runs once when you press reset:
14 void setup()  {
15   // declare pin 9 to be an output:
16   pinMode(led, OUTPUT);
17 }
18
19 // the loop routine runs over and over again forever:
20 void loop()  {
21   // set the brightness of pin 9:
22   analogWrite(led, brightness);
23
24   // change the brightness for next time through the loop:
25   brightness = brightness + fadeAmount;
26
27   // reverse the direction of the fading at the ends of the fade:
28   if (brightness == 0 || brightness == 255) {
29     fadeAmount = -fadeAmount ;
30   }
31   // wait for 30 milliseconds to see the dimming effect
32   delay(30);
33 }
34
35
36
37
38
39
40
41
42
```

Bloque 1

Bloque 2

Bloque 3

Bloque 4

Figura 6.2: Bloques de un programa

En *Arduino* pueden distinguirse cuatro bloques principales comunes a todos los programas:

- **Bloque 1**: En estas líneas se suelen crear y definir variables y constantes, definir pines que se utilizarán; y cargar las librerías que se vayan a usar durante el programa. Al definir las variables y las constantes se indica de qué tipo van a ser. (Líneas 1 a 13).

- **Bloque 2**: La función *setup* es una de las funciones identificativas de *Arduino*. Es una función vacía (*void*), no devuelve ningún valor tras ejecutarse, sólo ejecuta el código en ella contenida. En este bloque se suelen realizar las configuraciones de los diferentes pines, puertos e/s, dispositivos, etc. que se utilicen a lo largo del programa. Este código sólo se ejecuta una vez al inicio del programa. (Líneas 14 a 17).

- **Bloque 3**: La función *loop* es otra de las funciones identificativas de *Arduino*. Al igual que *setup* *e*s una función vacía (*void*), no devuelve ningún valor tras ejecutarse, sólo ejecuta el código en ella contenida. Aquí es donde suele incluirse el cuerpo del programa. Éste se ejecuta una y otra vez ininterrumpidamente hasta que se apague el dispositivo. (Líneas 20 a 33).

- **Bloque 4**: Este bloque no siempre se utiliza. Sirve para declarar y definir funciones que se utilizan dentro de la función *loop*, pero que por cuestiones de organización y claridad no se definen allí. Estas funciones, aunque se invocan desde *loop*, su código está escrito en el bloque 4. (Líneas 34 en adelante).

En la figura también puede observarse texto en color gris que constituye los comentarios. Los comentarios se pueden incluir en una única línea, iniciando el comentario con los símbolos "//". Todo el texto en esta línea que sigue a esos símbolos será ignorado por el compilador. O también puede indicarse un fragmento de varias líneas de comentario, iniciándolo con los símbolos "/*" y cerrándolo con los símbolos "*/". Todas las líneas que se encuentren entre esos símbolos serán ignoradas por el compilador.

6.2 Estructura del Código Desarrollado

En este caso se comentará la estructura que tienen los programas que se han desarrollado para este proyecto. Todos los programas consistentes en una versión final o de prueba para el control del robot tienen una estructura similar, donde normalmente sólo se ha ido modificando las líneas de código correspondiente a las estrategias de control. El código correspondiente al controlador no se detallará en este apartado, sino en otros apartados posteriores.

En la figura 6.3 se detalla la estructura común de estos programas con las diferentes secciones. El programa que se muestra de ejemplo es uno correspondiente a la utilización de control LQR, donde se han eliminado la mayoría de comentarios aclaratorios para reducir la extensión. Se puede observar que cada bloque de los mencionados en anteriores párrafos se divide en diferentes secciones dedicadas a cada uno de los dispositivos o tareas que se necesitan llevar a cabo.

Cabe destacar que no se ha incluido todo el código utilizado en este programa, pues no está presente el código correspondiente a la librería *Kalman.h*, ni tampoco el código donde están definidas las funciones para la comunicación I2C, que correspondería al bloque 4, pero que está incluido en otro archivo, de nombre *I2C.ino*; aunque como se dijo previamente, a efectos prácticos funciona como si estuviera en el bloque 4.

Archivo: CONTROLADOR_LQR.ino

```
1   #include <Wire.h>
2   #include "Kalman.h"
3   Kalman kalmanX;
4   Kalman kalmanY;
5
6   /* PARA LOS DATOS DE LA IMU */
7   double accX, accY, accZ;
8   double gyroX, gyroY, gyroZ;
9   int16_t tempRaw;
10  double gyroXangle, gyroYangle;
11  double compAngleX, compAngleY;
12  double kalAngleX, kalAngleY;
13
14  uint32_t timer;
15  uint8_t i2cData[14];
16  ////////////////////////////
17
18  /* PARA LOS ENCODERS, VELOCIDAD, FILTRADO Y ACELERACIÓN */
19  //Izquierdo
20  #define encoder0PinA 18
21  #define encoder0PinB 19
22  volatile float encoder0Pos = 0;
23  unsigned long time1;
```

Carga de librerías

Definición de variables para la IMU

Definición de variables para *encoders*, posición, velocidad, filtrado y aceleración. […]

```
24      float pos0;
25      float veli;
26
27      //Derecho
28      #define encoder1PinA 2
29      #define encoder1PinB 3
30      volatile float encoder1Pos = 0;
31      unsigned long time2;
32      float pos0d;
33      float veld;
34
35      //Filtro y acel
36      float wf, wf_1, wf_2, wf0;
37      float vel_1, vel_2, acelf;
38      ////////////////////////
39
40      /* PARA EL MOTOR */
41      #define PinPWMD 7
42      #define PinPWMI 9
43      #define PinIn1I 40
44      #define PinIn2I 41
45      #define PinIn1D 43
46      #define PinIn2D 42
47      float PWMD;
48      float PWMI;
49      ////////////////////////
50
51      /* PARA EL CONTROL LQR*/
52      float u;
53      float phi_r,dphi_r,dtheta_r;
54      float phi, dphi;
55      float Ahora;
56      float Cambiot, Ultimot;
57      float Kp,Ki;
58      float r,e;
59      float Ahora1, Cambiot1, Ultimot1;
60      float ITerm;
61      float u_m;
62      ////////////////////////
63
64      /* PARA LAS MANIOBRAS DE PRUEBA */
65      float Tiempo0, Tiempo1;
66      ////////////////////////
67
68  void setup() {
69    Serial.begin(115200);
70
71    ////////// CÓDIGO IMU ////////////
72    Wire.begin();
73    TWBR = ((F_CPU / 400000L) - 16) / 2;
74    i2cData[0] = 7
75    i2cData[1] = 0x00;
76    i2cData[2] = 0x00;
77    i2cData[3] = 0x00;
78    while (i2cWrite(0x19, i2cData, 4, false));
79    while (i2cWrite(0x6B, 0x01, true));
80    while (i2cRead(0x75, i2cData, 1));
81    if (i2cData[0] != 0x68) {
82      Serial.print(F("Error leyendo sensor"));
83      while (1);
84    }
85    delay(100);
86
87    /* Ángulos iniciales cálculo kalman y con giróscopos*/
88    while (i2cRead(0x3B, i2cData, 6
89    accX = (i2cData[0] << 8) | i2cData[1];
90    accY = (i2cData[2] << 8) | i2cData[3];
91    accZ = (i2cData[4] << 8) | i2cData[5];
92
93    // Posteriormente se convierte de radianes a grados
94    double roll =atan2(accY,accZ)*RAD_TO_DEG;
95    double pitch=atan(-accX/sqrt(accY*accY+accZ*accZ))*RAD_TO_DEG;
96
97    // Inicializar ángulos
98    kalmanX.setAngle(roll);
99    kalmanY.setAngle(pitch);
100   gyroXangle = roll;
101   gyroYangle = pitch;
102   compAngleX = roll;
103   compAngleY = pitch;
104   timer = micros();
105   ////////////////////////////////////////////////////
106
107
108   /////// CÓDIGO PARA LAS INTERRUPCIONES //////
109   //Izquierdo
110   pinMode(encoder0PinA, INPUT);
111   digitalWrite(encoder0PinA, HIGH);
112   pinMode(encoder0PinB, INPUT);
113   digitalWrite(encoder0PinB, HIGH);
114   attachInterrupt(5, doEncoderA0, CHANGE);
115   attachInterrupt(4, doEncoderB0, CHANGE);
116   //Derecho
117   pinMode(encoder1PinA, INPUT);
118   digitalWrite(encoder1PinA, HIGH);
119   pinMode(encoder1PinB, INPUT);
120   digitalWrite(encoder1PinB, HIGH);
121   attachInterrupt(0, doEncoderA1, CHANGE);
122   attachInterrupt(1, doEncoderB1, CHANGE);
123   time1=millis();
```

[…]

Definición de variables y pines para las controladoras de motores.

Definición de variables para el control.

Código inicialización y configuración de la IMU […]

Configuración de interrupciones y sus pines.

```
124     time2=millis();
125     ////////////////////////////////////////////
126
127     ////////// MOTOR /////////////
128     TCCR4B = TCCR4B & 0b000 | 0x01;
129     TCCR2B = TCCR2B & 0b000 | 0x01;
130     pinMode(PinPWMD, OUTPUT);
131     pinMode(PinPWMI, OUTPUT);
132     pinMode(PinIn1I, OUTPUT);
133     pinMode(PinIn2I, OUTPUT);
134     pinMode(PinIn1D, OUTPUT);
135     pinMode(PinIn2D, OUTPUT);
136
137     ///CONTROLADOR////
138     phi_r=0;dphi_r=0;dtheta_r=0;
139     u=0;
140     Kp=0.2;
141     Ki=0.1;
142     ////////////////////////////////
143
144     ///FILTRO VELOCIDAD////
145     wf_2=1.0;
146     wf_1=1.0;
147     wf=1.0;
148     vel_2=1.0;
149     vel_1=1.0;
150     wf0=1.0;
151     ///////////////////////
152
153     //CONFIGURACIÓN PARA MANIOBRAS//
154     Tiempo0=millis();
155     ////////////////////////////////
156 }
157
158
159 void loop() {
160
161     /////CÓDIGO PARA LA IMU/////
162     /* Actualización de todos los valores */
163     while (i2cRead(0x3B, i2cData, 14));
164     accX = ((i2cData[0] << 8) | i2cData[1]) - 432.0;
165     accY = ((i2cData[2] << 8) | i2cData[3]) - 291.16;
166     accZ = ((i2cData[4] << 8) | i2cData[5]) - 307.52;
167     tempRaw = (i2cData[6] << 8) | i2cData[7];
168     gyroX = (i2cData[8] << 8) | i2cData[9];
169     gyroY = (i2cData[10] << 8) | i2cData[11];
170     gyroZ = (i2cData[12] << 8) | i2cData[13];
171     double dt = (double)(micros()- timer) / 1000000; // Cálculo de dt
172     timer = micros();
173
174     // atan2 devuelve valores de -π a π (radianes)
175     // Posteriormente se convierte de radianes a grados
176     double roll  = atan2(accY, accZ) * RAD_TO_DEG;
177     double pitch = atan(-accX / sqrt(accY * accY + accZ * accZ)) * RAD_TO_DEG;
178     double gyroXrate = gyroX / 131.0;
179     double gyroYrate = gyroY / 131.0;
180
181     if ((roll < -90 && kalAngleX > 90) || (roll > 90 && kalAngleX < -90)) {
182       kalmanX.setAngle(roll);
183       compAngleX = roll;
184       kalAngleX = roll;
185       gyroXangle = roll;
186     } else
187       kalAngleX = kalmanX.getAngle(roll, gyroXrate, dt);
188
189     if (abs(kalAngleX) > 90)
190       gyroYrate = -gyroYrate;
191     kalAngleY = kalmanY.getAngle(pitch, gyroYrate, dt);
192
193     // Cálculo del ángulo mediante giróscopos sin ningún filtro
194     gyroXangle += gyroXrate * dt;
195     gyroYangle += gyroYrate * dt;
196     // Cálculo del ángulo usando un filtro complementario
197     compAngleX = 0.93 * (compAngleX + gyroXrate * dt) + 0.07 * roll;
198     compAngleY = 0.93 * (compAngleY + gyroYrate * dt) + 0.07 * pitch;
199    // Reseteo del ángulo de los giróscopos cuando se ha producido mucha deriva
200     if (gyroXangle < -180 || gyroXangle > 180)
201       gyroXangle = kalAngleX;
202     if (gyroYangle < -180 || gyroYangle > 180)
203       gyroYangle = kalAngleY;
204     ////////////////////////////////
205
206
207
208
209     //////// CÓDIGO PARA POSICIÓN, VELOCIDAD Y ACELERACIÓN ////////
210     if ((millis()-time1)>=10){
211         veli=((encoder0Pos-pos0)*1000.0/(millis()-time1))*(PI/180.0);
212         pos0=encoder0Pos;
213         veld=((encoder1Pos-pos0d)*1000.0/(millis()-time1))*(PI/180.0);
214         pos0d=encoder1Pos;
215         vel=(veli+veld)/2;
216
217         //Filtrado de la velocidad(BUTTERWORTH CON N=2; Wn=0.0784):
218         wf=1.6544*wf_1 - 0.7059*wf_2 + 0.0129*vel + 0.0257*vel_1 + 0.0129*vel_2;
219         wf_2=wf_1;
220         wf_1=wf;
221         vel_2=vel_1;
222         vel_1=vel;
223
```

Configuración señal PWM y salidas para las controladoras de motores.

Inicialización de variables para:

-controlador

-filtro de velocidad

Código IMU para la obtención del ángulo y velocidad angular.

Cálculo de la velocidad, filtrado y aceleración. […]

```
      acelf=(wf-wf0)*1000.0/(millis()-time1);
      wf0=wf;
      time1=millis();}
////////////////////////////////////////////////////////////////////

/////// CÓDIGO PARA MANIOBRAS DEL ROBOT ///////
Tiempo1=millis();
if ((Tiempo1-Tiempo0)<4000){
   dtheta_r=0;}
else {if ((Tiempo1-Tiempo0)<9000){
        dtheta_r=3.8*PI;}
      else{ if ((Tiempo1-Tiempo0)<13000){
              dtheta_r=0;}
            else{if ((Tiempo1-Tiempo0)<18000){
                    dtheta_r=-3.8*PI;}
                 else{dtheta_r=0;}
                }
          }
     }
////////////////////////////////////////////

//////// CÓDIGO PARA CONTROLADOR ////////
Ahora=millis();
Cambiot=Ahora-Ultimot;

if (Cambiot>=40){
  phi=kalAngleY*(PI/180);
  dphi=gyroYrate*(PI/180);
  u=-332.8746*(phi_r-phi)-52.7835*(dphi_r-dphi)-3.1623*(dtheta_r-wf);
  r=-u*1.9;
  Ultimot=Ahora;}
Ahora1=millis();
Cambiot1=Ahora1-Ultimot1;

if (Cambiot1>=15){
  Ultimot1=Ahora1;
  e=r-acelf;
  ITerm += Ki*e;
  if((abs(ITerm))>255.0){
    if (ITerm>0) ITerm=255.0;
    else ITerm=-255.0;}
  u_m=Kp*e+ITerm;}


if (u_m<=0){
  digitalWrite(PinIn1I, HIGH);
  digitalWrite(PinIn2I, LOW);
  digitalWrite(PinIn1D, HIGH);
  digitalWrite(PinIn2D, LOW);
  PWMI=17+abs(u_m);
  PWMD=16+abs(u_m);}
else {
  digitalWrite(PinIn1I, LOW);
  digitalWrite(PinIn2I, HIGH);
  digitalWrite(PinIn1D, LOW);
  digitalWrite(PinIn2D, HIGH);
  PWMI=18+abs(u_m);
  PWMD=17+abs(u_m);}

if (PWMI>255){
  PWMI=255;
  PWMD=PWMI;}
if (abs(kalAngleY)>30){
  PWMI=0;
  PWMD=0;}
analogWrite(PinPWMI, PWMI);
analogWrite(PinPWMD, PWMD);
////////////////////////////////////////////////////////////////////

/* ENVÍO DE DATOS */
#if 0 // Cambiar a 1 para activar
  Serial.print(r); Serial.print("\t");
  Serial.print(acel); Serial.print("\t");
  Serial.print(phi); Serial.print("\t");
  Serial.print(dphi); Serial.print("\t");
  Serial.print(vel); Serial.print("\t");
  Serial.print(millis()); Serial.print("\t");
  Serial.print("\t");
#endif
#if 0 // Cambiar a 1 para activar
  Serial.print(accX); Serial.print("\t");
  Serial.print(accY); Serial.print("\t");
  Serial.print(accZ); Serial.print("\t");

  Serial.print(gyroX); Serial.print("\t");
  Serial.print(gyroY); Serial.print("\t");
  Serial.print(gyroZ); Serial.print("\t");
  Serial.print("\t");
#endif
#if 0 // Cambiar a 1 para activar
  Serial.print(roll); Serial.print("\t");
  Serial.print(gyroXangle); Serial.print("\t");
  Serial.print(compAngleX); Serial.print("\t");
  Serial.print(kalAngleX); Serial.print("\t");
  Serial.print("\t");
  Serial.print(pitch); Serial.print("\t");
  Serial.print(gyroYangle); Serial.print("\t");
```

Programación de maniobras de prueba.

Código para el controlador y mando de los motores.

Envío de datos por puerto serie […]

```
      Serial.print(compAngleY); Serial.print("\t");
      Serial.print(kalAngleY); Serial.print("\t");
    #endif
    #if 0 // Cambiar a 1 para activar
      Serial.print("\t");
      double temperature = (double)tempRaw / 340.0 + 36.53;
      Serial.print(temperature); Serial.print("\t");
    #endif
    #if 0
      Serial.print("\t");
      Serial.print(encoder0Pos); Serial.print("\t");
      Serial.print(encoder1Pos); Serial.print("\t");
      Serial.print(velf); Serial.print("\t");
      Serial.print(velfd); Serial.print("\t");
      Serial.print(acel); Serial.print("\t");
      Serial.print(aceld); Serial.print("\t");
      Serial.print("\t");
      Serial.print(Kp); Serial.print("\t");
      Serial.print(Ki); Serial.print("\t");
      Serial.print(Kd); Serial.print("\t");
    #endif
    #if 0 // Cambiar a 1 para activar
      Serial.print(u); Serial.print("\t");
      Serial.print(PWMI); Serial.print("\t");
      Serial.print(kalAngleY); Serial.print("\t");
    #endif
      Serial.print("\r\n");
      delay(1);
    //////////////////////////////////////////
  }

  ///// A CONTINUACIÓN VIENEN LAS FUNCIONES DE LAS
  /// INTERRUPCIONES PARA LOS ENCODERS ///////
  //INTERRUPCIONES ENCODERS MOTOR IZQUIERDO
  void doEncoderA0(){
    if (digitalRead(encoder0PinA) == HIGH) {
      if (digitalRead(encoder0PinB) == LOW)
        encoder0Pos++;}        // Horario
      else {encoder0Pos--;}    // Antihorario
    }
    else{
      if (digitalRead(encoder0PinB) == HIGH) {
        encoder0Pos++;}          // Horario
      else {encoder0Pos--;}      // Antihorario
    }
   }
  void doEncoderB0(){
    if (digitalRead(encoder0PinB) == HIGH) {
      if (digitalRead(encoder0PinA) == HIGH) {
        encoder0Pos++;}          // Horario
      else {encoder0Pos--;}      // Antihorario
    }
    else {
      if (digitalRead(encoder0PinA) == LOW) {
        encoder0Pos++;}          // Horario
      else {encoder0Pos--;}      // Antihorario
    }
   }

   //INTERRUPCIONES ENCODERS MOTOR DERECHO
  void doEncoderA1(){
    if (digitalRead(encoder1PinA) == HIGH) {
      if (digitalRead(encoder1PinB) == LOW) {
        encoder1Pos--;}        // Horario
      else {encoder1Pos++;}    // Antihorario
    }
    else{
      if (digitalRead(encoder1PinB) == HIGH) {
        encoder1Pos--;}          // Horario
      else {encoder1Pos++;}      // Antihorario
    }
   }
  void doEncoderB1(){
    if (digitalRead(encoder1PinB) == HIGH) {
      if (digitalRead(encoder1PinA) == HIGH) {
  gira
        encoder1Pos--;}          // Horario
      else {encoder1Pos++;}      // Antihorario
    }

    else {
      if (digitalRead(encoder1PinA) == LOW) {
        encoder1Pos--;}          // Horario
      else {encoder1Pos++;}      // Antihorario
    }
   }
```

[…]

Funciones de las interrupciones para los *encoders*.

Figura 6.3: Estructura de un Programa

Aunque ya se ha indicado dentro de la figura 6.3, se volverá a hacer un esquema, para mayor claridad, de la división de este programa tipo, en bloques y secciones, indicando las líneas de código que abarca cada sección:

- **Bloque 1: Definiciones** (líneas 1 a 67)
  - Carga de las librerías *Wire.h* y *Kalman.h* (líneas 1 a 4).
  - Definición de las variables para la IMU (líneas 6 a 16).
  - Definición de variables y pines para los *encoders*, cálculo de velocidad, filtrado y aceleración (líneas 18 a 38).
  - Definición de variables y pines para las controladoras de los motores (líneas 40 a 49).
  - Definición de variables para el control (líneas 51 a 66).

- **Bloque 2: *Setup*** (líneas 68 a 156)
  - Iniciar puerto serie a 115200 baudios (línea 69).
  - Código de inicialización y configuración de la IMU (líneas 71 a 105).
  - Configuración de interrupciones y sus pines (líneas 108 a 125).
  - Configuración de las señales PWM y pines de salida para las controladoras de los motores (líneas 127 a 135).
  - Inicialización de variables para: el controlador, el filtro de velocidad y configuración de maniobras de prueba (exclusivo de este programa); (líneas 137 a 155).

- **Bloque 3: *Loop*** (líneas 159 a 353)
  - Código de la IMU para la obtención del ángulo y la velocidad angular (líneas 161 a 204).
  - Cálculo de la velocidad, filtrado y aceleración (líneas 209 a 227).
  - Programación de maniobras de prueba (exclusivo de este programa); (líneas 230 a 244).
  - Programación del controlador y mando de los motores (líneas 247 a 293).
  - Envío de datos por el puerto serie (líneas 296 a 352).

- **Bloque 4: Funciones no definidas en *loop*** (líneas 355 en adelante)
  - Funciones que se ejecutan con las interrupciones de los *encoders* (líneas 355 a 409).
  - Funciones para la comunicación I2C (contenidas en el archivo *I2C.ino*).

En los siguientes apartados se explicará la mayoría de las secciones anteriormente comentadas, así como los cálculos y estrategias utilizadas en la obtención de los datos necesarios para el control, y cómo se le ha dado solución a los diferentes retos que han ido apareciendo a lo largo del proyecto. De esta manera también se hará hincapié en algunas decisiones en cuanto al código.

6.3 Interrupciones para los *Encoders*

El funcionamiento de los *encoders* ya se detalló en el apartado 4.7, donde además se indicó que éstos tienen una resolución de 360 pulsos por vuelta de rueda cuando se utilizan dos interrupciones por cada eje, que es la opción escogida. Por tanto se puede considerar que un pulso corresponde a un grado de ángulo girado por la rueda.

Para afrontar la escritura del código correspondiente a estos sensores, hay que tener claro que la idea consiste en conseguir una variable que almacene la posición, en grados de la rueda. De manera que si gira en el sentido declarado como positivo, el ángulo se vaya incrementando correctamente; y que si cambia de sentido, el ángulo vaya disminuyendo como sucede en la realidad.

Para realizar esto lo más intuitivo es introducir la señal de salida de los sensores por un pin digital cualquiera de *Arduino* y desarrollar unas líneas de código dentro del bloque *loop*, de manera que exista una variable que sume o reste un valor 1 cuando se reciba un pulso por el correspondiente pin de entrada. Pero esta estrategia tiene un problema, y es que si la rueda gira a una velocidad alta, de

manera que el tiempo entre pulsos del *encoder* sea menor que lo que tarda Arduino en ejecutar todas las líneas de código del programa, habría casos en los que no se detectaran pulsos del *encoder*. Es por esta razón que se utilizan los pines de interrupciones hardware que tiene *Arduino Mega*.

Una interrupción hardware implica que cuando el microcontrolador detecta un valor alto, o bajo, o un cambio de bajo a alto, o un cambio cualquiera (esto es definible por el usuario), se dispara la ejecución de una función, sea cual sea el punto por el que se está recorriendo el programa. De esta manera no importa que la velocidad de giro de las ruedas sea muy elevada, pues ya se sabe que la función de interrupción se ejecutará siempre que el sensor de *encoder* emita un pulso, no importa cuán lento se esté recorriendo el programa. Por eso se llaman interrupciones, pues el encoder interrumpe lo que está haciendo el procesador y lo manda a ejecutar las funciones de interrupción.

Variables Encoders

```
1   /* PARA LOS ENCODERS, VELOCIDAD, FILTRADO Y ACELERACIÓN */
2   //Izquierdo
3   #define encoder0PinA 18
4   #define encoder0PinB 19
5   volatile float encoder0Pos = 0;
6   unsigned long time1;
7   float pos0;
8   float veli;
9
10  //Derecho
11  #define encoder1PinA 2
12  #define encoder1PinB 3
13  volatile float encoder1Pos = 0;
14  unsigned long time2;
15  float pos0d;
16  float veld;
17
```

Figura 6.4: Definición de Variables para los Encoders

En la figura 6.4 se ha vuelto a incluir el trozo de código donde se definen las variables y pines para la lectura de los encoders. Se observa que los pines se definen con la orden *#define*, seguida del nombre que se le da al pin y su valor. Esta orden es heredada del lenguaje C y permite al programador dar un nombre a una constante antes de que el programa sea compilado, de esta forma estas constantes no ocupan ningún espacio de memoria en *Arduino*. También se observa que los pines escogidos son 2, 3, 18 y 19 (recordar que hay dos canales o sensores por cada encoder), que como se explicó en el apartado 4.4 son los únicos dedicados a interrupciones externas en *Arduino Mega*, además del pin 21; pero éste ya se está empleando para las comunicaciones I2C.

Las variables *encoder0Pos* y *encoder1Pos* son las destinadas a almacenar la posición de cada rueda y son las que se manipularán en las funciones de interrupción que se verán más adelante. Se puede observar que al definir estas variables han sido precedidas por el calificador *volatile.* Esto indica al compilador que cargue la variable desde la RAM y no desde un registro de almacenamiento, pues bajo ciertas condiciones el valor de una variable almacenada en estos registros puede ser impreciso. Como estas variables se modifican en las funciones de interrupción, es más preciso declararlas como volátiles y así evitar errores. Además se decidió que estas variables fueran de tipo flotante para evitar desbordamientos.

Configuración Interrupciones

```
1   /////// CÓDIGO PARA LAS INTERRUPCIONES //////
2   //Izquierdo
3   pinMode(encoder0PinA, INPUT);
4   digitalWrite(encoder0PinA, HIGH);       // Activar resistencias pullup
5   pinMode(encoder0PinB, INPUT);
6   digitalWrite(encoder0PinB, HIGH);       // Activar resistencias pullup
7
8   attachInterrupt(5, doEncoderA0, CHANGE);  // Encoder A en la interrupción 5 (pin 18)
```

```
9     attachInterrupt(4, doEncoderB0, CHANGE);  // Encoder B en la interrupción 4 (pin 19)
10
11    //Derecho
12    pinMode(encoder1PinA, INPUT);
13    digitalWrite(encoder1PinA, HIGH);       // Activar resistencias pullup
14    pinMode(encoder1PinB, INPUT);
15    digitalWrite(encoder1PinB, HIGH);       // Activar resistencias pullup
16
17    attachInterrupt(0, doEncoderA1, CHANGE);  // Encoder A en la interrupción 0 (pin 2)
18    attachInterrupt(1, doEncoderB1, CHANGE);  // Encoder B en la interrupción 1 (pin 3)
19
```

Figura 6.5: Configuración de las interrupciones

En la figura 6.5 puede verse el trozo de código correspondiente a la configuración de pines e interrupciones. Este código esta dentro de la función vacía *setup* (anteriormente denominado bloque 2). Cada pin de entrada o salida hay que configurarlo como tal, lo que puede observarse en las líneas 3, 4, 12, 14. Además se activan las resistencias *pull-up* internas del procesador (de las que se habló en el apartado 4.2), mediante una orden de escritura digital en valor alto, a pesar de que los pines se han declarado como entrada. Esto puede verse en las líneas 4, 6, 13, 15.

La configuración de las interrupciones se hace mediante la orden *attachInterrupt*, que tiene como argumentos el número de la interrupción (diferente del pin correspondiente a la interrupción), el nombre de la función de interrupción que se ha de ejecutar, y el evento que disparará la función de interrupción. En este caso el evento elegido es cualquier cambio en el valor, como se explicó en el apartado 4.7. Esto puede verse en las líneas de código 8, 9, 17 y 18.

Funciones de Interrupción

```
1   ///// A CONTINUACIÓN VIENEN LAS FUNCIONES DE INTERRUPCIÓN PARA LOS ENCODERS /////
2   //INTERRUPCIONES ENCODERS MOTOR IZQUIERDO
3   void doEncoderA0(){
4     if (digitalRead(encoder0PinA) == HIGH) {  //busca un cambio de bajo-a-alto en el canal A
5       if (digitalRead(encoder0PinB) == LOW) { //comprueba el canal B para saber sentido giro
6         encoder0Pos++;}        // Horario
7       else {encoder0Pos--;}    // Antihorario
8     }
9
10    else{  //en otro caso sería un cambio de alto-a-bajo en canal A
11      if (digitalRead(encoder0PinB) == HIGH) { //comprueba el canal B para saber sentido giro
12        encoder0Pos++;}          // Horario
13      else {encoder0Pos--;}      // Antihorario
14    }
15  }
16
17  void doEncoderB0(){
18    if (digitalRead(encoder0PinB) == HIGH) {   //busca un cambio de bajo-a-alto en el canal B
19      if (digitalRead(encoder0PinA) == HIGH) { //comprueba el canal A para saber sentido giro
20        encoder0Pos++;}          // Horario
21      else {encoder0Pos--;}      // Antihorario
22    }
23
24    else {  // en otro caso sería un cambio de alto-a-bajo en canal B
25      if (digitalRead(encoder0PinA) == LOW) {    //comprueba el canal A para saber sentido giro
26        encoder0Pos++;}          // Horario
27      else {encoder0Pos--;}      // Antihorario
28    }
29  }
30
31   //INTERRUPCIONES ENCODERS MOTOR DERECHO
32  void doEncoderA1(){
33    if (digitalRead(encoder1PinA) == HIGH) {   //busca un cambio de bajo-a-alto en el canal A
34      if (digitalRead(encoder1PinB) == LOW) {  //comprueba el canal B para saber sentido giro
35        encoder1Pos--;}        // Horario
36      else {encoder1Pos++;}    // Antihorario
37    }
38
39    else{  // en otro caso sería un cambio de alto-a-bajo en canal A
40      if (digitalRead(encoder1PinB) == HIGH) {  //comprueba el canal B para saber sentido giro
41        encoder1Pos--;}          // Horario
42      else {encoder1Pos++;}      // Antihorario
43    }
44   }
45
46  void doEncoderB1(){
```

```
47      if (digitalRead(encoder1PinB) == HIGH) {   //busca un cambio de bajo-a-alto en el canal B
48        if (digitalRead(encoder1PinA) == HIGH) { //comprueba el canal A para saber sentido giro
49          encoder1Pos--;}          // Horario
50        else {encoder1Pos++;}      // Antihorario
51      }
52
53      else {  // en otro caso sería un cambio de alto-a-bajo en canal B
54        if (digitalRead(encoder1PinA) == LOW) {  //comprueba el canal A para saber sentido giro
55          encoder1Pos--;}          // Horario
56        else {encoder1Pos++;}      // Antihorario
57      }
58    }
59
```

Figura 6.6: Funciones de Interrupción

La figura 6.6 incluye las funciones de interrupción, es decir, el código que se ejecuta cuando se disparan las interrupciones configuradas en el código mostrado en la figura 6.5. No se explicará la lógica seguida, puesto que ya se explicó en el apartado 4.7, y además está bastante detallado mediante los comentarios. En lo único que consisten estas funciones es detectar si se está girando en sentido horario o anti horario. Si es lo primero se resta uno a la variable y si es lo segundo se suma uno.

El que el sentido horario corresponda al valor negativo únicamente depende de la posición de montaje de los motores y lo que se ha considerado como sentido hacia adelante y hacia atrás en el robot.

6.4 Cálculo de la Velocidad y la Aceleración

6.4.1 Cálculo de la Velocidad

En líneas generales existen dos estrategias o algoritmos para calcular la velocidad utilizando *encoders* incrementales, a tiempo fijo o a espacio fijo.

- **Tiempo fijo:** consiste en contar el número de pulsos (en este caso grados) que transcurren durante un tiempo constante determinado. La velocidad se obtiene dividiendo el ángulo (en este caso los grados contados) por el tiempo fijo transcurrido.

- **Espacio fijo:** consiste en cronometrar el tiempo que pasa en incrementar el ángulo un grado. La velocidad se obtiene de la misma forma, dividiendo el ángulo (en este caso un grado) por el tiempo medido.

La estrategia a espacio fijo proporciona más precisión a velocidades bajas, pero a velocidades altas, el tiempo que transcurre entre un pulso y el siguiente es tan pequeño que es menor que el tiempo que el programa tarda en ejecutarse, por lo que la precisión del cálculo disminuye, alterándose los valores. De forma análoga, la estrategia a tiempo fijo proporciona más precisión a velocidades altas, pero si la velocidad va bajando, se llega a un punto en el que es tan baja que entre un pulso y el siguiente el tiempo transcurrido es mayor que el tiempo fijado, por lo que lo considera velocidad cero.

Inicialmente se consideró que las velocidades a las que funcionaría el aparato superarían al límite de precisión para el algoritmo a tiempo fijo, por lo que fue éste el algoritmo escogido, puesto que daba menos problemas. Pero el problema que tiene el cálculo de la velocidad mediante estos algoritmos es que se introducen ruidos en la medida. Estos ruidos se deben al hecho de que los *encoders* incrementales son dispositivos que miden la posición (el ángulo en este caso) de forma discreta, por lo que a veces, con una misma velocidad real constante, se mide algún pulso de más o algún pulso de menos, dando oscilaciones en la medida. Esto es lo que se conoce como error de truncamiento.

Un ejemplo del código desarrollado para implementar el algoritmo a tiempo fijo puede verse en la figura 6.7.

Cálculo de la velocidad

```
1   if ((millis()-time1)>=10){
2       veli=((encoder0Pos-pos0)*1000.0/(millis()-time1));
3       pos0=encoder0Pos;
4       veld=((encoder1Pos-pos0d)*1000.0/(millis()-time1));
5       pos0d=encoder1Pos;
6       vel=(veli+veld)/2;
7       time1=millis();
8    }
9
```

Figura 6.7: Cálculo de la velocidad

En ese código las variables *veli* y *pos0* corresponden a la velocidad y posición de la rueda izquierda, mientras que las variables *veld* y *pos0d* corresponden a la velocidad y posición de la rueda derecha. El cuerpo del código se encuentra dentro de un condicional *if*, de manera que sólo se ejecuta cuando se cumple la condición. La condición establecida es que entre cálculo y cálculo hayan pasado 10 ms. Por tanto ese es el tiempo fijo escogido en este caso.

Además se observa, en las líneas 2 y 4, (se multiplica por 1000 para pasar de ms a segundos), que el cálculo de la velocidad se realiza como:

$$velocidad = \frac{pos\ actual - pos\ anterior}{tiempo\ transcurrido}$$

Los valores de la posición anterior se actualizan en las líneas 3 y 5, después del cálculo. Y el valor final de la velocidad se obtiene como valor medio de las velocidades de ambas ruedas, aunque esto no se ha realizado en todos los programas. El tiempo anterior también se actualiza en la línea 7, donde se introduce mediante la función *millis()*, que devuelve el tiempo actual en milisegundos.

6.4.2 Algoritmo combinado

Cuando se estaba desarrollando la estrategia de control LQR, que se verá más adelante, se detectó que el control sucedía demasiado lento, como si hubiera algo que ralentizara el sistema. Se llegó a la conclusión de que el problema residía en que la actualización del cálculo de la velocidad se realizaba demasiado lento.

Al principio, este tiempo de actualización se fijó en 50 ms, creyendo que sería suficientemente rápido para un control óptimo del vehículo. Además se decidió esta duración porque para tiempos más cortos el error en la medida se incrementaba. Esto se pudo comprobar comparando las mediciones al utilizar tiempos fijos muy largos y tiempos fijos muy cortos, observándose que con tiempos fijos muy cortos se producía mucho más ruido en la medida. Todo esto se debía al error de truncamiento, pues al ser el tiempo fijo más corto el error relativo era mayor (por ejemplo, el que se cuele un pulso de *encoder* frente a un total de 2 pulsos medidos supone mucho mayor error que el colarse un pulso frente a 50 pulsos medidos).

Intentando resolver este problema para poder reducir el tiempo fijo, se consultó un artículo, que está incluido en la bibliografía, donde se detallaba un algoritmo que combinaba las estrategias a tiempo fijo y a espacio fijo, de manera que el error de truncamiento se reducía, y al combinar ambas estrategias el algoritmo era válido tanto para bajas como para altas velocidades. A continuación se expone este algoritmo tal y como lo explican sus autores.

6.4.2.1 Descripción del algoritmo combinado por sus autores

El corazón del algoritmo está en la sincronización de los impulsos que indican posición y tiempo. El objetivo fundamental es eliminar el error de truncamiento, y en su lugar tener una medida de velocidad estable, sin necesidad de filtros.

El único tratamiento que se le hace a la señal proveniente del *encoder* es la detección de los flancos de subida, con lo que se genera el tren de impulsos denominado Iep como se muestra en la figura 6.8. Si la velocidad es constante, la separación entre impulsos Iep es constante. Los pulsos son contados por medio de un acumulador, lo cual genera la señal Cep. Cuando ese conteo se va a 0, porque un evento hasta ahora no descrito reinicia el acumulador, el bloque retenedor conserva el último conteo y así se ha contado el número de pulsos, Nep.

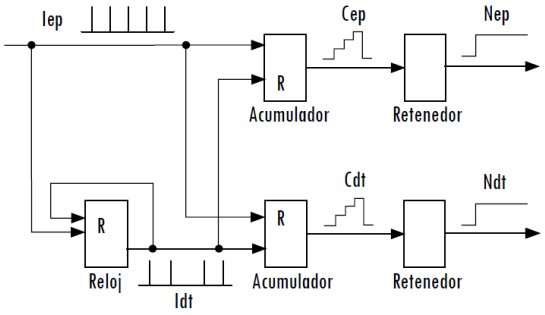

Figura 6.8: Diagrama de bloques del algoritmo combinado

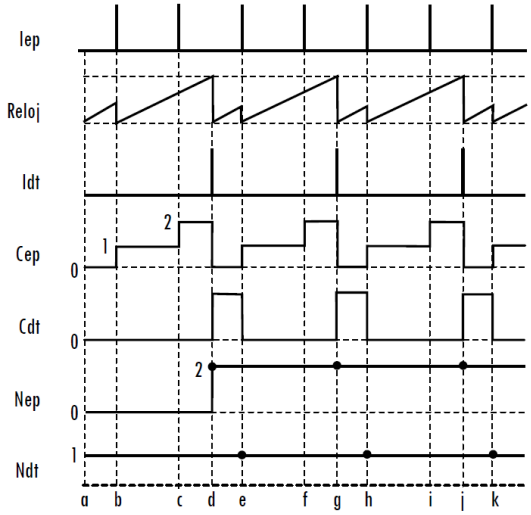

Figura 6.9: Señales cuando w=1.5wlím. El eje X es tiempo

El esquema en la figura 6.8 presenta además una serie de bloques en cascada en la parte inferior; estos son los encargados de contar el número de instantes de tiempo, dt. Al comienzo del programa se inicia el reloj, y cada vez que el reloj llega a dt aparece un impulso, Idt, el cual es acumulado por un integrador en la variable Cdt. Cuando un impulso Iep reinicia el acumulador, entonces el retenedor almacena el último valor en Cdt. Así se ha contado el número de impulsos dt en la variable Ndt.

La velocidad que entrega el algoritmo es $\omega_m = (Nep / Ndt)\,\omega_{\text{lim}}$. Por la naturaleza del conteo, Nep puede ser cualquier número entero, mientras que Ndt es un número natural. Si la velocidad de giro es superior o igual a ω_{lim} [ω_{lim} es la velocidad que corresponde a cuando se cuenta un pulso por dt, siendo dt el tiempo fijo del algoritmo a tiempo fijo], entonces hay más impulsos Iep que Idt, y Ndt es 1. A su vez, si la velocidad de giro es inferior a ω_{lim}, y por tanto hay más pulsos Idt que Iep, Nep es 1. Lo

descrito hasta el momento es similar a los algoritmos a espacio o a tiempo fijo, la diferencia fundamental está en que los impulsos Idt e Iep están sincronizados.

La figura 6.9 presenta un ejemplo en el cual la velocidad que se está midiendo es 1.5 ω_{lim} . Mientras que el primer pulso Iep puede aparecer en cualquier momento, el reloj es iniciado con el programa, como indica el evento (a), según se indica en el eje x de la figura 6.9. El conteo de tiempo en el reloj se reinicia para que ese conteo coincida con la aparición de un impulso Iep, de esta manera la misma cantidad de pulsos Iep será contada en el mismo tiempo, y se elimina la influencia de la incertidumbre por la aparición aleatoria del primer pulso Iep. Si la velocidad es mayor que la velocidad límite, de igual manera se reinicia el conteo de Idt, y el resultado es otra vez que el conteo es independiente de la aparición aleatoria del primer pulso Iep.

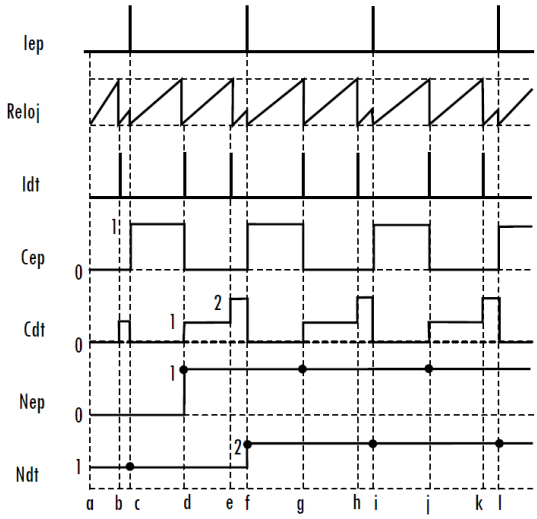

Figura 6.10: Señales cuando w=0.25wlím. El eje X es tiempo

En la figura 6.10 se presenta otro ejemplo con w=0.25 ω_{lim} . Como la velocidad es lenta, comparada con ω_{lim} , aparecen más impulsos Idt que Iep. Se supondrá, como ejemplo, que el primer impulso Iep aparece después de que el reloj ya ha contado un impulso Idt.

6.4.2.2 Intento de implementación del algoritmo combinado

Dadas las buenas características anunciadas para el algoritmo combinado, se decidió implementar el código para este algoritmo. Como se ha podido leer en el apartado anterior, en este algoritmo se utiliza una señal de reloj que tiene que ser precisa y dispararse siempre a intervalos iguales. Para ello no podemos utilizar condicionales que se ejecuten durante el programa, pues esto no es preciso. Es necesario hacer uso de las interrupciones internas del microcontrolador.

Las interrupciones internas únicamente son funciones de interrupción disparadas por *Timers*, que no es otra cosa que contadores o cronómetros. Es decir, estas interrupciones se disparan cada cierto tiempo, que es configurable, no se disparan por otro tipo de evento. En el microcontrolador *Atmel Atmega 2560,* que es el que tiene *Arduino Mega 2560*, existen 6 timers:

- ***Timer* 0:** Temporizador de 8 bits (registrará como máximo 256 valores). Es usado en las funciones *delay()* y *millis()*, por lo que se debe tener en cuenta a la hora de programar.

- ***Timer* 1:** Temporizador de 16 bits (registrará como máximo 1024 valores). Es usado en la librería *servo,* por lo que se debe tener en cuenta a la hora de programar.

- ***Timer* 2:** Temporizador de 8 bits. Es muy similar al Timer 0 y es usado por la función *tono()*.

- ***Timer* 3, 4 y 5:** Estos tres Timers son de 16 bits y funcionan de manera muy similar al Timer 1.

Se decidió utilizar el *Timer 1*, puesto que la librería *servo* no se utiliza en este proyecto, además de que se debía tener en cuenta que los 2 y 4 estaban utilizados para configurar la señal PWM para los motores (se explicará más adelante). Toda esta información se consultó en el *datasheet* del microcontrolador, que se incluye como bibliografía.

Debido a que los *timers* dependen de una fuente de reloj, la unidad más pequeña medible es el periodo que ofrece el reloj del microcontrolador:

$$T = \frac{1}{f} = \frac{1}{16MHz} = 62.5ns$$

Como el tiempo fijo de salto de la interrupción que queremos es mayor que este (buscamos que la interrupción salte cada 10ms), no podemos utilizar el Timer 1 tal cual está, sino que se necesita configurar. Para ello se hace uso del *preescaler* y del CTC.

El *preescaler* es algo similar a un divisor de la frecuencia de reloj. Se comprueba que si se utiliza el *preescaler* 8, se tiene un periodo:

$$T = \frac{1}{f\ /\ preescaler} = \frac{1}{16MHz\ /\ 8} = 5 \cdot 10^{-7} s$$

Y como se conoce que este *timer* es de 16 bit, se puede calcular:

$$\left(2^{16} - 1\right) \cdot 5 \cdot 10^{-7} = 0.0327675s \simeq 33ms$$

Se tiene que con el uso del *preescaler* se provoca que el *timer* finalice su ciclo cada 33ms. Para conseguir que se finalice el ciclo cada 10ms se hace uso del CTC:

$$valor\ CTC = \frac{tiempo\ deseado}{resolución\ timer} - 1 = \frac{10ms}{5 \cdot 10^{-4} ms} - 1 = 19999$$

Como marca el *datasheet*, este valor habrá que almacenarlo en el registro OCR1A. De esta manera se ejecutará la función de interrupción cada 10ms. En la figura 6.11 se incluye el código desarrollado para configurar el *Timer 1* (dentro de la función *setup*), utilizando los valores antes calculados. No se comentará en más detalle puesto que el código en sí está bastante comentado.

Configuración Timer 1 para interrupciones internas

```
1   ////// CONFIGURACION DEL TIMER PARA INTERRUPCIÓN INTERNA //////////////
2     cli();      // Desactiva las interrupciones temporalmente para configurarlas
3     TCCR1A=0;   // Uso del Timer 1, puesto que los timer 2 y 4 están usados.
4     TCCR1B=0;
5     OCR1A=19999;  // Este es el valor CTC para que la interrupción se lance cada 10ms
6     TCCR1B |= (1<<WGM12);
7     TCCR1B |= (1<<CS11);   // Esta es la configuración para utilizar el prescaler 8.
8     TIMSK1=(1<<OCIE1A);
9     sei();      // Reactiva las interrupciones
10
11  ////////////////////
12
```

Figura 6.11: Configuración del Timer 1 para la interrupción interna

Además la función de interrupción se tiene que definir en lo que anteriormente se ha llamado bloque 4, fuera de las funciones vacías *loop* y *setup*. Esta función de interrupción se incluye en la figura 6.12. Puede observarse que en la línea 4 se pone a cero el contador Cep y se actualiza el valor Np cuando corresponde como se vio en la descripción del algoritmo.

Función de interrupción interna

```
1  ///// INTERRUPCIÓN INTERNA PARA CONTADOR DE TIEMPOS PARA LA VELOCIDAD////
2  ISR(TIMER1_COMPA_vect){
3    Cdt++;
4    if (Cep!=0){Np=Cep;}
5    Cep=0;
6  }
7
```

Figura 6.12: Función de interrupción interna

Por otra parte también se modificaron las funciones de interrupción externas, para adaptarlas a las actualizaciones del nuevo algoritmo combinado. En la figura 6.13 se incluyen las correspondientes al motor izquierdo, con el que se hicieron las pruebas.

Funciones interrupciones externas modificadas

```
1  //INTERRUPCIONES ENCODERS MOTOR IZQUIERDO
2  void doEncoderA0(){
3    if (digitalRead(encoder0PinA) == HIGH)
4      if (digitalRead(encoder0PinB) == LOW)
5        encoder0Pos++; Cep++; if (Cdt!=0){Nt=Cdt;} Cdt=0;}      // Horario
6      else {encoder0Pos--; Cep--;Cep=-abs(Cep);Nt=Cdt; Cdt=0;} // Antihorario
7    }
8
9    else{
10     if (digitalRead(encoder0PinB) == HIGH) {
11       encoder0Pos++; Cep++; if (Cdt!=0){Nt=Cdt;} Cdt=0;}      // Horario
12     else {encoder0Pos--; Cep--;Cep=-abs(Cep);Nt=Cdt; Cdt=0;} // Antihorario
13   }
14  }
15
16 void doEncoderB0(){
17   if (digitalRead(encoder0PinB) == HIGH) {
18     if (digitalRead(encoder0PinA) == HIGH) {
19       encoder0Pos++; Cep++; if (Cdt!=0){Nt=Cdt;} Cdt=0;}      // Horario
20     else {encoder0Pos--;Cep--;Cep=-abs(Cep);Nt=Cdt; Cdt=0;}  // Antihorario
21   }
22
23   else {
24     if (digitalRead(encoder0PinA) == LOW) {
25       encoder0Pos++; Cep++; if (Cdt!=0){Nt=Cdt;} Cdt=0;}      // Horario
26     else {encoder0Pos--; Cep--;Cep=-abs(Cep);Nt=Cdt; Cdt=0;} // Antihorario
27   }
28  }
29
```

Figura 6.13: Modificación de funciones de interrupción externas

Al haberse configurado la interrupción interna cada 10ms, el cálculo de la velocidad puede realizarse las veces que se quiera, a la frecuencia que se desee, sin necesidad de que el cálculo en sí se vea alterado por esto. De esta manera, se decidió incluir una línea de código en el programa para el cálculo de la velocidad, sin condicionales de tiempo, de forma que se ejecuta una vez por cada ciclo del bucle del programa. La línea del cálculo de la velocidad puede verse en la figura 6.14, donde el factor que multiplica es simplemente la velocidad angular límite de la que se habla en la descripción del algoritmo combinado, y es aquella en la que se produce un pulso de *encoder* cada dt, es decir, un grado cada 10ms.

$$\omega_{lím} = \frac{1^o}{0.01s} = 100^o / s$$

Y pasándolo a revoluciones por minuto (que es la unidad que se ha usado en ese programa):

$$\omega_{lim} = 100\frac{^o}{s}\cdot\frac{60s}{1\min}\cdot\frac{1vuelta}{360^o} = \frac{600}{36}rpm$$

Cálculo de la velocidad con el algoritmo combinado

```
1   //// NUEVA VELOCIDAD /////
2
3   w=(Np/Nt)*(600/36);
4
```

Figura 6.14: Cálculo de la velocidad con el algoritmo combinado

Tras implementar todo el código detallado en este apartado correspondiente al algoritmo combinado, cuando se realizaron diferentes pruebas no se encontraron diferencias en la precisión y tampoco en la eliminación del error de truncamiento. Además este algoritmo añadía problemas que no se tenían con el anterior.

Uno de ellos consistía en que cuando la velocidad real era nula (el vehículo estaba detenido) el algoritmo nunca podía devolver velocidad nula, pues en ese momento estaba ejecutando la estrategia a espacio fijo, de manera que esperaba al siguiente pulso para calcular la nueva velocidad. Este problema se podía haber resuelto añadiendo condicionales que indicaran que cuando el tiempo entre pulsos era demasiado bajo, la velocidad se considerase nula. Otro problema que aparecía es que no se supo cómo detectar el cambio de sentido de giro de las ruedas con este algoritmo.

Todos estos problemas, sumados a que no se reducía el ruido de la medida de la velocidad, hicieron que se desestimase este algoritmo, y se decidió volver a utilizar el algoritmo a tiempo fijo utilizando filtros. Con esto no se pretende desacreditar ni menospreciar el algoritmo combinado desarrollado por estos investigadores, sino una autocrítica de que el código desarrollado para este algoritmo por el autor de este proyecto no era adecuado. De esta manera queda como trabajo futuro intentar desarrollar un código que represente fielmente ese algoritmo, pues se debía estar cometiendo algún error en la programación del algoritmo.

6.4.3 Filtrado de la Velocidad

Dado lo sucedido con el anterior algoritmo, se decidió seguir el empleando el algoritmo a tiempo fijo, pero intentando implementar un filtro de paso bajo para reducir ruidos de alta frecuencia. Los filtros que se han ensayado son de primer orden, de media móvil y filtros Butterworth. No se pretende aquí profundizar en la teoría de filtros, simplemente comentar los que se han probado, sus resultados y la elección final.

6.4.3.1 Filtros de Primer Orden

Todos los filtros utilizados han sido de carácter digital, esto es, se utilizan de forma discreta que es como se pueden tratar los datos. En este caso, la expresión matemática discreta del filtro es muy sencilla:

$$y_k = y_{k-1} + \alpha\left(x_k - y_{k-1}\right) \;\; donde\; k = 0,1,2,3...$$

En la expresión anterior “y” es la señal filtrada, mientras que “x” es la señal sin filtrar; y “α” es el factor de filtrado con valores posibles de 0 a 1. La expresión está escrita de forma que cuanto menor sea “α”

mayor es el filtrado y viceversa. Se observa cómo este algoritmo sólo emplea un valor anterior, por eso se denomina de primer orden.

A continuación se incluyen algunas gráficas con ensayos utilizando este tipo de filtros. El ensayo realizado es siempre el mismo: consiste en hacer funcionar un motor a diferentes valores PWM, concretamente se inicia con valor PWM=20, a los 3 segundos se pasa súbitamente a PWM=50, a los 6 segundos se pasa súbitamente a PWM=100 y a los 9 segundos se pasa a PWM=255. El ensayo se ha realizado así para abarcar el mayor rango posible de velocidades de giro del motor, y además se incluyen escalones en la tensión para observar el comportamiento de los filtros frente a éstos.

En la figura 6.15, a modo introductorio se incluyen las velocidades obtenidas durante este ensayo mediante el algoritmo a tiempo fijo, usando incrementos de tiempo de 100 ms y 10 ms. El cálculo usando 100 ms se usará como referencia de la velocidad más precisa y fiel a la realidad a la que se aspira llegar, pues ya se explicó que a mayores incrementos de tiempo, mayor precisión. Lamentablemente, como también se explico anteriormente, no se pueden utilizar estos incrementos de tiempo para el control del vehículo porque es demasiado lento.

Aquí se puede observar cómo al reducir el incremento de tiempo a 10ms el ruido es mucho mayor y de alta frecuencia y también se puede observar que este ruido tiene su origen en el error de truncamiento antes comentado, pues la forma de la onda es como cuadrada y siempre oscila entre valores similares. En la figura 6.15.1 se incluye esta misma gráfica pero más ampliada para observar este fenómeno.

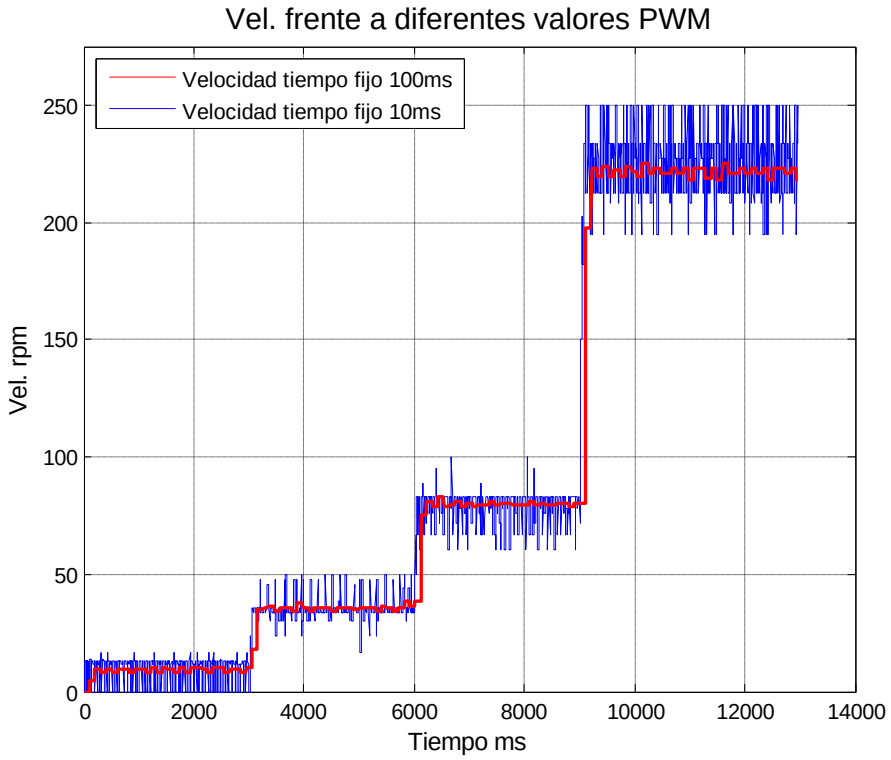

Figura 6.15: Velocidad calculada con diferentes incrementos de tiempo

En las figuras 6.16 y 6.17 se han incluido los efectos de un filtro de primer orden con factor de filtrado 0.09 y 0.05, respectivamente sobre la velocidad calculada cada 10ms; comparándolos con la velocidad de mejor precisión a la que aspirar calculada cada 100ms que se incluyó en la figura 6.15. Se observa cómo los filtros reducen mucho el ruido que podía verse en la primera gráfica. Aún así todavía queda un ruido de alta frecuencia y baja amplitud en la señal. Estos factores de filtrado no se han escogido aleatoriamente, sino que ha sido por experimentación. Si fueran mayores, el filtrado sería muy pequeño, mientras que si son menores, aunque el filtrado es mayor, el retraso de la señal es muy alto y se pierde información fundamental. Incluso entre la figura 6.16 y la 6.17 se puede observar un retraso mayor en la última que en la primera. Este retraso no se puede permitir que sea mayor de unos pocos

milisegundos, pues en otro caso se volvería a tener una respuesta lenta y un control inadecuado del vehículo.

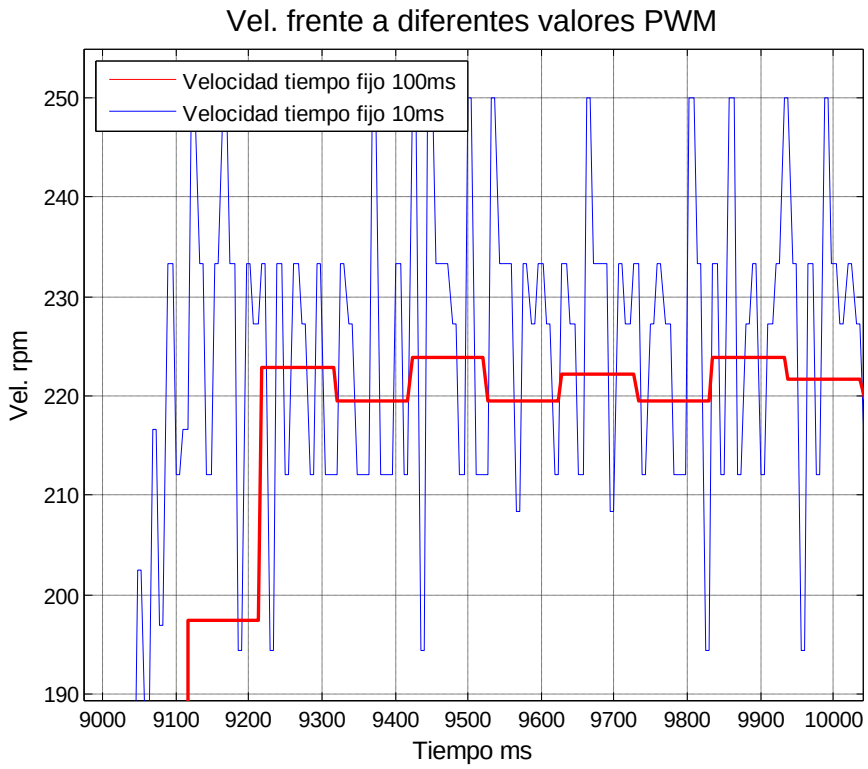

Figura 6.15.1: Detalle oscilaciones de velocidad con diferentes incrementos de tiempo

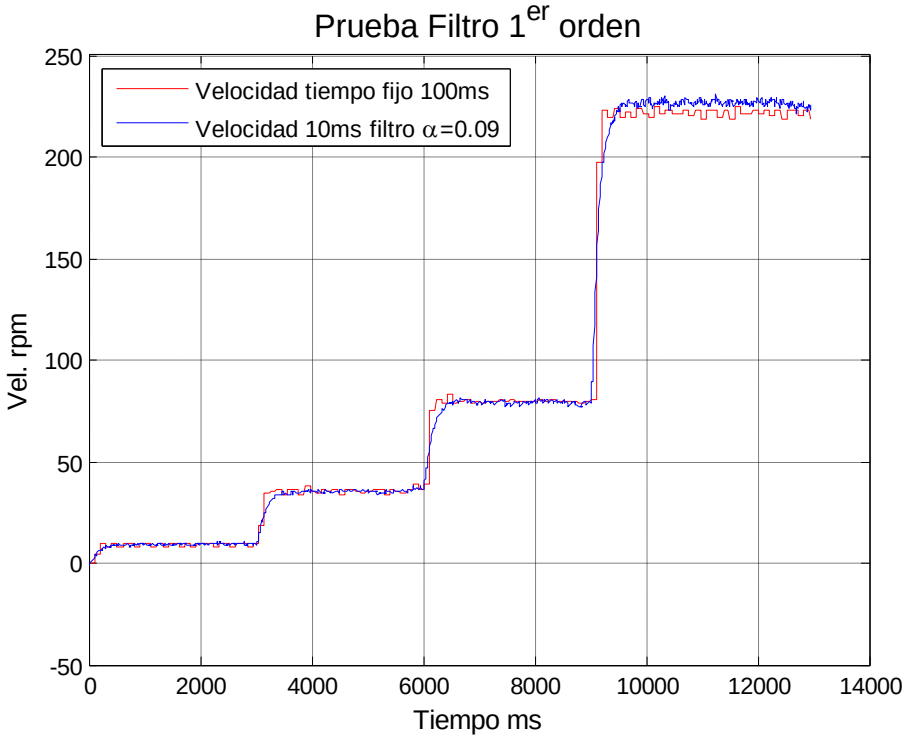

Figura 6.16: Filtro de primer orden con α=0.09

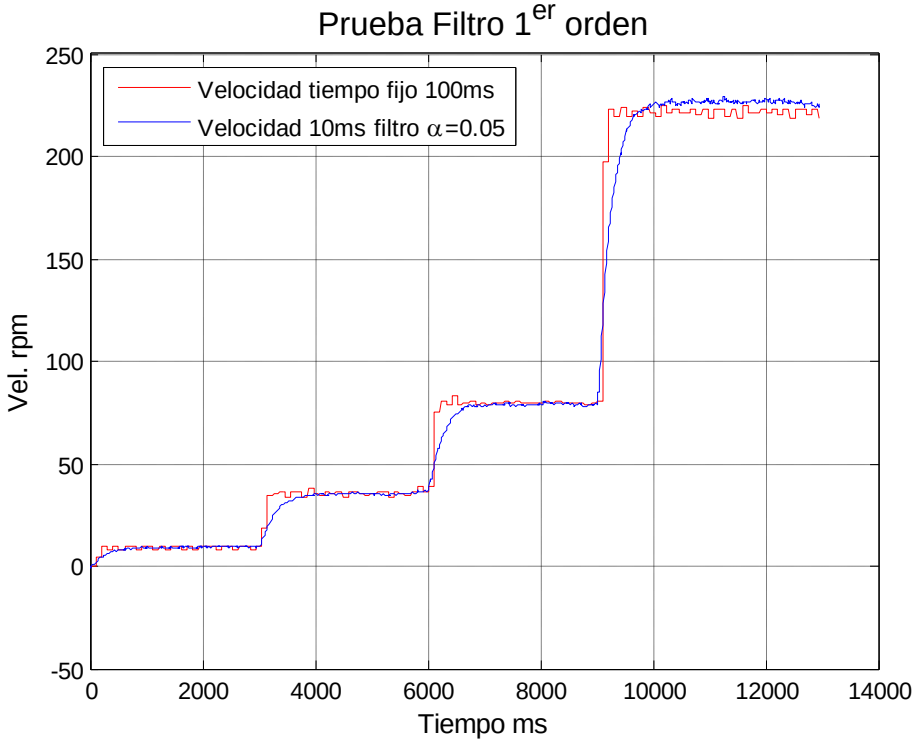

Figura 6.17: Filtro de primer orden con α=0.05

Para mejorar estos pequeños inconvenientes de los filtros de primer orden, se siguió probando con otros filtros.

6.4.3.2 Filtros de Media Móvil

Los filtros de media móvil, como su propio nombre indica, funcionan calculando una media móvil de diferentes valores de la sucesión de valores de la señal original. En este caso siempre se han utilizado de tipo media centrada, de manera que se usan el mismo número de valores de la señal que se pretende filtrar por delante y por detrás del valor que se está calculando. Su fórmula general discreta puede resumirse en la siguiente expresión:

$$y_k = \frac{x_{k-n} + \ldots + x_{k-2} + x_{k-1} + x_k + x_{k+1} + x_{k+2} + \ldots + x_{k+n}}{2 \cdot n + 1} \quad donde\ k = n, n+1, n+2, \ldots$$

Se pueden usar tantos valores como se desee, incrementándose o reduciéndose así el orden del filtro utilizado. En la figura 6.18 se incluye el resultado de un filtro de media móvil de 5 puntos para el mismo ensayo presentado en la sección anterior. También se incluye la velocidad calculada con un tiempo fijo de 100ms. Se puede comprobar cómo con este filtro se eliminan prácticamente todas las altas frecuencias. El principal problema que tiene es que retrasa mucho la medida, en concreto la retrasa unos 300ms, lo que supone demasiado.

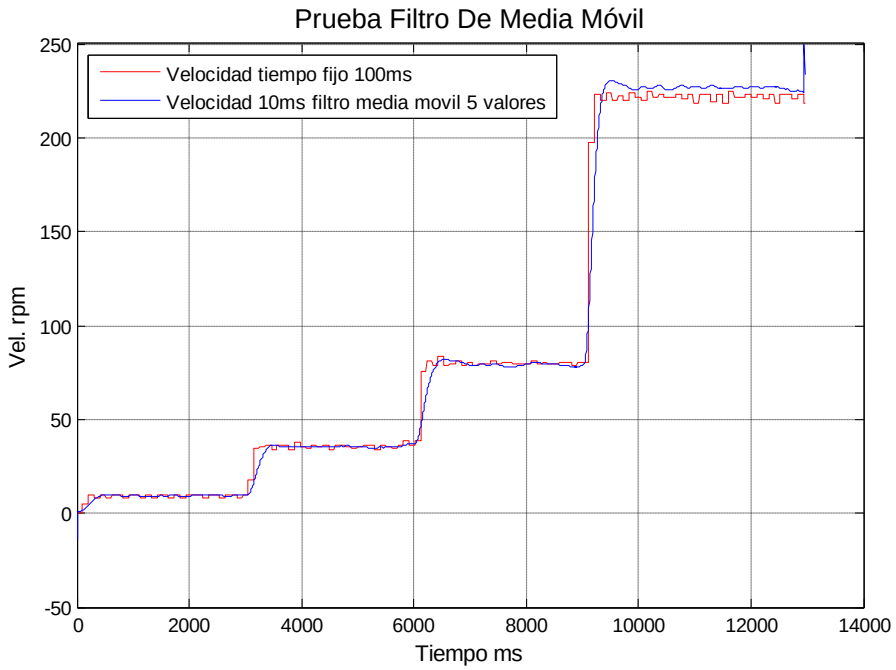

Figura 6.18: Filtro de media móvil de 5 puntos

6.4.3.3 Filtros Butterworth

El filtro de Butterworth es un filtro de tipo IIR (*Infinite Impulse Response*). La respuesta en frecuencia del filtro es extremadamente plana (con mínimas ondulaciones) en la banda pasante, es decir, mantiene la información de frecuencias bajas casi intacta, hasta que llega a la frecuencia de corte.

Este filtro puede implementarse de diferentes órdenes. En este caso se ha utilizado únicamente un filtro Butterworth de orden 2. Para este orden, la función de transferencia discreta del filtro es del tipo:

$$H(z) = \frac{B_1 + B_2 z^{-1} + B_3 z^{-2}}{A_1 + A_2 z^{-1} + A_3 z^{-2}} = \frac{Y(z)}{X(z)}$$

Y por tanto su ecuación en diferencias puede escribirse como:

$$y_k = \frac{-A_2 y_{k-1} - A_3 y_{k-2} + B_1 x_k + B_2 x_{k-1} + B_3 x_{k-2}}{A_1}$$

Para calcular los coeficientes del filtro se ha utilizado la función de Matlab denominada *butter*, introduciendo como argumentos el orden deseado y la frecuencia de corte normalizada. De esta forma, la función devuelve como salida dos vectores, el primero con los coeficientes correspondientes al numerador de la función de transferencia y el segundo con los correspondientes al denominador, de la forma:

$$numerador \quad B = [B_1 \ B_2 \ B_3]$$
$$denominador \ A = [A_1 \ A_2 \ A_3]$$

Las gráficas que se incluyen en las figuras 6.19 y 6.20 corresponde al mismo ensayo que en los apartados anteriores, pero el filtro utilizado es un Butterworth de orden 2 con frecuencias de corte Wn=0.0354 y Wn=0.0784.

En el primer caso se puede observar que se corrigen muchas de las altas frecuencias que no se conseguían con los otros filtros, consiguiéndose una señal suave y ajustada a la referencia de precisión que se tiene (cálculo con 100ms). El problema de esta gráfica es que aunque el cálculo de la velocidad se retrasa menos que en los casos de otros filtros con respecto a la señal original de 10ms, sigue aún retrasándose bastante, en torno a 250ms.

Por ello se decidió utilizar una frecuencia de corte normalizada mayor. De esta manera, aunque se eliminan únicamente frecuencias más altas que antes y por tanto permanece menos ruido, se consigue que la señal se retrase un poco menos. La frecuencia normalizada que se decidió utilizar, tras realizar varios experimentos es Wn=0.0784.

Este caso es el que puede observarse en la figura 6.21. Se puede ver que ahora existe un poco de más ruido en la señal filtrada, pero se ha conseguido lo que se buscaba, reducir el retraso. Se debe llegar así a una solución de compromiso entre ruido y retraso de la señal. Se puede comprobar que ahora el retraso es tan sólo de alrededor de 100 ms.

Este último es el filtro implementado en el código final para el control del vehículo.

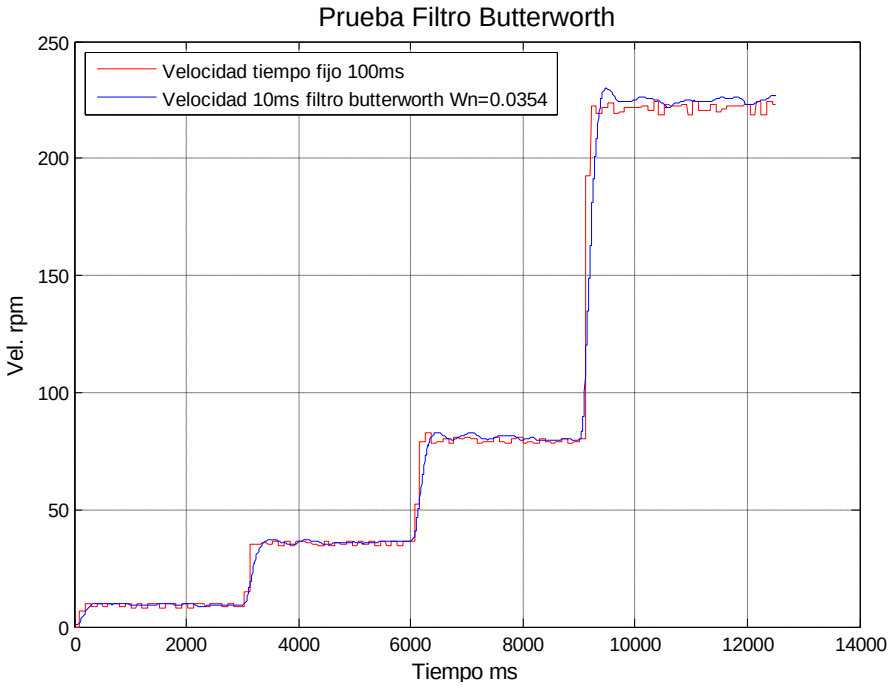

Figura 6.19: Velocidad filtrada con Butterworth Wn=0.0354

Para implementar este filtro en el código, sencillamente se realiza dentro del condicional para calcular la velocidad a tiempo fijo de 10 ms, utilizando los coeficientes obtenidos con *Matlab* como se explicó anteriormente. Esto puede verse en el código de la Figura 6.20, donde se incluye el código íntegro utilizado en una de las versiones finales para el control del robot, por lo que puede apreciarse también que el cálculo de la aceleración se realiza en el mismo condicional. Las líneas correspondientes al filtro Butterworth van de la 11 a la 16. Hay que tener en cuenta la actualización de los términos anteriores, que tiene lugar en las líneas 13 a 16.

Filtro Velocidad y Cálculo Aceleración

```
1   ///////// CÓDIGO PARA POSICIÓN, VELOCIDAD Y ACELERACIÓN ////////
2
3     if ((millis()-time1)>=10){
4         veli=((encoder0Pos-pos0)*1000.0/(millis()-time1))*(PI/180.0);
5         pos0=encoder0Pos;
6         veld=((encoder1Pos-pos0d)*1000.0/(millis()-time1))*(PI/180.0);
7         pos0d=encoder1Pos;
8         vel=(veli+veld)/2;
```

```
9
10        //Filtrado de la velocidad(BUTTERWORTH CON N=2; Wn=0.0784):
11        wf=1.6544*wf_1 - 0.7059*wf_2 + 0.0129*vel + 0.0257*vel_1 + 0.0129*vel_2;
12
13          wf_2=wf_1;
14          wf_1=wf;
15          vel_2=vel_1;
16          vel_1=vel;
17
18        acelf=(wf-wf0)*1000.0/(millis()-time1);
19
20        wf0=wf;
21        time1=millis();
22
23      }
24
```

Figura 6.20: Filtro de la velocidad y cálculo de la aceleración

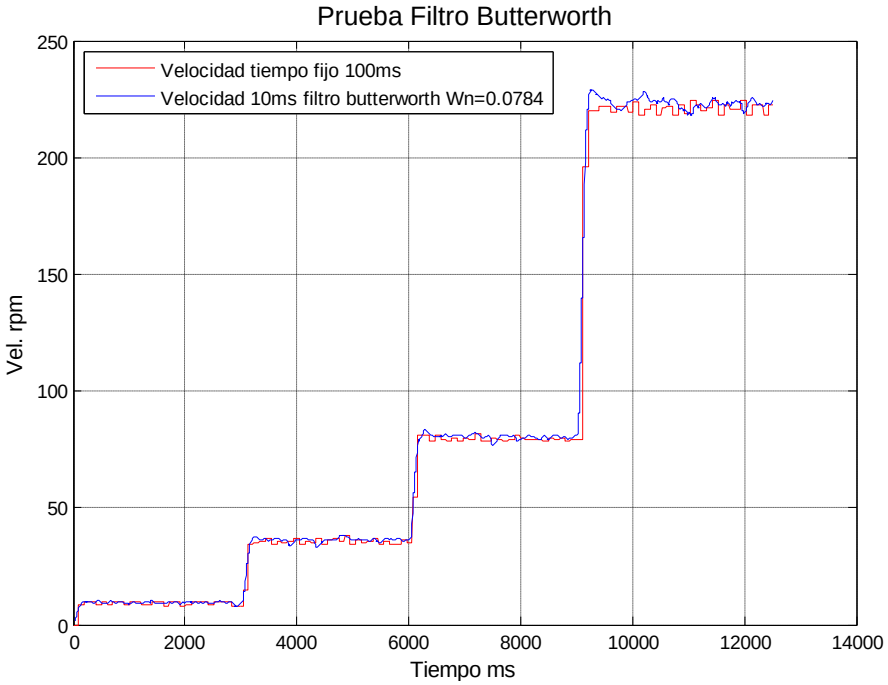

Figura 6.21: Velocidad filtrada con Butterworth Wn= 0.0784

6.4.4 Cálculo de la Aceleración

Para calcular la aceleración el algoritmo que se sigue es análogo al que se decidió utilizar para la velocidad, tras la argumentación en apartados anteriores. Esto es, se trata de un algoritmo a tiempo fijo.

En concreto, para calcular la aceleración en la práctica, se realiza dentro del mismo condicional usado para la velocidad, de manera que la aceleración también se calcula cada 10 ms. La expresión que resume el cálculo de la aceleración es:

$$aceleración = \frac{velocidad\ filtrada\ actual - velocidad\ filtrada\ anterior}{tiempo\ transcurrido}$$

El cálculo de la aceleración puede verse en el código de la figura 6.21 en la línea 18. También debe tenerse en cuenta la actualización de lo que se denomina velocidad filtrada anterior en la anterior expresión, que se realiza en la línea 20.

6.5 Obtención del Ángulo de Inclinación

En este apartado se trata la obtención del ángulo de inclinación del robot, así como de la velocidad angular del mismo, mediante la adquisición de datos de la IMU *MPU6050*, cuyas características se describieron en el apartado 4.5.

6.5.1 Trigonometría

La obtención del ángulo de inclinación del vehículo se hace con ayuda de la aceleración de la gravedad. La gravedad provoca que los acelerómetros de la IMU den un valor distinto de cero aunque ésta esté en reposo. De esta manera, cuando el eje z de la IMU está alineado con el eje "Z" del sistema de coordenadas fijo (ligado al suelo), el valor que proporciona el acelerómetro del eje "z" de la IMU es -1 (en realidad por convención está cambiado de signo, +1), y los otros dos acelerómetros proporcionan valor cero.

Si en esta misma posición se mantiene la IMU con su eje "y" fijo y ésta se gira alrededor del eje "y" un ángulo ϕ, el acelerómetro del eje "y" de la IMU seguirá indicando un valor 0, pero el del eje "x" ya no dará valor cero, y el del eje "z" tampoco dará un valor 1. Esta situación puede verse en la figura 6.22. Ahora el acelerómetro del eje "x" marcará $Acc_x = \sin\phi$ (en color rojo), y el del eje "z" marcará $Acc_z = \cos\phi$ (en color azul).

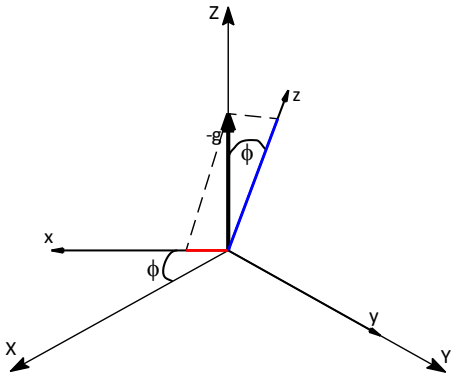

Figura 6.22: Proyección del vector gravedad

Aunque para este proyecto sólo se va a utilizar un eje de giro, que determinará el ángulo de inclinación del vehículo, como ya se ha indicado múltiples veces la IMU utilizada dispone de 3 acelerómetros. En teoría podría obtenerse el ángulo de inclinación utilizando sólo dos de los acelerómetros, pues el vector gravedad siempre se encontraría contenido en uno en uno de los planos coordenados. Pero en la práctica esto no es tan sencillo porque nunca se consigue que el vector gravedad se encuentre en un único plano coordenado del sistema de referencia de la IMU.

En la realidad, aunque la IMU se calibre muy bien, siempre existirán pequeñas perturbaciones, tales como irregularidades del terreno, imperfecciones de las ruedas, o falta de alineación de los ejes de los motores, que harán que desde el sistema de referencia de la IMU el vector gravedad se vea como un vector en el espacio tridimensional.

Es por este motivo por el que para obtener el ángulo de inclinación del robot, que es el que interesa en este proyecto, hace falta trabajar con los ángulos de *Euler* en un espacio tridimensional (la realidad) y la conversión del vector gravedad entre los ejes de un sistema de referencia fijo (ligado a la Tierra) y los ejes de un sistema de referencia móvil (el vehículo, o más concretamente la IMU).

En general, considerando que no existen otras aceleraciones, la IMU proporciona el vector gravedad proyectado sobre los ejes de su sistema de referencia, que son unos ejes móviles. A este vector se le denominará según (1).

$$\vec{a} = \begin{pmatrix} g_{xm} \\ g_{ym} \\ g_{zm} \end{pmatrix} \tag{1}$$

Por otra parte, se conoce que el vector gravedad en los ejes fijos es (2). Se considerará que el módulo del vector gravedad es 1.

$$\vec{A} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{2}$$

La relación entre ambos vectores se lleva a cabo mediante una matriz de rotación o transformación R, según la expresión (3).

$$\vec{a} = R\,\vec{A} = R \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{3}$$

En este apartado, y para la obtención de los ángulos de inclinación se considerará la notación de ángulos de la figura 6.23.

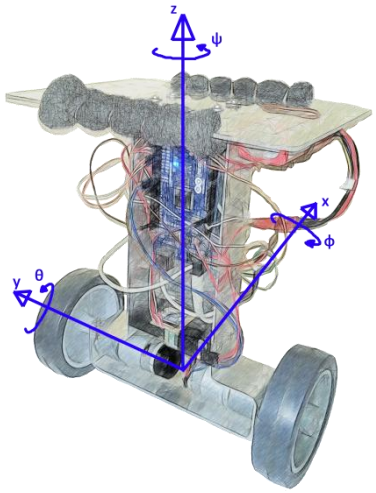

Figura 6.23: Convenio de ángulos de Euler

En la figura 6.24 se puede ver una posición arbitraria del sistema de referencia móvil (ejes xyz), que en este caso sería la IMU, con respecto al sistema de referencia fijo (ejes XYZ). Para llegar a esa posición, los ejes móviles, partiendo de la misma posición que los fijos, han efectuado 3 giros sucesivos marcados con los números 1, 2 y 3 en el dibujo.

Cada uno de estos giros se puede describir con una matriz de giro. La forma de obtener estas matrices no se detallará aquí para no extenderse demasiado, pero se ha de tener en cuenta que se hace uso de los ejes intermedios x'y'z' y x"y"z" para facilitar la labor.

- ①La matriz de giro alrededor del eje Z (giro de ángulo ψ) es:

$$T_z(\psi) = \begin{bmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4}$$

- ②La matriz de giro alrededor del eje y' (giro de ángulo θ) es:

$$T_y(\theta)=\begin{bmatrix}\cos\theta & 0 & -\sin\theta\\ 0 & 1 & 0\\ \sin\theta & 0 & \cos\theta\end{bmatrix} \tag{5}$$

- ③La matriz de giro alrededor del eje x (giro de ángulo ϕ) es:

$$T_x(\phi)=\begin{bmatrix}1 & 0 & 0\\ 0 & \cos\phi & \sin\phi\\ 0 & -\sin\phi & \cos\phi\end{bmatrix} \tag{6}$$

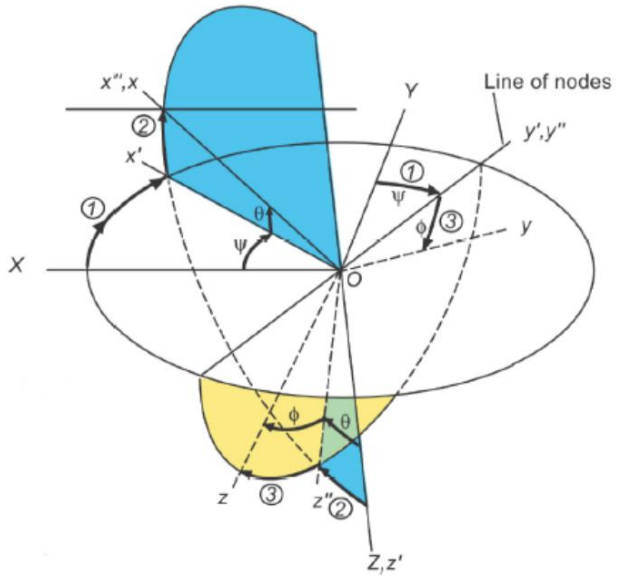

Figura 6.24: Relación entre los dos sistemas de referencia

Para obtener la expresión en los ejes móviles de un vector expresado en los ejes fijos se deben recorrer los giros en sentido inverso a como se ha descrito, es decir, multiplicar las matrices en orden inverso, de la forma que aparece en (7).

$$\begin{bmatrix}vector\\ en\ \text{ejes}\\ \text{móviles}\end{bmatrix}=T_x(\phi)T_y(\theta)T_z(\psi)\begin{bmatrix}vector\\ en\ \text{ejes}\\ \text{fijos}\end{bmatrix} \tag{7}$$

De esta forma se calcula la matriz general de rotación entre estos sistemas:

$$R=T_x(\phi)T_y(\theta)T_z(\psi)=\begin{bmatrix}\cos\theta\cos\psi & \cos\theta\sin\psi & -\sin\theta\\ \cos\psi\sin\theta\sin\phi-\cos\phi\sin\psi & \sin\psi\sin\theta\sin\phi+\cos\phi\cos\psi & \cos\theta\sin\phi\\ \cos\psi\sin\theta\cos\phi+\sin\phi\sin\psi & \sin\psi\sin\theta\cos\phi-\sin\phi\cos\psi & \cos\theta\cos\phi\end{bmatrix} \tag{8}$$

Por tanto, sustituyendo (8) en la expresión (3) se tiene:

$$\begin{bmatrix}g_{xm}\\ g_{ym}\\ g_{zm}\end{bmatrix}=\begin{bmatrix}\cos\theta\cos\psi & \cos\theta\sin\psi & -\sin\theta\\ \cos\psi\sin\theta\sin\phi-\cos\phi\sin\psi & \sin\psi\sin\theta\sin\phi+\cos\phi\cos\psi & \cos\theta\sin\phi\\ \cos\psi\sin\theta\cos\phi+\sin\phi\sin\psi & \sin\psi\sin\theta\cos\phi-\sin\phi\cos\psi & \cos\theta\cos\phi\end{bmatrix}\begin{bmatrix}0\\ 0\\ 1\end{bmatrix} \tag{9}$$

$$\begin{bmatrix} g_{xm} \\ g_{ym} \\ g_{zm} \end{bmatrix} = \begin{bmatrix} -\sin\theta \\ \cos\theta\sin\phi \\ \cos\theta\cos\phi \end{bmatrix} \tag{10}$$

Se observa que con tres acelerómetros sólo se pueden obtener los ángulos θ, ϕ. Para obtener ψ harían falta sensores magnéticos. Esto es fácilmente entendible puesto que el primer giro en la figura 6.24 se realiza alrededor del eje z cuando la aceleración de la gravedad está alineada con este eje, de forma que la IMU no percibe variación alguna.

Para obtener los ángulos θ, ϕ, se considera la expresión (10):

$$\tan\phi = \frac{g_{ym}}{g_{zm}} \tag{11}$$

$$\tan\theta = \frac{-g_{xm}}{\frac{g_{zm}}{\cos\phi}} \tag{12}$$

De (10) puede desarrollarse:

$$\left.\begin{aligned} \cos\theta &= \frac{g_{ym}}{\sin\phi} \\ \cos\theta &= \frac{g_{zm}}{\cos\phi} \end{aligned}\right\} \frac{g_{ym}}{\sin\phi} = \frac{g_{zm}}{\cos\phi};\ \left(g_{ym}\cos\phi\right)^2 = \left(g_{zm}\sin\phi\right)^2;\ \cos^2\phi = \left(\frac{g_{zm}}{g_{ym}}\right)^2 \sin^2\phi \tag{13}$$

Conociendo que $\sin^2\phi = 1 - \cos^2\phi$, e introduciéndolo en (13):

$$\cos^2\phi = \left(\frac{g_{zm}}{g_{ym}}\right)^2 - \left(\frac{g_{zm}}{g_{ym}}\right)^2 \cos^2\phi \tag{14}$$

$$\cos^2\phi = \frac{\left(\frac{g_{zm}}{g_{ym}}\right)^2}{1+\left(\frac{g_{zm}}{g_{ym}}\right)^2} = \frac{g_{zm}^2}{g_{ym}^2 + g_{zm}^2} \tag{15}$$

$$\cos\phi = \frac{g_{zm}}{\sqrt{g_{ym}^2 + g_{zm}^2}} \tag{16}$$

Introduciendo (16) en (12):

$$\tan\theta = \frac{-g_{xm}}{\frac{g_{zm}}{\frac{g_{zm}}{\sqrt{g_{ym}^2 + g_{zm}^2}}}} = \frac{-g_{xm}}{\sqrt{g_{ym}^2 + g_{zm}^2}} \tag{17}$$

Finalmente, de (11) y (17) se obtienen las expresiones de los ángulos:

$$\phi = \tan^{-1}\left(\frac{g_{ym}}{g_{zm}}\right) \tag{18}$$

$$\theta = \tan^{-1}\left(\frac{-g_{xm}}{\sqrt{g_{ym}^2 + g_{zm}^2}}\right) \tag{19}$$

6.5.2 Filtro de Kalman

Como se explicó en el apartado anterior, los ángulos de cabeceo y balanceo (θ y ϕ) se pueden obtener mediante el uso de la proyección del vector gravedad sobre el sistema de referencia móvil formado por los acelerómetros de la IMU. Anteriormente se dijo también que para realizar este cálculo se consideraba que el robot estaba en reposo, pues cualquier aceleración del sistema estaría falseando la medida.

Esto es precisamente lo que sucede en la realidad, el vehículo está en movimiento y por tanto existen continuas aceleraciones externas de menor o mayor medida que afectan a la medida de ángulos proporcionada por los acelerómetros. El resultado en la práctica es que se obtiene una señal muy ruidosa de los ángulos calculados, aunque el ruido no es estrictamente ruido aleatorio, sino que son aceleraciones reales que experimenta el sistema, pero carecen completamente de interés para esta aplicación. Por esta razón es por la que se ha intentado situar la IMU lo más cerca posible del eje de rotación del péndulo, para que se viera afectada lo mínimamente posible por las aceleraciones derivadas de la inclinación del vehículo.

Para reducir o filtrar este ruido se utiliza en este proyecto un filtro de Kalman. Esto consiste en considerar que el valor real de los ángulos nunca es conocido, de manera que sólo se conoce un valor observado o medido mediante los acelerómetros. Lo mismo sucede con las aceleraciones angulares que proporcionan los giróscopos. El efecto del filtro de Kalman es utilizar las medidas de los acelerómetros y los giróscopos para obtener una estimación del estado real (ángulos y velocidades angulares) del sistema que sea lo más cercana posible al estado real.

Visto de otro modo, ya se ha dicho que los acelerómetros proporcionan una medida fiel de los ángulos de posición, aunque con mucho ruido. De la misma manera los giróscopos proporcionan una medida de las velocidad angulares, que integrándose en el tiempo pueden proporcionar los ángulos. Aunque esta medida de los giróscopos tiene menos ruido, sufre un efecto de deriva y acumulación de error con el tiempo. De esta manera puede considerarse que los giróscopos proporcionan una medida precisa de los ángulos a corto plazo, mientras que los acelerómetros proporcionan una medida precisa de los ángulos a largo plazo. Así, la función del filtro de Kalman puede entenderse, de manera muy simplificada, como la utilización de la señal de los giróscopos para filtrar el ruido de alta frecuencia de la señal de los acelerómetros y obtener así una estimación de los ángulos más precisa y menos ruidosa.

El filtro de Kalman opera produciendo una estimación estadísticamente óptima del estado del sistema basada en las medidas de los sensores. Para hacer esto necesita conocer el ruido de entrada al filtro, pero también el ruido del sistema en sí mismo llamado ruido del proceso. El ruido tiene que tener una distribución gaussiana y tener media cero, pero afortunadamente la mayoría del ruido aleatorio tiene estas características.

6.5.2.1 El estado del sistema

En primer lugar se hace mención a la notación para designar los diferentes vectores de estado del sistema. Se debe tener en cuenta que en el razonamiento aquí seguido el filtro de Kalman sólo se aplica

a la obtención de un solo ángulo utilizando las medidas de los acelerómetros y del giróscopo correspondientes. Esto es aplicable de forma análoga al otro ángulo restante.

A continuación se escribe la notación para lo que se denomina la estimación del estado anterior. Consiste en la estimación del estadio anterior basado en el estado anterior y las estimaciones de estados anteriores a él:

$$\hat{x}_{k-1|k-1}$$

El siguiente es el estado estimado a priori. Consiste en la estimación del vector de estados en el tiempo actual k, basado en el estado anterior del sistema y las estimaciones del estado anteriores:

$$\hat{x}_{k|k-1}$$

El último es el estado estimado a posteriori. Se trata de la estimación del estado en el tiempo k dadas las observaciones hasta e incluyendo el tiempo k:

$$\hat{x}_{k|k}$$

El problema es que el estado del sistema está oculto y sólo puede ser observado mediante la observación del estado z_k que es la medida que proporcionan los sensores. Esto significa que el estado actual proporcionado estará basado en el estado en el tiempo k y en todos los estados anteriores. Y también significa que no se puede confiar en la estimación del estado antes de que el filtro de Kalman se haya estabilizado.

El gorro sobre la $\hat{x}$ significa que se trata de una estimación del estado, mientras que si se escribe únicamente x se trata del estado real, el que se está tratando de estimar. De esta manera, el estado del sistema en el tiempo k es:

$$x_k$$

El estado del sistema en el tiempo k está dado por (1):

$$x_k = Fx_{k-1} + Bu_k + w_k \tag{1}$$

Donde x_k es el vector de estado que consiste en (2):

$$x_k = \begin{bmatrix} \theta \\ \dot{\theta}_b \end{bmatrix}_k \tag{2}$$

Como puede observarse, la salida del filtro será el ángulo θ, pero también la desviación $\dot{\theta}_b$ basada en las mediciones de los acelerómetros y el giróscopo. La desviación es el valor que corresponde a la deriva del giróscopo. Esto significa que se puede obtener la velocidad angula real restando la desviación de la medida del giróscopo.

La matriz F es el modelo de transición del estado, que se aplica al estado anterior x_{k-1}. En este caso F se define como (3):

$$F = \begin{bmatrix} 1 & -\Delta t \\ 0 & 1 \end{bmatrix} \tag{3}$$

La entrada de control u_k es la medida del giróscopo en grados por segundo en el tiempo k, lo que también se conoce como la velocidad angular $\dot{\theta}$. De hecho se reescribirá la ecuación de estado de la forma (4):

$$x_k = Fx_{k-1} + B\dot{\theta}_k + w_k \tag{4}$$

La matriz B es el modelo de la entrada de control, y se define como (5). Esto tiene sentido, puesto que el ángulo θ se puede conseguir multiplicando la velocidad angular $\dot{\theta}$ por el incremento de tiempo Δt ; y puesto que no se puede calcular la desviación directamente de la velocidad angular, la segunda componente es cero.

$$B = \begin{bmatrix} \Delta t \\ 0 \end{bmatrix} \tag{5}$$

El último término w_k es el ruido del proceso que consiste en una distribución gaussiana con media cero y con covarianza Q en el tiempo k:

$$w_k \sim N(0, Q_k)$$

La matriz Q_k es la matriz de covarianza del ruido del proceso y en este caso es la matriz de covarianza de la estimación del estado del acelerómetro y la desviación. En este caso consideraremos independientes la estimación de la desviación y el acelerómetro, así que en realidad es igual a la varianza de la estimación del acelerómetro y la desviación. La matriz final se define como en (6). Se puede ver que la matriz de covarianza Q_k depende del tiempo actual, así que la varianza del acelerómetro Q_θ y la varianza de la desviación (deriva) se multiplican por el incremento de tiempo Δt . Esto tiene sentido puesto que el ruido del proceso será cada vez mayor con el tiempo desde la última actualización del estado, pudiendo haberse producido, por ejemplo, una deriva en la medida del giróscopo.

$$Q_k = \begin{bmatrix} Q_\theta & 0 \\ 0 & Q_{\dot{\theta}_b} \end{bmatrix} \Delta t \tag{6}$$

Estas constantes se deberán conocer para que el filtro de Kalman funcione. Hay que tener en cuenta también que mientras mayor sea el valor, mayor será el ruido en la estimación del estado. De esta forma, si por ejemplo el ángulo estimado comienza a mostrar una variación por deriva, se deberá incrementar el valor de $Q_{\dot{\theta}_b}$. Por el contrario, si la estimación tiende a ser demasiado lenta, indica que se está confiando demasiado en la estimación del ángulo y se debería reducir el valor de $Q_{\dot{\theta}_b}$ para hacerlo más ágil.

6.5.2.2 El estado observado

Ahora se prestará atención a la observación del estado z_k , esto es, la medida del estado que proporcionan los sensores. Ésta es dada por (7). Se puede observar que el estado observado está compuesto por el estado actual multiplicado por la matriz H más el ruido de medida v_k .

$$z_k = Hx_k + v_k \tag{7}$$

La matriz H se denomina modelo de observación y es utilizada para mapear el estado real en el estado observado. Ya se ha comentado que el estado real no puede ser conocido. Puesto que la medida es únicamente la medida de los acelerómetros, la expresión de H es (8).

$$H = \begin{bmatrix} 1 & 0 \end{bmatrix} \tag{8}$$

El ruido de la medida tiene que ser también una distribución gaussiana con media cero y covarianza R:

$$v_k \sim N(0, R)$$

Como R no es una matriz, el ruido de la medida es igual a la varianza de la medida, puesto que la covarianza de la misma variable es igual a la varianza. De esta manera se puede definir R como (9).

$$R = E\left[v_k \quad v_k{}^T\right] = \text{var}(v_k) = \text{var}(v) \tag{9}$$

Además se asumirá que el ruido de la medida no depende del instante de tiempo. Se debe tener en cuenta que si se selecciona la varianza del ruido de la medida demasiado alta, el filtro responderá muy lento puesto que está confiando menos en las nuevas medidas, pero si es demasiado baja, el valor podría sobreoscilar y ser ruidoso puesto que se confía demasiado en la medida de los acelerómetros.

En resumen, se necesita encontrar las varianzas del ruido del proceso Q_θ y $Q_{\dot{\theta}_b}$ y la varianza del ruido de la medida var(v). Hay múltiples formas de hacer esto, aunque en la práctica suele hacerse por experimentación.

6.5.2.3 Las ecuaciones del filtro de Kalman

A continuación se expondrán las ecuaciones del algoritmo que nos permitirán estimar el estado del sistema en el tiempo k. Este algoritmo se divide en dos etapas: Predicción y Actualización.

Predicción

En las dos primeras ecuaciones, ecuaciones (10) y (11), se intenta predecir el estado actual y la matriz de covarianza del error en el tiempo k. En primer lugar el filtro intentará estimar el estado actual basándose en todos los estados anteriores y la medida del giróscopo:

$$\hat{x}_{k|k-1} = F\hat{x}_{k-1|k-1} + B\dot{\theta}_k \tag{10}$$

Esta es otra razón por la que $\dot{\theta}_k$, que en la expresión (1) aparecía como u_k , es una entrada de control, puesto que se utiliza como una entrada extra para estimar el estado en el tiempo actual k, llamado estimación del estado a priori, como se comentó al inicio del apartado.

Lo siguiente es intentar estimar la matriz de covarianza del error a priori $P_{k|k-1}$, basada en la matriz de covarianza del error anterior $P_{k-1|k-1}$:

$$P_{k|k-1} = FP_{k-1|k-1}F^T + Q_k \tag{11}$$

Esta matriz se utiliza para estimar cuánto se confía en los valores actuales del estado estimado. Mientras más pequeña sea, más se confía en el estado estimado actual. La matriz de covarianza del error en este caso es una matriz 2x2, como puede verse en (12).

$$P = \begin{bmatrix} P_{00} & P_{01} \\ P_{10} & P_{11} \end{bmatrix} \tag{12}$$

Actualización

Lo primero que se hace es calcular la diferencia entre la medida y el estado anterior, esto se denomina innovación (13):

$$\tilde{y}_k = z_k - H\hat{x}_{k|k-1} \tag{13}$$

El modelo de observación H se utiliza para mapear el estado a priori $x_{k|k-1}$ en el espacio observado que es la medida del acelerómetro, por tanto la innovación no es una matriz. A continuación se calcula lo que se llama la covarianza de innovación:

$$S_k = HP_{k|k-1}H^T + R \tag{14}$$

Esto intenta predecir cuánto se debería confiar en la medida basada en la matriz a priori de covarianza del error y la covarianza del ruido de la medida R. El modelo de observación H se utiliza para mapear la matriz a priori de covarianza del error en el espacio observado. Mientras mayor sea el valor del ruido de la medida, mayor será el valor de S, lo que significa que no se confía tanto en la medida que llega. En este caso S tampoco es una matriz.

El siguiente paso es calcular la ganancia de Kalman. La ganancia de Kalman se utiliza para indicar cuánto se confía en la innovación y se define como en (15):

$$K_k = P_{k|k-1}H^T S_k^{-1} \tag{15}$$

Se puede ver que si no se confía tanto en la innovación, la covarianza de innovación S será alta y si se confía en la estimación del estado entonces la matriz de covarianza del error P será pequeña; la ganancia de Kalman será por tanto pequeña y sucederá lo contrario si se confía en la innovación pero no se confía en la estimación del estado actual.

Si se presta atención puede observarse que la traspuesta del modelo de observación H se utiliza para mapear el estado de la matriz de covarianza del error P en el espacio observado. Entonces se compara la matriz de covarianza del error multiplicando con la inversa de la covarianza de innovación S. Esto tiene sentido puesto que se usará el modelo de observación H para extraer información de la covarianza del error del estado y compararlo con la estimación actual de la covarianza de innovación.

En este caso, la ganancia de Kalman es una matriz 2x1:

$$K = \begin{bmatrix} K_0 \\ K_1 \end{bmatrix}$$

Ahora se puede actualizar la estimación a posteriori del estado actual (16):

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k \tilde{y}_k \tag{16}$$

Se debe recordar que la innovación es la diferencia entre el estado observado y la estimación del estado a priori, así que la innovación puede ser tanto positiva como negativa. De forma muy simplificada esta ecuación puede entenderse como si simplemente se corrigiese la estimación del estado a priori, que fue calculada usando el estado anterior y la medida del giróscopo, con la medida del acelerómetro. Lo último que se necesita hacer es actualizar la matriz a posteriori de covarianza de error (17).

$$P_{k|k} = \left(I - K_k H\right) P_{k|k-1} \tag{17}$$

Donde I es la matriz de identidad 2x2. Lo que el filtro está realizando es básicamente auto-corregir la matriz de covarianza del error tanto como se corrigió la estimación. Esto tiene sentido puesto que se corrigió el estado basándose en la matriz a priori de la covarianza del error $P_{k|k-1}$, pero también en la covarianza de innovación S_k.

6.5.2.4 Implementación del Filtro de Kalman

En esta sección se utilizarán las ecuaciones anteriores para calcular cada uno de los términos de forma detallada y extendida, y su expondrá su posterior conversión a código.

Paso 1:

$$\boxed{\hat{x}_{k|k-1} = F\hat{x}_{k-1|k-1} + B\dot{\theta}_k}$$

$$\begin{bmatrix} \theta \\ \dot{\theta}_b \end{bmatrix}_{k|k-1} = \begin{bmatrix} 1 & -\Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \theta \\ \dot{\theta}_b \end{bmatrix}_{k-1|k-1} + \begin{bmatrix} \Delta t \\ 0 \end{bmatrix} \dot{\theta}_k = \begin{bmatrix} \theta + \Delta t \left(\dot{\theta} - \dot{\theta}_b \right) \\ \dot{\theta}_b \end{bmatrix}$$

Paso 2:

$$\boxed{P_{k|k-1} = F P_{k-1|k-1} F^T + Q_k}$$

$$\begin{bmatrix} P_{00} & P_{01} \\ P_{10} & P_{11} \end{bmatrix}_{k|k-1} = \begin{bmatrix} 1 & -\Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} P_{00} & P_{01} \\ P_{10} & P_{11} \end{bmatrix}_{k-1|k-1} \begin{bmatrix} 1 & 0 \\ -\Delta t & 1 \end{bmatrix} + \begin{bmatrix} Q_\theta & 0 \\ 0 & Q_{\dot{\theta}_b} \end{bmatrix} \Delta t =$$

$$= \begin{bmatrix} P_{00} - \Delta t \left(\Delta t P_{11} - P_{01} - P_{10} + Q_\theta \right) & P_{01} - \Delta t P_{11} \\ P_{10} - \Delta t P_{11} & P_{11} + Q_{\dot{\theta}_b} \Delta t \end{bmatrix}$$

Paso 3:

$$\boxed{\tilde{y}_k = z_k - H \hat{x}_{k|k-1}}$$

$$\tilde{y}_k = z_k - \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} \theta \\ \dot{\theta}_b \end{bmatrix}_{k|k-1} = z_k - \theta_{k|k-1}$$

Paso 4:

$$\boxed{S_k = H P_{k|k-1} H^T + R}$$

$$S_k = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} P_{00} & P_{01} \\ P_{10} & P_{11} \end{bmatrix}_{k|k-1} \begin{bmatrix} 1 \\ 0 \end{bmatrix} + R = P_{00 k|k-1} + \text{var(v)}$$

Paso 5:

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

$$\begin{bmatrix} K_0 \\ K_1 \end{bmatrix}_k = \begin{bmatrix} P_{00} & P_{01} \\ P_{10} & P_{11} \end{bmatrix}_{k|k-1} \begin{bmatrix} 1 \\ 0 \end{bmatrix} S_k^{-1} = \begin{bmatrix} P_{00} \\ P_{10} \end{bmatrix}_{k|k-1} S_k^{-1} = \frac{\begin{bmatrix} P_{00} \\ P_{10} \end{bmatrix}_{k|k-1}}{S_k}$$

Paso 6:

$$\boxed{\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k \tilde{y}_k}$$

$$\begin{bmatrix} \theta \\ \dot{\theta}_b \end{bmatrix}_{k|k} = \begin{bmatrix} \theta \\ \dot{\theta}_b \end{bmatrix}_{k|k-1} + \begin{bmatrix} K_0 \\ K_1 \end{bmatrix}_k \tilde{y}_k = \begin{bmatrix} \theta \\ \dot{\theta}_b \end{bmatrix}_{k|k-1} + \begin{bmatrix} K_0 \tilde{y} \\ K_1 \tilde{y} \end{bmatrix}_k$$

Paso 7:

$$\boxed{P_{k|k} = \left(I - K_k H \right) P_{k|k-1}}$$

$$\begin{bmatrix} P_{00} & P_{01} \\ P_{10} & P_{11} \end{bmatrix}_{k|k} = \left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \begin{bmatrix} K_0 \\ K_1 \end{bmatrix}_k \begin{bmatrix} 1 & 0 \end{bmatrix} \right) \begin{bmatrix} P_{00} & P_{01} \\ P_{10} & P_{11} \end{bmatrix}_{k|k-1} = \begin{bmatrix} P_{00} & P_{01} \\ P_{10} & P_{11} \end{bmatrix}_{k|k-1} - \begin{bmatrix} K_0 P_{00} & K_0 P_{01} \\ K_1 P_{00} & K_1 P_{01} \end{bmatrix}$$

Estos cálculos son los que convertidos a código incluyen en la librería Kalman.h que se ha utilizado para aplicar el filtro de Kalman en el código para el control del vehículo. El código de esta librería, que sea obtenido con ayuda de diferentes fuentes que se incluyen en la bibliografía, se muestra en la figura 6.25. No se explicará aquí la estructura que deben tener las librerías en *Arduino.* Puede observarse que muchas variables reciben su nombre del inglés.

Librería Kalman.h

```
#ifndef _Kalman_h
#define _Kalman_h

class Kalman {
public:
    Kalman() {
        /* Se configurarán las variables así. Estos valores también pueden ajustarse por el usuario */
        Q_angle = 0.001;
        Q_bias = 0.003;
        R_measure = 0.03;

        angle = 0; // Resetear el ángulo
        bias = 0;  // Resetear la desviación por deriva

        P[0][0] = 0; // Al inicio se considera desviación nula
        P[0][1] = 0; // Se usa como ángulo inicial el de los acelerómetros con la función setAngle
        P[1][0] = 0;
        P[1][1] = 0;
    };
    // El ángulo debe estar en °, la velocidad (rate) en °/s y el dt en segundos
    double getAngle(double newAngle, double newRate, double dt) {

        // Filtro de Kalman discreto. Predicción.
        /* Paso 1 */
        rate = newRate - bias;
        angle += dt * rate;

        // Actualizar la estimación de covarianza de error
        /* Paso 2 */
        P[0][0] += dt * (dt*P[1][1] - P[0][1] - P[1][0] + Q_angle);
        P[0][1] -= dt * P[1][1];
        P[1][0] -= dt * P[1][1];
        P[1][1] += Q_bias * dt;

        // Filtro de Kalman Discreto. Actualización.
        // Calcular la ganancia de Kalman
        /* Paso 4 */
        S = P[0][0] + R_measure;

        /* Paso 5 */
        K[0] = P[0][0] / S;
        K[1] = P[1][0] / S;

        // Calcular el ángulo y desviación por deriva
        // Actualizar estimacion con medida zk (newAngle)
        /* Paso 3 */
        y = newAngle - angle;
        /* Paso 6 */
        angle += K[0] * y;
        bias += K[1] * y;

        // Calcular la estimación de la covarianza de error
        // Actualizar la covarianza de error
        /* Step 7 */
        P[0][0] -= K[0] * P[0][0];
        P[0][1] -= K[0] * P[0][1];
        P[1][0] -= K[1] * P[0][0];
        P[1][1] -= K[1] * P[0][1];

        return angle;
    };
    void setAngle(double newAngle) { angle = newAngle; }; // Utilizado para configurar
el ángulo. Este debería configurarse como el ángulo inicial.
    double getRate() { return rate; }; // Devuelve la velocidad angular sin deriva

    /* Esto se usa para ajustar el filtro Kalman*/
    void setQangle(double newQ_angle) { Q_angle = newQ_angle; };
    void setQbias(double newQ_bias) { Q_bias = newQ_bias; };
    void setRmeasure(double newR_measure) { R_measure = newR_measure; };
```

```
69
70      double getQangle() { return Q_angle; };
71      double getQbias() { return Q_bias; };
72      double getRmeasure() { return R_measure; };
73
74  private:
75      /* Variables del filtro de Kalman */
76      double Q_angle;   // Varianza del ruido de proceso para los acelerómetros.
77      double Q_bias;    // Varianza del ruido de proceso para la desviación del giroscopo
78      double R_measure; // Varianza de ruido de la medida
79
80      double angle; // Ángulo calculado por el filtro Kalman
81      double bias;  // Desviación de los giróscopos calculados por el Filtro
82      double rate;  // Vel angular sin desviación calculada con la vel. angular y la
83                    // desviación calculada
84
85      double P[2][2]; // Matriz de covarianza del error. Es una matriz 2x2.
86      double K[2];    // Ganancia Kalman - Es un vector 2x1
87      double y;       // Ángulo diferencia
88      double S;       // Error estimado
89  };
90
91  #endif
92
```

Figura 6.25: Código para el Filtro de Kalman

6.5.3 Filtro Complementario

Aunque para el control del robot se confía plenamente en el filtro de Kalman para la obtención del ángulo, también se programó un filtro complementario para comparar su resultado con el filtro de Kalman.

Un filtro complementario se basa en el hecho de que para estimar una variable (en este caso el ángulo) se dispone de dos fuentes de medidas diferentes (el ángulo de los acelerómetros y la velocidad angular del giróscopo), tales que una de ellas proporciona sólo buena información en regiones de baja frecuencia (los acelerómetros), mientras que la otra sólo es buena en la región de alta frecuencia (el giróscopo).

De esta manera se utiliza un filtro paso alto para la medida del ángulo estimado por integración de la velocidad angular que proporciona el giróscopo, y un filtro paso bajo para la medida del ángulo que proporcionan los acelerómetros. Puede verse un esquema en la figura 6.26.

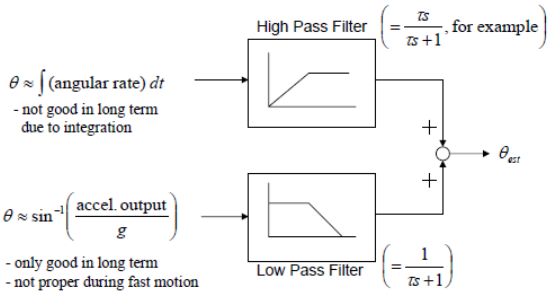

Figura 6.26: Filtro complementario

A continuación se incluye la expresión que permite obtener el ángulo estimado mediante el filtro complementario, donde puede observarse que se trata de un algoritmo recursivo, pues además utiliza la información de la estimación anterior:

$$\hat{\theta}_k = c \cdot \left(\hat{\theta}_{k-1} + \dot{\theta}_{giro} \Delta t\right) + d \cdot \theta_{acel}$$

En la expresión anterior "c" y "d" son constantes de ajuste que se obtienen por experimentación, mientras que θ_{acel} y $\dot{\theta}_{giro}$ son el ángulo y la velocidad angular obtenida directamente de los acelerómetros y el giróscopo, respectivamente. El código correspondiente a esta expresión está incluido en el código final del vehículo, tanto para el ángulo de cabeceo como de balanceo y puede verse en la figura 6.27.

Filtro Complementario

```
// Cálculo del ángulo usando un filtro complementario
compAngleX = 0.93 * (compAngleX + gyroXrate * dt) + 0.07 * roll;
compAngleY = 0.93 * (compAngleY + gyroYrate * dt) + 0.07 * pitch;
```

Figura 6.27: Código Filtro Complementario

6.5.4 Implementación del Código para la Obtención del Ángulo

Como ya se ha indicado, para elaborar el código correspondiente a la obtención del ángulo se han utilizado diferentes fuentes que se incluyen en la bibliografía, y sin las cuales no podría haber sido posible. A continuación se detalla el código que se ha utilizado, aunque ya se pudo ver en la figura 6.3, que puede dividirse en dos secciones principales: la correspondiente a inicialización y configuración de la IMU, que está incluida en la función *setup()*; y la correspondiente al cálculo recursivo del ángulo, que está incluido en la función *loop().*

6.5.4.1 Inicialización y configuración

El código correspondiente a esta parte se incluye en la figura 6.28. En la primera línea se inicia la comunicación del bus I2C (comunicación *IMU-Arduino)*, y en la segunda línea se sube la frecuencia del bus I2C al máximo que puede trabajar la IMU, que es 400KHz.

Obtención del Ángulo: inicialización y configuración

```
Wire.begin();
TWBR = ((F_CPU / 400000L) - 16) / 2;
i2cData[0] = 7
i2cData[1] = 0x00;
i2cData[2] = 0x00;
i2cData[3] = 0x00;
while (i2cWrite(0x19, i2cData, 4, false));
while (i2cWrite(0x6B, 0x01, true));
while (i2cRead(0x75, i2cData, 1));
if (i2cData[0] != 0x68) {
  Serial.print(F("Error leyendo sensor"));
  while (1);
}
delay(100);

/* Ángulos iniciales cálculo kalman y con giróscopos*/
while (i2cRead(0x3B, i2cData, 6
accX = (i2cData[0] << 8) | i2cData[1];
accY = (i2cData[2] << 8) | i2cData[3];
accZ = (i2cData[4] << 8) | i2cData[5];

// Posteriormente se convierte de radianes a grados
double roll =atan2(accY,accZ)*RAD_TO_DEG;
double pitch=atan(-accX/sqrt(accY*accY+accZ*accZ))*RAD_TO_DEG;
```

```
25
26      // Inicializar ángulos
27      kalmanX.setAngle(roll);
28      kalmanY.setAngle(pitch);
29      gyroXangle = roll;
30      gyroYangle = pitch;
31      compAngleX = roll;
32      compAngleY = pitch;
33      timer = micros();
34
35
```

Figura 6.28: Inicialización y Configuración para la obtención del Ángulo

De la líneas 3 a la 6, se configura la tasa de muestreo a 1KHz, el filtrado de las señales de salida de los giróscopos y acelerómetros. Y el rango y precisión de los acelerómetros al caso de precisión más alta. Los valores necesarios se explican en el *datasheet* de la IMU que incluye su mapa del registro.

De la línea 7 a la 14 se introducen esos valores de configuración en el bus I2C y se espera a que de confirmación de éxito.

Posteriormente, de la línea 18 a la 20 se forman los valores de aceleración procedente de los acelerómetros. Se dice que se forman porque el valor de la aceleración correspondiente a un acelerómetro no cabe en un solo registro de la IMU y está dividido en 2, por lo que hay que hacer una operación de composición. Estos valores de la aceleración son los que se utilizarán como iniciales para el filtro de Kalman, por lo que es recomendable mantener la unidad en reposo cuando se inicia.

En las líneas 23 y 24 se calculan los ángulos de cabeceo y balanceo con las expresiones (18) y (19) obtenidas en el apartado 6.5.1. Como se ha dicho esta servirán como iniciales para los cálculos del filtro Kalman, lo que se realiza en las líneas 27 y 28.

De las líneas 29 a 32 se inicializan las estimaciones del ángulo que se calcularán con el filtro complementario y por integración simple de las velocidades angulares de los giróscopos.

6.5.4.2 Cálculo Recursivo de los Ángulos

El código correspondiente a esta parte se incluye en la figura 6.29. En la primera línea se produce la lectura del bus I2C.

Actualización del Cálculo de los Ángulos

```
1       /////CÓDIGO PARA LA IMU/////
2       /* Actualización de todos los valores */
3       while (i2cRead(0x3B, i2cData, 14));
4       accX = ((i2cData[0] << 8) | i2cData[1]) - 432.0;
5       accY = ((i2cData[2] << 8) | i2cData[3]) - 291.16;
6       accZ = ((i2cData[4] << 8) | i2cData[5]) - 307.52;
7       tempRaw = (i2cData[6] << 8) | i2cData[7];
8       gyroX = (i2cData[8] << 8) | i2cData[9];
9       gyroY = (i2cData[10] << 8) | i2cData[11];
10      gyroZ = (i2cData[12] << 8) | i2cData[13];
11      double dt = (double)(micros()- timer) / 1000000; // Cálculo de dt
12      timer = micros();
13
14      // atan2 devuelve valores de -π a π (radianes)
15      // Posteriormente se convierte de radianes a grados
16      double roll  = atan2(accY, accZ) * RAD_TO_DEG;
17      double pitch = atan(-accX / sqrt(accY * accY + accZ * accZ)) * RAD_TO_DEG;
18      double gyroXrate = gyroX / 131.0;
19      double gyroYrate = gyroY / 131.0;
20
21      if ((roll < -90 && kalAngleX > 90) || (roll > 90 && kalAngleX < -90)) {
22        kalmanX.setAngle(roll);
23        compAngleX = roll;
24        kalAngleX = roll;
25        gyroXangle = roll;
26      } else
27        kalAngleX = kalmanX.getAngle(roll, gyroXrate, dt);
```

```
28
29      if (abs(kalAngleX) > 90)
30        gyroYrate = -gyroYrate;
31      kalAngleY = kalmanY.getAngle(pitch, gyroYrate, dt);
32
33      // Cálculo del ángulo mediante giróscopos sin ningún filtro
34      gyroXangle += gyroXrate * dt;
35      gyroYangle += gyroYrate * dt;
36      // Cálculo del ángulo usando un filtro complementario
37      compAngleX = 0.93 * (compAngleX + gyroXrate * dt) + 0.07 * roll;
38      compAngleY = 0.93 * (compAngleY + gyroYrate * dt) + 0.07 * pitch;
39     // Reseteo del ángulo de los giróscopos cuando se ha producido mucha deriva
40      if (gyroXangle < -180 || gyroXangle > 180)
41        gyroXangle = kalAngleX;
42      if (gyroYangle < -180 || gyroYangle > 180)
43        gyroYangle = kalAngleY;
44      ////////////////////////////////
45
```

Figura 6.29: Actualización del Cálculo de los Ángulos

De la línea 4 a la 6 se almacenan y componen los valores de aceleración de los acelerómetros. Además estos valores se corrigen sumándoles un offset calculado en su calibración. Conviene comentar que aunque la IMU se ha calibrado de fábrica, siempre existirán pequeñas desviaciones de su sistema de referencia debido a imperfecciones en el montaje o fabricación. Por ello hay que efectuar una calibración manual una única vez.

En la línea 7 se almacena y compone el valor de la temperatura. La IMU incluye un sensor de temperatura.

De la línea 8 a la 10 se almacenan y componen los valores de velocidad angular procedente de los giróscopos. Hay que destacar que todos los valores procedentes de la IMU se reciben en crudo, esto es, en unas unidades diferentes de las usadas. Por ello se deben aplicar factores de conversión que se pueden consultar en el *datasheet* como se observa en las líneas 18 y 19 para las velocidades angulares. Cabe destacar que las aceleraciones no se convierten porque nunca se utilizan sus valores absolutos, sino siempre proporciones entre ellas, así que no es necesario.

En las líneas 16 y 17 se calculan los ángulos de cabeceo y balanceo con las expresiones (18) y (19) obtenidas en el apartado 6.5.1.

De la línea 21 a la 31 se resuelve el problema de transición que sucede cuando el ángulo de los acelerómetros salta entre los dos límites de intervalo. Además se invoca al filtro de Kalman mediante un objeto que se creó al inicio del programa. En *Arduino* también existe la capacidad heredada de crear objetos, y en este caso se utiliza cuando se hace uso de la librería *Kalman.h*

De la línea 33 a la 35 se realiza el cálculo de los ángulos de cabeceo y balanceo mediante simple integración de la velocidad angular de los giróscopos en el intervalo de tiempo. Estos ángulos calculados mediante integración se reinicializan de la línea 40 a 43 cuando se ha producido mucha deriva.

Finalmente, en las líneas 37 y 38 se calculan los ángulos usando un filtro complementario como se indicó en el apartado anterior.

En este código se observa que se han obtenido los dos ángulos, de cabeceo y balanceo, aunque sólo interesa para este proyecto el de cabeceo. Esto es así porque se mantuvo desde el inicio cuando se implementó el código para su depuración. Como el coste de calcular el otro ángulo es muy reducido, no importa mantenerlo en el código.

Hay que tener en cuenta que las lecturas y escrituras del bus I2C no son automáticas en *Arduino.* Aunque se haya utilizado la librería *Wire*, también se han definido unas funciones para enviar y recibir los valores deseados, hacia y desde las direcciones de registro adecuadas. Esto no se detallará aquí para no complicar más la memoria, pero este código se incluye en el apéndice.

6.6 Aplicación en *Processing*

Durante la realización del proyecto, después de la implementación del código para la posición, velocidad y aceleración, y del código para la obtención de datos de la IMU, se decidió realizar una aplicación en *Processing* (lenguaje en el que está basado *Arduino*) a modo de ayuda para la depuración.

Processing es un lenguaje basado en *Java* fundamentalmente destinado para realizar programas y entornos visuales. De esta manera se decidió integrar en una misma ventana la mayor parte de la información procedente de la IMU y de los encoders, así como los resultados de los cálculos realizados con esa información. El objetivo era comprobar si todo funcionaba correctamente, y también se pensó que podría servir en el futuro como monitorización de lo que sucedía en el robot en tiempo real, aunque posteriormente se desestimó esta idea porque la carga de trabajo extra que supone enviar toda la información por el puerto serie ralentizaba demasiado la ejecución de los programas.

En la figura 6.30 puede verse una captura de pantalla de esta aplicación en funcionamiento. Se observa que en la primera gráfica incluida en la aplicación se representa el ángulo calculado directamente con los acelerómetros, donde puede verse el ruido en la señal; junto con el ángulo calculado simplemente por integración de la velocidad angular, donde se comprueba la deriva al cabo del tiempo, si se deja correr la aplicación.

En la segunda gráfica de la aplicación se incluye el ángulo estimado con el filtro de Kalman y con el filtro complementario, comprobándose que para pequeños movimientos efectuados con la mano, ambos filtros devuelven señales casi idénticas.

En las últimas gráficas se representa la velocidad y aceleración del motor derecho. Y numéricamente se muestra la posición del ángulo en grados obtenida con los *encoders*. Del mismo modo, al lado de cada gráfica existe una leyenda para cada curva junto con el valor numérico de la variable en tiempo real.

Por último también se incluye en la esquina inferior derecha la temperatura proporcionada por el sensor integrado en la IMU, convertida a grados centígrados.

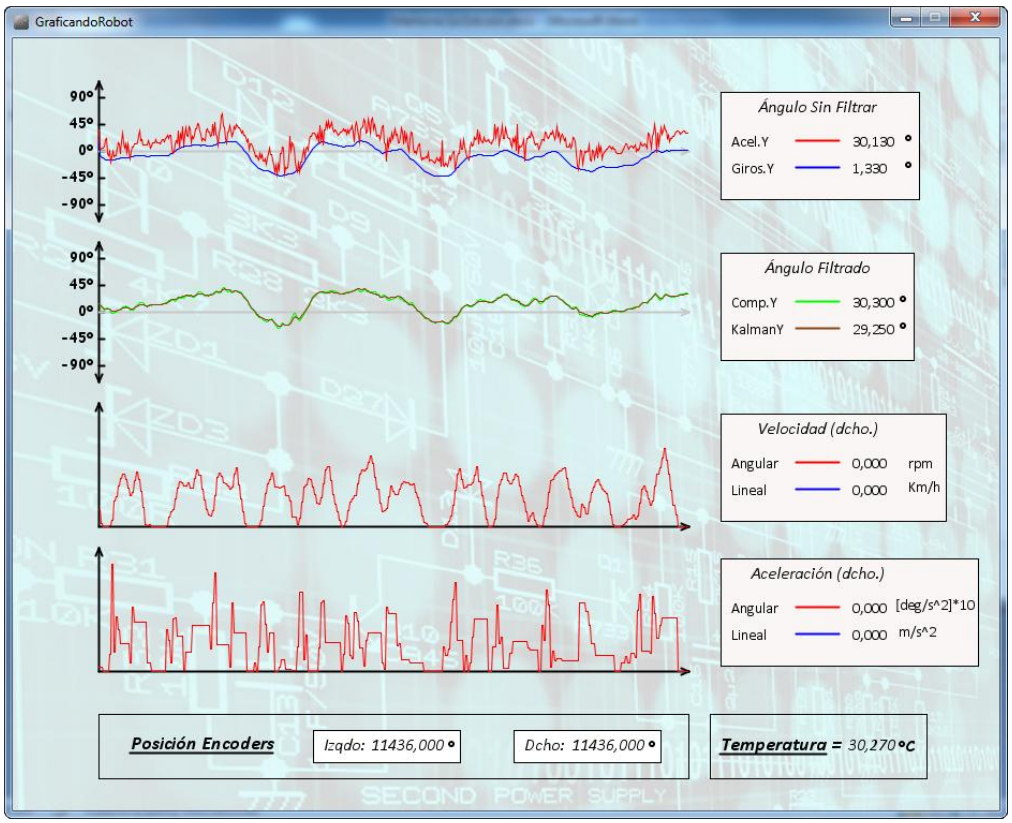

Figura 6.30: Captura de la Aplicación en *Processing*

6.7 Configuración y Mando de las Controladoras de Motores

En el apartado 4.6.3 ya se explicaron los posibles modos de funcionamiento y configuración de la controladora de motores elegida TB6612FNG. En este caso únicamente se hará mención a cómo se ha implementado el código para su utilización en Arduino.

En este sentido existe una primera parte de definición de variables y constantes para la posición de los pines escogidos, que se incluye en la figura 6.40. La elección de los pines 7 y 9 para la señal PWM (líneas 2 y 3 del código) no es aleatoria, pues esto pines permiten la configuración de dicha señal como se verá a continuación. Cada una de las dos controladoras utilizadas, necesita una combinación de dos entradas digitales para seleccionar el modo de funcionamiento, por eso se han necesitado 4 pines digitales en total.

Definición de variables y pines TB6612FNG

```
1   /* PARA EL MOTOR */
2   #define PinPWMD 7
3   #define PinPWMI 9
4   #define PinIn1I 40
5   #define PinIn2I 41
6   #define PinIn1D 43
7   #define PinIn2D 42
8   float PWMD;
9   float PWMI;
10  ////////////////////////
11
```

Figura 6.40: Definición de variables y pines TB6612FNG

Posteriormente, e incluido en la función *setup()*, existe un código para la configuración de los pines digitales como entradas, y de la señal PWM. Este código puede verse en la figura 6.41.

Configuración de pines TB6612FNG y señal PWM

```
1   /////////// MOTOR /////////////
2   //Para subir la frecuencia del PWM pin 7 del Mega 2560 de 489Hz a 31333Hz
3   TCCR4B = TCCR4B & 0b000 | 0x01;
4   //Para subir la frecuencia del PWM pin 9 del Mega 2560 de 489Hz a 31333Hz
5   TCCR2B = TCCR2B & 0b000 | 0x01;
6
7   pinMode(PinPWMD, OUTPUT);
8   pinMode(PinPWMI, OUTPUT);
9   pinMode(PinIn1I, OUTPUT);
10  pinMode(PinIn2I, OUTPUT);
11  pinMode(PinIn1D, OUTPUT);
12  pinMode(PinIn2D, OUTPUT);
13  ////////////////////////
14
```

Figura 6.41: Configuración de pines y señal PWM

La frecuencia que *Arduino* trae por defecto para sus señales de salida PWM es de 489Hz. Pero consultando el *datasheet* del microcontrolador Atmel se observa que existen 6 salidas a las que se les puede modificar esta frecuencia. Al igual que para las interrupciones internas, de generar la señal PWM se encargan los *timers*. De esta forma, se pueden elegir diferentes *preescalers* para elegir las diferentes frecuencias.

En este caso se decidió configurar los *Timer*s 2 y 4, que eran de los pocos que quedaban que se encargan de las señales PWM y que se cercioró que al modificarlos no se alterarían otras funciones utilizadas en el programa. La configuración elegida consiste en elegir el *preescaler* 1, con el que se consigue la mayor frecuencia posible, que es lo que interesa para esta aplicación, de 31333Hz. Como la controladora de motores escogida permite una frecuencia de hasta 100KHz, se comprobó que no había problema a la

hora de utilizar esa frecuencia. Esta modificación de la frecuencia tiene lugar en las líneas 3 y 5 del código.

La razón fundamental para subir la frecuencia de la señal PWM es que así se consigue un mejor comportamiento de los motores, que se percibe principalmente en la reducción de la zona muerta. Esto se pudo percibir auditivamente cuando se utilizó esta controladora con la señal antigua de 489Hz, pues al funcionar los motores se emitía un ruido agudo. Al subir la frecuencia de la señal PWM se consiguió reducir la zona muerta en alrededor de 15 valores PWM.

El resto de líneas del código de esta sección se dedican a declarar como entradas los pines definidos con anterioridad.

Finalmente se incluyen en la función *setup()* unas líneas de código para mandar directamente a las controladoras si se requiere un funcionamiento en un sentido u otro de los motores, así como la señal de tensión (en este caso PWM) que se le quiere dar a los motores. Esto se observa en la figura 6.42.

Combinación para el sentido de giro y escritura PWM

```
1   if (u_m<=0){
2     digitalWrite(PinIn1I, HIGH);
3     digitalWrite(PinIn2I, LOW);
4     digitalWrite(PinIn1D, HIGH);
5     digitalWrite(PinIn2D, LOW);
6     PWMI=17+abs(u_m);
7     PWMD=16+abs(u_m);}
8   else {
9     digitalWrite(PinIn1I, LOW);
10    digitalWrite(PinIn2I, HIGH);
11    digitalWrite(PinIn1D, LOW);
12    digitalWrite(PinIn2D, HIGH);
13    PWMI=18+abs(u_m);
14    PWMD=17+abs(u_m);}
15
16  analogWrite(PinPWMI, PWMI);
17  analogWrite(PinPWMD, PWMD);
18
```

Figura 6.42: Combinación para el sentido de giro y escritura PWM

En el caso de que el controlador emita que el robot tiene que moverse hacia adelante (condición de la línea 1), se escribe en los pines correspondiente la combinación que permite este sentido de giro (líneas 2 a la 5). En caso de que se tenga que mover en sentido contrario (condición de la línea 8), se escribe la combinación para ese otro sentido de giro (líneas 9 a la 12).

La estrategia para la compensación de la zona muerta es muy sencilla y puede verse en las líneas 6,7, 13 y 14. Finalmente, se manda la señal PWM deseada mediante las líneas 16 y 17.