Laboratorio de Principios de Mecatrónica
Proyecto Final

Leonardo Antonio García Osuna
Rebeca Baños García

Resumen—El proyecto consistía en implementar un robot diferencial utilizando 2 actuadores, uno para cada rueda. El robot debe de recorrer una trayectoria con la posición inicial y la posición final ya preestablecidas, además debe de ser capaz de esquivar cualquier obstáculo que se presente.

1. INTRODUCCIÓN

El problema fue interesante ya que tuvimos que implementar todo lo aprendido hasta ahora en el curso. Además, tuvimos que aprender a conectar todo lo que vimos para que trabajara en conjunto con Arduino y que lo pudiéramos manejar desde ROS. Esto fue para la parte del control del robot, pero para el hardware también tuvimos que verificar detalles que podrían variar o afectar en el momento en el que el robot avanzara. El objetivo principal era que el robot funcionara controlándolo de manera inalámbrica desde ROS utilizando el Arduino para conectarlo con los XBee's. A continuación se presentan los elementos esenciales que debimos tener en cuenta para el proyecto, el desarrollo que tuvimos y los resultados.

2. MARCO TEÓRICO

El proyecto final como se mencionaba anteriormente, era implementar un robot diferencial. El que nuestro robot sea diferencial implica que el tipo de direccionamiento viene dado por la diferencia de velocidad en las ruedas laterales. La tracción se consigue también con las mismas dos ruedas. Dos ruedas montadas en un único eje son independientemente propulsadas y controladas, proporcionando ambas, tracción y direccionamiento. La combinación del movimiento de las dos ruedas provoca el movimiento. Este sistema es muy útil si consideramos la habilidad del movimiento del móvil, presentando la oportunidad de cambiar su orientación sin movimientos de traslación. Las variables de este sistema son las velocidades angulares de las ruedas izquierda y derecha.[1].

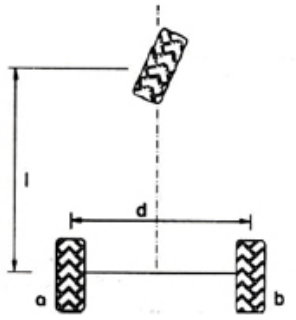

Figura 2.1: Robot Diferencial con 2 ruedas. [1]

Otro elemento esencial fue el controlador Proporcional, Integral y Derivativo, conocido como controlador PID por sus siglas. Este es un mecanismo de control simultáneo por realimentación, el cual calcula la desviación o error entre un valor medido y un valor deseado. Como lo indica su nombre, el controlador PID depende de tres parámetros distintos, el proporcional, el integral y el derivativo.

- **Proporcional:** Depende del error actual.
- **Integral:** Depende de los errores pasados.
- **Derivativo:** Es una predicción de los errores futuros.

La suma de estas tres acciones es usada para ajustar al proceso por medio de un elemento de control como la posición de una válvula de control o la potencia suministrada a un calentador. Ajustando estas tres variables en el algoritmo de control del PID, el controlador puede proveer una acción de control diseñado para los requerimientos del proceso específico [2].

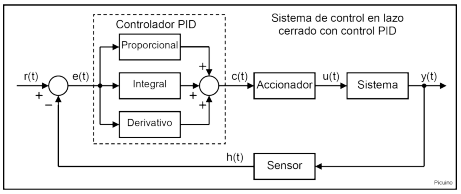

Figura 2.2: Controlador PID.

Otro elemento importante para controlar el robot de manera inalámbrica fue adaptar la conexión de ROS con Arduino. Esto fue para programar el Hardware de forma rápida y eficiente. Para usar la conexión utilizamos el paquete de rosserial-arduino para poder utilizar ROS directamente con el IDE de Arduino. Este paquete proporciona un

protocolo de comunicación ROS que funciona a través de la UART del Arduino, de esta forma permite que el Arduino sea un nodo ROS completo que puede publicar y suscribirse directamente a los mensajes ROS [3].

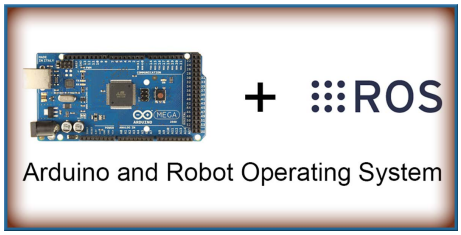

Figura 2.3: Conexion entre ROS y Arduino por medio del paquete rosserial-arduino.

Finalmente, requerimos de comprender de manera muy básica algoritmos para la planeación de trayectorias y navegación; para mantenerlo sencillo decidimos integrar una librería de manejo de polígonos y geometría del plano para ayudarnos con las intersecciones (colisiones) de la trayectoria deseada.

3. DESARROLLO

3.1. Robot físico

Comenzamos armando la parte de Hardware de robot ya que teníamos que unir todas las piezas y conectar el Arduino de manera adecuada para que las conexiones necesarias fueran eficientes. Atornillamos las partes impresas en 3D que sujetan a los dos motores y a las dos ruedas. Posteriormente conectamos los pines del Arduino a los motores que queríamos que moviera cada uno, colocando las resistencias necesarias. Obtuvimos el siguiente resultado:

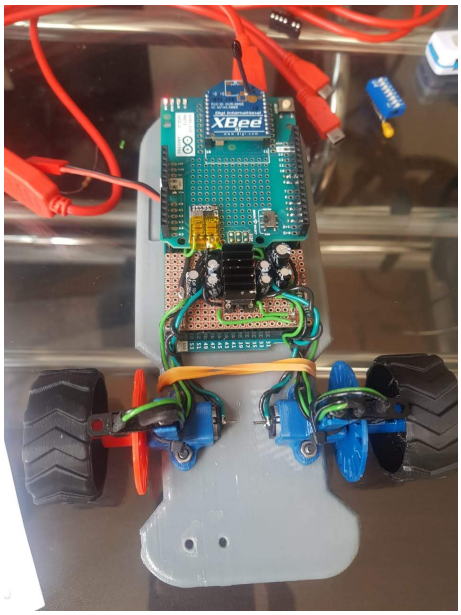

Figura 3.1: Resultado de la conexión física del robot.

A continuación, se presenta el esquemático del circuito detallado, salvo por la comunicación vía un puerto UART al módulo XBee.

La hoja de datos del codificador óptico TCST2103, detalla que es necesario tener un voltaje de $1{,}35V$, por lo que para fijar dicho voltaje usamos una señal PWM con un capacitor de $1\mu F$ para suavizarla.

Asimismo, como la velocidad de los motores se espera que cambie continuamente –a veces bruscamente–, añadimos capacitores a cada media-H del IC L293D, esto para prevenir picos de voltaje en las terminales del puente.

Además, instalamos un circuito regulador de voltaje (SBEC) para alimentar a la electrónica con $5V$, esto permite tener conectado simultáneamente el USB ($@5V$) con la batería ($@7{,}4V$ nominal) y poder *debuggear* cómodamente, además de dar un voltaje *limpio* a los componentes que lo requieren –el MCU y el XBee.

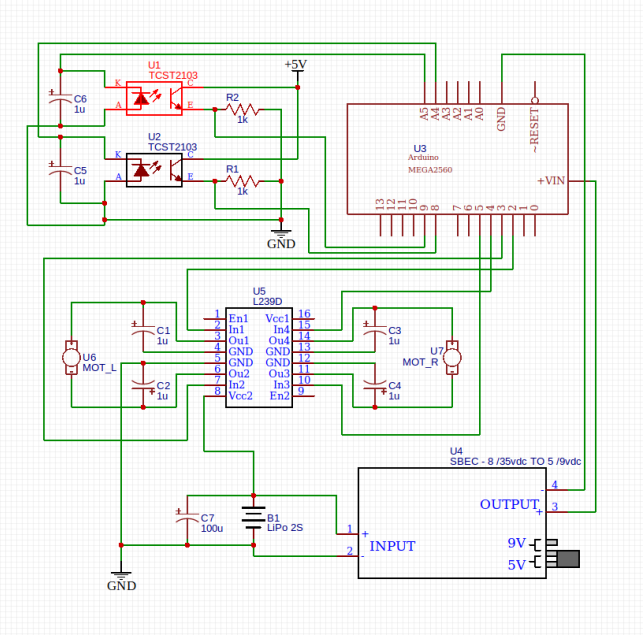

Figura 3.2: Esquemático del circuito

En el diseño esquemático se muestra como MCU un Arduino UNO, esto por la cantidad de pines no usados en el Arduino MEGA, para comodidad de lectura.

3.2. Programación Robot Físico –Arduino/ROS

Dadas las condiciones de la sección anterior, tenemos los siguientes requerimientos en cuanto a pines para el Arduino MEGA

- 2 pines para interrupciones (Entrada del fototransistor)
- 4+2 pines PWM –4 para los motores y 2 para el LED emisor
- 2 pines Enable de L293D (digitales)

Configuramos los pines de entrada y salida de acuerdo a la tabla anterior para el Arduino.

```
#include <ros.h>
#include <std_msgs/String.h>
#include <geometry_msgs/Pose2D.h>

#define MOTOR_L1 4
#define MOTOR_L2 5
#define MOTOR_R1 3
#define MOTOR_R2 2
#define LED_PHOTO_L 9
#define LED_PHOTO_R 8
#define PHOTO_DET_L 21
#define PHOTO_DET_R 20
#define PULLUP1 22
```

```
#define PULLUP2 23
#define H_ENA 24
#define H_VCC 25

#define DEBUG 1
#define ENC_DEBOUNCE 2 //ms
#define LOOP_DELTA 80 //ms loop time
#define LOOP_NAV 200 //ms vel loop tiene que
    ser de 500ms
#define Kp 3    // Ganancia Proporcional
#define Ki 1.6  //Ganancia control Integral
#define Kd 0.95  //Ganancia parte Derivativa
```

Fig 3.3: Declaración de pines y dependencias.

```
void setup() {
  if (DEBUG>1){
    Serial.begin(57600);
  }
  pinMode(PHOTO_DET_L,INPUT);
  pinMode(PHOTO_DET_R,INPUT);

  pinMode(13, OUTPUT);
  pinMode(MOTOR_L1, OUTPUT);
  pinMode(MOTOR_L2, OUTPUT);
  pinMode(MOTOR_R1, OUTPUT);
  pinMode(MOTOR_R2, OUTPUT);
  pinMode(LED_PHOTO_L, OUTPUT);
  pinMode(LED_PHOTO_R, OUTPUT);
  pinMode(PULLUP1,OUTPUT);
  pinMode(PULLUP2,OUTPUT);
  pinMode(H_ENA,OUTPUT);
  pinMode(H_VCC,OUTPUT);

  attachInterrupt(digitalPinToInterrupt(
    PHOTO_DET_L),left_encoderISR,RISING);
  attachInterrupt(digitalPinToInterrupt(
    PHOTO_DET_R),right_encoderISR,RISING);

  digitalWrite(PULLUP1,HIGH);
  digitalWrite(PULLUP2,HIGH);
  digitalWrite(H_ENA,HIGH);
  digitalWrite(H_VCC,HIGH);
  analogWrite(LED_PHOTO_L, 250); //esto deja
    el voltaje filtrado en 1.35V
  analogWrite(LED_PHOTO_R, 250);

  nh.initNode();
  nh.advertise(chatter);
  nh.subscribe(sub);

  v=w=0;
}
```

Fig 3.4: Configuración de pines.

Luego, implementamos el controlador PID para los motores DC. Para dicho controlador debemos tener una medida de la velocidad de la rueda, para poder calcular el error respecto de la velocidad deseada; esto trajo complicaciones dado que había que encontrar un valor de resistencia que dotara al codificador óptico de la sensibilidad adecuada y además, la implementación de un *debounce* de software para evitar entradas (cuentas) múltiples ante un mismo impulso –acción– y así poder sensar la velocidad de la rueda de la forma más aproximada a la realidad. Probamos con diferentes tiempos de cuenta ya que observamos que cambia la respuesta ante esta variable, y tamién ante el tiempo de debounce, que no podía ser muy grande dado que interfería con la frecuencia de las interrupciones. (En 100ms perdíamos una gran parte de las velocidades, al cambiar la ventana a 20-80 ms obtuvimos la mejor respuesta –linealidad en la velocidad observada vs. el output real a los motores)

```
ros::NodeHandle  nh;
std_msgs::String str_msg;
geometry_msgs::Pose2D pose;

ros::Publisher chatter("/mobile_robot_ack", &
    str_msg);
ros::Subscriber<geometry_msgs::Pose2D> sub("/
    nav_target", &messageCb );
```

Fig 3.5: Declaración nodo ROS.

```
void messageCb(const geometry_msgs::Pose2D&
    data){
  analogWrite(13,180);
  v= data.x;
  w= data.theta;
  str_msg.data= hello;
   chatter.publish(&str_msg);
}
```

Fig 3.6: Función callback del nodo de ROS

Dentro de la función `loop()` del MCU, pusimos condiciones para discriminar tareas de control de velocidad y de actualización de valores deseados, estos son los parámetros `LOOP_NAV` y `LOOP_DELTA` , que definen el período tras el cual se ejecutará cada tarea.

De esta forma se evitan cálculos innecesarios y damos más tiempo para permitir la convergencia a un valor fijo por un período no tan corto como el de `LOOP_NAV` que es cercano a los $80ms$.

```
void loop() {
  if(millis()-lastNav> LOOP_NAV){
    if (DEBUG>1) {
      if(stringComplete){
        Serial.println(inputString);
        v=inputString.toDouble();
        inputString = "";
        stringComplete = false;
        calcSetpoint();
      }
    }
    else{
      calcSetpoint();
      str_msg.data= hello;
      nh.spinOnce();

      digitalWrite(13,LOW);
    }
    lastNav=millis();
  }
  if(millis()-lastLoop> LOOP_DELTA){
    calcVel();
    calcPID();


    setMotor(pwmLeft,pwmRight);

    if(DEBUG>1){
      Serial.print("Vel izq:\t");
      Serial.print(vel_L);
      Serial.print("\t\tVel der:\t");
      Serial.print(vel_R);
      Serial.println("");
      Serial.print("Salida izq:\t");
      Serial.print(pwmLeft);
      Serial.print("\t\tSalida der:\t");
```

```
      Serial.print(pwmRight);
      Serial.println("\n");
    }
    lastLoop=millis();
  }
}
```

Fig 3.4: Función `oop()` .

Para terminar, con los valores obtenidos de las impresiones –con `DEBUG == 2`–, ajustamos poco a poco las ganancias del controlador PID. Primero usamos un período muy largo para calibrar con detalle el número de ciclos hasta la convergencia y fuimos acortando el período y ajustando las ganancias hasta llegar a una convergencia suave y relativamente rápida al valor deseado.

Consideramos que en este tipo de implementaciones (tanto por los componentes como por la practicidad), no es de utilidad hacer un modelo de la planta dado que se desconocen muchos parámetros tanto del motor, como de la fricción, entre otros. Por lo que el control fue *ajustado* a mano. Además requeríamos de realizar una conversión más a unidades de velocidad conocidas (mms^{-1}, ms^{-1}, etc), en este caso simplemente usamos $numerodeinterrupciones/periododetiempo$ y ajustamos el controlador de acuerdo a los rangos de velocidad obtenidos. Incluso ahorramos al Arduino unas cuantas conversiones.

```
void calcSetpoint(){
  setpoint_L= 0.047*v-2.71*w;
  setpoint_R= 0.047*v+2.71*w;
}
void calcPID(){
  error_L= setpoint_L- vel_L;
  error_R= setpoint_R- vel_R;

  changeError_L= error_L- lastError_L; //
    derivative term
  changeError_R= error_R- lastError_R;
  totalError_L+= error_L; //accumalate errors
    to find integral term
  totalError_R+= error_R;
  pidTerm_L= (Kp*error_L)+(Ki*totalError_L)+(
    Kd*changeError_L); //total gain
  pidTerm_R= (Kp*error_R)+(Ki*totalError_R)+(
    Kd*changeError_R);

  pwmLeft= floor(constrain(pidTerm_L,-255,255)
    );//constraining to appropriate value
  pwmRight= floor(constrain(pidTerm_R
    ,-255,255));

  lastError_L= error_L;
  lastError_R= error_R;
  if(setpoint_L==0 &&setpoint_R==0){
    pwmLeft=pwmRight=0;
  }

}
void calcVel(){
  vel_L= counter_left*150.0/LOOP_DELTA; //
    ticks per loop
  vel_R= counter_right*150.0/LOOP_DELTA;
  if(pwmLeft<0)
    vel_L= -1.0*vel_L;
  if (pwmRight<0)
    vel_R= -1.0*vel_R;
  counter_left=0;
  counter_right=0;
}
void setMotor(int16_t izq, int16_t der){
  if(izq>0){
    analogWrite(MOTOR_L1,izq);
    analogWrite(MOTOR_L2,0);
  }else{
    analogWrite(MOTOR_L1,0);
    analogWrite(MOTOR_L2,abs(izq));
  }
  if(der>0){
    analogWrite(MOTOR_R1,der);
    analogWrite(MOTOR_R2,0);
  }else{
    analogWrite(MOTOR_R1,0);
    analogWrite(MOTOR_R2,abs(der));
  }
}
```

Fig 3.5: Transformación, PID, cálculo de velocidad observada, escritura a motores.

Como se puede ver en la figura 3.5, en la función `calcSetpoint()` hacemos una transformación simple de v, w a los objetivos de velocidad de cada rueda (setPoint_L, setPoint_R), dado que esperamos desde el nodo del navegador un vector de la forma v, w con $v = rapidez$ y $w = direccion$. Esto toma en cuenta la separación de las ruedas, el radio de las mismas, entre otros parámetros; es decir, fueron obtenidas a partir del modelo cinemático del robot diferencial.

Para finalizar mostramos las funciones asignadas a las interrupciones, que es la vía para medir la velocidad de cada rueda.

```
void right_encoderISR(){
  if(millis()-last_rcounter_compare>
    ENC_DEBOUNCE){
    counter_right++;
    last_rcounter_compare= millis();
  }
}
```

Fig 3.6: IRQs para los encoders

3.3. Planeación de Trayectoria

Después de tener al roboto físico funcionando, procedimos a programar la trayectoria que este debía de seguir. El algoritmo es sencillo, aunque tuvimos que utilizar librerías geométricas para poder considerar los obstáculos como círculos, para que al trazar la trayectoria en línea recta detectáramos el cruce de la línea con algún círculo de obstáculos. Para esto, en el código primero instanciamos los obstáculos como variables globales, después definíamos nuestra posición inicial y llamamos esa posición como coordenada y también la posición final como coordenada.

```
import rospy
import time
from geometry_msgs.msg import Pose2D
from graphical_client.msg import Pose2D_Array
import sympy
import sympy as sym
from sympy.abc import s
from sympy.geometry import Point, Circle,
    Segment, intersection, Ray

radioObstaculo=180
poseSelf= Pose2D()
poseTarget= Pose2D()
poseObs0= Pose2D()
poseObs1= Pose2D()
```

```
poseObs2= Pose2D()
poseObs3= Pose2D()
poseObs4= Pose2D()
poseObs5= Pose2D()
poseObs6= Pose2D()
poseObs7= Pose2D()


def init_pose():
    pose = Pose2D()
    pose.x = 0
    pose.y = 0
    pose.theta = 0
    return pose
def callbackSelf(dataSelf):
    poseSelf.x =int(dataSelf.x)
    poseSelf.y =int(dataSelf.y)
    poseSelf.theta =int(dataSelf.theta)

def callbackTarget(dataTarget):
    poseTarget.x =int(dataTarget.x)
    poseTarget.y =int(dataTarget.y)
    poseTarget.theta = int(dataTarget.theta)
```

Fig 3.6 Instancia de obstáculos, punto inicial y punto final,

Después llamamos a cada obstáculo para obtener las coordenadas donde se encontraba.

```
def callbackObs0(dataObs):
    poseObs0.x = int(dataObs.x)
    poseObs0.y = int(dataObs.y)
    poseObs0.theta = int(dataObs.theta)

def callbackObs1(dataObs):
    poseObs1.x = int(dataObs.x)
    poseObs1.y = int(dataObs.y)
    poseObs1.theta = int(dataObs.theta)

def callbackObs2(dataObs):
    poseObs2.x = int(dataObs.x)
    poseObs2.y = int(dataObs.y)
    poseObs2.theta = int(dataObs.theta)

def callbackObs3(dataObs):
    poseObs3.x = int(dataObs.x)
    poseObs3.y = int(dataObs.y)
    poseObs3.theta = int(dataObs.theta)

def callbackObs4(dataObs):
    poseObs4.x = int(dataObs.x)
    poseObs4.y = int(dataObs.y)
    poseObs4.theta = int(dataObs.theta)

def callbackObs5(dataObs):
    poseObs5.x = int(dataObs.x)
    poseObs5.y = int(dataObs.y)
    poseObs5.theta = int(dataObs.theta)

def callbackObs6(dataObs):
    poseObs6.x = int(dataObs.x)
    poseObs6.y = int(dataObs.y)
    poseObs6.theta = int(dataObs.theta)

def callbackObs7(dataObs):
    poseObs7.x = int(dataObs.x)
    poseObs6.y = int(dataObs.y)
    poseObs7.theta = int(dataObs.theta)
```

Fig 3.7 Llamada de cada uno de los obstáculos

Después con dos métodos calculamos las circunferencias de cada obstáculo con un radio suficiente para poder esquivarlo a salir del radio y otro método en donde calcula la intersección entre la linea recta y las circunferencias de los obstáculos. La intersección puede ser cero si estos no se intersecan.

```
def calcCircs():
```

Fig 3.8 Cálculo de circunferencias e intersección de la ruta con los obstáculos.

Después de obtener los obstáculos donde se intersecaba nuestra trayectoria principal, creamos métodos que recalculaban la trayectoria haciendo que se abriera el punto de intersección hacia donde estuviera fuera del radio, después de esto, recuperábamos la trayectoria principal para no hacer recalculo de ruta cada que se encontraba un nuevo obstáculo.

```
def distanceVector(point, vec):
    distances=[]
    n=len(vec)
    for x in xrange(n):
        laux=vec[x]
        distances.append(point.distance(laux
    [0]))

    return distances;
def arrange(point, vec):
    aux=[]
    distances= distanceVector(point, vec)
    m=len(distances)
    while m != 0 :
        ind_min= distances.index(min(distances
    ))
        aux.append(ind_min)
        del distances[ind_min]
        m=len(distances)

    return aux;

# regresa un objeto Point para librar el
    obstaculo en cuestion
def recalc(inter, centro,circ):
    a= inter[0]
    b= inter[1]
    seg= Segment(a,b)
    c= seg.midpoint
    line= Ray(centro,c)
    return intersection(line,circ);

def intDiscriminator(list):
    n=len(list)
    out=0
    for x in xrange(n):
        aux= len(list[x])
        if(aux==2):
            out= out+aux
    return out
```

Fig 3.9 Evitar el obstáculo y recuperar la trayectoria inicial.

Por último juntamos todos los cálculos para que nos desplegara la ruta terminada evitando los obstáculos.

```
def planner():
    pub = rospy.Publisher('/trajectory',
    Pose2D_Array, queue_size=10)
    rospy.Subscriber('/y_r0', Pose2D,
    callbackSelf)
    rospy.Subscriber('/ball', Pose2D,
    callbackTarget)
    rospy.Subscriber('/b_r0', Pose2D,
    callbackObs0)
    rospy.Subscriber('/b_r2', Pose2D,
    callbackObs2)
```

```
    rospy.Subscriber('/b_r1', Pose2D,
    callbackObs1)
    rospy.Subscriber('/b_r3', Pose2D,
    callbackObs3)
    rospy.Subscriber('/b_r4', Pose2D,
    callbackObs4)
    rospy.Subscriber('/b_r5', Pose2D,
    callbackObs5)
    rospy.Subscriber('/b_r6', Pose2D,
    callbackObs6)
    rospy.Subscriber('/b_r7', Pose2D,
    callbackObs7)
    rospy.init_node('planner')
    rate = rospy.Rate(1)
    # time.sleep(1)

    while not rospy.is_shutdown():
        pointSelf= Point(int(poseSelf.x),int(
    poseSelf.y))
        pointTarget= Point(int(poseTarget.x),
    int(poseTarget.y))
        arr = Pose2D_Array()  #contiene la
    salida del nodo
        arr.poses.append(poseSelf) #el primer
    punto siempre es donde se encuentra el
    robot actualmente
        obstacles= calcCircs()
        trajectory = Segment(pointSelf,
    pointTarget)
        intersections= calcIntersect(obstacles
    , trajectory)
        numIntersect= intDiscriminator(
    intersections)

        if numIntersect==0:
            arr.poses.append(poseTarget)
        else:
            while not numIntersect==0 :
                newP=[]
                index=[]
                if len(intersections[0])==2 :
                    index.append(0)
                    newP.append(recalc(
    intersections[0],Point(poseObs0.x,poseObs0
    .y), obstacles[0]))
                if len(intersections[1])==2 :
                    index.append(1)
                    newP.append(recalc(
    intersections[1],Point(poseObs1.x,poseObs1
    .y), obstacles[1]))
                if len(intersections[2])==2 :
                    index.append(2)
                    newP.append(recalc(
    intersections[2],Point(poseObs2.x,poseObs2
    .y), obstacles[2]))
                if len(intersections[3])==2 :
                    index.append(3)
                    newP.append(recalc(
    intersections[3],Point(poseObs3.x,poseObs3
    .y), obstacles[3]))
                if len(intersections[4])==2 :
                    index.append(4)
                    newP.append(recalc(
    intersections[4],Point(poseObs4.x,poseObs4
    .y), obstacles[4]))
                if len(intersections[5])==2 :
                    index.append(5)
                    newP.append(recalc(
    intersections[5],Point(poseObs5.x,poseObs5
    .y), obstacles[5]))
                if len(intersections[6])==2 :
                    index.append(6)
                    newP.append(recalc(
    intersections[6],Point(poseObs6.x,poseObs6
    .y), obstacles[6]))
                if len(intersections[7])==2 :
                    index.append(7)
                    newP.append(recalc(
    intersections[7],Point(poseObs7.x,poseObs7
    .y), obstacles[7]))
                print "New points calced: ",
    newP
                puntosOrdenados= arrange(
    pointSelf, newP)
                ntraj=[]
                ntraj.append(pointSelf)
                for x in range(len(
    puntosOrdenados)):
                    poseAdd= init_pose()
                    laux=newP[puntosOrdenados[
    x]]
                    poseAdd.x=int(laux[0].x)
                    poseAdd.y= int(laux[0].y)
                    poseAdd.theta= 0
                    ntraj.append(laux[0])
                    arr.poses.append(poseAdd)
                ntraj.append(pointTarget)
                arr.poses.append(poseTarget)
                # nintersect=[]
                # for x in xrange(len(
    puntosOrdenados)):
                #     seg= Segment(ntraj[x],
    ntraj[x+1])
                #     nintersect=
    calcIntersect(obstacles, seg)
                #     numIntersect=
    intDiscriminator(nintersect)
                numIntersect=0

            print "The array is:", arr
            pub.publish(arr)
            # while True:
            #     b=1
            rate.sleep()

if __name__ == '__main__':
    try:
        planner()
    except rospy.ROSInterruptException:
        pass
```

Fig 3.10 Trayectoria completa

Después de tener el calculo de la trayectoria completa, creamos otro archivo de código para que este mande a llamar al que crea la trayectoria y se encargue de desplegar el resultado en el simulador. En este archivo de código mandamos llamar a la trayectoria y a nuestro punto inicial para que desplegara el camino completo desde el inicio hasta el punto de meta.

```
import rospy
import time
import math
from geometry_msgs.msg import Pose2D
from graphical_client.msg import Pose2D_Array
import sympy
import sympy as sym
from sympy.abc import s
from sympy.geometry import Point, Circle,
    Segment, intersection, Ray

poseSelf= Pose2D()
poseNav= Pose2D()
velVector= Pose2D()

def callbackTrajectory(data):
    n=len(data.poses)
    poseNav.x= int(data.poses[1].x)
    poseNav.y= int(data.poses[1].y)
    poseNav.theta= data.poses[1].theta
```

```
def callbackSelf(dataSelf):
    poseSelf.x = int(dataSelf.x)
    poseSelf.y = int(dataSelf.y)
    poseSelf.theta = dataSelf.theta

def navigator():
    pub= rospy.Publisher('/nav_target', Pose2D
    , queue_size=10)
    rospy.Subscriber('/trajectory',
    Pose2D_Array, callbackTrajectory)
    rospy.Subscriber('/y_r0', Pose2D,
    callbackSelf)
    rospy.init_node('navigation')
    rate= rospy.Rate(1) # Hz
    while not rospy.is_shutdown():
        pSelf=Point(poseSelf.x,poseSelf.y)
        pNav=Point(poseNav.x,poseNav.y)
        dist= pSelf.distance(pNav)
        s= math.atan2(poseNav.y-poseSelf.y,
    poseNav.x-poseSelf.x)
        velVector.theta= s-poseSelf.theta
        if(dist>500):
            v= 350
        elif(dist>100):
            v=0.3*dist +230
        else:
            v=200
        velVector.x= v

        pub.publish(velVector)
        rate.sleep()

if __name__ == '__main__':
    try:
        navigator()
    except rospy.ROSInterruptException:
        pass
```

Fig 3.11 Resultado de la trayectoria en el simulador

4. RESULTADOS

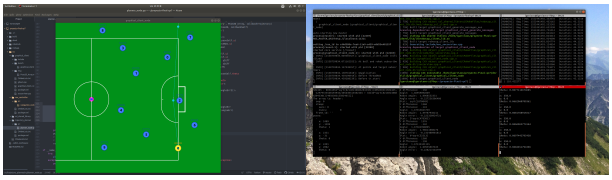

Fig 4.1: Los nodos de ROS en ejecución.

En la parte del hardware, al armar el robot tuvimos ciertos problemas ya que procuramos que los motores que utilizáramos fueran lo más similares posibles para que al encontrar el error en el controlador del PID no fuera tan complicado la modificación para que estuviera sincronizado. Al cambiar los motores tuvimos un problema ya que el motor que elegimos no lo probamos primero y al volver a conectar y soldar todo, no funcionaba una rueda, por lo que tuvimos que desarmar de nuevo el robot para poder cambiar el motor. Otro aspecto que decidimos cambiar fue una de las piezas impresas en 3D que sostenía la rueda del robot ya que los colores eran diferentes, uno era rojo y otro azul, y los encoders utilizados eran opto-interruptores, por lo que el reflejo de la luz podría alterar los valores que se leyeran por la diferencia de color, optamos por cambiarlo por tener 2 rojos.

En cuanto a la parte de programación, nos costo al inicio poder calibrar el robot al asignarle valores a las variables de controlador PID, a veces el valor asignado era mayor o menor por lo que las velocidades de las ruedas diferían y el robot se iba de lado. Nos dimos cuenta que el motor derecho giraba más rápido que el motor izquierdo, por lo que decidimos darle más velocidad al izquierdo y con esto logramos que avanzara derecho sin girar.

Para el calculo de la trayectoria el principal reto fue lograr evadir los obstáculos, ya que al tener las intersecciones con cada obstáculo, fue un problema averiguar a que lado del círculo de ese obstáculo moveríamos nuestra trayectoria, para que fuera la solución más eficiente. Tuvimos que aprovechar las herramientas que nos brindaban las librerías importadas en Python de geometría y con esto logramos utilizar vectores que nos indicaban la dirección en donde se intersecaba al obstáculo y con esta herramienta logramos irnos al lado de la circunferencia más cerca a la trayectoria original. Después de esto, decidimos dividir la trayectoria completa en varios segmentos que se partían al momento de evadir un obstáculo pero al final se conectaban entre ellos para lograr la trayectoria final.

Fig 4.2: *graphical_client* con el resto de nodos en ejecución, trayectoria en azul.

5. CONCLUSIONES

- **Rebeca Baños:** Fue muy interesante poder aplicar todo lo aprendido en teoría en una práctica tan elaborada y con muchos elementos importantes que aprendimos en el curso. Es importante tratar de implementar todo lo que se aprende teóricamente en la práctica para entender mejor como funcionan los controladores y los códigos en algo físico. La práctica también nos impuso retos de materias pasadas que nos ayudaron a solucionar los problemas presentes recordando lo visto anteriormente. Fue una práctica muy divertida en dónde aprendimos mucho.
- **Leonardo García:** La experiencia ganada y toda la investigación necesaria para el proyecto hacen que sea muy provechoso para nuestra formación. Probablemente hizo falta énfasis en la etapa de pruebas finales, que es la integración, dado que no contábamos con un simulador, toda la geometría había que imaginársela o simularla en ros con filtros para los

rosbags e inyectando manualmente posiciones para ver cómo respondía el controlador de navegación –algo lento, fuera de eso; permite ver que de la teoría a la realización de cualquier sistema existe algo que se llama implementación y que, en mi opinión, es, algunas veces, más importante que la teoría subyacente.

6. ROL O PAPEL

- **Rebeca Baños:** Se encargo de agregar en el reporte todo lo aprendido. Corrigió la pieza 3D de color y los motores. Ayudó a escribir el código.
- **Leonardo García:** Se encargó de conectar el Arduino con los motores y los encoders. Escribió todo el código de conexión entre ROS y Arduino así como el de la trayectoria del robot.

7. FUENTES CONSULTADAS

BIBLIOGRAFÍA

[1] M. Gómez, "Robot con tracción diferencial," https://www.tamps.cinvestav.mx/~mgomez/Control_Lineal/node6.html.

[2] Wikipedia, "Controlador pid," https://es.wikipedia.org/wiki/Controlador_PID.

[3] ROS.org, "Arduino ide setup," http://wiki.ros.org/rosserial_arduino/Tutorials/Arduino%20IDE%20Setup.