Reporte del Proyecto Final

Andrea Marín Alarcón
Luis Felipe Landa Lizarralde
Miguel González Borja

Resumen—Existen múltiples situaciones en las cuales un robot requiere ir de un punto inicial a un punto final evitando obstáculos. Sea en una planta donde el robot busque transportar alguna mercancía o en una cancha de fútbol y el robot se dirija al balón, la problemática es similar en muchos respectos. Para superar el problema de que nuestro robot se traslade de un inicio a una meta evitando obstáculos usamos: dos motores de corriente directa, opto-interruptores, un Arduino, controles *PID*, un sistema de visión y una algoritmo de búsqueda llamado *A**. Con la información provista por el sistema de visión, se logró calcular una ruta del inicio al fin y los motores con el control y los opto-interruptores siguen dicha ruta.

1. Introducción

Los robots autónomos se vuelven cada vez una realidad más cercana. Un robot autónomo es aquel que no requiere una aportación humana durante su operación para ejecutar su función. Dentro de estos robots autónomos existen aquellos que se desplazan de un lugar a otro evitando obstáculos.

Por ejemplo, parte del trabajo de Eduardo Morales en el Instituto Nacional de Astrofísica, Óptica y Electrónica consiste en un robot autónomo que interactúa con no expertos. A este robot se le pide que obtenga un objeto y el robot lo busca dentro de una serie de cuartos usando un sistema de visión. Genera una ruta óptima con base en una heurística de distancia a los cuartos, tamaño de los cuartos y la probabilidad de encontrar el objeto deseado en el cuarto. El robot puede agarrar objetos por medio de una mano operada con motores que igualmente requieren control. Este robot en particular se piensa que sea parte de la vida diaria en el hogar.

Otra empresa que está a la vanguardia de lo que concierne robots autónomos es Tesla. Su concepto de un vehículo autónomo incorpora elementos similares a los que se presentan en nuestro problema. Por medio de sensores ultrasónicos y cámaras, se crea una imagen del entorno. Con base en la información recolectada se determina si el vehículo debe acelerar, frenar, mantener la velocidad constante o girar en alguna dirección. Este tipo de tecnologías es capaz de alertar al conductor momentos antes de que ocurra un accidente y evitarlo. El trazado de rutas, el movimiento del vehículo y la necesidad no solo de seguir un carril, sino evitar colisiones con otros vehículos y seguir los señalamientos exaltan el nivel al que puede llegar nuestra problemática inicial.

Nuestro objetivo entonces es: dado un vehículo con dos motores y un Arduino, ser capaces de que de manera autónoma reconozca un punto objetivo y siga una ruta hasta él, evitando obstáculos fijos a lo largo de su ruta.

A continuación, en el Marco Teórico, se esbozan los conceptos principales utilizados en la solución a la problemática. En el Desarrollo se explica la solución de manera explícita, detallando lo que se llevó a cabo en el experimento. Los Resultados evalúan el éxito del proceso descrito en el Desarrollo junto con sus limitantes. La sección de Conclusiones incluye una aportación individual de cada miembro del equipo que resalta los aspectos principales del experimento. En Rol se enuncia la participación que tuvo cada miembro del equipo en el desarrollo de la solución. Finalmente, en Fuentes Consultadas se exhiben las fuentes citadas de manera directa, indirecta o que aportaron los conocimientos para el desarrollo del experimento.

2. Marco Teórico

Un robot autónomo que siga una ruta tiene varios componentes, los cuales se pueden dividir en dos categorías principales: la parte de la lógica, que se encarga de trazar la ruta y que el robot la siga; y la parte del control, es decir, lo que se encarga de regular la velocidad de los motores, así como de seguir instrucciones acerca del movimiento del robot.

En este caso se hizo un robot con tracción diferencial. La tracción diferencial se caracteriza por el uso de dos motores independientes para controlar las llantas de cada lado, a diferencia de la tracción de un coche donde la rotación de las llantas fronteras o traseras es igual en ambos lados. La Figura 1 muestra un diagrama de un robot simple con tracción diferencial. Este robot se caracteriza por el radio de cada rueda y la distancia entre las llantas [1].

El uso de motores independientes para cada llanta permite al robot tener un mayor rango de movimientos. Por ejemplo, además de poder moverse hacia adelante y atrás girando las llantas a la misma velocidad, puede también controlar la velocidad angular del robot haciendo diferir la velocidad de los motores, e inclusive puede rotar en su propio eje cuando hace girar los motores en velocidades opuestas. En otras palabras, la tracción diferencial permite un mayor control sobre la velocidad y la velocidad angular. Específicamente, la relación entre la velocidad de cada rueda (ϕ_l, ϕ_r) y la velocidad v con velocidad angular ω es:

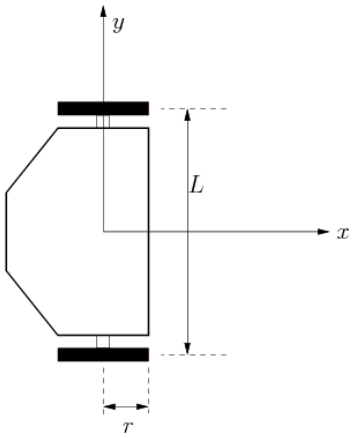


Figura 1: Diagrama de un robot diferencial

$$\begin{bmatrix}\phi_l \\ \phi_r\end{bmatrix} = \frac{1}{r}\begin{bmatrix}1 & \frac{-L}{2} \\ 1 & \frac{L}{2}\end{bmatrix}\begin{bmatrix}v \\ \omega\end{bmatrix}$$

A pesar de estas ventajas, la tracción diferencial trae consigo un gran problema. Debido al uso de motores independientes, es necesario tener un buen control no solo individual, sino sobre todos los motores en conjunto. Es decir, una tracción diferencial gana precisión sobre el movimiento del robot a costo de mayor precisión en el control de los robots.

Nuestro vehículo está conformado por dos motores de corriente directa (motores DC) y un Arduino. Este último es una plataforma de hardware y software libre, basada en una placa con un microcontrolador y un entorno (IDE) de desarollo [2].

Los motores DC son dispositivos que convierten la energía eléctrica en mecánica, provocando un movimiento rotatorio y tienen dos partes: un estator y un rotor. El primero da soporte mecánico al motor y es donde se encuentran los polos y el segundo, generalmente es de forma cilíndrica el cual se polariza por medio de una corriente la cual le llega mediante dos escobillas [3]. Para controlar cómo fluye el voltaje a través de los motores se utilizó un puente H y para controlar la velocidad se implementó un control PID (*Proportional Integrative Derivative*).

Un puente H es un circuito electrónico que permite que un motor eléctrico DC gire a la izquierda, a la derecha, entre en freno pasivo y en freno activo. Este circuito se construye con 4 interruptores. Así, dependiendo de cuáles interruptores estén abiertos/cerrados el motor se mueve en una dirección o se frena. En la Fig. 2 se puede ver el diagrama de un puente H y su funcionamiento.

Los controles PID funcionan con base en una retroalimentación de la señal de salida, conocida como señal de error, $e(t)$. Esta señal se calcula con la diferencia de la velocidad deseada y la velocidad actual del motor. Posteriormente se alimenta al controlador, el cual emite una señal de control al motor,$u(t)$, para intentar alcanzar el valor deseado [4]. Como su nombre lo indica un controlador PID está compuesto de tres partes: la proporcional, la diferencial y la integral (Fig. 3).

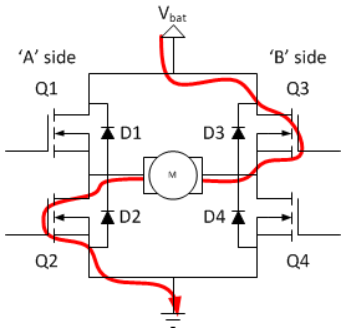


Figura 2: Diagrama de un puente H

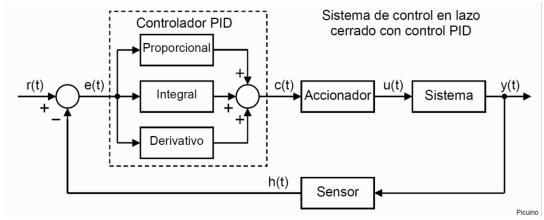


Figura 3: Diagrama de un controlador PID

La acción de control proporcional tiene como objetivo minimizar el error del sistema y da una salida proporcional a la señal de error. La acción de control integral tiene como objetivo reducir el error del sistema en régimen permanente. Es un control más lento basado en la señal de error acumulada. Por último la acción de control derivativa tiene como objetivo aumentar la estabilidad del sistema por medio del control de la rapidez con que cambia la señal del motor [5]. De esta manera la ecuación de la señal de control está dada por:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\delta\tau + K_d \frac{\delta}{\delta t} e(t)$$

Para poder determinar la velocidad a la que corren los motores se utilizaron dos optointerruptores, uno para cada rueda. Un optointerruptor es un sensor en forma de "U": en uno de los extremos está un diodo emisor de infrarrojos, y en el otro hay un fototransistor que recibe la señal. Este sensor detecta cuando un objeto para por la ranura pues se interrumpe el rayo de luz infrarroja (Fig. 4) [6]. A diferencia de los interruptores mecánicos, los optointerruptores son interruptores sin contacto, lo que mejora la confiabilidad al evitar que el sensor se desgaste debido a la abrasión (contacto) [7].

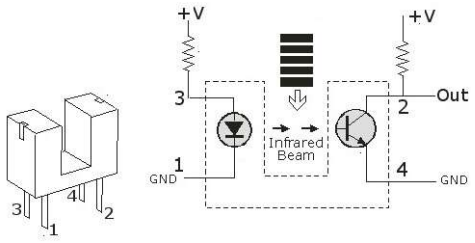


Figura 4: Diagrama de un optointerruptor

Tanto los obstáculos, como la meta y la posición del robot eran publicados por un sistema de visión a diferentes nodos

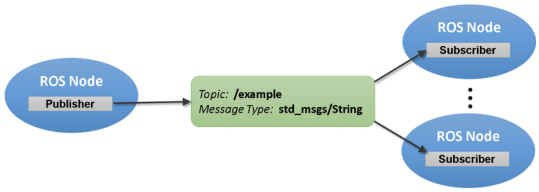


Figura 5: Sistema de publicación-inscripción de ROS

de ROS. Estos se procesaban y se calculaba la ruta utilizando un algoritmo de búsqueda. Se realizó una comunicación indirecta entre el Arduino y la computadora que procesa la trayectoria por medio de XBee.

ROS (*Robot Operating System*), es un meta sistema operativo para robots, es decir, es un intermedio entre un sistema operativo y un *middleware*. Tiene como objetivo principal facilitar la creación de comportamientos de robots que sean complejos y robustos entre diversas plataformas [8]. ROS está basado en una arquitectura *peer-to-peer*, acoplada a un sistema de almacenamiento en búfer y un sistema de búsqueda (*master*), los cuales permiten que cuales quiera dos componentes se comuniquen entre ellos de manera síncrona o asíncrona. La comunicación asíncrona se realiza por medio de un sistema de publicación-inscripción en el que se publican mensajes, que pueden ser de distintos tipos, a diferentes tópicos (Fig. 5)

Encontrar una trayectoria óptima dado un grafo es un problema bastante común y para resolverlo se pueden utilizar algoritmos de búsqueda. Un algoritmo de búsqueda se plantea en una situación particular: se tiene un estado base, un estado objetivo, en cualquier punto en el tiempo se pueden ejecutar una serie de acciones y una combinación de acciones puede o no llevarnos al estado objetivo; el proceso de búsqueda es la exploración de tales combinaciones y elegir una que llegue al estado final[9]. Comúnmente las acciones tienen asociados ciertos costos y el algoritmo de búsqueda busca no solo llegar al objetivo, sino hacerlo con el costo mínimo asociado.

El espacio de estados posibles, con sus correspondientes acciones posibles que llevan a otros estados forman un grafo dirigido. Los estados son vértices y las acciones son aristas que van del estado inicial al siguiente estado. Los algoritmos de búsqueda se pueden reducir a algoritmos que buscan recorrer este grafo de manera óptima. Para hacer esto posible, se implementan funciones llamadas heurísticas que tienen como propósito dirigir al algoritmo en el proceso de búsqueda.

Una ejemplo heurística es aproximar el costo de ir de un nodo al estado objetivo. Para obtener este costo generalmente se toma como la distancia del nodo que se está estudiando al estado objetivo. Con una heurística implementada, el algoritmo ya no recorre el grafo a fuerza bruta, es una búsqueda dirigida. Un algoritmo de particular interés que utiliza una búsqueda dirigida es el llamado A^*. Este algoritmo combina la distancia del nodo actual al nodo objetivo y el costo del camino hasta el nodo actual para recorrer el grafo. A^* es un algoritmo completo y óptimo, es decir, a diferencia de otro tipo de algoritmos, este siempre encuentra una ruta y siempre es la ruta óptima.

Finalmente, para comunicar el Arduino y el procesador se realizó de manera indirecta por medio de XBees. Los XBee son chips que se pueden comunicar de manera inalámbrica entre ellos. De esta manera puedes evitar colocar un cable físico que una los dos dispositivos. Fueron diseñados para aplicaciones de requieren un alto tráfico de datos, baja latencia y una sincronización de comunicación predecible. Los XBee están basados en el protocolo Zigbee [10].

3. DESARROLLO

3.1. Búsqueda de Ruta

Para implementar el algoritmo A* se utilizó Python. Primero dividimos la cancha con una malla para generar una matriz sobre la cual correría el algoritmo. Después se leyó la posición del robot, los centros de los obstáculos y la meta para alterar la matriz. Las coordenadas (x, y) se convierten en posiciones (i, j) que corresponden a renglón y columna en la matriz. Una vez posicionados los obstáculos, se vetan las celdas cuyo centro está a menos de 150mm para darle al robot suficiente espacio para evitar los obstáculos. Sobre la nueva matriz se genera la ruta que deberá seguir el robot. Una vez generada la ruta como conjunto de posiciones (i, j), se convierten en posiciones (x, y, θ). La θ corresponde al ángulo del vector entre el dos puntos (x, y) consecutivos. Este ángulo proporciona al robot la velocidad angular necesaria para obtener la velocidad lineal de las llantas.

```
def map_position(position):
    j = math.floor(position[0]/finesse)
    i = math.floor((-position[1]+top)/finesse)

    return int(i),int(j)
```

Figura 6: Mapeo de coordenadas espaciales a matriciales

```
def map_inverse(matrix_pos):

    half = finesse/2

    x = (matrix_pos[1]*finesse) + half
    y = top - (matrix_pos[0]*finesse) - half

    return x,y
```

Figura 7: Mapeo de coordenadas matriciales a espaciales

```
def check_neighbors(obstacle):
    global maze
    center_pos = map_position(obstacle)

    print obstacle
    print center_pos

    if not validate(center_pos):
        print 'Out of bounds'
        return None

    maze[center_pos[0]][center_pos[1]] = 0.3

    for new_position in get_neighbors():
        #Get node position
        pos = (center_pos[0] + new_position[0], center_pos[1] + new_position[1])

        if pos[0] > (len(maze) - 1) or pos[0] < 0 or pos[1] > (len(maze[0]) - 1) or pos[1] < 0:
            print 'continue'
            continue

        res = is_blocked(obstacle, pos)

        print 'Position {} is {}'.format(pos,res)

        maze[pos[0]][pos[1]] = res
```

Figura 8: Eliminación de vecinos de obstáculos

```
def get_theta(start, end):
    x = end[0] - start[0]
    y = end[1] - start[1]

    theta = 0

    if y >= 0 and x >= 0:
        theta = math.atan2(x,y)
    elif y < 0 and x >= 0:
        theta = math.pi/2 + math.atan2(-y, x)
    elif y < 0 and x < 0:
        theta = math.pi + math.atan(-x, -y)
    else:
        theta = math.pi*2 - math.atan2(-x, y)

    theta = theta if theta <= 2*math.pi else 2*math.pi

    return theta
```

Figura 9: cálculo de ángulo

3.2. ROS

Se crearon dos nodos de ROS, uno para calcular la trayectoria y otro para hacer que el robot siga la ruta deseada. Ambos se programaron utilizando Python.

El primer nodo, llamado *receiver*, estaba subscrito a los tópicos *y_r0*, *ball* y al 9 obstáculos, es decir de *br_0* a *br_8*. Se declararon variables globales para tener control sobre el estado de la ruta, las más relevantes eran:

- *ready*: funciona como bandera para saber si ya se tiene la posición de todos los obstáculos, junto con la posición inicial y final. De esta manera se evita calcular la ruta muchas veces.
- *no_robots*: número de obstáculos a esquivar
- *no_topics*: número de tópicos a escuchar para obtener la posición de los obstáculos
- *obstacles*: diccionario con la posición de los obstáculos

También se tienen variables globales para guardar la posición inicial y final.

Se creó una sola función *callback* para todos los obstáculos la cual recibe la posición del obstáculo junto con su id

```
def robot_position(msg):
    global pos

    print "Recibi inicio"

    pos = Pose2D()
    pos.x = msg.x
    pos.y = msg.y
    pos.theta = msg.theta

    if no_robots == 0 and (not end is None):
        calculate_trajectory()
```

Figura 10: Función callback para la posición del robot, nodo *receiver*

```
def obstacle_position(msg, args):
    global obstacles
    global ready

    robot_id = args
    obstacles[robot_id] = msg

    if not ready and len(obstacles.keys()) == no_robots and not pos is None:
        ready = True
        calculate_trajectory()
```

Figura 11: Función callback para la posición de los obstáculos, nodo *receiver*

(Fig. 11). La posición de los obstáculos se guarda en un diccionario donde el id funciona como llave. Cuando el número de obstáculos en el diccionario es igual al número de obstáculos deseados, entonces se manda a llamar a una subrutina que calcula la trayectoria a seguir y se activa la bandera *ready*. Para realizar lo anterior primero se checa que la bandera esté desactivada, de esta manera no se calcula la trayectoria más de una vez.

Si el número de obstáculos a esquivar es cero, entonces la función *callback* para la posición del robot es la encargada de activar la bandera y lo hace cuando la variable global *end* sea diferente de `None` (Fig. 10). En este caso no se calcula la trayectoria por medio del algoritmo de búsqueda, si no que se define una trayectoria con solo dos puntos: la posición inicial del robot y la meta.

Una vez que se tienen los puntos de la trayectoria se calcula el ángulo entre cada uno de ellos para así saber qué dirección tiene que seguir el robot para llegar al siguiente punto. Una vez que se ha calculado esto, se publica la trayectoria al tópico *final_path*.

El segundo nodo está subscrito al tópico *final_path* y tiene una bandera la cual indica si ya recibió la trayectoria o no. La trayectoria final se maneja como una cola de manera que el primer elemento siempre es el siguiente punto al que se quiere llegar.

Este nodo también se subscribe al tópico *y_r0* el cual actualiza la posición del robot (Fig. 12). Cada que recibe la posición y el ángulo del robot, checa si la distancia de éste al siguiente punto de la ruta es menor a una tolerancia fija, si sí entonces descarta el punto. Una vez que se descartaron los puntos de la trayectoria por los que ya pasó el robot, se calcula la diferencia entre el ángulo actual del robot y el ángulo al que debería de estar para llegar al siguiente

```
def update_robot(pos):
    global path
    global ready

    if not path is None:
        print path

    if ready and len(path) > 0:
        current_pos = (pos.x, pos.y)
        distance_next = get_distance(current_pos, path[0])

        while distance_next < tolerance:
            p = path.pop()
            print "Point Reached: " + str(p)
            print "Distance: " + str(distance_next)
            distance_next = get_distance(current_pos, path[0])

        if len(path) > 0:
            theta = path[0][2] - pos.theta
            if theta > math.pi:
                theta -= 2*math.pi
            if theta < -math.pi:
                theta += 2*math.pi

        else:
            theta = -np.inf

        print "New angle " + str(theta)

        get_vel(theta)
```

Figura 12: Función callback para la posición del robot, nodo *xbee*

punto. Con esta diferencia se calcula la velocidad de cada llanta y se envía al Arduino por medio del XBee. Al Arduino se le envía un arreglo de bytes el cual contiene: velocidad llanta derecha, dirección llanta derecha, velocidad llanta izquierda, dirección llanta izquierda. La dirección de las llantas es 1 si es positiva, es decir, van para adelante y 0 si es negativa.

3.3. Robot

Para controlar cada motor del robot, se utilizo un puente H conectado a una batería de 7.4 V. Luego, se agregó un encoder de 18 ranuras con un opto-interruptor al eje del motor para poder medir su velocidad. Todos estos componentes se conectaron al ATmega. Desde ahí, se corrió un programa que contaba los *ticks* de cada interruptor y con esto calculaba la velocidad del motor cada 250 ms. Después, comparaba la velocidad contra el objetivo, y, usando un control PID, determinaba el voltaje necesario para el motor. Este voltaje se conseguía utilizando un PWM al control del puente H, reduciendo el voltaje original de 7.4 V al determinado por el ciclo de trabajo de la señal.

Finalmente, el Arduino se comunicaba con el sistema de ROS que determinaba la ruta a través de un XBee. Al recibir un mensaje del sistema central, el Arduino actualiza la velocidad objetivo de cada motor, así como la dirección en que este debía girar. La Figura 13 muestra el robot diferencial completo.

4. RESULTADOS

4.1. Búsqueda de Ruta

La generación de la ruta dio resultados mixtos. El vetado de las celdas vecinas de los robots no logró eliminar todos los vecinos y por ende A* fue incapaz de generar rutas que evitara los obstáculos completamente. Sin embargo, la generación de la ruta como tal y el cálculo de los ángulos se hizo correctamente.

4.2. ROS

La comunicación con el sistema de visión se realizó de manera exitosa. Para que los dos nodos de nuestro programa se comunicaran de manera correcta fue necesario primero correr el nodo que calcula la trayectoria y luego el nodo que guía al robot por la ruta, pues si se hacía de la otra manera, el segundo nodo no recibía la ruta publicada. De igual manera las actualizaciones de la velocidad por medio del XBee se hacían correctamente.

La tolerancia utilizada para la distancia del robot con respecto al siguiente punto de la ruta fue de 90mm. Para calcular la velocidad de cada rueda se utilizó la siguiente fórmula:

$$\begin{bmatrix}\Phi_l \\ \Phi_r\end{bmatrix} = \frac{1}{r}\begin{bmatrix}1 & \frac{-L}{2} \\ 1 & \frac{L}{2}\end{bmatrix}\begin{bmatrix}v \\ \omega\end{bmatrix}$$

donde $r = 21$ era el radio de las llantas, $L = 57{,}5$ era el ancho del robot, $v = 20$ y ω era la diferencia entre el ángulo deseado y el ángulo actual del robot.

4.3. Robot

El Cuadro 1 muestra la configuración de los pines del ATmega 2560 utilizado en la construcción del robot. Además, para alimentar el Arduino y los motores DC se utilizó una batería LIPO de 7.4 V.

El Cuadro 2 muestra la configuración de los dos XBee utilizados para la comunicación entre el ATmega 2560 y la PC que determinaba la velocidad deseada de cada motor. La PC enviaba, en orden, la velocidad del motor derecho en *ticks/seg*, la dirección del motor derecho (1 hacia adelante, 0 hacia atrás), la velocidad del motor izquierdo y la dirección del motor izquierdo. Cada uno de estos elementos se enviaba como un entero de 8 bits sin signo. Al recibirlos, el AVR actualizaba los valores correspondientes para que el robot se moviera de la manera deseada.

Para los controles PID, se obtuvieron las siguientes constantes:

$$\begin{aligned} k_p^l &= 0{,}9 & k_p^r &= 0{,}7 \\ k_i^l &= 0{,}6 & k_i^r &= 0{,}7 \\ k_d^l &= 0{,}1 & k_d^r &= 0{,}15 \end{aligned}$$

Por último, la Figura 13 muestra las vistas lateral, trasera y aérea del robot creado.

Pin	I/O	Elemento
12	Output	Puente H Motor Derecho - 0
11	Output	Puente H Motor Derecho - 1
2	Output	Puente H Motor Izquierdo - 0
5	Output	Puente H Motor Izquierdo - 1
42	Input	Opto-interruptor Motor Derecho
44	Input	Opto-interruptor Motor Izquierdo
50	I/O	XBee Tx
51	I/O	XBee Rx

Cuadro 1: Configuración del ATmega2560 para el sistema

Propiedad	XBee PC	XBee AVR
ID	0x3332	0x3330
DH	0x0000	0x0000
DL	0x0030	0x0032
MY	0x0032	0x0030
Baud Rate	9,600	9,600

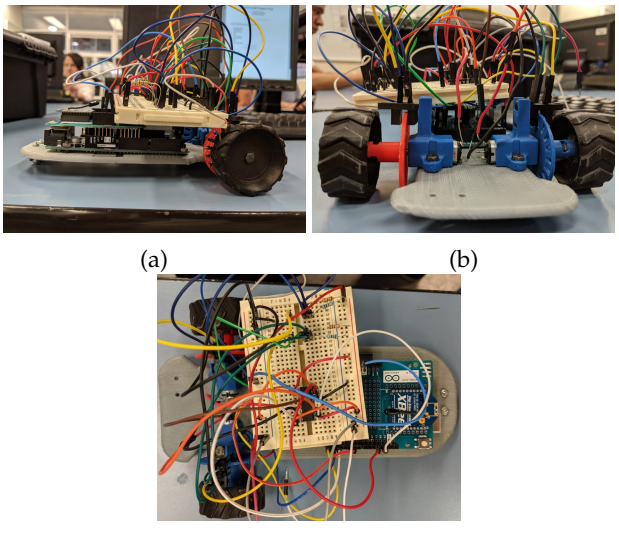
Cuadro 2: Configuración de XBee para el sistema

5. CONCLUSIONES

El uso de tracción diferencial en el robot represento un desafío debido a la necesidad de tener un control con comportamiento similar en ambos motores. Si bien se logró tener un control relativamente uniforme de ambas ruedas, una mayor precisión en la medición de la velocidad del motor ayudaría mucho para tener un mejor manejo. En específico, una mayor resolución en el encoder permitiría bajar el tiempo de muestreo, que en este caso tuvo que ser 250 ms, sin perder precisión en la velocidad, pues en este caso si se intentaba medir la velocidad en intervalos de menos de 250 ms, las lecturas del opto-interruptor solo reportaban valores de 1, 2 y 3, lo cual ocasionaba demasiada varianza en la velocidad real del motor. Esto afectó fuertemente el tiempo de convergencia del control PID al valor deseado, y en ocasiones hacía que el voltaje final oscilara entre valores.

- Miguel González Borja

La combinación de tanto ROS, comunicación inalámbrica y los dos motores fue problemática. Asegurar la comunicación correcta entre todos los puntos y una respuesta correcta de los nodos implicó grandes desafíos. Si cambiamos la arquitectura de una de censado y planeación a una reactiva, podemos tener obstáculos y metas móviles. Si la meta puede ser móvil, entonces también podemos tener una optimización de ruta multiobjetivo. También, si los armazones de las ruedas tuvieran hoyos más finos, el censado de velocidad podría ser más preciso. Con un censado más preciso, el control PID tendría mejor retroalimentación. Con el enfoque reactivo y la mejora en la precisión el robot podría cumplir objetivos de búsqueda en un entorno que evoluciona en el tiempo.



(a) (b)

(c)

Figura 13: Robot Diferencial

- Luis Felipe Landa Lizarralde

La creación del robot representó todo un reto, pues si bien el algoritmo de búsqueda sí encontraba una buena ruta que esquiva los obstáculos no fue sencillo hacer que el robot la siguiera. Esto se debe a que el movimiento del robot no era perfecto, así que cuando le decíamos que siguiera una dirección no siempre se iba en una línea recta. Así, muchas veces el algoritmo no detectaba exactamente cuando el robot ya había pasado por un punto de la ruta. Una mejora al código sería tener un solo nodo de ROS para evitar problemas con la comunicación entre ambos. De igual manera se debe de mejorar la manera en la que el robot siga la ruta para tener más flexibilidad. Este proyecto nos permitió darnos cuenta que pasar de la teoría a la práctica no siempre es fácil pues los componentes físicos casi nunca se comportan de manera ideal. Esto se debe de tener en cuenta al momento de crear los algoritmos para poder manejar fallos de mejor manera.

- Andrea Marín Alarcón

6. ROL

- Miguel González Borja - Diseño e implementación del robot diferencial con controles PID.
- Luis Felipe Landa Lizarralde - Diseño e implementación de algoritmo de búsqueda
- Andrea Marín Alarcón - Diseño del algoritmo de búsqueda y de los nodos de ROS.

7. FUENTES CONSULTADAS

[1] S. M. LaValle, "Planning algorithms," accesed 2019-05-22. [Online]. Available: http://planning.cs.uiuc.edu/node659.html

[2] Sparkfun, "What is an arduino?" accessed 2019-05-22. [Online]. Available: https://learn.sparkfun.com/tutorials/what-is-an-arduino/all

[3] Geekbot Electronics, "Motores de dc," accessed 2019-05-21. [Online]. Available: http://www.geekbotelectronics.com/motores-de-dc/

[4] L. Vélez de Guevara, "Sistemas de control de lazo cerrado," accessed 2019-05-21. [Online]. Available: https://makinandovelez.wordpress.com/2018/02/15/sistemas-de-control-de-lazo-cerrado/

[5] V. Mazzone, "Controladores pid," accessed 2019-04-11. [Online]. Available: http://www.eng.newcastle.edu.au/~jhb519/teaching/caut1/Apuntes/PID.pdf

[6] L. Llamas, "Hacer un encoder óptico con un optointerruptor y arduino," accessed 2019-05-22. [Online]. Available: https://www.luisllamas.es/usar-un-optointerruptor-con-arduino/

[7] ROHM Semiconductor, "What is a photointerrupter?" accessed 2019-05-22. [Online]. Available: https://www.rohm.com/electronics-basics/photointerrupters/what-is-a-photointerrupter

[8] Open Source Robotics Foundation, "About ros," accessed 2019-05-22. [Online]. Available: http://www.ros.org/about-ros/

[9] U. Endriss, "Search techniques for artificial intelligence," accessed 2019-05-22. [Online]. Available: http://www.cs.ubbcluj.ro/~csatol/log_funk/prolog/slides/7-search.pdf
[10] XBee.cl, "¿qué es xbee?" accessed 2019-05-20. [Online]. Available: https://xbee.cl/que-es-xbee/