Reporte del prototipo **SMART**HAWK

Contenido

El Reto

Descripción del Problema Por Resolver

La problemática a resolver propuesta por BMW consiste en un programar un vehículo para que de manera autónoma localice la ubicación de 4 cajas de colores que se encuentran sobre un piso de color blanco con una cuadricula negra, cada caja tiene asignado un número y color distinto que no se repite ni se modifica a lo largo de la competencia (1 es para la caja roja, 2 para la verde, 3 para la azul y 4 para la amarilla) y sólo se pueden colocar en intersecciones.

El robot realizará 2 recorridos, el primero con el objetivo de hacer una exploración del mapa, para que con el segundo se enfoque en resolverlo de la manera más rápida posible ruta más eficiente para cruzar por todas las puertas en el orden establecido.

La cancha:

El reto se realizará en una cancha de 2 x 2 metros, esta se encuentra demarcada por una cuadrícula de 8x8 en la cual cada cuadrado mide 25 x 25 cm. En esta cancha se van a poner 4 cajas (De ahora en adelante llamadas “Gates”) de diferente color en las intersecciones de las líneas. Las líneas de la cuadrícula tienen un grosor de 1 pulgada.

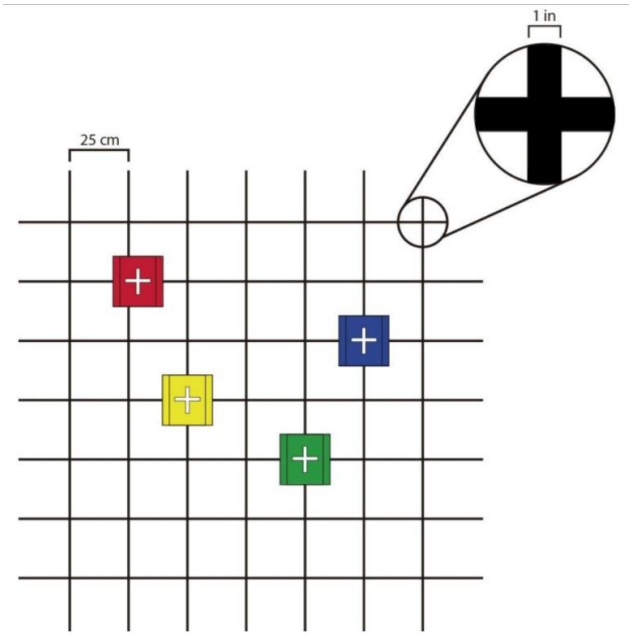


Figura 1. Ejemplo de escenario

Gates

El posicionamiento de cada Gate tiene las siguientes consideraciones:

1.- Las gates sólo pueden colocarse en las orillas de tal forma que permitan que el robot las atraviese, lo que implica que no podrán haber gates en las esquinas. Ver Figura 2.

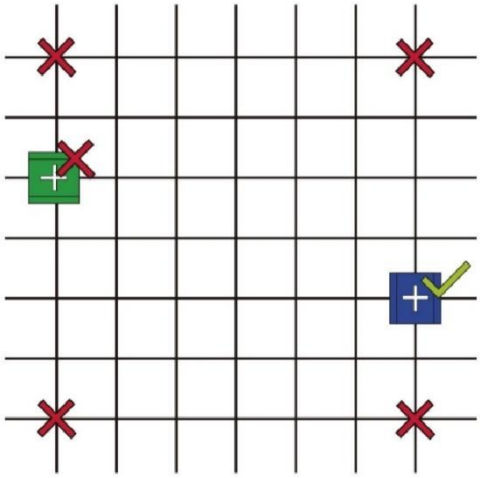


Figura 2

2.- Las gates no pueden ser colocadas en ningunas de las intersecciones adyacentes a alguna otra gate. Siempre se debe dejar al menos una intersección de separación.

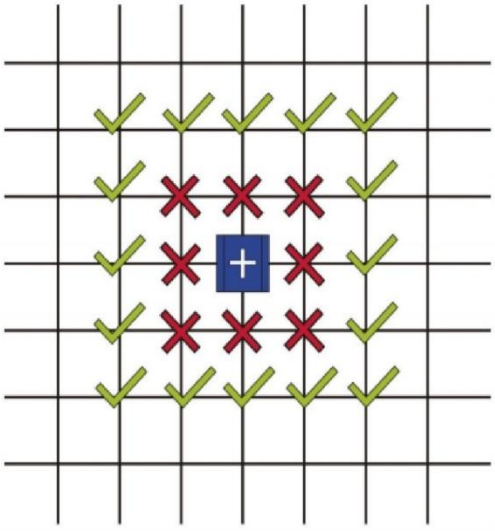


Figura 3

Monitoreo en Tiempo Real

Además de hacer un recorrido en el menor tiempo posible, el vehículo deberá mostrar en tiempo real la información de sus sensores, ya sea en una computadora o un dispositivo portátil.

El Hardware

El vehículo que se utilizará para resolver el reto consta de los siguientes elementos de entrada y salida:

- 2 motores con Encoder
- 1 sensor ultrasónico
- 1 Pi Camera versión 3.5
- 2 sensores de luz y color
- 1 Raspberry Hat con sensor de potencia e IMU

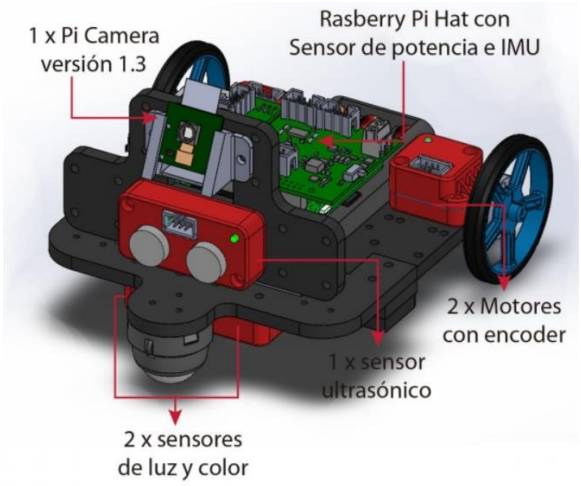


Figura 4

Otras Condiciones.

1.- El Robot va a empezar en una esquina en del mapa.

2.- La orientación inicial puede ser elegida por el equipo.

3.- Al finalizar, el robot se debe detener en la esquina opuesta de donde comenzó.

Nuestro Plan de Ataque

Nuestro acercamiento inicial del reto fue atacarlo como pequeños problemas en lugar de un solo gran problema, es así como decidimos dividirlo en 4 partes:

- **Mapeo:** Consiste en la obtención de los puntos en donde se localizan la entrada y la salida de cada caja.
- **Optimización de Ruta:** En este paso, un programa se encarga de que, con los puntos obtenidos en el mapeo, se encuentre la trayectoria más corta para recorrer en orden las cajas.
- **Navegación:** En el punto anterior el robot resolvió de manera teórica los puntos que debía de seguir en su recorrido, esta parte utiliza esa información y la pone en práctica operando al robot.
- **Desarrollo de aplicación:** Se refiere a la programación de la aplicación que hará el trabajo de mostrar en tiempo real la información recabada por el robot.

Mapeo

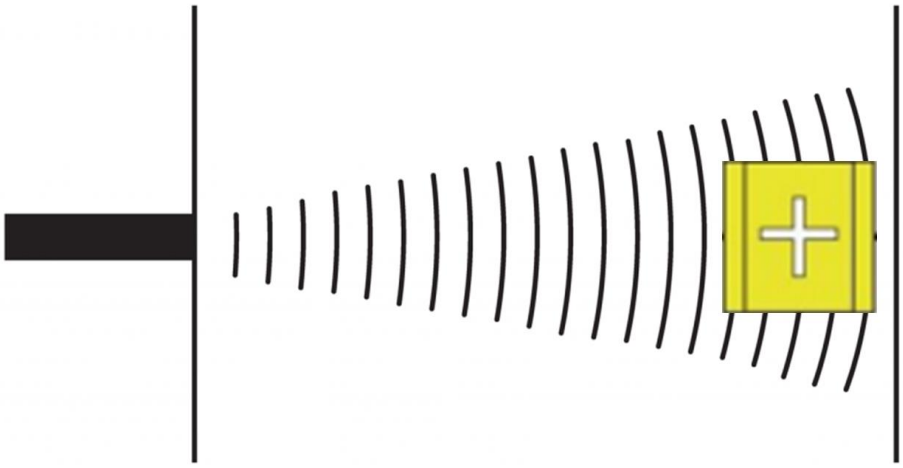
En esta parte del proyecto decidimos apegarnos lo más posible a la filosofía KISS (Keep It simple Stupid), es decir, en lugar de intentar resolver el problema de una manera innecesariamente complicada, encontrar una solución haciendo uso de los sensores y algoritmos más simples, es así como decidimos que en nuestra primer propuesta de solución sólo utilizar los sensores de color ubicados en la parte inferior del robot y el sensor ultrasónico para el mapeo, almacenando la posición actual del robot y de las cajas encontradas, así como la posición de sus entradas en una matriz de 7x7.

Nuestro primer trabajo fue realizar pruebas con los sensores

Pruebas ultrasónico

La propuesta original consistía en que el robot sólo iba a visitar las líneas externas de la cuadrícula y cada que llegara a una intersección giraría para ver el centro y tomar una medida con el ultrasónico, si detectaba un objeto se acercaría hasta estar frente a la caja y entraría para ver el color, posteriormente regresaría al punto de donde hizo la medición y continuaría su recorrido.

Esta idea en un principio nos pareció buena, sin embargo, al momento de hacer mediciones con el sensor ultrasónico nos dimos cuenta de que no funcionaban de una manera tan ideal como lo habíamos planeado, esto debido a que cuando se envía la señal que hará la medición, esta se dispersa en forma de cono y no en una línea recta como lo suponíamos.



Esto nos hizo eliminar la opción de usar el ultrasónico de lejos debido a la cantidad de lecturas erróneas que podíamos tener y decidimos que su uso se limitaría a verificar si había o no un objeto inmediatamente adelante para evitar chocar.

Mapeo usando sensores de color

Nuestra segunda propuesta consistía en utilizar los sensores de color, en este caso, en lugar de determinar la posición de las cajas por sensores de distancia, recorreríamos en línea recta el mapa hasta llegar a la caja, posteriormente, nos moveríamos dentro de la caja y al final regresaríamos a la intersección donde comenzamos.

Debido a que no usábamos el sensor ultrasónico para determinar la posición, usamos los sensores de color para encontrar y contar las intersecciones.

Para esta parte utilizamos el modo “White2” que nos arrojaba un valor en escala de grises en donde 255 era un blanco absoluto y 0 un negro absoluto.

Por medio de mediciones definimos un umbral, cualquier medición arriba de ese umbral se consideraría un color blanco, cualquier valor igual o menor se consideraría como color negro.

Debido a que solo son 2 sensores, en total hay 4 casos que se pueden presentar a la hora de hacer la lectura:

1. Sensor Izquierdo = Blanco; Sensor Derecho Blanco
2. Sensor Izquierdo = Blanco; Sensor Derecho **Negro**
3. Sensor Izquierdo = **Negro**; Sensor Derecho Blanco
4. Sensor Izquierdo = **Negro**; Sensor Derecho **Negro**

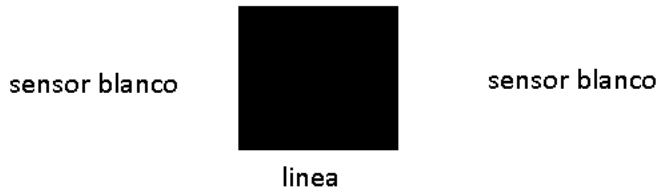


Figura 5.Ejemplo de lo que verían los sensores si el vehículo estuviera centrado

En los casos 2 y 3 podemos interpretar que el robot se esta desviando ligeramente a la izquierda (caso 2) o a la derecha (caso 3), así que la solución seria reducir ligeramente la velocidad de un motor para que se haga la corrección adecuada (Como lo manejamos fue reducir la velocidad de 255 a 200)

Para el caso 1 y 4 podemos concluir que ambos sensores ven el mismo color y por lo tanto el robot debería mantenerse moviendo en línea recta, así que no seria necesario hacer correcciones de

velocidad, sin embargo, si se presenta el caso 4 significa que acaba de pasar sobre una intersección, por lo tanto modificaría en una unidad su posición en la matriz en donde va guardando el mapa, además de esto, activaría una bandera que se mantendría en alto mientras los sensores siguieran viendo color negro y se reiniciaría hasta que ambos sensores detectaran color blanco, esto para evitar múltiples lecturas sobre el mismo cruce. (Ver Apéndice B).

Este método aparentemente nos dio muy buenos resultados al inicio, sin embargo, en algunas ocasiones fallaba y contaba más o menos intersecciones que las que había en realidad.

Haciendo una investigación cuidadosa nos dimos cuenta de que por pequeños que fueran los dobleces de la lona, alteraban las mediciones, dando falsos positivos en algunos puntos.

Debido a esto, se consideraron 2 soluciones, la primera consistía en hacer el conteo de los cruces utilizando el encoder, la segundas utilizando la cámara.

Mapeo del robot con encoders.

A pesar de que el método anterior tenía problemas para contar los cruces, el robot parecía seguir la línea sin mayor problema, así que consideramos dejar la función de seguir la línea intacta y usar el encoder para contar los cruces.

El encoder que se maneja en el carro da 600 pulsos por revolución y el diámetro de la llanta es de 6 cm. Por la fórmula 2*pi*r que es igual al perímetro de la llanta tenemos que:

L=2*Pi*r → 2*Pi*3= 18.84954 cm.

Quiere decir que por cada revolución de la llanta se avanza 18.84 cm. Por regla de tres determinamos los pulsos requeridos para avanzar en la cuadrícula.

18.84954 → 600 pulsos

25 → X pulso

Sabemos que cada cuadro por lado mide 25 cm, es decir que para recorrer la distancia de 25cm se necesitan 795.7753 pulsos aproximadamente. Con esto y la librería de motor_turn determinamos el recorrido del carro sobre la cuadrícula usando los encoders.

Este nuevo conocimiento además nos permitió crear algunas funciones que nos servirían para dar vueltas más precisas de 90° (Ver Apéndice A).

También, al momento de que contáramos que se avanzó hasta llegar a una intersección, se haría una medición de color, con el objetivo de saber si nos encontrábamos sobre una caja. (Ver Apéndice C)

Detección de línea y cruces usando Open CV.

Cómo se mencionó anteriormente, también se probó determinar la corrección de línea recta usando la cámara y OpenCV. Se cambio el sentido de la cámara para que viera hacia la cuadrícula. Al adquirir la imagen, esta se transforma a escalas de grises y se pone un umbral binario ya que solo nos interesa detectar las líneas negras. Para determinar en qué momento el robot debe corregir, se corta la imagen para solo distinguir una línea recta y un cruce como se ejemplifica a continuación.

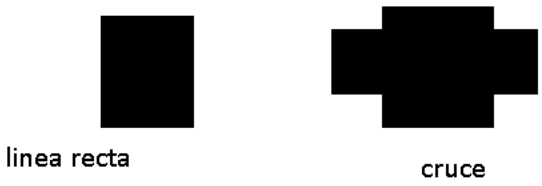


Figura 6 Imagen muestra de línea y cruce usando Open CV

Para hacer la corrección y el conteo de cruces se saca el centroide de la imagen y el área de la figura detectada. Se usó la función findContours de Open Cv, para detectar los contornos en la imagen. Los contornos se pueden ejemplificar como una curva que une todos los puntos continuos (a lo largo del límite), teniendo el mismo color o intensidad.

En dicha función se usan tres parámetros donde el primero es la imagen de origen, el segundo es el modo de recuperación de contorno, el tercero es el método de aproximación de contorno y tiene como salida la imagen, los contornos y la jerarquía, con esto podemos detectar diferentes objetos en una imagen. En ocasiones los objetos están en diferentes lugares, pero en algunos casos, algunas formas están dentro de otras formas. Al igual que las figuras anidadas. En este caso, llamamos a uno externo como “padre” y uno interno como “hijo”. De esta forma, los contornos en una imagen tienen alguna relación entre sí. Y podemos especificar cómo un contorno está conectado entre sí, por ejemplo, si es “hijo” de algún otro contorno, o es un “padre”, etc.

La representación de esta relación se llama Jerarquía.

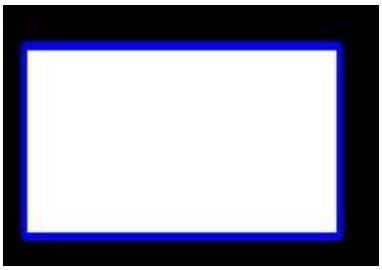


Figura 7. Detección de contorno de un rectángulo.

Teniendo los contornos obtenemos diferentes datos como área, perímetro, centroide, etc. Los momentos de imagen ayudan a calcular algunas características como el centro de masa y el área del objeto, etc.

La función cv2.moments () nos proporciona todos los valores de momento calculados. A partir de estos obtenemos el centroide que viene dado por las relaciones:

$$C_x = \frac{M_{10}}{M_{00}}$$
$$C_y = \frac{M_{01}}{M_{00}}$$

Ecuación 1. Cálculo de centroides del objeto.

Con la función cv2.contourArea() obtenemos el área del objeto. El área de las cruces debe ser mayor que el área de la línea recta. Con estos datos, obtenemos la posición de los cruces en la cuadrícula y los guardamos en la matriz de mapeo. Usando lo sensores piso verificamos que en el cruce no exista una caja de color, y si existe guarda la posición con un multiplicador que llamaremos peso para identificar una caja en la matriz.

Determinamos el error cada que el robot se mueve hacia alguno de los lados haciendo la resta del centroide en el eje Y menos 80 (pixeles de la línea) y del mismo modo se corrige usando las funciones de movimiento del motor.

En ambos métodos se emplea una corrección usando un control proporcional al error.

Detección de las cajas usando Open CV.

Para detectar las cajas, se hace un cambio de espacio de color (inicialmente BGR) a HSV para determinar el rango de valores de color de las cajas. El espacio de color HSV tiene las siguientes componentes.

H - Hue (longitud de onda dominante).

S - Saturación (Pureza / tonos del color).

V - Valor (Intensidad).

Lo que lo hace ideal para trabajar es que usa solo un canal para describir el color (H), por lo que es muy intuitivo especificar el color.

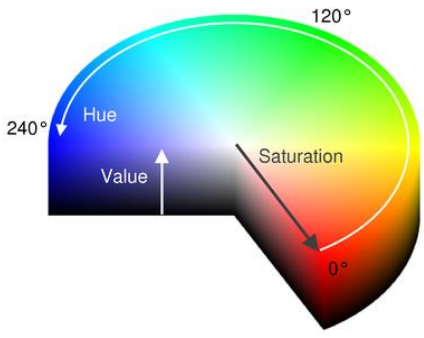


Figura 8. Espacio de color HSV.

En nuestro programa hacemos uso de un trackbar para detectar y calibrar el color de cada caja ya que estos valores varían dependiendo de la luz ambiente. Se usa la siguiente instrucción:

hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

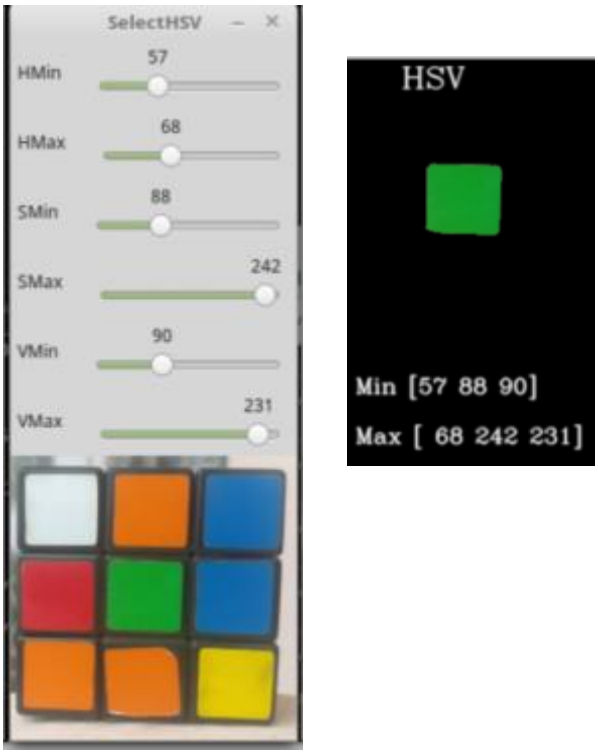


Figura 9. Ejemplo de calibración de color verde usando trackbars en el espacio HSV.

Se guardan los valores máximo y mínimo de H, S y V de cada caja. Al momento de implementar la detección de color de cada caja junto con el sensor ultrasónico que la encontraba y esto pasaba cuando había 5 cm de separación entre sensor y caja, por las sombras que el mismo carro generaba la detección de color no era del todo efectiva por lo cual declinamos por detectar las cajas usando la cámara. (Ver Apéndice D)

Planeación de Ruta usando Ant Colony Optimization Meta-Heuristic

La planeación de ruta fue realizada basada en un algoritmo llamado Ant Colony Optimizaton[1], el cual da solución al problema del viajero, que consiste en visitar un número total de locaciones recorriendo la menor distancia posible; no obstante, el problema que se le trata de dar solución es distinto, ya que, más allá de intentar recorrer un número de locaciones de manera aleatoria, se necesita recorrer los túneles en orden, partiendo de una esquina para llegar a la esquina contraria, cruzando por los túneles rojo, azul, verde y amarillo. Debido a esto, un algoritmo derivado de Ant Colony Optimization es usado, conocido como Ant Colony Optimization Meta-Heuristic[2], el cual consiste en hacer que agentes virtuales conocidos como "Hormigas" realicen la búsqueda de una ruta entre dos puntos por distintas generaciones hasta que una ruta lo suficientemente buena, con base en criterios dados, sea encontrada o que un número *n* de épocas hayan sido usadas para ejecutar dicha búsqueda.

Supóngase un entorno como en la Figura 10a, donde un plano coordenado derecho es asignado. En la Figura 1b se observa el área con los cuatro túneles colocados en distintas posiciones, las posiciones de las entradas libres al túnel se muestran encerradas en círculos con el color perteneciente al túnel.

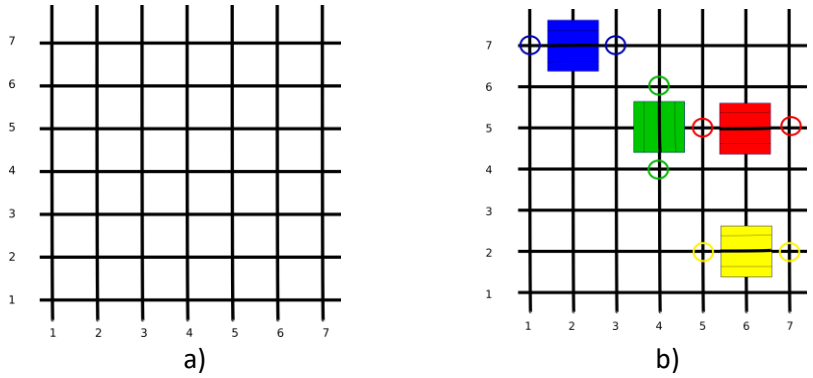


Figura 10. a) Entorno en donde el móvil se desplazará desde las coordenadas (1,1) a (7,7). b) Entorno con los tuneles colocados y encerrados con círculos las entradas de cada tunel con su respectivo color.

[1] M. Dorigo, V. Maniezzo, Colorni A., Ant System: Optimization by a Colony of Cooperating Agents, Cybernetics, Vol. 26, No. 1, 1996.
[2] M.A. Porta Garcia, O. Montiel , O. Castillo, R. Sepulveda ,P. Melin , Path planning for autonomous mobile robot navigation with antcolony optimization and fuzzy function evaluation, Analysis and Design of Intelligent Systems using Soft Computing Techniques SP, Volume 41, pp 790-798, 2009.

Desarrollo de Optimización.

Un objeto de la clase **BMWRoute** debe ser creado, dando como entrada una lista conteniendo las posiciones de cada una de las entradas de cada túnel de manera ordenada. Un ejemplo de lista con base en la Figura 10b sería como se muestra a continuación

Posiciones = [(5,5) , (7,5), (1,7) , (3,7), (4,4) , (4,6), (5,7) , (7,2)]

Una vez que las posiciones han sido dadas, el método BMW.Start() es llamado, comenzando la búsqueda de ruta. Primero, dadas las coordenadas de las entradas a los túneles, es posible encontrar las coordenadas en las cuales cada túnel está colocado, siguiendo la fórmula

Coordenadas de Túnel = [(Punto1.x + Punto2.x) /2 , (Punto1.y + Punto2.y)/2]

Estas coordenadas son tratadas como obstáculos, ya que dichas posiciones no pueden ser alcanzadas a menos que sea el turno de dicho túnel a ser recorrido.

Después de que se calculan las coordenadas de cada túnel, se inicializa la lista de nombre **orderedPoints** con la primera posición siendo la coordenada de partida (1,1), para después ser ingresado la coordenada de túnel siguiente que se encuentra más cercana al inicio, en este ejemplo sería la posición (5,5), y se agrega a la lista **orderedPoints** seguido por la coordenada de la otra entrada del túnel, que es (7,5); después se obtiene la coordenada del siguiente túnel más cercana a la última coordenada agregada a la lista **orderedPoints,** siendo la coordenada (7,5). Este proceso se repite hasta que todas las posiciones han sido ordenadas y al final el punto final (7,7) es agregado a la lista. El resultado una lista del ejemplo actual es:

orederPoints = ((1,1) , (5,5) , (7,5) , (3,7) , (1,7) , (4,6) , (4,4) , (5,2) , (7,2) , (7,7))

Después, una matriz que representa el área mostrada en la Figura 1b es creada, si bien el plano coordenado es de 7,7; la matriz que se crea es de 9,9 debido a que se agregan las fronteras que son prohibidas. A continuación, se muestra una matriz representativa del área

Area = [0 0 0 0 0 0 0 0 0

0 1 1 1 1 1 1 1 0

0 1 1 1 1 1 1 1 0

0 1 1 1 1 1 1 1 0

0 1 1 1 1 1 1 1 0

0 1 1 1 1 1 1 1 0

0 1 1 1 1 1 1 1 0

0 1 1 1 1 1 1 1 0

0 0 0 0 0 0 0 0 0]

Después las posiciones de los túneles son agregadas. Los ceros representan las posiciones que no deben ser alcanzadas por el móvil.

Area = [0 0 0 0 0 0 0 0 0 0

0 1 0 1 1 1 1 1 0

0 1 1 1 1 1 1 1 0

0 1 1 1 0 1 0 1 0

0 1 1 1 1 1 1 1 0

0 1 1 1 1 1 1 1 0

0 1 1 1 1 1 0 1 0

0 1 1 1 1 1 1 1 0

0 0 0 0 0 0 0 0 0]

Una vez que el área es terminada, un objeto tipo **Colony** es inicializado dando como entradas el número de objetos **Ant** a ser creados, el número de épocas a ejecutar el algoritmo, la posición inicial, la posición final, el Área a explorar, constantes de evaporación de feromona, constante para refuerzo de feromona y valor de feromona inicial).

Col = Colony (10, 10, orderedPoints[i], orderedPoints[i+1], Area, 0.001, 1,0.1)

Los puntos que son entregados al objeto **Colony** como inicio y objetivo son en parejas, el primer punto de partida es la esquina en la cual el móvil es colocado al inicio y el primer punto objetivo es la entrada del tunel número uno más cercana; la segunda posición de partida es la entrada del tunel número uno que no ha sido usada y la segunda posición objetivo es la entrada del tunel número dos más cercana al tunel uno y el proceso sigue sucesivamente hasta que la última pareja es conformada por una entrada del tunel número cuatro y las coordenadas 7,7 que son la posición final del recorrido. En el ejemplo, estás parejas estarían dadas por las siguientes parejas

[1 1] [5 5]

[7 5] [3 7]

[1 7] [4 6]

[4 4] [5 2]

[7 2] [7 7]

Dentro de la clase **Colony** n objetos **Ant** son creados e inicializados recibiendo como entrada la posición de donde comenzará la búsqueda y el punto donde la búsqueda termina. Después de esto,

la búsqueda de la posición objetivo comienza. Este proceso es ejemplificado con el área mostrada en la Figura 10b.

La primera ruta por encontrar es desde la posición (1,1) a la posición (5,5). Está búsqueda será realizada por cada uno de los objetos tipo **Ant**. Supóngase que el primero objeto tipo **Ant** comienza su recorrido, éste se encuentra en la posición (1,1); para tomar su siguiente posición un proceso de decisión es realizado de manera probabilística. Cada uno de los objetos tipo **Ant** solo tiene permitido moverse trazando cruces, como se muestra en la Figura 2, los puntos verdes son las posiciones a las cual el objeto **Ant** puede tener acceso.

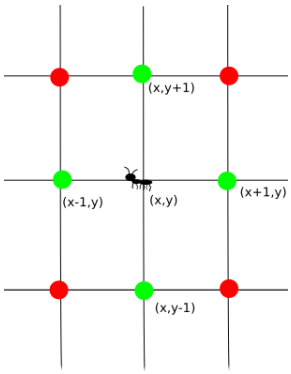


*Figura 11. Posiciones a las cuales el objeto tipo **Ant** puede tener acceso. Cada uno de estos puntos en color verde tienen sus coordenadas con base en la posición del objeto **Ant**.*

Para determinar la siguiente posición, se calcula cuán probable es que alguno de los puntos en el vecindario del objeto **Ant** sea elegido utilizando la siguiente ecuación

$$p_{ij}^k = \begin{cases} \frac{(\tau_{ij}^k)^\alpha (\eta_{ij}^k)^\beta}{\sum_{l \epsilon N_i^k} (\tau_{il}^k)^\alpha (\eta_{il}^k)^\beta}, & \text{si } j \in N_i^k \\ 0, & \text{si } j \notin N_i^k \end{cases}$$

Ecuación 2

donde p^k_{ij} es la probabilidad de que el nodo *j* sea elegido por el objeto k tipo **Ant** encontrado en la posición i, τ_{ij} es el valor de feromona encontrado en el nodo, η_{ij} es el inverso de la distancia entre el nodo j y la posición objetivo, α y *β son constantes de peso, es decir, el factor en el que la feromona o la distancia intervendrá en la probabilidad.* Esta probabilidad solo es calculada para los nodos encontrados en el vecindario del nodo i, que es la posición del objeto **Ant**. Una vez que las probabilidades se tienen, se elige alguna de las posibles siguientes posiciones tomando en cuenta la probabilidad de cada posición.

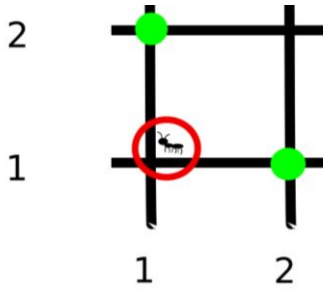


Figura 12. Posibles posiciones partiendo de la posición inicial.

Tomando como ejemplo la Figura 12, solo hay dos posibles posiciones a ser tomadas, los nodos (2,1) y (1,2). Debido a que es la primera iteración, las feromonas conservan el valor con el que se inicializaron, que es 0.1; el valor de α y *β son constantes e iguales a 0.5*. Tomando lo anterior en cuenta y que la posición destino en esta primera parte es (5,5), las probabilidades de cada uno de los puntos son como se muestra a continuación:

$p^{1}_{(1,1)(2,1)}$ = $[0.1^{0.5} * (1/dist((2,1), (5,5)))^{0.5}]$ / $\{[0.1^{0.5} * (1/dist((2,1), (5,5)))^{0.5}] + [0.1^{0.5} * (1/dist((1,2), (5,5)))^{0.5}]\}$ = 0.0548478 / (0.0548478 +0.0548478) = 0.5

$p^{1}_{(1,1)(2,1)}$ = $[0.1^{0.5} * (1/dist((1,2), (5,5)))^{0.5}]$ / $\{[0.1^{0.5} * (1/dist((2,1), (5,5)))^{0.5}] + [0.1^{0.5} * (1/dist((1,2), (5,5)))^{0.5}]\}$ = 0.0548478 / (0.0548478 +0.0548478) = 0.5

Debido a que ambas posiciones son equiprobables, la decisión será hecha de manera aleatoria. Supóngase que la posición (2,1) es elegida, el proceso de cálculo de probabilidades se repite de nuevo con las posiciones a las que se tiene acceso (Figura 13, puntos verdes) y que no hayan sido visitados (Figura 13, punto amarillo).

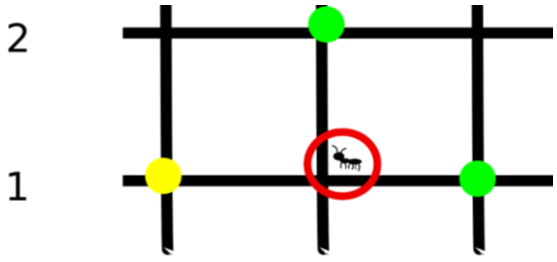


Figura 13. Desde la posición actual (2,1), las posibles siguientes posiciones son (3,1) y (2,2). La posición (1,1) no es una opción ya que ya fue visitada y el algoritmo prohíbe repetir dicha posición.

Este proceso se repite sucesivamente hasta que el objeto de la clase **Ant** llega a la posición deseada; en este momento el objeto **Ant** en turno termina su trabajo y el siguiente comienza al igual que el anterior, buscando una serie de posiciones desde (1,1) hasta (5,5). Una vez que los *n* objetos **Ant** han encontrado la posición deseada, la ruta de menor número de posiciones es elegida para que dos procesos que afectan las feromonas depositadas en el área toman lugar, evaporación y refuerzo. El primero consiste en hacer que la feromona depositada en el área sea reducida un factor ρ en función de la ecuación siguiente:

$$\tau_{ij} \longleftarrow (1-\rho)\tau_{ij}$$

Ecuación 2

Una vez que la evaporación ha sido realizada, el refuerzo entra en acción. El refuerzo de feromona consiste en incrementar la feromona en las posiciones que forman la ruta que es formada por el menor número de posiciones. El refuerzo está dado por la siguiente ecuación

$$\tau_{ij} \longleftarrow \tau_{ij} + \sum_{k=1}^{m} (\Delta\tau_{ij}^{k})$$

Ecuación 3

Donde $\Delta\tau_{ij}$ es el inverso del número de posiciones que componen la ruta encontrada.

Cuando se llega a este punto se dice que una iteración o época ha terminado. Este proceso se repite tantas veces como le haya sido indicado al algoritmo.

Una vez que el número de épocas ha sido agotado, la ruta de menor número de posiciones es almacenada y un nuevo punto de inicio y posición de interés son ingresados al algoritmo, siguiendo el ejemplo mostrado en la Figura 10b, las posiciones de inicio y objetivo serían (7,5) y (3,7) respectivamente.

Una vez que todas las posiciones han sido ingresadas al algoritmo y éste ha encontrado una ruta, todas las rutas de los anteriores recorridos son unidas y regresadas en un vector.

(Ver Apéndice E)

Resultados:

La lista de puntos que es entregada con base en el ejemplo de la Figura 10b es la siguiente

Ruta = [[1, 1], [2, 1], [3, 1], [2, 1], [3, 1], [3, 2], [4, 2], [5, 2], [5, 3], [5, 4], [5, 5], (6.0,5.0), [7, 5], [7,6], [6, 6], [6, 7], [5, 7], [4, 7], [3, 7], (2.0, 7.0), [1, 7], [1, 6], [2, 6], [3, 6], [4, 6], (4.0, 5.0), [4, 4], [4, 3], [5,3], [5, 2], (6.0, 2.0), [7, 2], [7, 3], [7, 4], [7, 5], [7, 6], [7, 7]]

En la Figura 14 se muestra la ruta que el móvil seguiría si la Figura 10b fuera el entorno con el cual se está trabajando.

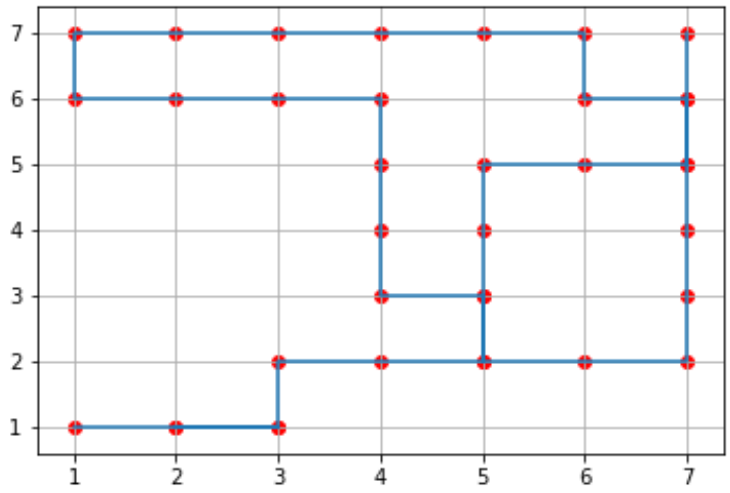


Figura 14. Ruta final entregada por el algoritmo

Interpretación de la ruta planeada a sentencias de control del vehículo

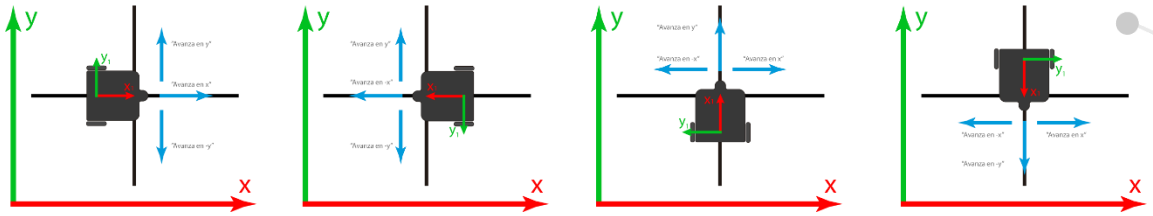
Una vez que la planeación de la ruta fue calculada mediante la implementación de un algoritmo Ant Colony Optimization, se obtiene una lista de puntos, es decir, las coordenadas a las cuales el vehículo tendrá que dirigirse partiendo de una de las esquinas de la cancha para realizar el recorrido en el orden adecuado.

La lista de puntos entregada servirá como una base para realizar los desplazamientos y rotaciones con el vehículo. Para poder interpretar dicha lista de puntos fue necesario identificar todos los posibles casos que podían presentarse, es decir, la orientación en el punto actual (punto en el que se encuentra el vehículo), la orientación en el punto siguiente y los desplazamientos a lo largo del eje x y eje y.

Es importante mencionar las consideraciones que se hicieron para interpretar de puntos entregados por el algoritmo de planeación de ruta a sentencias de control en el vehículo:

- La cancha se restringió a un sistema de coordenadas cartesianas derecho.
- La orientación inicial del vehículo en la esquina, punto (1,1), era hacia el eje x positivo.
- El vehículo sólo podía realizar desplazamientos hacia adelante y rotaciones de 90° tanto en sentido horario como antihorario.

En la Figura 155 se muestran tanto el referencial de la cancha como el del vehículo, las orientaciones que el vehículo podía presentar en la cancha y los avances que se podían realizar dependiendo de la orientación en la que se encontraba el mismo.



a) ori_car = 1 b) ori_car = 2 c) ori_car = 3 d) ori_car = 4

Figura 15 Posibles orientaciones del vehículo dentro de la cancha.

Desarrollo de la navegación

Para identificar el tipo de avance requerido, en la interpretación de puntos del algoritmo de planeación a sentencias de control de movimiento en el vehículo, fue necesario realizar una substracción de el punto de estado siguiente y el punto actual, $avance = pt\_sig - pt\_act$.

Una vez realizada la substracción, era posible identificar el tipo de avance de acuerdo con el valor y el signo de la componente x, y del punto avance, es decir, se podían presentar cuatro casos distintos:

1. Avance en x: $(1\ ,0)$
2. Avance en -x: $(-1\ ,0)$
3. Avance en y: $(0\ ,1)$
4. Avance en -y: $(0\ ,-1)$

Posterior a la identificación del tipo de avance se verificaba la orientación del vehículo para determinar si era necesario rotar 90° (sentido horario o antihorario) y avanzar o sólo avanzar, por lo cual se propuso una máquina de estados (Figura 16) para que una vez realizada una rotación, el registro de orientación cambiara y al registrar un nuevo avance se consideraran los posibles casos anteriormente mostrados en la Figura 155.

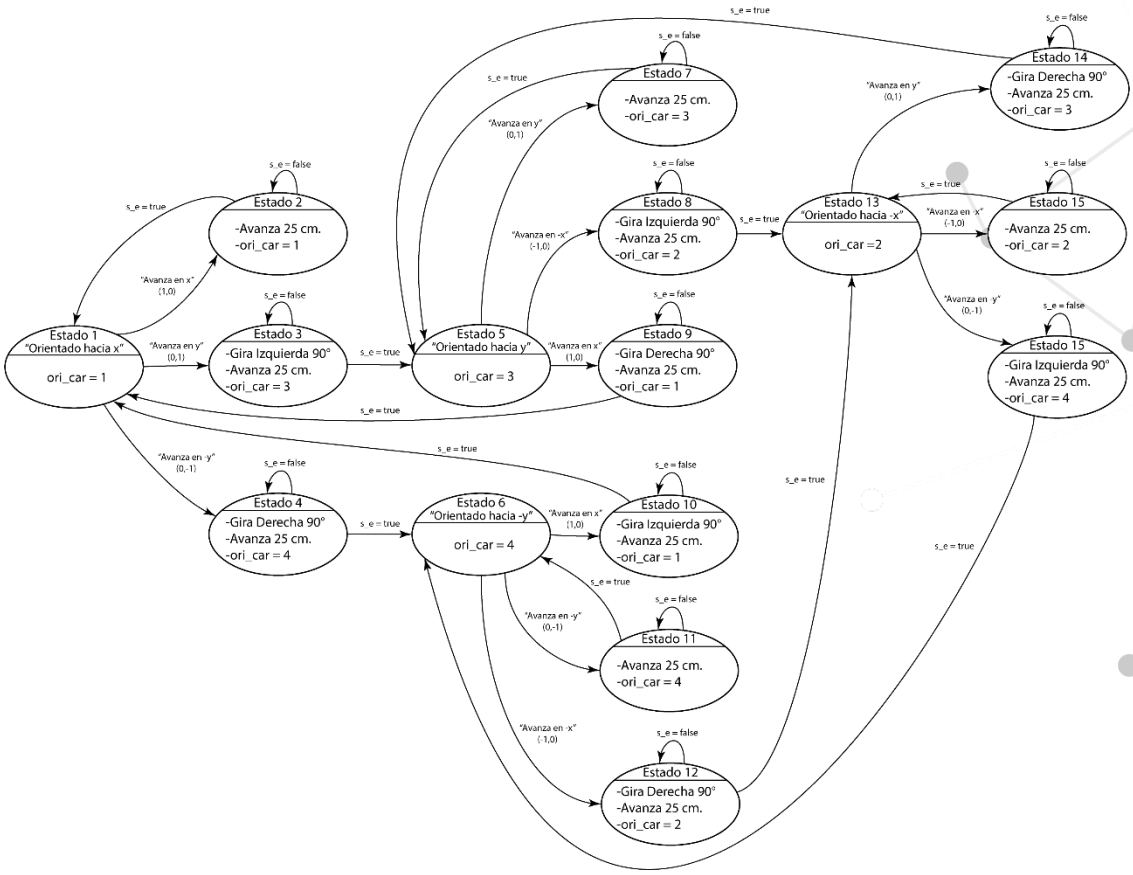


Figura 16 Máquina de estados propuesta para interpretar los puntos de la ruta planeada a sentencias de control del vehículo.

En la Figura 16, *s_e* una bandera lógica que es *false* mientras la función wait() bloquea el programa y espera a que los motores se hayan movido a las posiciones deseadas y *true* cuando el movimiento se ha completado. Para que exista una transición de los estados 1,5,6 y 13 hacia los demás estados es necesario realizar el cálculo del tipo de avance $n-1$ veces, donde n es la cantidad de puntos que componen la lista entregada por el algoritmo de planeación de la ruta.

Se definieron tres funciones que sirven para controlar el avance y rotación del vehículo:

1. avanza_odom(vel, dist): Tiene como argumentos la velocidad de los motores (0 a 255) y la distancia a recorrer, es decir, una conversión de distancia en cm. a recorrer a número de revoluciones necesarias para avanzar dicha distancia.
2. gira_i_odom(vel): Velocidad de los motores (0 a 255) como argumento y es una rutina de giro hacia la izquierda de 90°.
3. gira_d_odom(vel): Velocidad de los motores (0 a 255) como argumento y es una rutina de giro hacia la derecha de 90°.

Considerando la lista de puntos que es entregada por el algoritmo de planeación de ruta con base en el ejemplo de la Figura 10b se tiene que:

Ruta = [[1, 1], [2, 1], [3, 1], [3, 2], [4, 2], [5, 2], [5, 3], [5, 4], [5, 5], (6.0,5.0), [7, 5], [7,6], [6, 6], [6, 7], [5, 7], [4, 7], [3, 7], (2.0, 7.0), [1, 7], [1, 6], [2, 6], [3, 6], [4, 6], (4.0, 5.0), [4, 4], [4, 3], [5,3], [5, 2], (6.0, 2.0), [7, 2], [7, 3], [7, 4], [7, 5], [7, 6], [7, 7]]

Por lo que al interpretar los puntos de la ruta a sentencias de control para el vehículo quedan como se muestra en la

Tabla 1.

No.	Secuencia de control	No.	Secuencia de control
0	Avanza	18	Gira izquierda y avanza
1	Avanza	19	Gira izquierda y avanza
2	Gira izquierda y avanza	20	Avanza
3	Gira derecha y avanza	21	Avanza
4	Avanza	22	Gira derecha y avanza
5	Gira izquierda y avanza	23	Avanza
6	Avanza	24	Avanza
7	Avanza	25	Gira izquierda y avanza
8	Gira derecha y avanza	26	Gira derecha y avanza
9	Avanza	27	Gira izquierda y avanza
10	Gira izquierda y avanza	28	Avanza
11	Gira izquierda y avanza	29	Gira izquierda y avanza
12	Gira derecha y avanza	30	Avanza
13	Gira izquierda y avanza	31	Avanza
14	Avanza	32	Avanza
15	Avanza	33	Avanza
16	Avanza	34	Paro
17	Avanza		

Tabla 1 Secuencia de comandos de control del vehículo.

En la Figura 17 se muestra la ruta por la cual el vehículo se tendría que desplazar y rotar para lograr cumplir el objetivo de cruzar todas las puertas en orden y llegar a la esquina opuesta (7,7) de donde se inició el recorrido. (Ver Apéndice F)

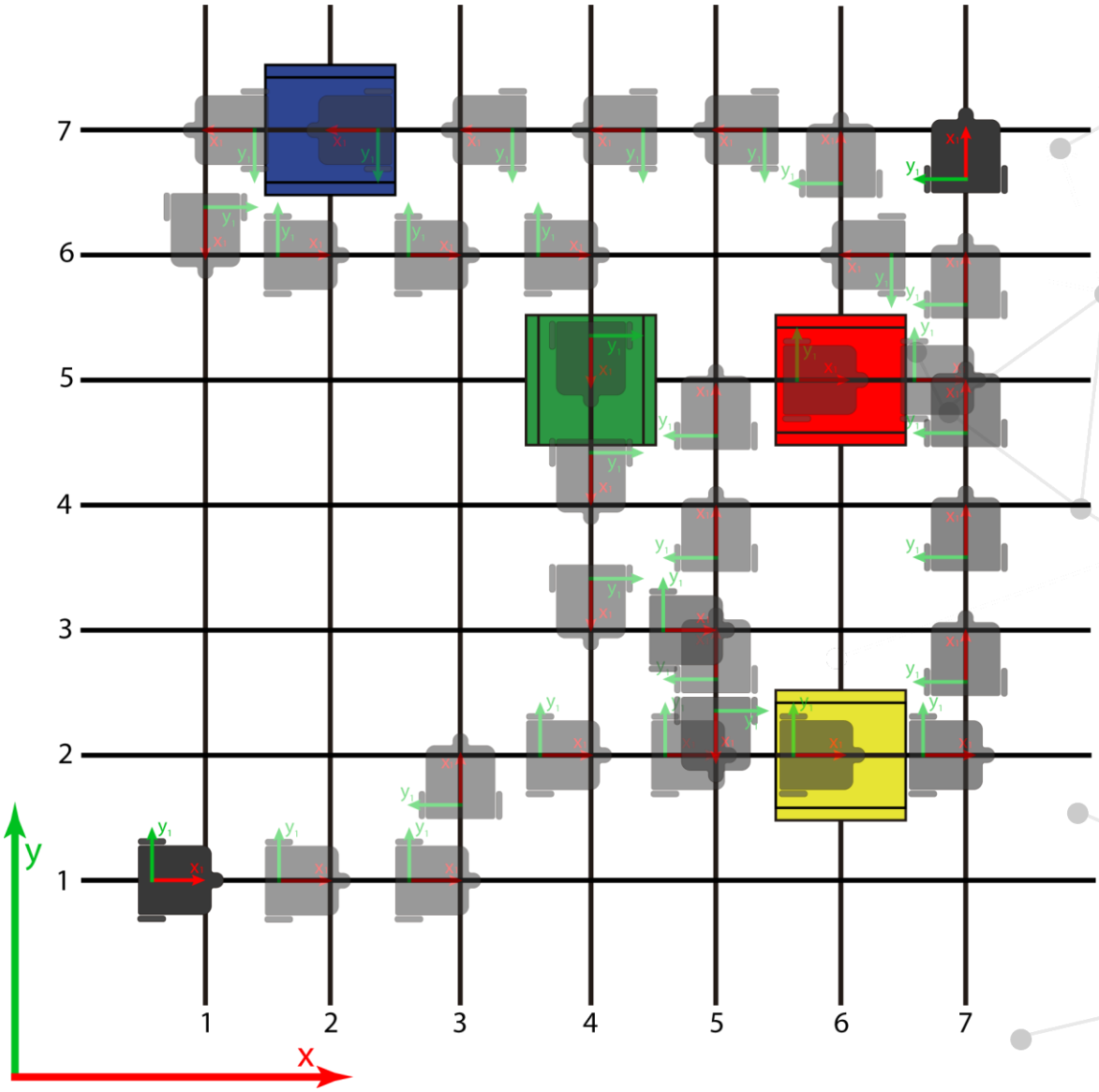


Figura 17 Ruta a realizar con las secuencias de control del vehículo.

Es importante mencionar que para la realización de las pruebas de movimiento a través de la ruta planeada sólo se hizo la consideración de la estimación de la posición con la odometría, es decir, calculando el desplazamiento del vehículo mediante la información obtenida de los encoders en las ruedas. Es posible obtener mejores resultados para estimar la posición fusionando la información obtenida de los encoders con la información obtenida de los sensores de luz y color para detectar cuando se ha llegado a una intersección y corregir la orientación del vehículo con las líneas de la cancha.

Desarrollo de la aplicación

Dentro de la RASPBERRY se creó un servidor el cual se encargaba únicamente de enviar datos, que serían recibidos por el cliente que estaba en una computadora. La computadora únicamente iba a estar encargada de mostrar los datos recibidos del servidor mediante una interfaz gráfica o app.

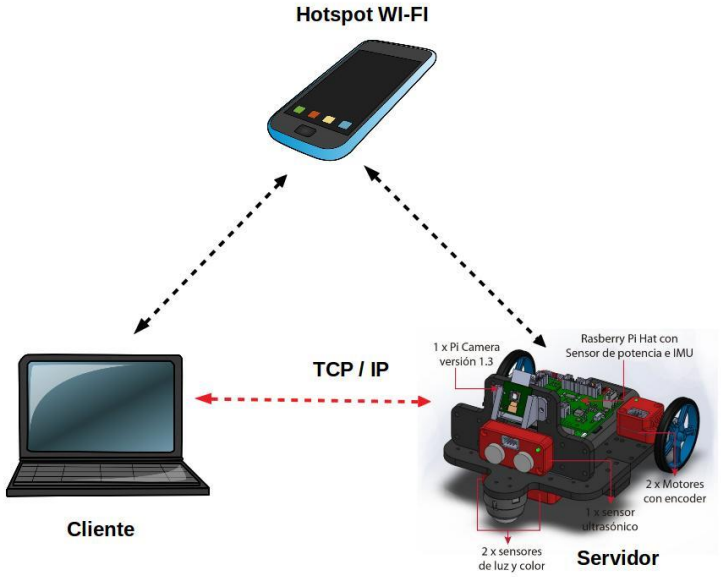


Figura 18

Se muestra una imagen de la interfaz gráfica, en donde se mostrarían datos en tiempo real de los sensores del carro. La interfaz fue hecha en python para una computadora.

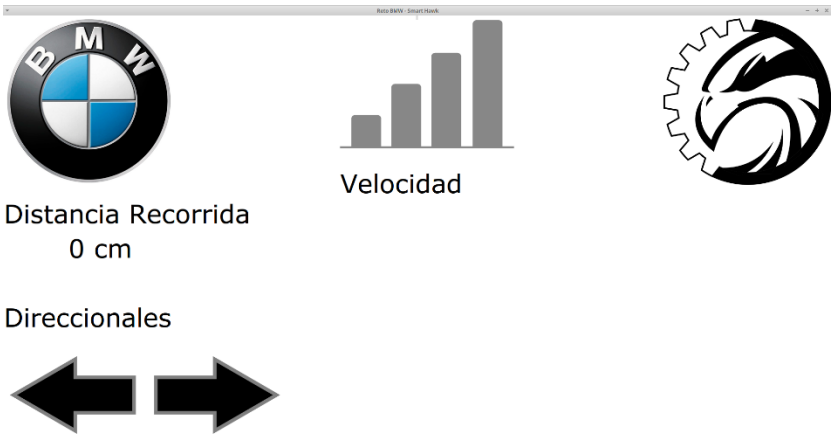


Figura 19

El siguiente diagrama de flujo muestra el funcionamiento lógico de la interfaz, en donde primero se hace una conexión al servidor y se selecciona el puerto por el que se va a comunicar el cliente. Una vez que la comunicación es establecida se recibe el mensaje y es clasificado para asignar los valores correspondientes en el cliente.

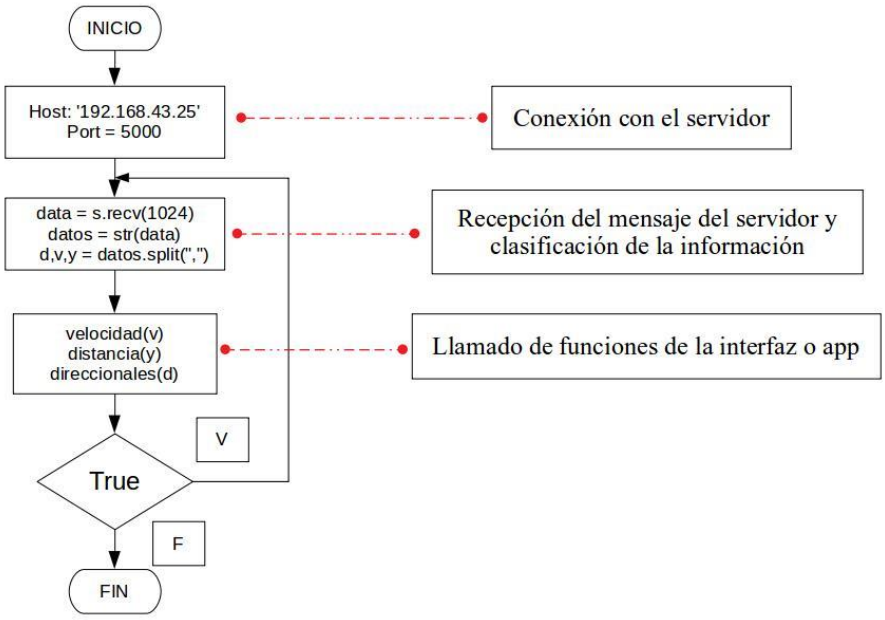


Diagrama 1

El siguiente diagrama de flujo describe la función de distancia, la cual también funciona para estimar la velocidad utilizando los encoders del carro. Se toma lectura de la posición actual del encoder y después se toma la última posición del encoder. Con esto únicamente es necesario realizar una resta para conocer el número de pulsos totales y multiplicarlos por un factor, para conocer la distancia recorrida en ese trayecto.

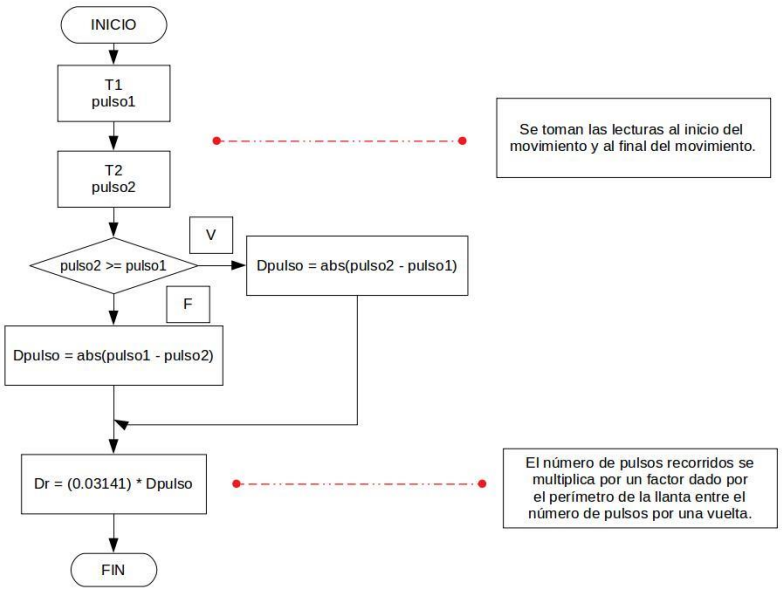


Diagrama 2

Ahora el diagrama de flujo de la función de velocidad toma un valor recibido del servidor. El cual es estimado usando la función de distancia. El valor es comparado para ir mostrando el incremento o decremento de la velocidad en la interfaz gráfica.

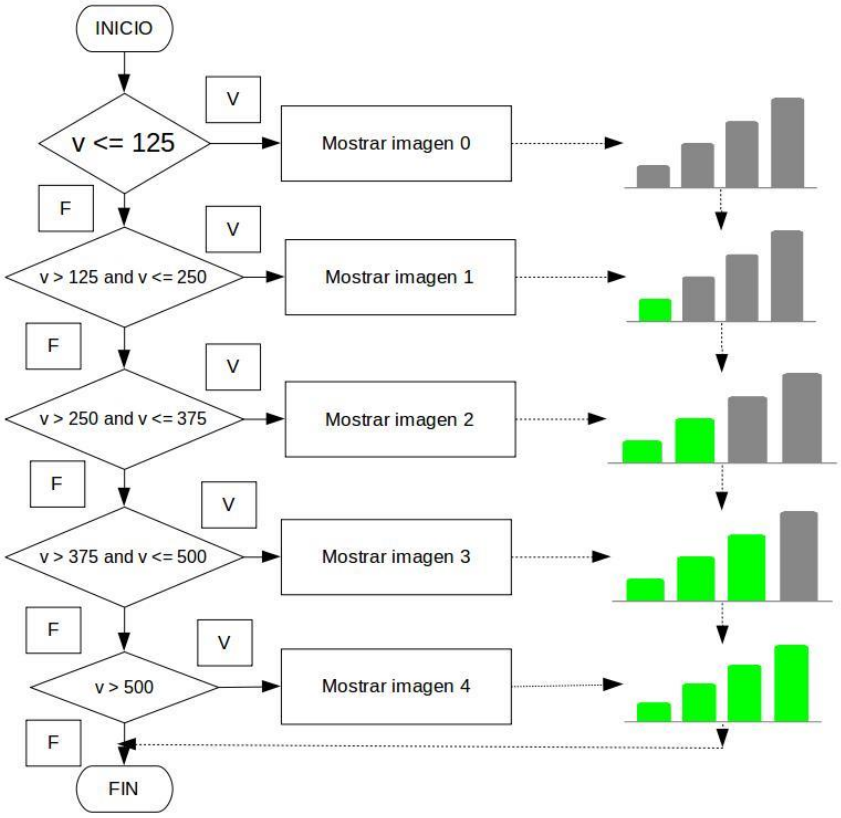


Diagrama 3

CONCLUSIONES

El Reto BMW sin duda es una experiencia que hemos disfrutado bastante y que sin duda nos ha dejado importantes enseñanzas.
Al momento de realizar este documento, como equipo hemos hecho algunas reflexiones:

¿Qué hicimos bien?

- Pudimos actuar frente a los distintos problemas que encontramos en el camino y los abordamos de distintas maneras.
- Nuestro nivel de exigencia con nosotros y nuestros algoritmos garantizó que el robot siempre estuviera trabajando a máxima velocidad, lo que no consiguieron la mayoría de los demás equipos.
- Cada integrante del equipo fue capaz de realizar las tareas que se le encomendaban, además de esto, todos eran proactivos, apoyando en donde se requería sin necesidad de que alguien tuviera que decirlo.
- Todos fueron capaces de admitir aciertos y errores durante el trabajo, permitiendo así que se pudieran corregir las cosas que estaban mal sin dar muchas vueltas.

¿Qué hicimos mal?

- Nos faltó mayor comunicación a lo largo del reto, esto quedo demostrado a la hora de intentar unir el trabajo que se le asignó a cada uno.
- No haberle dado la suficiente importancia a la aplicación y concentrar la mayoría de nuestros recursos solo en el robot.
- Algunas propuestas de solución les dedicamos más tiempo del que debimos, mientras que a otras les hizo falta trabajarlas más.
- Estábamos tan concentrados en intentar mandar los datos en tiempo real vía wifi que no se nos ocurrió intentar hacer una app más simple vía bluetooth

¿Qué podemos mejorar?

- La manera de organizar el trabajo en este tipo de concursos debe ser distinta a cualquier otro tipo de concurso, es decir, en lugar dividirnos para intentar atacar todo el reto de una sola vez, consideramos que debemos ir avanzando poco a poco todos juntos y así ir construyendo una solución sólida que no cueste trabajo unir al final.
- Antes de intentar implementar una solución establecer criterios de éxito y fracaso a lo largo de su desarrollo y pruebas que nos ayuden a decidir si vale la pena seguir explorando ese camino o mejor evitarlo.
- No menospreciar los que parecen problemas “pequeños”
- Siempre hacer un pequeño ejercicio de imaginar una solución que no creamos viable y ver si algo de eso complementa y mejora alguna de nuestras otras propuestas, esto con la intención de evitar la “Visión de Túnel”

En resumen, este reto fue una vivencia sin precedentes para nosotros que esperamos podamos repetir. Damos gracias al equipo de BMW por todas las atenciones con nosotros durante el evento, en especial a nuestro asesor Javier Ríos y a Rubén Ruiz, que siempre mostraron su apoyo a nosotros y a los demás competidores, ¡Esperamos verlos el siguiente año para la revancha!

¡¡¡GRACIAS POR TODO A EL EQUIPO DE BMW Y TALENT LAND!!!

Apéndices

Apéndice A

Rutinas precisas para girar 90°

```
# velg es una constante que se refiere a la velocidad a la que se realizaran los giros
# Siempre se manejó a la máxima velocidad (velg = 255)
def giro(cuadrante): #Giro a la Izquierda
  if cuadrante==1:
    motor1 = motor.motorbk(7)
    motor2 = motor.motorbk(6)
    motor1.move(-velg,70)
    motor2.move(velg,70)
    motor1.wait()
    motor2.wait()
    #time.sleep(5)
    motor1.move(velg,240)
    motor2.move(velg,240)
    motor1.wait()
    motor2.wait()
    #time.sleep(5)
    motor1.move(velg,85)
    motor2.move(-velg,85)
    motor1.wait()
    motor2.wait()
    #time.sleep(2)

def giro2(cuadrante): #Giro a la Derecha
  if cuadrante==1:
    motor1 = motor.motorbk(7)
    motor2 = motor.motorbk(6)
    motor1.move(-velg,70)
    motor2.move(velg,70)
    motor1.wait()
    motor2.wait()
    motor1.move(velg,-240)
    motor2.move(velg,-240)#232
    motor1.wait()
    motor2.wait()
    motor1.move(velg,85)
    motor2.move(-velg,85)
    motor1.wait()
    motor2.wait()
```

Apéndice B

Código prototipo para Mapeo utilizando sensores de piso.

```
#Inicialización
import sys
sys.path.append("/home/pi/bmw/libs")
import motorbk as motor
import colorbk as colorx
import distancebk as distance
import numpy as np
import time

matrix = np.zeros(8,8)
t=0
# Función que asigna un número según el color leído, este número se guardará en la matriz
def colores():
   SI=sensorI.read()
   SD=sensorD.read()
   if (SD==SI):
      if(SD=="red"):
         return (11)
      else:
         if(SD=="green"):
            return (22)
         else:
            if(SD=="yellow"):
               return (44)
            else:
               if(SD=="blue"):
                  return (33)
               else:
                  return 0
# Rutina de Giro a la Izquierda 90°
def giro(cuadrante):
   if cuadrante==1:
      motor1 = motor.motorbk(7)
      motor2 = motor.motorbk(6)
      motor1.move(-255,70)
      motor2.move(255,70)
      motor1.wait()
      motor2.wait()
      #time.sleep(5)
      motor1.move(255,232)
      motor2.move(255,232)
      motor1.wait()
```

```
        motor2.wait()
        #time.sleep(5)
        motor1.move(255,76)
        motor2.move(-255,76)
        motor1.wait()
        motor2.wait()
        #time.sleep(2)
# Inicialización de motores
motorI = motor.motorbk(7)
motorD = motor.motorbk(6)
motorI.set(-255)
motorD.set(255)
# Inicialización de sensores y otras variables
sensorI = colorx.colorbk(1)
sensorD = colorx.colorbk(2)
SI = 0
SD = 0
b=0
x=1
y=1
pos=(x,y)
ultra = distance.distancebk(3)
cruce = 0

umbralDiff = 20 # Umbral que detecta si los sensores ven cosas distintas. Falta hacerlo dinámico
umbralLight = 70 # Umbral que detecta si es blanco o negro. Falta hacerlo dinámico

# Inicia la exploración, si cuenta 6 cruces o encuentra una caja de a menos de 7 cm
# el robot se detiene y regresa.
while((ultra.read()>7) and (y < 7)):
    SI = sensorI.read("white2")
    SD = sensorD.read("white2")
    #print(SI, SD)
    if((abs(SI-SD))>umbralDiff): #
        if((SI-SD)>0):
            motorI.set(-255)
            motorD.set(200)
            #print("Derecha")
        else:
            motorI.set(-200)
            motorD.set(255)
            #print("Izquierda")
    else:
        t+=1
```

```
        if(SI>umbralLight and SD>umbralLight):
            motorI.set(-255)
            motorD.set(255)
            #print("Centro")
            if(b):
                print("Negro ",t)
                t=0
                b=0
        else:
            motorI.set(-255)
            motorD.set(255
            if (b==0):
                print("Cruce")
                print(SI, SD)
                cruce+=1
                y=+1
                b=1
                print("Blanco ",t)
                t=0
                matrix[x][y]=-1
if(y<7):
    matrix[x][y+1]=5

#break
print(cruce)
print (matrix)

giro(1)
giro(1)

motorI.set(-255)
motorD.set(255)

while(y > 1):
    SI = sensorI.read("white2")
    SD = sensorD.read("white2")
    #print(SI, SD)
    if((abs(SI-SD))>umbralDiff): #
        if((SI-SD)>0):
            motorI.set(-255)
            motorD.set(200)
            #print("Derecha")
        else:
            motorI.set(-200)
```

```
            motorD.set(255)
            #print("Izquierda")
        else:
            if(SI>umbralLight and SD>umbralLight): #
                motorI.set(-255)
                motorD.set(255)
                #print("Centro")
                if(b):
                    b=0 # Bandera, se reinicia si detecta blanco
            else:
                motorI.set(-255)
                motorD.set(255) # Checar caso Azul
                if (b==0): # Solo se ejecuta si la bandera no esta activa y cuenta un cruce
                    print("Cruce")
                    print(SI, SD)
                    y=-1
                    b=1 # Bandera, se pone en alto si se detecta negro
                    matrix[x][y]=-1 # Guarda valor en la matriz
giro(1)

motorI.set(0)
motorD.set(0)
```

Apéndice C

Código de Mapeo completo usando encoders para detectar ubicación. Cabe mencionar que el código NO ha sido optimizado, esto debido al tiempo que se disponía, por tal motivo se puede observar que hay 4 funciones muy similares, cuando en realidad pudo haber sido 1 sola.

```
import sys
sys.path.append("/home/pi/bmw/libs")
import motorbk as motor
import colorbk as colorx
import distancebk as distance
import numpy as np
import time

# Función para calibrar sensores antes de comenzar
def Sopas():
   sensor = colorx.colorbk(1)
   sensor2 = colorx.colorbk(2)
   print("start")
   time.sleep(1)
   sensor.set("black")
   sensor2.set("black")
   print("Ponte verga")
   time.sleep(4)
   sensor.set("white")
   sensor2.set("white")
   a=input()
   print("¡INICIA!")

# Inicialización de motores y algunas variables.

motorI = motor.motorbk(7)
motorD = motor.motorbk(6)
encoD = motorD.readkappa()
encoD = encoD
encoDD = encoD
GG = 690
GG2 = 750
GG3 = 460
i=0
tt=0
u=0
matrix = np.zeros((9,9))
```

```
vel=255
velg=vel # Solo se usara si es necesario

# Función de paro momentáneo, nos sirvió para debuggear.
def paro(tuna):
    motorI.set(0)
    motorD.set(0)
    time.sleep(tuna)
    motorI.set(-255)
    motorD.set(255)
# Variables que nos servirían para construir la lista de puntos que se usaran para optimización.
SI = 0
SD = 0
x1a = 0
y1a = 0
x2a = 0
y2a = 0
x3a = 0
y3a = 0
x4a = 0
y4a = 0
x1b = 0
y1b = 0
x2b = 0
y2b = 0
x3b = 0
y3b = 0
x4b = 0
y4b = 0

cajas = 0
# Función que asigna un número según el color leído, este número se guardará en la matriz
def colores():
    global SI, SD,x1,x2,x3,x4,y1,y2,y3,y4,x,y
    SI=sensorI.read()
    SD=sensorD.read()
    if (SD==SI):
        if(SD=="red"):
            cajas+=1
            return (11)
        else:
            if(SD=="green"):
                cajas+=1
                return (22)
```

```
            else:
                if(SD=="yellow"):
                    cajas+=1
                    return (44)
                else:
                    if(SD=="blue"):
                        cajas+=1
                        return (33)
                    else:
                        return (0)
# Función giro Izquierda
def giro(cuadrante):
    if cuadrante==1:
        motor1 = motor.motorbk(7)
        motor2 = motor.motorbk(6)
        motor1.move(-velg,70)
        motor2.move(velg,70)
        motor1.wait()
        motor2.wait()
        #time.sleep(5)
        motor1.move(velg,240)#232
        motor2.move(velg,240)#232
        motor1.wait()
        motor2.wait()
        #time.sleep(5)
        motor1.move(velg,85)#76
        motor2.move(-velg,85)#76
        motor1.wait()
        motor2.wait()
        #time.sleep(2)
# Función giro Derecha
def giro2(cuadrante):
    if cuadrante==1:
        motor1 = motor.motorbk(7)
        motor2 = motor.motorbk(6)
        motor1.move(-velg,70)
        motor2.move(velg,70)
        motor1.wait()
        motor2.wait()
        #time.sleep(5)
        motor1.move(velg,-240)#232
        motor2.move(velg,-240)#232
        motor1.wait()
        motor2.wait()
```

```
        #time.sleep(5)
        motor1.move(velg,85)
        motor2.move(-velg,85)
        motor1.wait()
        motor2.wait()
        #time.sleep(2)

#motorI.set(-255)
#motorD.set(255)

sensorI = colorx.colorbk(1)
sensorD = colorx.colorbk(2)
b=0
x=1
y=1
pos=0




ultra = distance.distancebk(3)

cruce = 0
umbralDiff = 15 #Umbral que detecta si los sensores ven cosas sitintas. Falta hacerlo dinamico
umbralLight = 65 #Umbral que detecta si es blanco o negro. Falta hacerlo dinamico

#time.sleep(1)
td=0
ti=0
ED=0
#1111111111111111111111111111111111111111111111111111111111111111111111111111111111111
# Función que hace mapeo de una sola línea, esta función se repite otras 3 veces,
# Pero con valores distintos para la Matriz, esto porque el robot recorrerá cada uno de los
# 4 lados del escenario usandola
def corre1():
    global
x,y,SI,SD,vel,encoD,ED,GG,cruce,umbralDiff,matrix,encoDD,velg,GG2,GG3,tt,td,ti,ED,pos,x1a,x2a,x
3a,x4a,y1a,y2a,y3a,y4a,x1b,x2b,x3b,x4b,y1b,y2b,y3b,y4b
    motorI.set(0)
    motorD.set(0)
    u=ultra.read()
    motorI.set(ti)
    motorD.set(td)
    while((ultra.read()>8) and (y < 7)):
```

```
SI = sensorI.read("white2")
SD = sensorD.read("white2")
#print(SI, SD)
if((abs(SI-SD))>umbralDiff): #
   if((SI-SD)>0):
      motorI.set(-vel)
      motorD.set(200)
      ti=-255
      td=200
      #print("Derecha")
   else:
      motorI.set(-200)
      motorD.set(255)
      #print("Izquierda")l
      ti=-200
      td=255
else:
   motorI.set(0)
   motorD.set(0)
   encoD = motorD.readkappa()
   ED = encoD-encoDD
   if (ED>=GG):

      encoDD = encoD+(ED-GG)*0 # SERA
      print("Enco D: ",ED-GG)
      cruce+=1
      y+=1
      if(matrix[x][y]==0):
         matrix[x][y]=-1
      #paro(tt)
      motorI.set(0)# COSA
      motorD.set(0)
      i=colores()
      if(i):
         matrix[x][y]=i
         matrix[x][y+1]=i/11
         matrix[x][y-1]=i/11
         if(i==11):
            x1a=x
            x1b=x
            y1a=y+1
            y1b=y-1
         else:
            if(i==22):
```

```
                x2a=x
                x2b=x
                y2a=y+1
                y2b=y-1
            else:
                if(i==33):
                    x3a=x
                    x3b=x
                    y3a=y+1
                    y3b=y-1
                else:
                    x4a=x
                    x4b=x
                    y4a=y+1
                    y4b=y-1


        motorI.set(-255)
        motorD.set(255)
        ti=-255
        td=255
        #*********************************
motorI.set(0)
motorD.set(0)
if(y<7):
    if(ED>(GG*0.8)):
        y+=1
    matrix[x][y+1]=55
    matrix[x-1][y+1]=5
    matrix[x+1][y+1]=5


#break
print(cruce)


giro(1)
giro(1)

encoD = abs(motorD.readkappa())

encoDD = encoD
```

```
#111
while(y > 1):
   SI = sensorI.read("white2")
   SD = sensorD.read("white2")
   #print(SI, SD)
   if((abs(SI-SD))>umbralDiff): #
      if((SI-SD)>0):
         motorI.set(-255)
         motorD.set(200)
         ti=-255
         td=200
         #print("Derecha")
      else:
         motorI.set(-200)
         motorD.set(255)
         #print("Izquierda")
         ti=-200
         td=255
   else:
      motorI.set(0)
      motorD.set(0)
      encoD = abs(motorD.readkappa())
      ED = encoD-encoDD
      if (ED>=GG2):

         encoDD = encoD+(ED-GG)*0 #SERA
         print("Enco D: ",ED-GG)
         cruce-=1
         y-=1
         if(matrix[x][y]==0):
            matrix[x][y]=-1
         #paro(tt)
      motorI.set(-255)
      motorD.set(255)
      ti=-255
      td=255
giro(1)


if(x<7):
   motorI.move(-velg,GG3)
   motorD.move(velg,GG3)
   motorI.wait()
   motorD.wait()
```

```
    x+=1
    i=colores()
    if(i):
      matrix[x][y]=i
      matrix[x+1][y]=i/11
      matrix[x-1][y]=i/11
      motorI.move(-velg,GG3)
      motorD.move(velg,GG3)
      motorI.wait()
      motorD.wait()
      if(i==11):
        x1a=x-1
        x1b=x+1
        y1a=y
        y1b=y
      else:
        if(i==22):
          x2a=x-1
          x2b=x+1
          y2a=y
          y2b=y
        else:
          if(i==33):
            x3a=x-1
            x3b=x+1
            y3a=y
            y3b=y
          else:
            x4a=x-1
            x4b=x+1
            y4a=y
            y4b=y
      x+=1
    giro(1)

  else:
    giro(1)
    motorI.move(-velg,GG3)
    motorD.move(velg,GG3)
    motorI.wait()
    motorD.wait()
    y+=1
    giro(1)
```

```
    encoD = motorD.readkappa()
    encoDD = encoD

    print(matrix)

#1111111111111111111111111111111111111111111111111111111111111111111111111111111
#2222222222222222222222222222222222222222222222222222222222222222222222222222222
def corre2():
    global
x,y,SI,SD,vel,encoD,ED,GG,cruce,umbralDiff,matrix,encoDD,velg,GG2,GG3,tt,td,ti,ED,pos,x1a,x2a,x
3a,x4a,y1a,y2a,y3a,y4a,x1b,x2b,x3b,x4b,y1b,y2b,y3b,y4b
    motorI.set(0)
    motorD.set(0)
    u=ultra.read()
    motorI.set(ti)
    motorD.set(td)
    while((ultra.read()>8) and (x > 1)):
        SI = sensorI.read("white2")
        SD = sensorD.read("white2")
        #print(SI, SD)
        if((abs(SI-SD))>umbralDiff): #
            if((SI-SD)>0):
                motorI.set(-vel)
                motorD.set(200)
                ti=-255
                td=200
                #print("Derecha")
            else:
                motorI.set(-200)
                motorD.set(255)
                #print("Izquierda")l
                ti=-200
                td=255
        else:
            motorI.set(0)
            motorD.set(0)
            encoD = motorD.readkappa()
            ED = encoD-encoDD
            if (ED>=GG):
                print("Enco D: ",ED)
                encoDD = encoD+(ED-GG)*0
                cruce+=1
                x-=1
                if(matrix[x][y]==0):
```

```
            matrix[x][y]=-1
        #paro(tt)
        motorI.set(0)# COSA
        motorD.set(0)
        i=colores()
        if(i):
            matrix[x][y]=i
            matrix[x-1][y]=i/11
            matrix[x+1][y]=i/11
            if(i==11):
                x1a=x-1
                x1b=x+1
                y1a=y
                y1b=y
            else:
                if(i==22):
                    x2a=x-1
                    x2b=x+1
                    y2a=y
                    y2b=y
                else:
                    if(i==33):
                        x3a=x-1
                        x3b=x+1
                        y3a=y
                        y3b=y
                    else:
                        x4a=x-1
                        x4b=x+1
                        y4a=y
                        y4b=y

    motorI.set(-255)
    motorD.set(255)
    ti=-255
    td=255
    #*******************************
motorI.set(0)
motorD.set(0)
if(x>1):
  if(ED>(GG*0.8)):
    x-=1
  matrix[x-1][y]=55
  matrix[x-1][y+1]=5
```

```
    matrix[x-1][y-1]=5


#break
print(cruce)


giro(1)
giro(1)

encoD = abs(motorD.readkappa())

encoDD = encoD

while(x < 7):
    SI = sensorI.read("white2")
    SD = sensorD.read("white2")
    #print(SI, SD)
    if((abs(SI-SD))>umbralDiff): #
        if((SI-SD)>0):
            motorI.set(-255)
            motorD.set(200)
            #print("Derecha")
            ti=-255
            td=200
        else:
            motorI.set(-200)
            motorD.set(255)
            #print("Izquierda")
            ti=-200
            td=255
    else:
        motorI.set(0)
        motorD.set(0)
        encoD = abs(motorD.readkappa())
        ED = encoD-encoDD
        if (ED>=GG2):
            print("Enco D: ",ED)
            encoDD = encoD+(ED-GG)*0
            cruce-=1
            x+=1
            if(matrix[x][y]==0):
                matrix[x][y]=-1
            #paro(tt)
```

```
        motorI.set(-255)
        motorD.set(255)
        ti=-255
        td=255
giro(1)


if(y<7):
    motorI.move(-velg,GG3)
    motorD.move(velg,GG3)
    motorI.wait()
    motorD.wait()
    y+=1
    i=colores()
    if(i):
        matrix[x][y]=i
        matrix[x1][y+1]=i/11
        matrix[x1][y-1]=i/11
        motorI.move(-velg,GG3)
        motorD.move(velg,GG3)
        motorI.wait()
        motorD.wait()
        if(i==11):
            x1a=x
            x1b=x
            y1a=y+1
            y1b=y-1
        else:
            if(i==22):
                x2a=x
                x2b=x
                y2a=y+1
                y2b=y-1
            else:
                if(i==33):
                    x3a=x
                    x3b=x
                    y3a=y+1
                    y3b=y-1
                else:
                    x4a=x
                    x4b=x
                    y4a=y+1
                    y4b=y-1
```

```
        y+=1
      giro(1)

    else:
      giro(1)
      motorI.move(-velg,GG3)
      motorD.move(velg,GG3)
      motorI.wait()
      motorD.wait()
      x-=1
      giro(1)

    encoD = motorD.readkappa()
    encoDD = encoD
    print(matrix)

#22222222222222222222222222222222222222222222222222222222222222222222222222222222
#33333333333333333333333333333333333333333333333333333333333333333333333333333333
def corre3():
    global
x,y,SI,SD,vel,encoD,ED,GG,cruce,umbralDiff,matrix,encoDD,velg,GG2,GG3,tt,td,ti,ED,pos,x1a,x2a,x
3a,x4a,y1a,y2a,y3a,y4a,x1b,x2b,x3b,x4b,y1b,y2b,y3b,y4b
    motorI.set(0)
    motorD.set(0)
    u=ultra.read()
    motorI.set(ti)
    motorD.set(td)
    while((ultra.read()>8) and (y > 1)):
      SI = sensorI.read("white2")
      SD = sensorD.read("white2")
      #print(SI, SD)
      if((abs(SI-SD))>umbralDiff): #
        if((SI-SD)>0):
          motorI.set(-vel)
          motorD.set(200)
          #print("Derecha")
          ti=-255
          td=200
        else:
          motorI.set(-200)
          motorD.set(255)
          #print("Izquierda")l
          ti=-200
          td=255
```

```
else:
    motorI.set(0)
    motorD.set(0)
    encoD = motorD.readkappa()
    ED = encoD-encoDD
    if (ED>=GG):
        print("Enco D: ",ED)
        encoDD = encoD+(ED-GG)*0
        cruce+=1
        y-=1
        if(matrix[x][y]==0):
            matrix[x][y]=-1
        #paro(tt)
        i=colores()
        if(i):
            matrix[x][y]=i
            matrix[x][y+1]=i/11
            matrix[x][y-1]=i/11
            if(i==11):
                x1a=x
                x1b=x
                y1a=y+1
                y1b=y-1
            else:
                if(i==22):
                    x2a=x
                    x2b=x
                    y2a=y+1
                    y2b=y-1
                else:
                    if(i==33):
                        x3a=x
                        x3b=x
                        y3a=y+1
                        y3b=y-1
                    else:
                        x4a=x
                        x4b=x
                        y4a=y+1
                        y4b=y-1
    motorI.set(-255)
    motorD.set(255)
    ti=-255
    td=255
```

```
    #*******************************
motorI.set(0)
motorD.set(0)
if(y>1):
  if(ED>(GG*0.8)):
    y-=1
  matrix[x][y-1]=55
  matrix[x-1][y-1]=5
  matrix[x+1][y-1]=5


#break
print(cruce)


giro(1)
giro(1)

encoD = abs(motorD.readkappa())

encoDD = encoD


#111
while(y > 1):
  SI = sensorI.read("white2")
  SD = sensorD.read("white2")
  #print(SI, SD)
  if((abs(SI-SD))>umbralDiff): #
    if((SI-SD)>0):
      motorI.set(-255)
      motorD.set(200)
      #print("Derecha")
      ti=-255
      td=200
    else:
      motorI.set(-200)
      motorD.set(255)
      #print("Izquierda")
      ti=-200
      td=255
  else:
    motorI.set(0)
    motorD.set(0)
```

```
        encoD = abs(motorD.readkappa())
        ED = encoD-encoDD
        if (ED>=GG2):
            print("Enco D: ",ED)
            encoDD = encoD+(ED-GG)*0
            cruce-=1
            y-=1
            if(matrix[x][y]==0):
                matrix[x][y]=-1
            #paro(tt)
        motorI.set(-255)
        motorD.set(255)
        ti=-255
        td=255
giro(1)

if(x>1):
    motorI.move(-velg,GG3)
    motorD.move(velg,GG3)
    motorI.wait()
    motorD.wait()
    x-=1
    i=colores()
    if(i):
        matrix[x][y]=i
        matrix[x+1][y]=i/11
        matrix[x-1][y]=i/11
        motorI.move(-velg,GG3)
        motorD.move(velg,GG3)
        motorI.wait()
        motorD.wait()
        if(i==11):
            x1a=x-1
            x1b=x+1
            y1a=y
            y1b=y
        else:
            if(i==22):
                x2a=x-1
                x2b=x+1
                y2a=y
                y2b=y
            else:
                if(i==33):
```

```
                x3a=x-1
                x3b=x+1
                y3a=y
                y3b=y
            else:
                x4a=x-1
                x4b=x+1
                y4a=y
                y4b=y
        x-=1
    giro(1)

  else:
    giro(1)
    motorI.move(-velg,GG3)
    motorD.move(velg,GG3)
    motorI.wait()
    motorD.wait()
    y-=1
    giro(1)

  encoD = motorD.readkappa()
  encoDD = encoD

  print(matrix)

#333333333333333333333333333333333333333333333333333333333333333333333333333333
#444444444444444444444444444444444444444444444444444444444444444444444444444444
def corre4():
  global
x,y,SI,SD,vel,encoD,ED,GG,cruce,umbralDiff,matrix,encoDD,velg,GG2,GG3,tt,td,ti,ED,pos,x1a,x2a,x
3a,x4a,y1a,y2a,y3a,y4a,x1b,x2b,x3b,x4b,y1b,y2b,y3b,y4b
  motorI.set(0)
  motorD.set(0)
  u=ultra.read()
  motorI.set(ti)
  motorD.set(td)
  while((ultra.read()>8) and (x < 7)):
    SI = sensorI.read("white2")
    SD = sensorD.read("white2")
    #print(SI, SD)
    if((abs(SI-SD))>umbralDiff): #
      if((SI-SD)>0):
        motorI.set(-vel)
```

```
            motorD.set(200)
            #print("Derecha")
            ti=-255
            td=200
        else:
            motorI.set(-200)
            motorD.set(255)
            #print("Izquierda")l
            ti=-200
            td=255
    else:
        motorI.set(0)
        motorD.set(0)
        encoD = motorD.readkappa()
        ED = encoD-encoDD
        if (ED>=GG):
            print("Enco D: ",ED)
            encoDD = encoD+(ED-GG)*0
            cruce+=1
            x+=1
            if(matrix[x][y]==0):
                matrix[x][y]=-1
            #paro(tt)
            i=colores()
            if(i):
                matrix[x][y]=i
                matrix[x+1][y]=i/11
                matrix[x-1][y]=i/11

                if(i==11):
                    x1a=x-1
                    x1b=x+1
                    y1a=y
                    y1b=y
                else:
                    if(i==22):
                        x2a=x-1
                        x2b=x+1
                        y2a=y
                        y2b=y
                    else:
                        if(i==33):
                            x3a=x-1
                            x3b=x+1
```

```
                    y3a=y
                    y3b=y
                else:
                    x4a=x-1
                    x4b=x+1
                    y4a=y
                    y4b=y
        motorI.set(-255)
        motorD.set(255)
        ti=-255
        td=255
        #*******************************
motorI.set(0)
motorD.set(0)
if(x<7):
    if(ED>(GG*0.8)):
        x+=1
    matrix[x+1][y]=55
    matrix[x+1][y+1]=5
    matrix[x+1][y-1]=5


#break
print(cruce)


giro(1)
giro(1)

encoD = abs(motorD.readkappa())

encoDD = encoD


#111
while(x > 1):
    SI = sensorI.read("white2")
    SD = sensorD.read("white2")
    #print(SI, SD)
    if((abs(SI-SD))>umbralDiff): #
        if((SI-SD)>0):
            motorI.set(-255)
            motorD.set(200)
            #print("Derecha")
```

```
            ti=-255
            td=200
        else:
            motorI.set(-200)
            motorD.set(255)
            #print("Izquierda")
            ti=-200
            td=255
    else:
        motorI.set(0)
        motorD.set(0)
        encoD = abs(motorD.readkappa())
        ED = encoD-encoDD
        if (ED>=GG2):
            print("Enco D: ",ED)
            encoDD = encoD+(ED-GG)*0
            cruce-=1
            x-=1
            if(matrix[x][y]==0):
                matrix[x][y]=-1
            #paro(tt)
        motorI.set(-255)
        motorD.set(255)
        ti=-255
        td=255
giro(1)

if(y>1):
    motorI.move(-velg,GG3)
    motorD.move(velg,GG3)
    motorI.wait()
    motorD.wait()
    giro(1)
    y-=1
else:
    giro(1)
    motorI.move(-velg,GG3)
    motorD.move(velg,GG3)
    motorI.wait()
    motorD.wait()
    y-=1
    i=colores()
    if(i):
        matrix[x][y]=i
```

```
        matrix[x][y+1]=i/11
        matrix[x][y-1]=i/11
        motorI.move(-velg,GG3)
        motorD.move(velg,GG3)
        motorI.wait()
        motorD.wait()
        if(i==11):
          x1a=x
          x1b=x
          y1a=y+1
          y1b=y-1
        else:
          if(i==22):
            x2a=x
            x2b=x
            y2a=y+1
            y2b=y-1
          else:
            if(i==33):
              x3a=x
              x3b=x
              y3a=y+1
              y3b=y-1
            else:
              x4a=x
              x4b=x
              y4a=y+1
              y4b=y-1
        x+=1
    giro(1)


  encoD = motorD.readkappa()
  encoDD = encoD

  print(matrix)

#4444444444444444444444444444444444444444444444444444444444444444444444444444444
Sopas()
t=0
while(t<7):
  corre1()
  t+=1
t=0
```

```
while(t<7):
   corre2()
   t+=1
t=0
while(t<7):
   corre3()
   t+=1
t=0
while(t<7):
   corre4()
   t+=1

motorI.set(0)
motorD.set(0)
#3

pos=[(np.array((x1a,y1a)))]
pos=[(np.array((x1b,y1b)))]
pos=[(np.array((x2a,y2a)))]
pos=[(np.array((x2b,y2b)))]
pos=[(np.array((x3a,y3a)))]
pos=[(np.array((x3b,y3b)))]
pos=[(np.array((x4a,y4a)))]
pos=[(np.array((x4b,y4b)))]
```

Apéndice D

Código implementado

```
# importa los paquetes a usar
from picamera.array import PiRGBArray
from picamera import PiCamera
import time
import cv2
 # inicializa la cámara y sus parámetros
camera = PiCamera()
camera.resolution = (640, 480) #resolución
camera.framerate = 32
rawCapture = PiRGBArray(camera, size=(640, 480))
 # tiempo de inicialización
time.sleep(0.1)
 # captura de los frames
for frame in camera.capture_continuous(rawCapture, format="bgr", use_video_port=True):
image = frame.array
#cambio a escala de grises y umbral binario
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
#hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)  espacio HSV
thresh = cv2.threshold(gray, 60, 255, cv2.THRESH_BINARY)[1]
#se corta la imagen
thresh = thresh[70:170, 440:540]
# encuentra los contornos de la imágen
cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
for c in cnts:
# encuentra el centroide
M = cv2.moments(c)
cX = int(M["m10"] / M["m00"])
cY = int(M["m01"] / M["m00"])
# dibuja el centroide en la imagen
cv2.circle(image, (cX, cY), 7, (255, 255, 255), -1)
# cálculo del área
area = cv2.contourArea(c)
# muestra el frame
cv2.imshow("Frame", image)
key = cv2.waitKey(1) & 0xFF
# limpia para recibir el siguiente frame
rawCapture.truncate(0)
# salir en caso de que la letra q se apriete
if key == ord("q"):
break
```

Apéndice E

Código de Optimización utilizando Algoritmo de colonia de hormigas

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Apr  4 19:07:57 2018

@author: jose
"""

import numpy as np

class Ant(object):        #class ant, every instance of the class is a virtual ant whose task is to explore
# an area where he is placed
    AreaF = np.zeros((9,9)) #The area where the Feromone is placed
    alpha = 0.5           #constant weights to be used when a new position is taken by an Ant
    beta = 0.5
    def __init__(self, goalPos, homePos, actPos, TabuArea):
        self.GoalPos = goalPos  #The position where the Ant has to reach
        self.HomePos = homePos  #The positio where the Ant start searching fo the GoalPosition
        self.ActPos = actPos       #np.Array, the Present location of the Ant in the given Area
        self.Journey = [[homePos[0], homePos[1]]]   #the list of locations the Ant has visited so far
        self.TabuArea = TabuArea    #The Area where the forbidden locations for the Ant are placed
        self.FixedTabuArea = TabuArea #The area where every Ant instace will exlore, including the
# forbidden places the ant can visit
        self.Memory = [[0,0]]      #Each Ant is given a Memory of six places, e.g. the Ant will remember
ten past position it has visited
    def getGoalPos(self):      #Methods to retrieve information about the Ant instance
        return self.GoalPos
    def getHomePos(self):
        return self.HomePos
    def getActPos(self):
        return self.ActPos
    def getJourney(self):
        return self.Journey
    def getJourneyLen(self):
        return len(self.Journey)
    def getTabuArea(self):
        return self.TabuArea
    def getFixedTabuArea(self):
        return self.FixedTabuArea
    def getMemory(self):
```

```
        return self.Memory

    def getAreaF(self):    #Methods to retrieve information about the Ant class
        return Ant.AreaF


    def Explore(self):     #Method to start the exploration
        #before starting the exploration, remember to initialize Ant.AreaF
        newPos = self.NextPos(self.ActPos, self.GoalPos, self.TabuArea) #The method NextPos() is called, retrieving the next position the ant is exploring based on the actual one
        self.Journey.append([newPos[0][0], newPos[0][1]]) #The new position is added in the Journey # list
        self.Memory.append([newPos[0][0], newPos[0][1]])          #The new position is added as well to the memory list
        self.ActPos = np.array((newPos[0][0],newPos[0][1]))       #The Actual position is updated with the next position coordinates
        while(self.dist(self.ActPos, self.GoalPos) != 0.001):       #If the new position is the Goal position, the search is over, otherwise, the Ant starts exploring until this condition is achieved
            self.updateTabuArea()                   #The method updateTabuArea() is called to place the new positions where the Ant cannot explore anymore
            newPos = self.NextPos(self.ActPos, self.GoalPos, self.TabuArea) #The method NextPos() is called, retrieving the next position the ant is visiting
            self.Journey.append([newPos[0][0], newPos[0][1]])     #The ner position is added to the Journey list
            if len(self.Memory)< 6:                 #Since the ant shall only remember six position it has visited, once the seventh position is reached, the Ant "forgets"  the fist position in the Memory list and adds the next position visited
                self.Memory.append([newPos[0][0], newPos[0][1]])
            else:
                self.Memory.reverse()
                self.Memory.pop()
                self.Memory.reverse()
            self.ActPos = np.array((newPos[0][0],newPos[0][1]))  #The actual position is updated


    def updateTabuArea(self):                   #Method used to update the positions the Ant cannot visit
        self.TabuArea = self.FixedTabuArea          #First, the TabuArea is reseted
        for i in range(0,len(self.Memory)):
            self.TabuArea[self.Memory[i][0]][self.Memory[i][1]] #After the TabuArea is as in the beggining of the exploration, the posotions saved in the Ant memory are now forbidden to it, e.g. the ant cannot visit the positions it has already visited
```

```
  def initializeAreaF(self,row,col, feroInit):      #Method used to start the Area of Feromones, every position has a feromone density and it has to be initialized with a value
    Area = np.zeros((row,col))
    for i in range(0,row):
      for j in range(0,col):
        if (i> 0) and (i < row-1 ) and (j > 0) and (j < col-1):
          Area[i][j] = feroInit* self.FixedTabuArea[i][j]
    Ant.AreaF = Area


  def NextPos(self, actual, goal, area):          #Method which retrieves the next position the Ant is taking based on the Actual position and the Feromone Map
    operations = np.array(((0,1),(1,0),(0,-1),(-1,0))) #This operations denote that the ant cannot move in diagonals
    probabilities = self.getProb(actual, goal, area, operations) #Every position the ant can reach from its actual position has a probability to be chosen based on the feromone the position has and how near the position is to the Goal Position
    Choice =np.random.choice([0, 1, 2, 3], 1, p = probabilities) #After the probabilities are given, one of the positions is chosen
    return operations[Choice] + actual        #The next position is returned

  def getProb(self,actual,goal, area, operations):     #actual has to be an np.array, this mehotd returns the probabilities of being chosen from every possible position the Ant can reach from its actual Position
    probs = {0:0, 1:0, 2:0, 3:0}           #a dictionary is initialized, the positions 0, 1, 2, 3 have at the beggining zero probability of being chosen
    summ = 0                       #The total addition is initialized to zero
    for i in probs:                #loop in the dictionary
      coord = operations[i]+actual        #get the coordinates of the position whose probability is being calculated
      val = ((Ant.AreaF[coord[0]][coord[1]]**Ant.alpha)*(1/(self.dist(coord,goal))**Ant.beta))*area[coord[0]][coord[1]] #This value is the (Feromone in the position)^(alpha, a weight value) * (1/euxclidean distance from the position to the Goal Position)^(beta, a weight value) * (the value of the area (1: position free of being visited, 0: forbidden position, it cannot be visited
      probs[i] = val               #the value inside the dictionary os updated
      summ += val                  #the summatory of the total result is updated
    probabilities = []             #the probabilities list is initialized
    for i in probs:                #loop in the Dictionary
      probabilities.append(  probs[i]/summ  )     #the probabilities of every possible position is calculated and stored inside the Dictionary
    if sum(probabilities) <0.9:            #If the summatory of the probabilities is less than 0.99, it meands the ant has reached a dead end, in which case another Method is called to overcome such situation
```

```
        probabilities = self.getProbTabuless(actual, goal, operations)
    return probabilities

  def getProbTabuless(self,actual,goal, operations):    #actual has to be an np.array, this method is called when the Ant has gotten to a dead end. The Tabu area is ignored and the probabilities are calculated as in the Method getProb()
    probs = {0:0, 1:0, 2:0, 3:0}
    summ = 0
    for i in probs:
      coord = operations[i]+actual
      val = ((Ant.AreaF[coord[0]][coord[1]]**Ant.alpha)*(1/(self.dist(coord,goal))**Ant.beta))
      probs[i] = val
      summ += val
    probabilities = []
    for i in probs:
      probabilities.append( probs[i]/summ )
    return probabilities


  def dist(self, actual, goal):   #Method which returns the euclidean distance from an actual point to a goal point
    d = ((actual[0]-goal [0])**2 + (actual[1]-goal[1])**2)**0.5
    if d == 0:          #if distance is zero, it means that the actual and goal position are the same, however, returning zero causes that the program gets a NaN (division by zero), so, a very very small value is returned
      d = 0.001
    return d

  def UpdateFeromone(self, SmallestJourney, rho, delta):  #SmallestJourney is an np.array, the feromone in the area goes through a process where it is "evaporated" in every position and "reinforced" in specific positions
    Ant.AreaF *=(1-rho)    #the evaporation of the feromone is given by this line, implying that the feromone is being decreased a rho factor
    for i in range(0, len(SmallestJourney)):    #after the feromone is evaporated, the feromone in the positions of the list SmallestJourney are reinforced by adding (1/the length of the list SmallestJourney)
      Ant.AreaF[SmallestJourney[i][0]][SmallestJourney[i][0]]                                    =
(Ant.AreaF[SmallestJourney[i][0]][SmallestJourney[i][0]])                                    +
(Ant.AreaF[SmallestJourney[i][0]][SmallestJourney[i][0]])*(delta/len(SmallestJourney))


class Colony(object):       #class Colony, the Colony will be formed by n Ants and will be responsible of the Feromone update and the smallest journey
```

```
    def __init__(self, numberOfAnts, Epochs, StartingPoint, EndingPoint, ObstacleArea, rho, delta,InitialFeromone):
        self.AntsNumber = numberOfAnts #int, The number of Ants the class will be formed by
        self.EPOCHS = Epochs          #int, Number of epochs the Colony will explore the Area
        self.StartPos = StartingPoint   #2 dimentional np.array, the position where every Ant in the Colony will start searching
        self.EndPos = EndingPoint       # 2 dimentional np.array, the final position every Ant in the Colony has to reach
        self.Area = ObstacleArea          #Row x Col dimentional np.array, the Area containing the position where no Ant of the Colony can visit
        self.Rho = rho              #float smaller than 1, the feromone evaporation factor
        self.Delta = delta           #float, the update feromone factor of the 1/(len of journey) value
        self.InitialPheromone = InitialFeromone #float, the Feromone Area is initialized with a constant value
    def SearchRoute(self):        #Method to start the Area exploration
        row, col = self.Area.shape #The dimentions of the Area are taken
        Perez = []     #the Colony instance is initialized
        PapaPerez = Ant(self.EndPos, self.StartPos, self.StartPos, self.Area) #the Colony Watcher is initialized, this Watcher is not used to explore, but to update the Feromone and Tabu Area
        PapaPerez.initializeAreaF(row,col,self.InitialPheromone)    #The watcher is used to initialize the Feromone Area, e.g. every Ant instance will share the same Feromone Area
        MinJourney = 0 #The smallest journey variable is initialized
        MinJourneyLen = 255*255        #the length of the smallest journey is initialized with a ridiculously big number
        for m in range(0,self.EPOCHS):  #the Search will take place EPOCHS times
            for i in range(0,self.AntsNumber):  #The Colony instance is initialized with AntsNumber members, and every Ant instance inside the Colony instance is initialized as well
                Perez.append(Ant(self.EndPos, self.StartPos, self.StartPos, self.Area))

            for i in range(0,self.AntsNumber): #every Ant inside the Colony initialize the Area with the initial Pheromone value and starts the exploration
                Perez[i].initializeAreaF(row,col,self.InitialPheromone)
                Perez[i].Explore()
            for i in range(0,self.AntsNumber): #Once every Ant inside the Colony finishes exploring, the smallest journey (the journey with the less positions visited) is found
                if Perez[i].getJourneyLen() < MinJourneyLen :
                    MinJourneyLen = Perez[i].getJourneyLen()
                    MinJourney = Perez[i].getJourney()
            #for i in range(0,numberOfAnts):
            #    print(Perez[i].getJourneyLen())
            Perez.clear()  #The Ants inside the Class are erased
            PapaPerez.UpdateFeromone(MinJourney,self.Rho, self.Delta)  #the Colony watcher is used to update the Pheromone in the Area that every Ant instance shares
```

```
        return MinJourney   #Once the EPOCHS are finished, the smallest Journey found within the EPOCHS is returned


'''
row = 9  #row and col have to be bigger by two units of the actual area
col = 9
Area = np.zeros((row,col))
for i in range(0,row):
   for j in range(0,col):
      if (i> 0) and (i < row-1 ) and (j > 0) and (j < col-1):
         Area[i][j] = 1
Area[2][2] = 0

Col = Colony(10, 10, np.array((1,1)), np.array((7,7)), Area, 0.001, 1,0.1)
m = Col.SearchRoute()
'''


class BMWRoute(object):    #BMW object, this object is used so that the Colony adn the Ant classes are initialized and used for the BMW challenge
   def __init__(self, listOfPoints):
      self.Points = listOfPoints      #list of 2 dimentional np.arrays, this points represent both of the tunnel entrances, every four points represent one tunnel

   def Start(self):       #Method used to start the route search after the points are given
      obst = []      #List of obstacles initialized
      for i in range(0, len(self.Points)):    #this loop is used to get the actual positions of the tunnels, to treat them as obstacles in the rout search
         px = (self.Points[i][0][0] + self.Points[i][1][0])/2
         py = (self.Points[i][0][1] + self.Points[i][1][1])/2
         obst.append((px,py))

      start = np.array((1,1))    #The initial point assumed to be [1,1] in the map
      d1 = 0              #two distances variables, d1 and d2, are initialized
      d2 = 0
      orderedPoints = [start]     #The first point to be inside the orderedPoints list is the start point
      for i in range(0, len(self.Points)): #This loop is intended to do 1 task, order the points so that the next point of the tunnel is the nearest to the last one
         d1 = self.distance(self.Points[i][0], start)
         d2 = self.distance(self.Points[i][1], start)
         if min(d1,d2) == self.distance(self.Points[i][0], start):
            orderedPoints.append(self.Points[i][0])
            orderedPoints.append(self.Points[i][1])
```

```
            start = self.Points[i][1]
        elif min(d1,d2) == self.distance(self.Points[i][1], start):
            orderedPoints.append(self.Points[i][1])
            orderedPoints.append(self.Points[i][0])
            start = self.Points[i][0]
        else:
            orderedPoints.append(self.Points[i][0])
            orderedPoints.append(self.Points[i][1])
            start = self.Points[i][1]
    orderedPoints.append(np.array((7,7)))  #The Final Goal Point is stored insite the orderedPoints
list


    row = 9  #row and col have to be bigger by two units of the actual area
    col = 9
    Area = np.zeros((row,col)) #The area is initialized as a zero array, and then just the out boundary
is preserved as zeros, the rest of the positions are set to a value of 1
    for i in range(0,row):
        for j in range(0,col):
            if (i> 0) and (i < row-1 ) and (j > 0) and (j < col-1):
                Area[i][j] = 1

    for i in range(0, len(obst)):   #after the area is initialized, the positions where the tunnels are
placed inside the matrix are set to a value of zero, meaning this position is an obstacle
        Area[int(obst[i][0])][int(obst[i][1])] = 0
        #print(i)
    FinalRoute = []  #The list which will contain the positions which form the route to be followed
    count = 0
    for i in range(0, len(orderedPoints)-1,2):  #in this loop the initial point and the goal point are
given to the Colony class so that the Ant instances find a way.
        #print(orderedPoints[i])              #the way these points are given to the program are as
follows: orderedPoints[i] = Start Position;  orderedPoints[i+1] = Goal position
        #print(orderedPoints[i+1])
        print(orderedPoints[i],orderedPoints[i+1])
        if self.distance(orderedPoints[i], orderedPoints[i+1]) == 0:  #if the starting point and the goal
point are the same, this point is added to the list and the nearest tunnel as well
            FinalRoute.append(orderedPoints[i])
            FinalRoute.append(obst[int((i+1)/2)])
            count +=1
            continue
        else:
```

```
            Col = Colony(10, 10, orderedPoints[i], orderedPoints[i+1], Area, 0.001, 1,0.1) #if the start and goal positions are different, they are given to the Colony class instance, the area where the Ant instanecs will explore, the initial pheromone value and constants
            m = Col.SearchRoute() #then the route of this search is added to the FinalRoute list
            for i in m:
                FinalRoute.append(i)
            #FinalRoute.append(m)

            if count < 4:
                FinalRoute.append(obst[count])    #the obstacle/tunnel in turn is added
            count +=1

        return orderedPoints, Area, FinalRoute

    def distance(self, a, b):      #method to calculate the euclidean distance between two points
        d = ((a[0]-b[0])**2 + (a[1]-b[1])**2)**0.5
        return d
```

Apéndice F

Código de la Navegación

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Apr  4 22:00:00 2018

@author: Rene
"""
import sys
sys.path.append("/home/pi/bmw/libs")
import motorbk as motor
import time

perimeter=18.85
db_p=25
vel=150
n=0
#Declaration of the motors
motorI = motor.motorbk(7)
motorD = motor.motorbk(6)

Route=[[1, 1], [2, 1], [3, 1], [3, 2], [4, 2], [5, 2], [5, 3], [5, 4], [5, 5], (6.0,5.0), [7, 5], [7,6], [6, 6], [6, 7], [5, 7], [4, 7], [3, 7], (2.0, 7.0), [1, 7], [1, 6], [2, 6], [3, 6], [4, 6], (4.0, 5.0), [4, 4], [4, 3], [5,3], [5, 2], (6.0, 2.0), [7, 2], [7, 3], [7, 4], [7, 5], [7, 6], [7, 7]]

act_state=[(0,0)]
nxt_state=[(0,0)]
calc_state=[(0,0)]
n_points=len(Route)-1
bx=1
by=1
ori_car=0
#count=0

def avanza_odom(v,dist): #Move forward n centimeters
    turns=dist/perimeter
    motorI.turn(-v,turns)
    motorD.turn(v,turns)
    motorI.wait()
    motorD.wait()

def gira_d_odom(v): #Rotate 90 degrees cw
```

```
    #Secuence of rotation with compensation x and y
    motorI.move(-v,70)
    motorD.move(v,70)
    motorI.wait()
    motorD.wait()
    #time.sleep(5)
    motorI.move(-v,240)#232
    motorD.move(-v,240)#232
    motorI.wait()
    motorD.wait()
    #time.sleep(5)
    motorI.move(-v,85)#76
    motorD.move(v,85)#76
    motorI.wait()
    motorD.wait()
    #time.sleep(2)

def gira_i_odom(v): #Rotate 90 degrees ccw
    #Secuence of rotation with compensation x and y
    motorI.move(v,70)
    motorD.move(-v,70)
    motorI.wait()
    motorD.wait()
    #time.sleep(5)
    motorI.move(v,240)
    motorD.move(v,240)
    motorI.wait()
    motorD.wait()
    #time.sleep(5)
    motorI.move(v,85)
    motorD.move(-v,85)
    motorI.wait()
    motorD.wait()
    #time.sleep(2)

while(n<n_points): #While the route has not been completed
    if(n==0):
        ori_car=1
    flag=False
    act_state=Route[n]
    nxt_state=Route[n+1]
    calc_state=[(nxt_state[0]-act_state[0]),(nxt_state[1]-act_state[1])]

    if(calc_state[0]==1 and calc_state[1]==0):#Move along axis x
```

```
    if(ori_car==1 and flag==False):
        avanza_odom(vel,db_p)
        #print(count,'Avanza')
        #count+=1
        ori_car=1
        flag=True
    elif(ori_car==3 and flag==False):
        gira_d_odom(vel)
        avanza_odom(vel,db_p)
        #print(count,'Gira derecha y avanza')
        #count+=1
        ori_car=1
        flag=True
    elif(ori_car==4 and flag==False):
        gira_i_odom(vel)
        avanza_odom(vel,db_p)
        #print(count,'Gira izquierda y avanza')
        #count+=1
        ori_car=1
        flag=True
    n+=1

elif(calc_state[0]==-1 and calc_state[1]==0):#Move along axis -x
    if(ori_car==2 and flag==False):
        avanza_odom(vel,db_p)
        #print(count,'Avanza')
        #count+=1
        ori_car=2
        flag=True
    elif(ori_car==3 and flag==False):
        gira_i_odom(vel)
        avanza_odom(vel,db_p)
        #print(count,'Gira izquierda y avanza')
        #count+=1
        ori_car=2
        flag=True
    elif(ori_car==4 and flag==False):
        gira_d_odom(vel)
        avanza_odom(vel,db_p)
        #print(count,'Gira derecha y avanza')
        #count+=1
        ori_car=2
        flag=True
    n+=1
```

```
elif(calc_state[0]==0 and calc_state[1]==1):#Move along axis y
    if(ori_car==1 and flag==False):
        gira_i_odom(vel)
        avanza_odom(vel,db_p)
        #print(count,'Gira izquierda y avanza')
        #count+=1
        ori_car=3
        flag=True
    elif(ori_car==2 and flag==False):
        gira_d_odom(vel)
        avanza_odom(vel,db_p)
        #print(count,'Gira derecha y avanza')
        #count+=1
        ori_car=3
        flag=True
    elif(ori_car==3 and flag==False):
        avanza_odom(vel,db_p)
        #print(count,'Avanza')
        #count+=1
        ori_car=3
        flag=True
    n+=1

elif(calc_state[0]==0 and calc_state[1]==-1):#Move along axis -y
    if(ori_car==1 and flag==False):
        gira_d_odom(vel)
        avanza_odom(vel,db_p)
        #print(count,'Gira derecha y avanza')
        #count+=1
        ori_car=4
        flag=True
    elif(ori_car==2 and flag==False):
        gira_i_odom(vel)
        avanza_odom(vel,db_p)
        #print(count,'Gira izquierda y avanza')
        #count+=1
        ori_car=4
        flag=True
    elif(ori_car==4 and flag==False):
        avanza_odom(vel,db_p)
        #print(count,'Avanza')
        #count+=1
        ori_car=4
```

```
            flag=True
        n+=1
    else:
    #Apaga los motores
    motorI.set(0)
    motorD.set(0)
        #print('Deten los motores')
        n+=1

print('Se completaron todos los puntos')
ori_car=0

#Apaga los motores
motorI.set(0)
motorD.set(0)
```