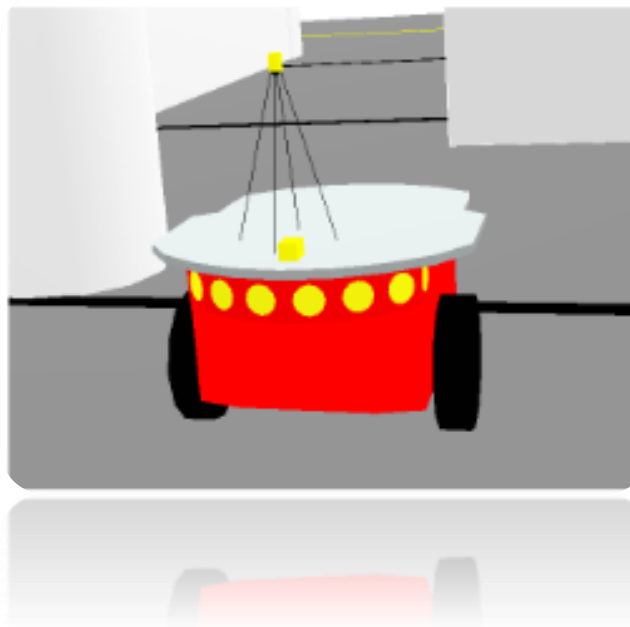




Universidad
Carlos III de Madrid










CONTROLADOR RESCUE PARA ROBOT DE RESCATE PIONEER 2

Javier Galán Méndez, Robótica

Departamento de Ingeniería de Sistemas y Automática— Grado en Electrónica Industrial y Automática

ÍNDICE

	1.Introducción	3
	2.Robot	3
	2.1. Sensores de distancia.....	3
	2.2. Brújula.....	3
	2.3.GPS.....	3
	2.3.Encoder	4
	2.4.Cámara Frontal y Cámara Esférica	4
	3.Entorno	5
	4.Controlador	6
	4.1. Métodos generales.....	6
	4.1. Go	8
	4.2.Search.....	10
	4.2.1.Camera_process().....	11
	4.3.Back.....	11
	5.Alternativas de proceso	12
	7.Conclusión y propuestas	15
	7.1.Conculsión	15
	7.2.Propuestas.....	15
	8.Bibliografía	16

1.INTRODUCCIÓN

El proyecto ha realizar tiene como objetivo la búsqueda y localización de dos posibles objetivos considerados como personas en un mapa concreto con 10 versiones distintas del mismo. El robot utilizado en esta práctica es el Pioneer2, robot que incorpora dos motores y una serie de sensores que le permitirán viajar por el mapa hasta llegar a su destino esquivando obstáculos, encontrar los objetivos y volver al punto de inicio del recorrido.

2.ROBOT

Como se ha comentado antes, el robot se trata del Pioneer 2. Este robot dispone de dos motores controlados por velocidad y una serie de sensores disponible para la realización de la tarea que le corresponda. Estos sensores se detallarán en sus siguientes apartados.



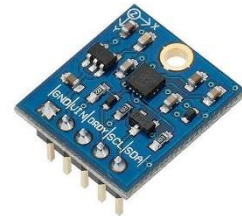
2.1. SENSORES DE DISTANCIA.

Son sensores ultrasonidos que contiene el robot alrededor de su estructura, estos dispositivos se encargarán de obtener información cercana del entorno del robot y así poder esquivar obstáculos. Cada sensor está colocado en una orientación y posición con respecto al robot para poder darle información de en que lado del robot se podría encontrar un obstáculo, pared o muro. Estos sensores en la práctica tienen una serie de desventajas las cuales principalmente son el ruido y la obtención de la señal de otro sensor equivalente por rebote del sonido en los obstáculos. Durante la práctica se han usado estos sensores para detectar y esquivar objetos simples como podría ser muros, cilindros o cubos. Estos sensores se pueden usar con la librería webots/DistanceSensor.



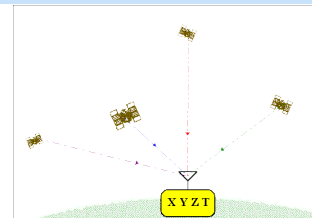
2.2. BRÚJULA

La brújula es un sensor electrónico que nos indica la orientación con respecto al norte. Su principio se basa en un magnetómetro que cambia su valor en base al campo magnético con respecto a la dirección. Este sensor da valores de fuerza y dirección del campo magnético de la Tierra. Estos valores pasan por un conversor analógico-digital y almacenados en una CPU e interpretados para obtener información sobre la orientación. En la práctica se utilizará para corregir la dirección del robot en torno a un ángulo específico permitiendo corregir errores causados por los motores. Este sensor se puede encontrar en la librería webots/Compass.



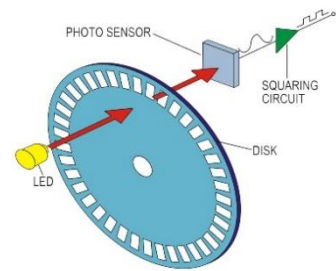
2.3.GPS

El GPS es un sensor basado en geolocalización usando tres satélites para triangular y otro extra para el error producido por el movimiento de la tierra. Con esto obtendremos las posiciones X,Y,Z para geolocalizar a nuestro robot y permitir que pueda ir a cierto punto buscando distintas rutas para llegar a dicho objetivo. Este sensor es usado en la práctica para poder dirigir al robot hacia un punto concreto del mapa y obtener la posición concreta del robot en el propio mapa. Este sensor se puede encontrar en la librería webots/GPS.



2.3. ENCODER

Este sensor es utilizado en odometría para obtener la posición del robot en torno a un punto de origen. El principio de este sensor es el uso de una plantilla taladrada con agujeros en que pasa una luz producida por un led y esta es recibida por un fotodiodo. La plantilla gira y deja pasar o no estos rayos, dando lugar a una serie de pulsos interpretados como un código binario en que son analizados para obtener la posición de las ruedas en radianes. A partir de estos pulsos codificados, es posible con las expresiones matemáticas mostradas a continuación conocer la posición y orientación del robot en torno a un punto origen:



$$p = f(x, y, \theta, \Delta s_r, \Delta s_l) = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} + \begin{bmatrix} \frac{\Delta s_r + \Delta s_l}{2} \cdot \cos\left(\theta + \frac{\Delta s_r - \Delta s_l}{2b}\right) \\ \frac{\Delta s_r + \Delta s_l}{2} \cdot \sin\left(\theta + \frac{\Delta s_r - \Delta s_l}{2b}\right) \\ \frac{\Delta s_r - \Delta s_l}{b} \end{bmatrix}$$

Donde Δs_r y Δs_l son el desplazamiento de las ruedas del robot, b es el ancho entre las ruedas y θ es el ángulo de orientación del robot.

Este sensor se puede encontrar integrado en la clase webots/DifferentialWheels y no se ha usado en el proyecto ya que suele depender de un punto inicial el cual no es conocido a priori, por lo que puede conducir a errores. En su lugar se han utilizado los sensores Compass y GPS para la orientación y posición global, permitiendo solucionar parcialmente el problema de la posición y orientación del robot.

2.4. CÁMARA FRONTAL Y CÁMARA ESFÉRICA

Son sensores añadidos al Pioneer 2 de forma externa que sirven principalmente para el reconocimiento de ciertos objetos u obstáculos, dependiendo del problema que se quiera resolver con estos sensores. El principio de las cámaras es la captura de la energía producida por los rayos de luz mediante unos fotosensores que mediante sensores CMOS o CCD y un filtro de Bayes es capaz de separarlo en tres espectros principales de color: rojo, verde y azul. A partir de esta conversión, se analiza cada captura realizada y se muestra por pantalla lo que llamamos píxel.



Cada píxel suele tener una cantidad de blanco o negro por canal de color que llamamos nivel de gris. A partir de estos conocimientos básicos, existe una rama del conocimiento dedicada a la visión por computador que se encarga de la detección de objetos, personas, entornos y su respectiva clasificación con el fin de ser igual o mejor a la visión humana. Este nivel de gris por canal es usado en la práctica para conocer entre que rangos del espacio RGB se encuentra el color del objetivo que queremos localizar.

Las cámaras usadas son la típica cámara frontal, el cual muestra lo que el robot ve de frente, por lo que no puede obtener información de los laterales ni de los obstáculos traseros, por lo cual suele ser usado para guiarse en torno a una dirección o el reconocimiento de objetos. Para resolver dicho problema, se utiliza la cámara esférica, que ofrece una información completa de los alrededores del robot permitiendo saber que obstáculos hay alrededor del robot o que señales podemos observar en el suelo. Se complementa con la frontal ya que esta no ofrece una visión definida del entorno y se encuentran en la librería webots/Camera.



3. ENTORNO

El entorno usado para la práctica es una pequeña zona de obstáculos que va cambiando a lo largo de las 10 versiones de las cuales el robot debe estar preparado sobre todo para situaciones generales. Podemos tener tres subconjuntos de mapas por su luminosidad, aunque en cada mapa presenta cierta variación de píxeles.

El primer subconjunto corresponde a los escenarios 1,2,8,9 y 10. En estos escenarios la luminosidad es normal, con ciertas sombras y obstáculos distintos para una realización distinta



Mapas con luminosidad normal

Luego tenemos el mapa 3, que puede considerarse un mapa nocturno, este presenta una luminosidad baja, por lo que las cámaras deben tener un rango de detección mayor para tonalidades oscuras de un color, pero presenta similitudes con el mapa 2 en obstáculos.



Mapa con luminosidad baja

Por último, tenemos a los mapas de niebla o humo, estos mapas representan distintos niveles de humo, los cuales el 4 y 5 presentan humo leve y los mapas 6 y 7 presentan humo denso. Estos mapas dificultan a las cámaras en una luminosidad mayor y contrastes de color distintos a los demás mapas, además de que en los mapas densos presentan obstáculos no dados en otros mapas, por lo que el controlador debe estar preparado para dichas situaciones.



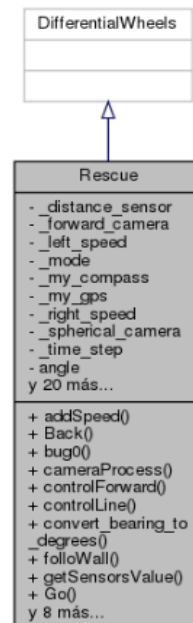
Mapas con humo o niebla

4.CONTROLADOR

El controlador es la parte más importante en un robot, se encarga de utilizar la información dada por los sensores para utilizarlos en diversos algoritmos y decisiones que permitirán que el controlador use los actuadores de manera diversa para realizar el objetivo propuesto de la manera más efectiva posible.

El controlador que se va a usar es llamado Rescue, el cual empleará tres fases generales para la realización de la práctica y estas fases tendrán unas subfases que permitirán que los objetivos de cada estado se realicen correctamente. Las fases principales son:

- Go: se encarga de guiar al robot por distintas rutas de acceso hasta la zona donde estarían los objetivos a localizar.
- Search: encargado de usar principalmente las cámaras para guiarse por la zona de búsqueda y localizar a los objetivos.
- Back: una vez localizados los objetivos, el robot deberá ser guiado de nuevo hasta el inicio del mapa realizando un tipo u otro de ejecuciones dependiendo de la ruta seguida en Go.



UML del controlador

4.1. MÉTODOS GENERALES

En este apartado de describiré los métodos usados en dos o más fases principales del controlador, considerándolas como métodos generales.

- getSensorsValue(): dan valores de los sensores internos del robot como viene a ser el compass, el GPS y los sensores de distancia y se actualizan estos valores cada vez que se llama. Este método se utiliza en las tres fases para evitar mismas llamadas a los sensores en cada fase

```
154 const double *gps_value; {
155 const double *compass_value;
156 //Valores del gps
157 gps_value = my_gps->getValues();
158 X=gps_value[2];
159 Y=gps_value[0];
160 //Valores de la brújula
161 compass_value = my_compass->getValues();
162 compass = convert_bearing_to_degrees(compass_value)-45;
163 //Valores de los Sensores delanteros de distancia
164 dist_sensor=0;
165 for(int i=0;i<5;i++){
166 ir_val[i] = distance_sensor[i]->getValue();
167 if(ir_val[i]>0) dist_sensor++;
168 }
169 for(int i=11;i<15;i++){
170 ir_val[i] = distance_sensor[i]->getValue();
171 if(ir_val[i]>0) dist_sensor++;
172 }
173 }
174 }
```

- bug0(): este método se basa en la técnica bug0 empleada en robótica, en este método se obtiene las coordenadas X e Y de un posible objetivo y las coordenadas actuales del robot. En base a ello se determina el ángulo entre el robot y ese objetivo para luego en la fase utilizar este ángulo para llamar a controlForward() para dirigirse hacia esa posición llamado en las tres fases, siempre en conjunto con controlForward() o controlLine(). Este algoritmo es de los más básicos dentro de los algoritmos tipo bug, el cual se explican en el apartado 5.

```
154 const double *gps_value; {
155 const double *compass_value;
156 //Valores del gps
157 gps_value = my_gps->getValues();
158 X=gps_value[2];
159 Y=gps_value[0];
160 //Valores de la brújula
161 compass_value = my_compass->getValues();
162 compass = convert_bearing_to_degrees(compass_value)-45;
163 //Valores de los Sensores delanteros de distancia
164 dist_sensor=0;
165 for(int i=0;i<5;i++){
166 ir_val[i] = distance_sensor[i]->getValue();
167 if(ir_val[i]>0) dist_sensor++;
168 }
169 for(int i=11;i<15;i++){
170 ir_val[i] = distance_sensor[i]->getValue();
171 if(ir_val[i]>0) dist_sensor++;
172 }
173 }
174 }
```

- controlForward(): este método es usado para corregir la dirección del robot en torno a un ángulo dado angle. Este método hace que el robot gire a la derecha cuando el compass es menor que el ángulo y a la izquierda cuando es mayor. Si el compass está dentro el rango

```
103 //Control del robot e torno a angle
104
105 if(compass<angle-1) addSpeed(MAX_SPEED,MAX_SPEED/2);
106 else if(compass>(angle+1)) addSpeed(MAX_SPEED/2,MAX_SPEED);
107 else addSpeed(MAX_SPEED,MAX_SPEED);
108
109 }
```

deseado, el robot irá a máxima velocidad en línea recta.

- `controlLine()`: es el método principal de guiado del robot por el mapa, realiza un movimiento u otro en base al modo que haya sido llamado de la lista de modos `_mode`.

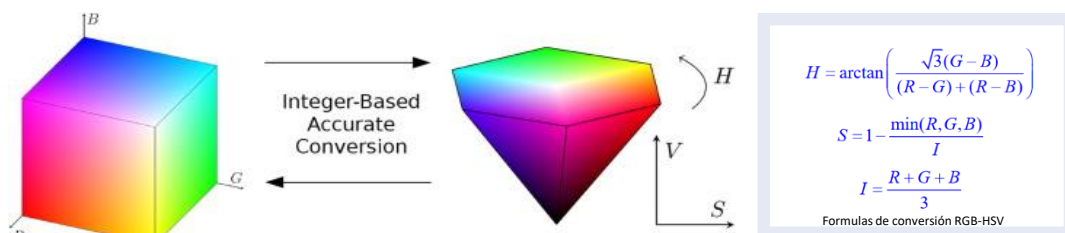
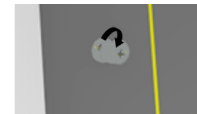
Si el modo es `TURN_LEFT`, el robot gira a la izquierda en torno a su eje vertical; `TURN_RIGHT` gira hacia la derecha en su eje vertical; `TURN_RIGHT_CONTROL` permite girar el robot hacia la derecha mientras avanza; `TURN_LEFT_CONTROL` gira el robot hacia la izquierda mientras avanza y `STOP` para el robot en seco. El modo `FORWARD` es el modo de control en línea recta en torno a un ángulo y lo que hace es llamar a `controlForward()`.

```
71 switch (_mode) {
72     case STOP:
73         cout<<"STOP"<<endl;
74         addSpeed(0,0);
75         break;
76     case FORWARD:
77         cout<<"FORWARD"<<endl;
78         controlForward();
79         break;
80     case TURN_LEFT:
81         cout<<"TURN LEFT"<<endl;
82         addSpeed(-MAX_SPEED/2,MAX_SPEED/2);
83         break;
84     case TURN_RIGHT:
85         cout<<"TURN RIGHT"<<endl;
86         addSpeed(MAX_SPEED/2,-MAX_SPEED/2);
87         break;
88     case TURN_LEFT_CONTROL:
89         cout<<"TURN LEFT CONTROL"<<endl;
90         addSpeed(MAX_SPEED/2,MAX_SPEED);
91         break;
92     case TURN_RIGHT_CONTROL:
93         cout<<"TURN RIGHT CONTROL"<<endl;
94         addSpeed(MAX_SPEED,MAX_SPEED/2);
95         break;
96     default:
97         break;
98 }
99 }
100 }
```

- `addSpeed (double left_speed,double right_speed)`: este método es usado como una función set para los atributos `_left_speed` y `_right_speed`. Pasa por parámetros unas velocidades que se añaden a dichos atributos. Este método es muy usado en muchos métodos como los descritos antes y la fase `Search()`.

```
149 this->_left_speed=left_speed;
150 this->_right_speed=right_speed;
151 }
152 }
```

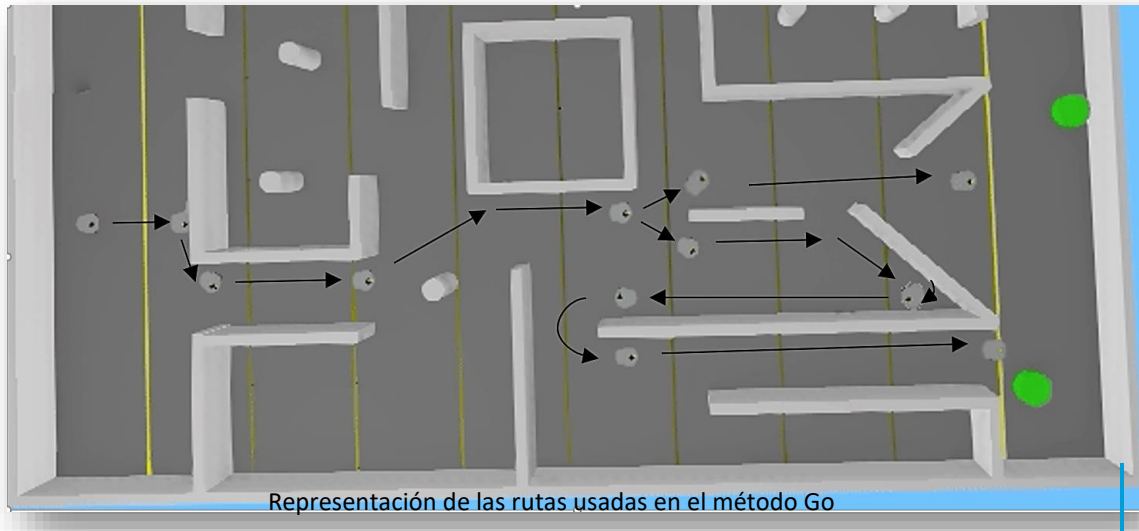
- `initialPosition(int angulo_salida)`: este método es usado en `Go` y `Back` para reorientar al robot en caso de no encontrarse en el ángulo de orientación deseado girará en torno a su eje vertical hasta que el compass esté dentro del rango deseado.
- `convert_bearing_to_degrees (const double *in_vector)`: permite al compass transformar sus valores vectoriales sacados de `getValues()` de la clase `Compass` en grados para una interpretación mejor y control de robot.
- `RGB2HSV (unsigned char red, unsigned char green, unsigned char blue)`: este método es usado para cambiar el espacio de color RGB, el cual presenta errores cuando se trata de colores intermedios como vendría a ser el amarillo, a un espacio de color más complejo como es el HSV, pero más efectivo. El HSV se basa en:
 - Hue: la longitud de onda del color;
 - Saturation: saturación de este, indica como de claro u oscuro se encuentra
 - Value: intensidad de brillo del píxel



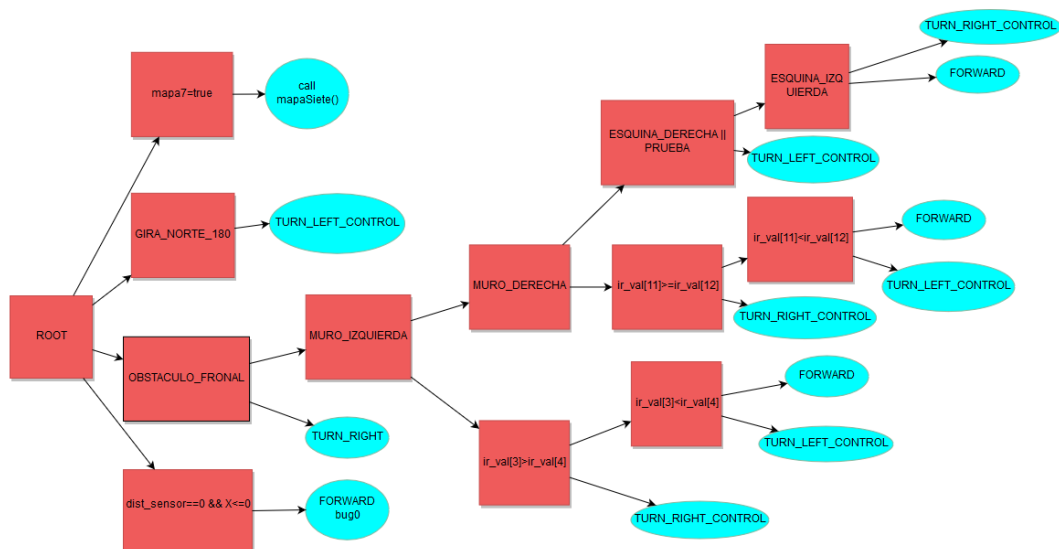
Este método es usado en `yellowLine ()` de `Go` como en `cameraProcess ()` de `Search`.

4.1. GO

En este método principal del controlador se usan varios métodos que se usan también en otras funciones principales, pero tienen una prioridad mayor en este caso. El objetivo de Go es poder llegar al objetivo sin que el robot sufra daños por golpes fuertes., por lo que se usan dos rutas principales en función de la situación correspondiente.

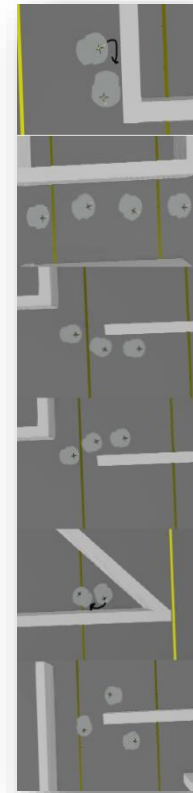


Como se ve en la imagen, el robot tiene una ruta principal que lo lleva a la salida intermedia del mapa y otra secundaria que suele usarse en mapas con la entrada principal bloqueada o por cuestión de errores del controlador. Por lo general esta ruta suele ser más larga puesto que tiene que retroceder para poder acceder a la entrada de abajo. Como se comentó en la introducción de este apartado, Go tiene unas subfases que en este caso sería el de un árbol de decisiones constituidas según qué situación se encuentre el robot. Estas situaciones corresponden a las declaraciones booleanas correspondientes al inicio del bucle de Go.



Los bloques de decisiones del árbol de decisiones corresponden a unos booleanos declarados en Go que se inicializan las condiciones correspondientes a cada sensor o suceso ocurrido durante el recorrido. Las condiciones dadas en el árbol de decisiones son:

- OBSTACULO_FRONTAL: si los sensores frontales detectan presencia y el robot se sitúa a -10/10 grados del norte, el robot girará a la derecha
- MURO_IZQUIERDA: esta condición se activa si el sensor lateral 3 detecta obstáculo. Dentro de su condición, hay tres subcondiciones de control para que siga el muro, pero dando problemas de oscilaciones que pueden llegar a errores como coger una ruta que no estaba planeado que la cogería
- MURO_DERECHA: es equivalente al muro izquierda, pero con el sensor lateral 12.
- ESQUINA_IZQUIERDA: cuando el sensor 1 detecta obstáculo, se trata de un obstáculo que se puede esquivar con un leve giro a la izquierda.
- ESQUINA_DERECHA: el caso es el mismo de esquina izquierda, pero con el sensor 14.
- RINCON: esta condición es anómala puesto que solo ocurre cuando el robot se encuentra en una esquina, este gira a la derecha, pero sin recurrir a los modos ya que necesita una velocidad menor para actuar.
- GIRA_NORTE_180: en esta condición, el robot está orientado hacia el sur del mapa, por lo que, si ninguno de los sensores de distancia frontales no detecta obstáculos, el robot se dará media vuelta.



Mapa7 se trata de una condición especial para el escenario 7, el cual presenta ciertos obstáculos que otros mapas no tienen y suponen un problema para el guiado del robot por dicho mapa, así que se llama a mapaSiete() para poder guiar al robot por dicho mapa. Esta condición se activa si el robot detecta que ha ido por una ruta que solo ocurre en dicho mapa.

La fase Go se debe terminar cuando encuentre una marca que le indique que se encuentra al final del mapa, que en este caso será la línea amarilla del final del mapa. Esta línea es detectada por la cámara esférica, la cual procesa la información de los píxeles en la función [yellowLine\(\)](#).

[yellowLine\(\)](#) devolverá un true si encuentra la línea correspondiente, haciendo que el método Go termine. En este método se analiza la imagen con `image->getImage()`, que dará la imagen de la cámara esférica y con bucles for se recorrerá la matriz de la imagen sacando los niveles de gris de los tres canales RGB. Estos canales, como se comentó en RGB2HSV, tienen el problema de la mezcla de colores, por lo que se transforma a HSV y se les da unas condiciones para que distinga el color amarillo en distintos niveles de brillo ya que, como se comentó en el apartado de mapas, los mapas contienen distintos niveles de brillo por lo cual no se puede condicionar a un solo nivel de amarillo.

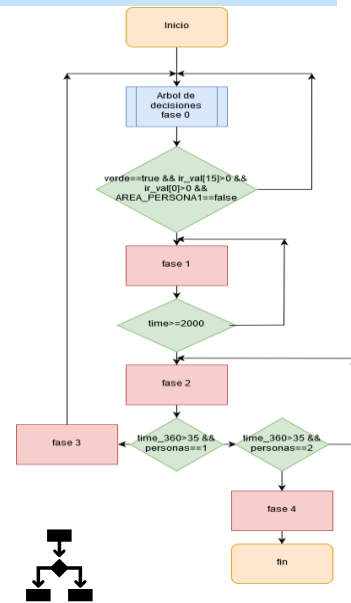


Al encontrar la línea, el robot guarda en memoria las coordenadas para luego poder volver al finalizar la función Search y empezar desde allí la fase de Back.

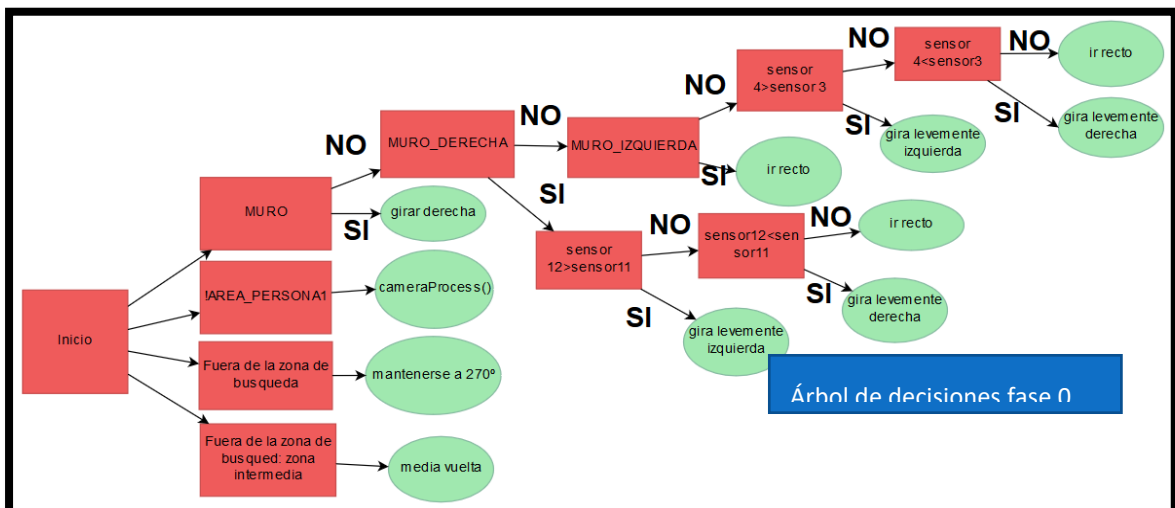
4.2. SEARCH

Esta es la segunda fase del proyecto. Una vez el robot haya llegado a su destino por las distintas rutas sin recibir un daño considerable, toca buscar los objetivos. Para esta fase, se emplea una selección de modos secuencial que se podrá ver en el diagrama de flujo.

La fase 0 es la fase de búsqueda, el robot empleará un árbol de decisiones como el mostrado a continuación para recorrer la zona de búsqueda sin sufrir daños y sin salir de la misma hasta que los dos objetivos sean localizados. Los objetivos son localizados con la cámara frontal, la cual accedemos a ella en el método [cameraProcess\(\)](#) del que se hablará más tarde. Si el robot ya ha encontrado un objetivo, se creará un área en torno al mismo que, controlando la distancia a ese objetivo, el robot lo interpretará como un obstáculo evitando así una confusión por parte del robot. Cuando uno de los objetivos es localizado, este



La fase 1 es simple, el robot debe permanecer estático durante dos segundos delante



del robot a menos de 1 metro, esto último debe haberse cumplido en la fase 0, ya unos de los requisitos en el cambio de fase es que los sensores frontales tengan un valor mayor que 0, indicando que han detectado un posible objetivo. Con el time step podemos averiguar que el robot se ha quedado parado dos segundos de simulación. A la vez que se encuentra parado, el robot comprueba que no tenía localizada otra persona antes para guardar la posición del actual objetivo encontrado en memoria y así no confundirlo cuando intente buscar al objetivo restante, para ello mediante el compass y trigonometría averigua un punto aproximado del objetivo y realizar lo comentado en la fase 0. Cuando esos dos segundos hayan pasado, se pasa a la fase 2.



La fase 2 trata de dar una vuelta completa en torno al eje vertical del robot, indicando el fin de localización de un objetivo. Si el robot ha encontrado una persona, pasa a la fase 3, si ha encontrado dos pasa a la fase 4.



La fase 3 trata de un giro pequeño a la izquierda para ayudar al robot a continuar la búsqueda. Tras ello vuelve a la fase 0 para continuar la búsqueda del segundo objetivo.

La fase 4 se activa cuando el robot ha encontrado los dos objetivos, el robot vuelve al lugar de donde llegó mediante `bug0` como objetivo las coordenadas guardadas en `Go`.

4.2.1. CAMERA_PROCESS()

Durante la realización de la búsqueda de objetivos, el robot usa la cámara frontal para encontrar el patrón de colores concreto para la localización. Para ello usa la llamada al método [cameraProcess\(\)](#) por el cual devolverá un booleano que le indicará al controlador que ha encontrado en su campo de visión un objetivo.

La función es parecida a la de [yellowLine\(\)](#), pero con unos matices diferentes que hace que este proceso también interfiera en el manejo del robot a lo largo de la búsqueda. Comenzamos obteniendo el ancho y alto de la imagen y la propia imagen de la cámara frontal, luego se analiza píxel por píxel como en la función anterior y se le aplica HSV para encontrar cierto rango de verde a distinta luminosidad ya que según el mapa su variación de nivel de gris es significativa.

Si se obtiene píxeles verdes, se localizará en qué lado de la imagen se encuentra, si la mitad izquierda o derecha y según esta condición se añadirá un contador para notificar cuántos píxeles verdes hay en cada mitad. Según estos últimos parámetros el robot podrá dirigirse a su objetivo corrigiendo hacia donde gira, de manera similar a [controlForward](#).

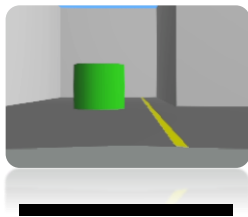


Imagen captada de mapas de niebla



Imagen captada de mapas comunes

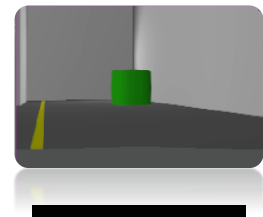


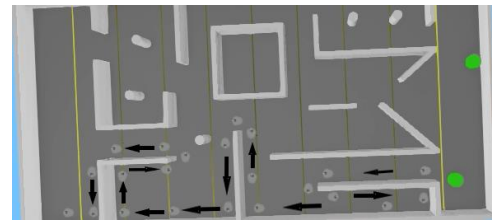
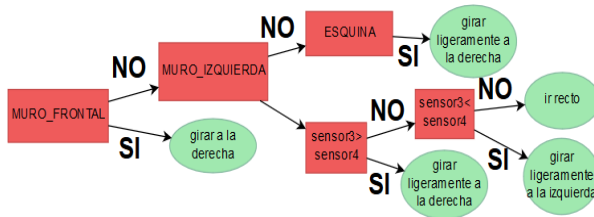
Imagen captada de mapas nocturnos

4.3. BACK

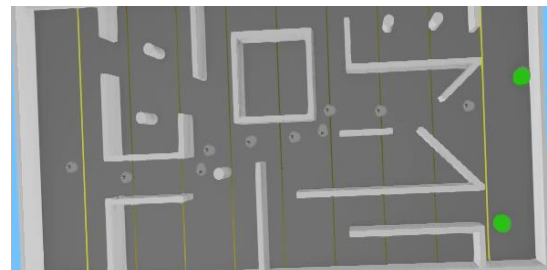
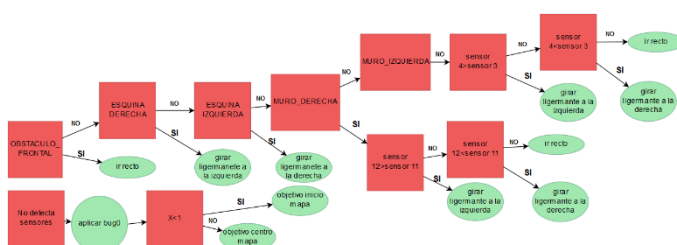
Tras haber llegado a la zona de búsqueda y localizado a los objetivos, toca volver a la zona de inicio, para ello `back` realizará pasos similares a los de `Go`, pero con una diferencia: localización. El robot empezará a estabilizar su posición inicial, para ello llama a [initialPosition](#) pero esta vez con dirección al sur del mapa.

Tras estabilizar el robot, este realizará un proceso u otro según de donde haya venido. Si el robot ha venido desde la salida inferior del mapa elegirá [followWall](#); por otro lado, si ha venido por las otras dos entradas entonces su proceso llamará a [homeComing](#).

folloWall es una de las funciones más básicas de la robótica : ¿Cómo resuelves un laberinto?, pues siguiendo el muro más externo. Este plantea un árbol de decisiones sencilla ya que solo depende de los sensores frontales y laterales izquierdos. La trayectoria que sigue es la siguiente junto a su árbol:



homeComing es la contraparte de Go, utiliza un árbol de decisiones más complejo que folloWall pero más sencillo que Go, puesto que a la vuelta no se contempla elementos como esquinas o cambios de sentido. El proceso que diferencia de este último es que, en caso de



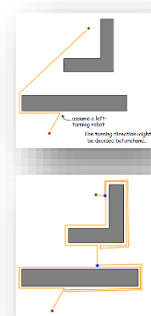
tener un obstáculo frontal, el robot girará a la derecha. Con respecto a los destinos el robot usa también bug0, pero con dos destinos: el centro del mapa y el inicio del mapa. El árbol de decisiones es el siguiente, junto con la ruta.

5.ALTERNATIVAS DE PROCESO

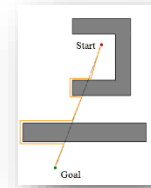
Durante el proceso de realización del proyecto se propusieron varias alternativas con relación a la estructura del código, la trayectoria del robot y como buscar los objetivos pero que no han tenido éxito por varios motivos, ya sea de procesamiento o de ruido.

Primero hablaremos de la técnica bug. Existen varias técnicas de poder controlar un robot con técnicas usada por insectos, de ahí el nombre de bug. Las técnicas propuestas fueron:

- Bug0: la técnica es basada en saber en que dirección se encuentra el objetivo y seguir dicha dirección. En caso de haber un obstáculo, este se sigue hasta volver a orientarse en la dirección del objetivo.
- Bug1: el bug 1 añade una mejora al bug0, el poder añadir memoria del recorrido seguido. El robot seguirá el obstáculo hasta encontrar el punto mas cercano al objetivo para luego volver a ese punto. Este proceso requiere cierta cantidad computacional.



- Bug2: esta técnica permite mejorar las dos técnicas anteriores. A parte de saber que orientación hay que seguir para llegar al objetivo, se traza una línea entre la posición inicial y final. Una vez trazada en memoria, el robot la seguirá hasta encontrar un obstáculo, el cual será seguido hasta volver a tocar la línea. Esta técnica es potente ya que evita problemas de orientación o seguimiento de muros del bug0 y el procesamiento del bug1. Pero aún sigue teniendo un problema: la precisión.



De estas técnicas de inicio se usó la bug2 ya que era la más adecuada para llegar a su destino ya que bug0 si solo sigue muros, puede llegar a atascarse en puntos del mapa y bug1 necesita odometría para memorizar y como de comenté en el apartado de sensores, con odometría no podíamos saber la posición inicial a priori. Se eligió al final bug0 debido a que el procesamiento del computador no podía captar perfectamente la línea trazada, por lo que el robot daba errores de localización. A bug0 no le ocurre eso, puesto que tener un porcentaje de error del punto de destino en esta técnica no es relevante.

Con respecto al código, se han implementado muchas alternativas que acabaron en errores de compilación o incluso en errores de IDE. Se intentó realizar el código en Python, ya que este lenguaje tiene menos dificultad con respecto al tratamiento de memoria y a su estructura simple, pero los errores del interprete eran mayores que en c++. Se probó realizar herencia entre Rescue y sus hijas Go, Search y Back, pero webots lo interpretaba como varias instancias en vez de una, por lo que el tratamiento de herencias no se realizó correctamente. Para mejorar la estructura del código, se probó poner los tres métodos principales en otra clase y los otros métodos en Rescue, así siendo más sencillo de interpretar; pero seguía siendo más trabajo del necesario para las líneas de código que existían. Al final se optó por las fases en función de modos que van pasando conforme el método termina su función en el programa.

Se optó por utilizar la cámara para poder guiar al robot por la arena, pero el procesamiento de píxeles no era el adecuado y las librerías openCV, las más adecuadas en estos casos, no conseguía obtener los resultados que se querían. Se optó por la simpleza de los sensores de distancia, brújula y GPS.

Con respecto a la Search, se pretendía desde el principio realizar el proceso completo con la cámara frontal, pero como se ha comentado anteriormente, manejar el robot con cámaras es complejo y no permitía realizar con precisión la posición del objetivo ya detectado, por lo que se realizó un diagrama de flujo que realizase la tarea sin recibir golpes.

Con Back fue sencillo, puesto que al ser las mismas técnicas que Go o más fáciles, no se realizaron muchos cambios. El único cambio era de como se hacía que el robot detectara la línea amarilla de inicio. Al ver que dicha línea no se detectaba siempre, se optó por parar cuando llegase a X posición.

6.RESULTADOS EXPERIMENTALES

Al realizar nuestro código para ir, buscar y volver se puso a prueba el Pioneer 2 para obtener los resultados. Esto fue los tiempos que se obtuvieron en las 10 versiones de prueba:

TIEMPOS		GO	SEARCH	BACK	TOTAL
Scenario1		0:31	0:33	0:36	1:40
Scenario2		1:11	0:58	1:22	3:31
Scenario3		1:05	1:13	1:11	3:29
Scenario4		0:30	0:47	0:41	1:58
Scenario5		0:55	2:31	0:54	4:20
Scenario6		0:46	1:00	0:53	2:39
Scenario7		1:04	1:16	0:49	3:09
Scenario8	Ruta 1	1:02	1:22	0:40	3:04
	Ruta 2	1:01	0:28	1:18	2:47
Scenario9		0:52	0:58	1:56	3:46
Scenario10		0:36	2:41	0:36	3:53

Como se puede comprobar, los mapas en los que el robot toma la segunda ruta, es decir, la que se encuentra en el extremo derecho del mapa, suelen tardar más que los que cogen la ruta principal. Esto es por lo que se explicó de que en esas rutas el robot tiene que dar media vuelta para poder acceder a esa parte, por lo que el recorrido es más largo. Con respecto a la búsqueda por lo general suelen tardar lo mismo, pero en el mapa 5 y 10 tardan más de la cuenta. Estos casos son debido a que el robot tiene problemas para detectar objetivos que se encuentran dentro de huecos no muy visibles, como ocurren en esos mapas. A la vuelta son relativamente parecidos los tiempos ya que las rutas de vuelta no son muy complejas. Esta situación no ocurre con el mapa 9, que suele tardar bastante debido a ciertos errores de orientación. El mapa 8 tiene las dos rutas ya que este mapa no suele ir por lo general por una ruta, sino que en unas ocasiones coge uno y en otras coge otro. En las dos rutas suele tardar parecido, con unos segundos de diferencia.

7.CONCLUSIÓN Y PROPUESTAS

Como final de este documento, se dará una conclusión de los resultados y algoritmos utilizados y que mejoras se pueden hacer al robot para futuras ideas o proyectos, tanto en el robot como en el código.

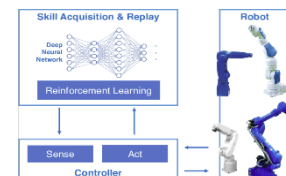
7.1.CONCLUSIÓN

Tras ver los resultados del robot se puede llegar a la conclusión que el robot en cuestión de tiempo no es el más adecuado, ya que en una fase u otra el robot llega a atascarse hasta que vuelve a la rutina normal. Con respecto a los golpes sufridos, el robot tiene ciertos golpes catalogados como leves debido a la mala precisión de los sensores o debido a poder evitar ese golpe, el robot tiende a dar vueltas por una zona hasta que dichos sensores se actualizan. Los árboles de decisiones han permitido que el robot realice los mapas correctamente a pesar de los errores de los sensores y en casos como el mapa7, se ha tenido que dar coordenadas conocidas, rompiendo la generalidad del robot en los entornos.

7.2.PROPUUESTAS.

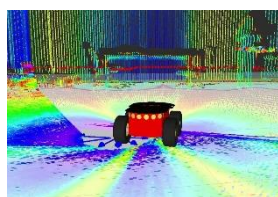
El robot tal y como está definido es un robot básico para conocer el entorno de desarrollo, pero es posible mejorarlo con ciertos sensores y técnicas que podrían ser más efectivas o corrijan errores que no han podido resolverse.

- Reinforcement Learning: utilizando dicha técnica, podríamos dotar al robot de cierta inteligencia haciéndole aprender de sus movimientos con castigos y recompensas, permitiendo que el robot aprenda a moverse por cualquier mapa.



- PID: el problema de seguir un muro son las oscilaciones que causan el control de la velocidad de las ruedas con los sensores. Un controlador PID podría haber resuelto este problema, puesto que el robot reduciría las oscilaciones a un nivel aceptable.

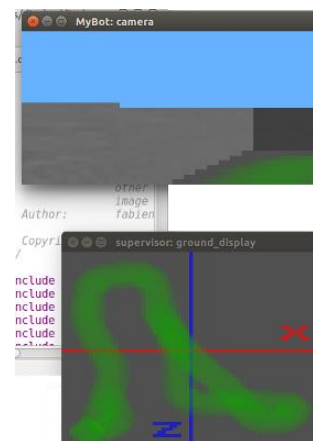
- LiDAR: este sensor no ha sido comentado en el apartado de los sensores, puesto que no



lo contenía el Pioneer 2. Con este sensor, el robot podría crear un entorno 3D mediante la proyección laser y conocer los obstáculos con mucha mayor precisión que los sensores a distancia, y poder reconocer los objetivos sin necesidad de contar

con las cámaras.

- Display: webots dispone de un display para poder analizar y modificar la imagen obtenida por las cámaras, con esto y procesamiento de imagen con openCV es posible detectar rutas y objetos con la cámara mientras el programados puede ver los resultados por pantalla.





8. BIBLIOGRAFÍA

Encoders: https://www.pc-control.co.uk/incremental_encoders.htm

Ultrasonidos: <http://www-personal.umich.edu/~johannb/Papers/paper33.pdf>

Bug: https://www.cs.cmu.edu/~motionplanning/lecture/Chap2-Bug-Alg_howie.pdf

LiDAR: <https://www.nobbot.com/futuro/tecnologia-lidar/>

GPS: <https://www.lifewire.com/how-car-gps-trackers-work-4147185>

<https://www.youtube.com/watch?v=6m0xGwkYYy0>

Machine Learning: <https://www.geeksforgeeks.org/what-is-reinforcement-learning/>

HSV: https://aulaglobal.uc3m.es/pluginfile.php/3310201/mod_resource/content/2/Clase%202-Imagenes.pdf

Webots Documentación: <https://cyberbotics.com/doc/guide/index>

PID: <https://www.luisllamas.es/teoria-de-control-en-arduino-el-controlador-pid/>