# Práctica Robótica

Carmen Carrero Hurtado, Andrés Fernández Pérez

January 21, 2019

# Contents

# 1 Práctica 1

## 1.1 Introdución

En la primera entrega de nuestro proyecto el objetivo principal es implementar un controlador con el cual el robot pueda moverse solo por la sala, evitando colisionar con las paredes y con las cajas. La entrega cuenta con un segundo objetivo, mejorar el código de tal manera que recorra la mayor parte de la habitación en el menor tiempo posible. Para ello hemos hecho uso de dos proxys, el robot y el láser que este porta.

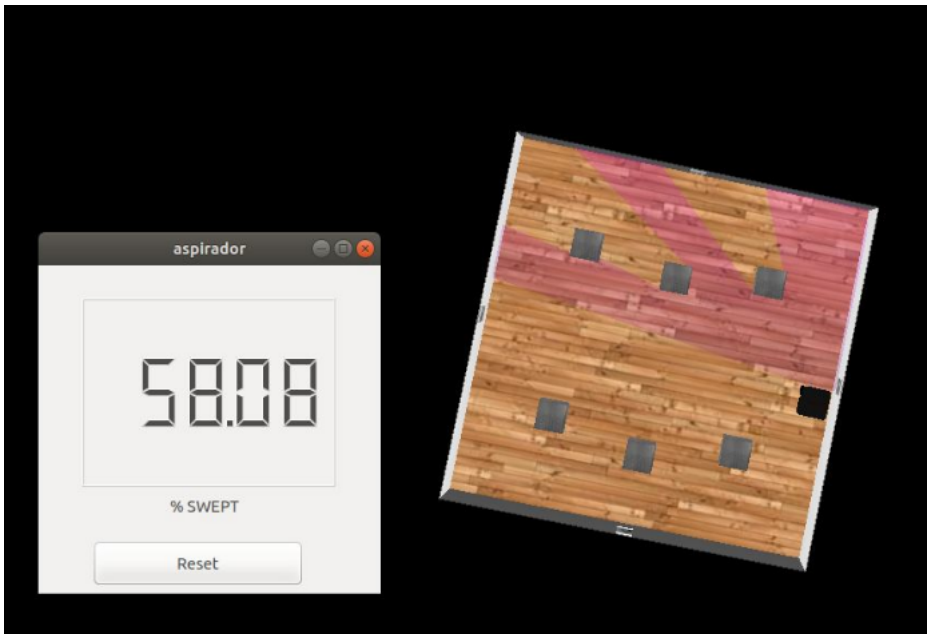
## 1.2 Código

Para conseguir llegar a nuestro objetivo hemos incluido en el código una serie de mejoras que detallamos a continuación. En primer lugar hemos calculado la velocidad de avance del robot, mediante la fórmula VelocidadMáxima * (distanciaMínima / 5000). La velocidad máxima será de 1000 y el número 5000 es por los 5 metros que mide la sala. La distancia mínima es la distancia al obstáculo más cercano que tiene enfrente. De una forma muy parecida hemos controlado la velocidad de giro, pero siendo esta inversamente proporcional, ya que cuanto más cerca está el objeto, más rápido gira. Hemos establecido un umbral y el robot actuará en función de este. Si está más cerca de un objeto por su parte derecha girará a la izquierda y viceversa. Hemos controlado el caso de las esquinas, ya que el robot al encontrarse con pared a su derecha y a su izquierda, entraba en una especie de bucle del que le costaba salir. Para esto, hemos puesto que si el robot empieza a girar a la derecha no pueda girar a la izquierda si antes no ha caminado recto. Por otro lado, si la distancia al obstáculo más cercano es mayor que el umbral el robot avanzará en línea recta. Se ha establecido un contador que suma uno cada vez que el robot avanza en línea recta; si después de 25 "pasos" el robot no se ha encontrando con ningún obstáculo, girará de forma aleatoria para continuar en otra dirección.

## 1.3 Autoevaluación

Para la evaluación de esta entrega utilizaremos otro controlador implementado por el profesor de la asignatura, llamado "aspiradora", el cual mostrará la cantidad de área por la que pasa el robot durante un tiempo determinado. Hemos establecido un tiempo de 5 minutos y estos han sido los resultados:
1o 58.08
2o 53.28
3o 55.36
4o 53.6
5o 55.2
Media : 55.104

Resultado de la primera ejecución:



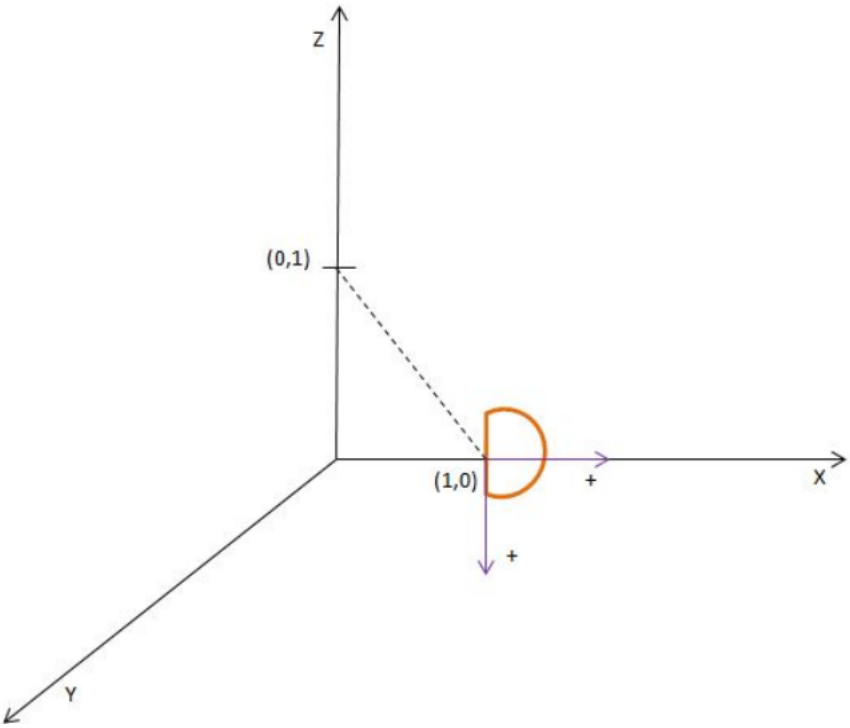
# 2 Práctica 2

## 2.1 Introducción

En esta entrega se nos pedía modificar el componente de la práctica anterior para que cuando pulsaramos sobre cualquier parte del tablero el robot se moviera hasta allí. Si pulsamos varias veces el ratón en distintos lugares, el robot irá a la última posición pulsada sin tener en cuenta los obstáculos, pudiente en esta entrega atravesarlos. Para llegar a esto, incluimos unas líneas en el archivo controller.cdsl para que reconociera el click del ratón y regeneramos el componente. Después editamos el archivo config y después de compilar de nuevo se creó un nuevo método setPick. Lo siguiente que había que hacer era definir una estructura thread-safe, por lo que creamos una clase nueva llamada Target, que explicaremos más adelante. El objetivo de esta estructura es almacenar las coordenadas del mundo real a las que tiene que dirigirse el robot. Una vez que teníamos la clase implementada, nos pusimos a desarrollar el código.

## 2.2 Código

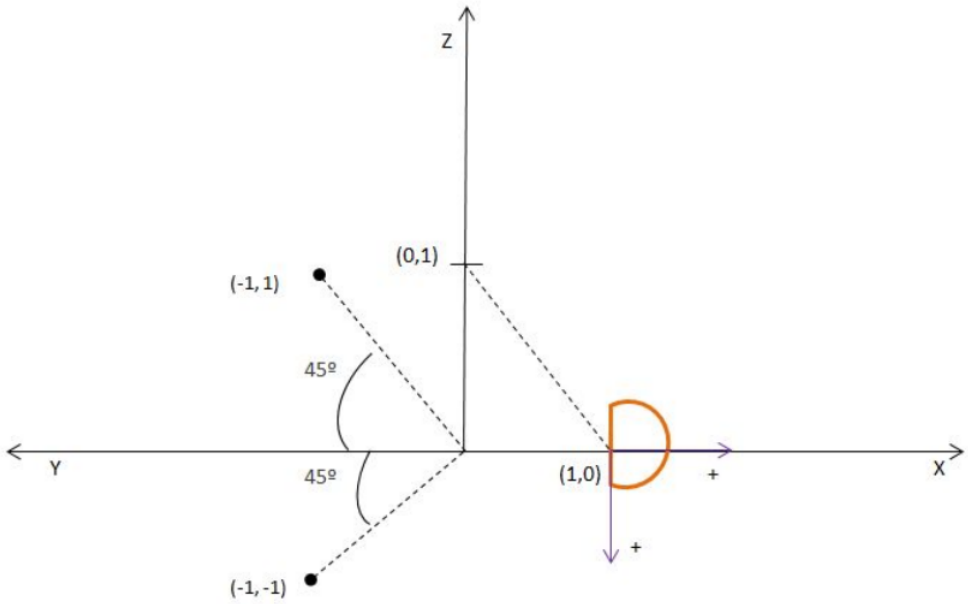
Vamos a empezar explicando la clase implementada para almacenar las coordenadas, llamada Target. Esta clase tendrá como atributos un vector de tamaño dos que almacenará las coordenadas, un mutex para garantizar la exclusión mutua y un boolean, inicialmente a falso, que se utilizará para comprobar si el robot tiene coordenadas pendientes o no. Target también cuenta con cuatro métodos: insert(float x, float y), extract(), setPendiente(bool pend) y getPendiente(). El primer método inserta los valores pasados por parámetro en el vector de coordenadas y pone el boolean a true para indicar que el robot tiene coordenadas pendientes. El método extract devuelve el vector de coordenadas, así como los métodos set y getPendiente que modifican y retornan el boolean, respectivamente. Todos los métodos garantizan una estructura segura para hilos mediante la variable mutex. Una vez explicada la clase Target, comentamos la clase specificWorker, ya que fue la única que modificamos para la creación de este componente.
El algoritmo seguido para el desarrollo de la práctica fue el siguiente:
Teniendo el cuenta que el mundo está en 3D y el robot ve en 2D, tenemos que pasar las coordenadas del mundo real (las que recibe del ratón) a las del robot. Para tener una mejor concepción de esto se desarrolla un ejemplo ilustrativo. Como vemos en la figura, el robot se encuentra en las coordenadas (1,0) orientado hacia la derecha, de tal manera que el eje del robot será el marcado en lila. Ponemos como coordenadas destino el punto (0,1), que si transformamos eso a cómo lo vería el robot sería el punto (-1, -1). La cuestión ahora es encontrar una fórmula para realizar eso.




La fórmula escogida fue:

$$X_R = R^T (Z_W - T_W)$$

Para representar esto gráficamente, seguimos con el ejemplo anterior donde Z - T = (0,1) - (1,0) = (-1, 1) y R = 90o (ya que el robot está mirando a la derecha y solo gira una vez -90o- para “mirar recto”).

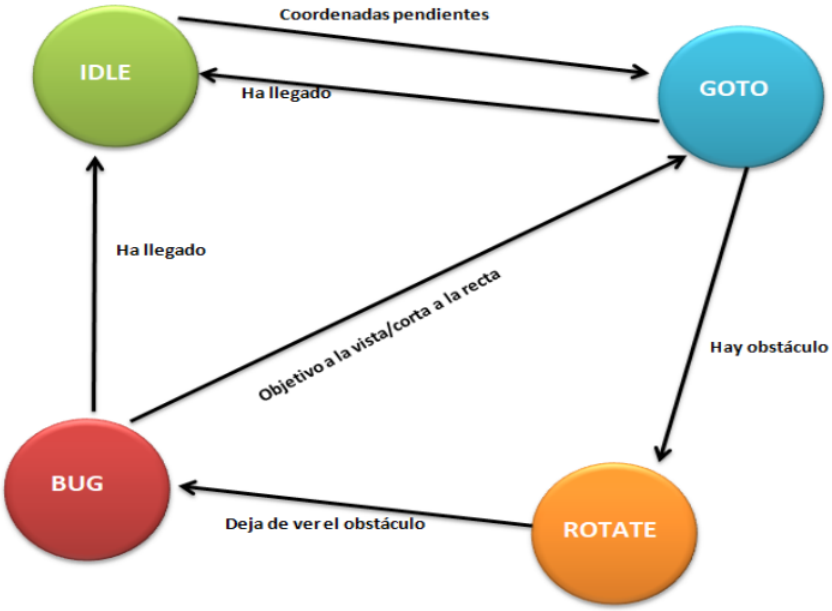



Efectivamente, comprobamos visualmente que una vez que te situas en el punto (-1, 1) giras 90o y acabas en el punto (-1, -1) que serían las coordenadas (0, 1) del mundo real en la posición del robot. Trasladamos esto al código y añadimos una fórmula más para calcular la velocidad que llevaría el robot en cada momento. Finalmente, suponemos que si la distancia es menor que 50 se ha llegado al objetivo, por lo que paramos al robot y actualizamos el boolean de la clase Target a false, pues no hay coordenadas pendientes.

# 3 Práctica 3

## 3.1 Introducción

En esta entrega se nos pedía extender el componente de la práctica anterior para que cuando pulsaramos sobre cualquier parte del tablero el robot se moviera hasta allí, esta vez evitando los obstáculos. Para esto último utilizamos el algoritmo bug, inspirando en el movimiento de los insectos. Este algoritmo consiste en que cuando el robot está cerca del objeto, lo rodea siempre en la misma dirección hasta que ve de nuevo el objetivo. Lo primero que hicimos antes de programar fue crear una máquina de estados sobre el papel, que nos serviría para controlar la lógica del componente. Esta máquina de estados es la siguiente:




## 3.2 Código

Tradujimos la máquina de estados del papel al código de la siguiente manera: (hay algunos cambios de estado que se encuentran dentro de los propios métodos, que veremos en detalle más adelante)

En un principio, el robot se encuentra en estado IDLE hasta que el usuario clicka sobre cualquier zona del mapa. En ese momento, se calcula la recta entre dos puntos (dónde se encuentra el robot al principio y donde tiene que llegar) y se almacena en tres variables llamadas A, B y C, correspondiendo con la ecuación de la recta ($Ax + By + C = 0$). Esto es importante ya que en un momento

```
switch(state) {

     case State::IDLE:

          if (coord.getPendiente())
               state = State::GOTO;
          break;

     case State::GOTO:
          gotoTarget();
          break;

     case State::BUG:
          bug();
          break;

     case State::ROTATE:
          rotar();
          break;
     }
```

dado el robot comprobará si desde la posición actual corta dicha recta, ya que eso significa que se encuentra en la línea hacia las coordenadas objetivo, por lo que lo único que tendrá que hacer será avanzar "siguiendo la línea" y no seguir rodeando el obstáculo.

Volviendo a la máquina de estados, hemos dicho que el robot se encuentra en IDLE y como tiene coordenadas pendientes, la condición del if se cumple y cambia su estado por GOTO, entrando en el método gotoTarget(). En dicho método se hace una primera comprobación para ver si el robot tiene algún obstáculo cerca (nosotros hemos puesto ese umbral en 250 mm de distancia); si esto se cumple, el robot cambia su estado a ROTATE y se para, saliendo también del método. Si no tiene ningún obstáculo próximo el robot avanza hacia las coordenadas objetivas, calculando siempre la distancia hasta las mismas. Finalmente, si la distancia es menor que 250 ha llegado al objetivo, por lo que se para y cambiamos el estado a IDLE, esperando nuevas coordenadas.

Hemos mencionado que si se encuentra con un objeto, cambia el estado a ROTATE. Vamos a suponer que así es por lo que ahora tenemos al robot en dicho estado, entrado en el método correspondiente. Este estado, junto con el estado BUG completan el algoritmo con el nombre del último; el método rotar() lo que hace es que gira el robot sobre sí mismo hasta que deja de ver el

obstáculo, y el método bug() lo que hace es que una vez que ha dejado de ver el obstáculo, lo bordea. Nosotros lo hemos programado para que siempre gire hacia la izquierda y si la distancia es mayor o igual que 450, entonces es que ha dejado de ver el objeto, por lo que se para y pasa al estado BUG.

Como hemos comentado antes, el estado BUG lo que hace es rodear el objeto. Cambia de estado a GOTO si se produce una de las dos condiciones, o bien tiene las coordenadas a la vista, o hay un momento que el robot corta la recta mencionada al principio. Si no se cumple ninguna de las dos opciones, el robot avanza rodeando el objeto. El robot para cuando se llega al objetivo.

# 4 Práctica 4

## 4.1 Introducción

Esta entrega tiene el mismo objetivo que la anterior, es decir, el usuario pulsa sobre un punto del mapa y el robot va esquivando los obstáculos. La única diferencia, es que en esta entrega se creará un camino por Dijkstra (será el camino más corto) que ya evitará los obstáculos y lo que hará el robot será seguir ese camino hasta las coordenadas objetivas. Para esto se ha importado la clase Grid, que crea un mapa del entorno e implementa el algoritmo Dijkstra. Esta entrega también cuenta con una interfaz gráfica, donde aparecerá el mapa con dos botones, Load y Save, que servirán para cargar y guardar el mapa del entorno.

## 4.2 Código

Lo primero de todo, el robot irá barriendo con el láser el mapa para crearlo y almacenarlo en un archivo llamado "map.txt", que será el que se cargue en la interfaz posteriormente (ya que tiene almacenado dónde se encuentran las paredes y obstáculos) y así no tiene que volver a recorrer el robot el entorno.

El cambio en el código ha sido totalmente en la clase specificworker, ya que la clase grid se ha quedado tal y como está.

Cuando el usuario clicka sobre un punto del mapa, se genera una serie de puntos que crean un camino, que será el que seguirá el robot. Estos puntos se almacenan en una variable path. Si el robot tiene coordenadas pendientes y la variable path está lista, obtiene el primer punto de dicho camino hasta que llega hasta a él, donde para y coge el siguiente punto. Así va uno a uno hasta llegar al final del camino, que serán las coordenadas objetivo.

Para que el robot siga el camino marcado y no se "desvíe" utilizamos la variable ángulo, que obtiene el ángulo de visión del robot. Si el ángulo es muy pequeño, es decir, está entre -0.5 y 0.5, tiene enfrente el punto al que quiere llegar, por lo que sigue totalmente recto. Si por el contrario, se ha girado un poco y tiene el punto fuera de ese ángulo, gira sobre sí mismo (a la izquierda o a la derecha) hasta llegar al punto deseado. La velocidad de giro del robot irá variando según la distancia a la que se encuentre.