Prácticas de Robótica V-REP

Ángel J. Maldonado Criado

Contents

.

Repositorio github con el código desarrollado:

https://github.com/AngelJMC/khepera-simulation

Video resumen de la implementación de cada tarea:

https://youtu.be/HrijEFNS6Hw

1 Tarea 1: Experimento de control de posición de un robot móvil.

En primer lugar se ha obtenido el modelo diferencial del robot móvil, que permite controlar la velocidad angular de cada rueda según la velocidad lineal y angular del robot. Es necesario calcular la velocidad angular de las ruedas, ya que el actuador a controlar (motor) genera un movimiento angular.

$$\begin{aligned} \omega_r &= (2v - \dot{\theta}L)/(2r) \\ \omega_l &= (2v + \dot{\theta}L)/(2r) \end{aligned} \tag{1}$$

Siendo L la distancia entre ruedas, r el radio de las ruedas, v la velocidad lineal y $\dot{\theta}$ la velocidad angular del robot.

Las velocidades v y ω son calculadas a partir de las leyes de control establecidas para resolver el experimento "point stabilization". En esta tarea se han puesto en práctica dos leyes de control y los resultados de cada una han sido comparados usando los Índices de Comportamiento (IAE, ISE, ITAE, ITSE).

La primera ley de control (**regla básica**) implementada es la presentada en los apuntes de las prácticas. La velocidad lineal es directamente proporcional a la distancia a la que se encuentra el robot de la posición objetivo, el coeficiente K_p permite ajustar la ganancia según la ecuación 2. La velocidad angular depende del error del ángulo de orientación $\theta_e = \alpha - \theta$, siendo su máximo valor ω_{max} según la ecuación 3.

$$v = K_p \cdot d \tag{2}$$

$$\omega = \omega_{max} sen(\theta_e) \tag{3}$$

La segunda ley de control (**regla avanzada**) propuesta es una evolución de la primera, añadiendo ciertas condiciones en el cálculo de la velocidad lineal. Dependiendo de la distancia del robot al punto de origen y al punto de destino, se calculará la velocidad lineal de tres formas distintas.

El robot iniciará el movimiento con una velocidad mínima $Vmin$ y se irá incrementando hasta V_{max} mientras la distancia recorrida por el robot sea menor a k_i. Una vez el robot supera esta distancia, mantiene la velocidad máxima V_{max} hasta que se aproxima a una distancia menor de K_r al punto de destino. En este punto comienza a reducir linealmente la velocidad hasta llegar al destino con velocidad próxima a cero. La ecuación 4 recoge la nueva función para calcular la velocidad lineal. Para el cálculo de la velocidad angular se ha mantenido la misma ley de control.

$$v(d, d_{acum}) = \begin{cases} max(d_{acum} * V_{max}/K_i, V_{min}) & si \quad d_{acum} < k_i \\ d * (V_{max}/K_r) & si \quad d < K_r \\ V_{max} & si \quad d \geq K_r, d_{acum} \geq k_i \end{cases} \tag{4}$$

Para la comparativa se ha realizado la simulación para 4 puntos de destino distintos: P1-1,-1.5,0.1, P21.5, 1,0.1, P31,0,0.1 y P4-1,1.5,0.1

	Control 1				Control 2			
	IAE	ISE	ITAE	ITSE	IAE	ISE	ITAE	ITSE
P1	4109	6118	5454786	4755109	1602	2734	682275	920324
P2	2114	1749	2663957	1129565	944	1136	335423	312236
P3	1087	488	1303712	295960	411	283	112553	57166
P4	3449	4083	4810254	3334732	1182	1642	458564	513088

Se puede comprobar cómo los Índices de comportamiento determinan que la ley de control 2 es claramente mejor que la ley de control 1. Se ha considerado que el error es la distancia entre el robot y el punto objetivo en cada instante de tiempo de simulación.

1.1 Programación en Lua e integración con V-rep

El desarrollo de esta se ha realizado en el proyecto V-rep llamado *khepera-sim-1.ttt*. La implementación de las funciones, que son comunes en todas las prácticas, se ha realizado en ficheros externos a los child scripts de V-rep. Al crear librerías independientes, podemos cargarlas desde diferentes proyectos V-rep, escribir el código desde editores más agradables y realizar control de versiones sobre estos ficheros. Los métodos que implementan las leyes de control o el algoritmo de detección, entre otros, se han implementado en el módulo *robot.lua*

En el *child script* asociado al robot Khepera de V-rep se ha implementado el uso de los métodos asociados a la clase *robot*, así como su configuración y rutina de simulación. Para esta tarea, el robot es de tipo máster, es decir, se desplazará de su posición inicial a una posición objetivo. Por defecto, el robot usará la regla de control avanzada, para usar la regla de control básica se habilita la opción controlRule = "advance" al inicio del script. Además, mediante la función *setNewControlParameter(Wmax, Vmax, Vmin, Kr, Ki)* se actualizan los parámetros de control establecidos por defecto. Una vez configurado el robot con las opciones deseados, se inicia el bucle de simulación.

El método *getMotorSpeed(p_target)* calcula la velocidad de las ruedas para desplazar el robot de acuerdo a la regla de control establecida. Esas velocidades son pasadas a la función *setTargetVelocity(wr, wl)*, encargada se controlar la velocidad de los motores.

En el bucle de simulación se ha añadido además la función *calculateStatics (d_toTarget , ori_err)*, que calcula los índices de comportamiento en cada iteración. La simulación es detenida cuando se detecta que el robot ha llegado al punto de destino.

Los métodos implementados en la clase *robot* y que son usados en esta tarea son:

robot:new(type, id, theta, dist)

Este método es el constructor de clase desde el que se crea un objetivo de robot. Tiene 4 parámetros de inicialización.

Para esta tarea, queremos que el comportamiento sea de tipo maestro, es decir, el robot irá hasta el punto de destino indicado sin tomar otras consideraciones. Como solo hay instanciado

un robot Khepera, el identificador es nulo.

robot:setTargetVelocity(wr, wl)

Este método configura la velocidad de las rudas izquierda y derecha, que es pasada como argumento, a los motores.

Para evitar cambios bruscos de velocidad que produzcan inestabilidad en el robot, en esta función también se realiza un filtrado mediante el promedio de las últimas 15 medidas. La implementación del filtro se encuentra en la fichero auxlib.lua.

robot:getMotorSpeed(p_target)

Esta función es la encargada de calcular la velocidad a asignar a los motores del robot. Para esta tarea, la velocidad será calculada únicamente a partir de las reglas de control que se implementan en las funciones runBasicControlRule ( ) y runAdvanceControlRule ( ).

robot:runBasicControlRule ( )

Este método implementa las funciones descritas en la primera ley de control. A partir de la distancia hasta el objetivo y el error en orientación, el método devuelve la velocidad lineal y angular del robot.

robot:runAdvanceControlRule ( )

Este método implementa las funciones descritas en la segunda ley de control. Devuelve la velocidad lineal y angular del robot.

robot:setNewControlParameter(Wmax, Vmax, Vmin, Kr, Ki)

Esta función permite modificar los parámetros de la segunda ley de control para usar otros diferentes a los establecidos por defecto.

robot:setBasicControlRule( )

Esta función se emplea para usar la primera ley de control, en vez de la segunda ley, que es la usada por defecto.

robot:getDistanceToTarget(p_target)

Este método devuelve la distancia desde la posición actual del robot hasta el objetivo.

robot:getOrientationErrorToTarget (p_target)

Este método devuelve el error del ángulo de orientación que tiene el robot respecto del punto de destino.

2 Tarea 2: Control de posición con evitación de obstáculos.

En esta tarea se han añadido diferentes obstáculos al escenario y se ha implementado el algoritmo de Braitenberg para la evitación de obstáculos mediante la manipulación de las velocidades del robot.

En este algoritmo de evitación, los sensores del robot están directamente "conectados" a sus motores, por lo que las velocidades se calculan directamente en función de los valores de los sensores. Para implementar este algoritmo se crea una matriz de pesos de [MxS] dimensiones, donde M representa el número de motores (2) y S el número de sensores (5) en este caso. Cada valor de la matriz representa el peso de cada sensor en el motor correspondiente.

El robot Khepera IV dispone de 8 sensores de infrarrojos para detección de obstáculos cercanos (25cm max) y 5 sensores de ultrasonidos de largo alcance (2m max). Para esta tarea se han elegido los sensores infrarrojos y se han descartado los 3 sensores traseros. Así, tenemos 5 sensores infrarrojos dispuestos tal y como se muestran en la siguiente figura.

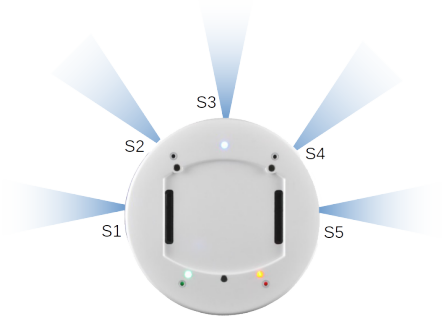

Figure 1: Posicionamiento sensores infrarrojos.

Los valores de la matriz de pesos se determinan empíricamente dependiendo de la posición de cada sensor en el robot (WIS1 representa la influencia del sensor 1 en la velocidad del motor izquierdo). Las velocidades de los motores derecho e izquierdo se obtienen de la siguiente forma:

$$v_{D,I} = W * (\frac{S}{S_{max}}) \tag{5}$$

Donde S_{max} es el valor máximo de los sensores y S es un vector que contiene el valor actual de cada sensor. En nuestro caso, el sensor devuelve la distancia a la que se detecta el obstáculo, por lo que este valor decrece según el robot se aproxima a él. La distancia máxima de detección $S_{max} = 0.25$

$$S = (S_1, ...S_5)^T \tag{6}$$

Mientras el robot no detecta obstáculos, se dirige hacia un punto destino ejecutando la segunda ley de control desarrollada en la Tarea 1. Al detectar un obstáculo, el robot pasa a ejecutar el algoritmo de evitación de obstáculos (según las ecuaciones 5 y 6).

2.1 Programación en Lua e integración con V-rep

El desarrollo de esta se ha realizado en el proyecto V-rep llamado *khepera-sim-2.ttt*. En el child script del robot Khepera se ha implementado el thread que ejecuta los diferentes algoritmos de control.

Se emplea el método *enableDetectionAlgorithm( )* implementado en la clase "robot" para habilitar el algoritmo Braitenberg de evitación de obstáculos. Como ya se ha comentando en el apartado anterior, el algoritmo de evitación de obstáculo se activa en el momento que detecta un obstáculo, el resto del tiempo el robot ejecuta la ley de control avanzada hasta llegar al punto de destino.

Los nuevos métodos añadidos a la clase "robot" necesarios para implementar el algoritmo Braintenberg son:

robot:enableDetectionAlgorithm( )

Habilita el algoritmo Braitenberg de evitación de obstáculos. Al estar habilitada la función getMotorSpeed(p_target), tendrá en cuenta los obstáculo próximos en el cálculo de las velocidades angulares de las ruedas.

robot:isAnyDetection( )

Devuelve *true* si se detecta un obstáculo en cualquiera de los 5 sensores infrarrojos habilitados.

robot:getDistanceVector( )

Devuelve un array con la distancia de detección de cada sensor. Si el sensor no detecta, este devuelve la distancia máxima de detección $S_{max} = 0.25$

robot:calculateBraitenbergAlgorithm(Sdist)

Este método toma el vector de distancias y obtiene el factor de proximidad ($Sdist/S_max$), que es multiplicado por la matriz de pesos W. El método devuelve la velocidad resultante a aplicar en cada motor.

3 Tarea 3: Control de formaciones con evitación de obstáculos.

El objetivo de esta tarea es implementar el control de formación, tanto en modo cooperativo como en no cooperativo, empleando para ello 4 robots Khepera.

La formación consiste en dirigir al maestro a un destino, al mismo tiempo que envía continuamente su posición al resto de los robots. Los esclavos toman la posición del maestro como referencia para alcanzar un punto a una distancia y orientación constante de éste. Para llevar a cabo la formación, cada robot implementa el "point stabilization", el maestro sobre el punto de destino y los esclavos sobre un punto relativo a la posición actualizada del maestro. La formación establecida para esta tarea es la mostrada en la figura 2.

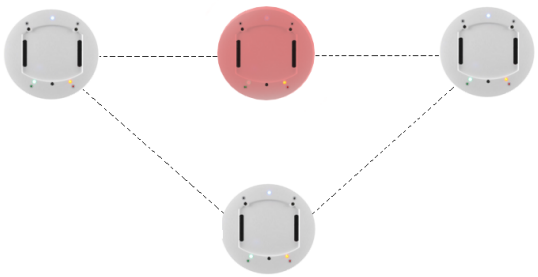

Figure 2: Formación de 4 robots.

En el control no cooperativo, el robot maestro calcula su velocidad solo teniendo en cuenta su error de posición. Por otro lado, en el control cooperativo el robot maestro tiene en cuenta tanto la posición de los esclavos en la formación como su propio error en posición para calcular su velocidad.

La ecuación 7 muestra el error de posición total de la formación que se expresa como el sumatorio del error de cada uno de los robots en la formación.

$$E_f(t) = \sum_{i=1}^{N} E_{pi}(t) \tag{7}$$

En la ecuación 8 se muestra el cálculo de la velocidad del maestro ($v_m(t)$) teniendo en cuenta su error de posición (aplicando el control avanzado de la ecuación 4) y el error de formación (Ef) multiplicado por el factor (Kf). Este factor indicará el nivel de cooperación: cuando este coeficiente es 0 el control es no cooperativo y cuando tiene un valor muy alto, la velocidad del maestro dependerá en mayor medida del error de posición de los esclavos [1].

$$v_m = v(d, d_{acum}) - K_f E_f(t) \tag{8}$$

A partir de los experimentos que se han realizado, se ha podido comprobar cómo al habilitar el modo no cooperativo, la formación es menos estable. Es decir, durante la mayor parte del trayecto, hasta que el robot maestro alcanza su destino, no se cumple la formación. Por otro lado, al aumentar el factor de cooperación se vuelve predominante ir en formación a cambio de reducir la velocidad de movimiento de todos los miembros del equipo: el maestro reduce su velocidad para esperar al resto de miembros del equipos cada vez que uno de ellos se sale de la formación.

3.1 Programación en Lua e integración con V-rep

El desarrollo de esta se ha realizado en el proyecto V-rep llamado *khepera-sim-3-simple.ttt*.

En este caso tenemos dos comportamientos distintos, programados en los child scripts de cada Khepera. Si el robot es de tipo "slave" será necesario pasarle al constructor la distancia y el ángulo respecto del maestro, en la que localizará su punto de estabilización.

Por ejemplo, robot:new('slave', robotID, math.pi, 0.5), el nuevo robot será de tipo slave y su posición estará a una distancia de 0.5 m y orientado 90º respecto del robot maestro.

El coeficiente de cooperación es configurado en el child script del robot Khepera maestro.

A continuación se listan los nuevos métodos implementados en la clase "robot".

robot:getSlaveTargetPos( )

Esta función es la encargada de calcular la posición objetivo que debe alcanzar el robot de tipo slave. Para ello, en primer lugar recibe la posición y orientación del robot maestro. Conociendo la distancia y ángulo relativo al que debe situarse el esclavo respecto del maestro, calcula la posición absoluta que debe ser alcanzada.

Esta posición objetivo es la que toma el robot de tipo slave para implementar el "point stabilization".

robot:getSlavesError(nslaves)

Esta función es utilizada por el maestro e implementa la ecuación 7- Es decir, obtiene el error total de la formación sumando el error en posición transmitido por cada uno de los esclavos.

robot:setCooperativeFactor(value)

Esta función permite establecer el factor de cooperación. Por defecto el valor es 0, por lo que se aplica el modo no cooperativo. Al establecer un valor mayor a 0, se habilita el modo cooperativo y se trabaja con el factor indicado.

3.2 Agregar drone a la formación

EL objetivo de esta tarea es agregar un dron a la formación y utilizar la cámara de a bordo para controlar la posición del dron encima del Khepera, por medio del procesamiento de la imagen que entrega la cámara del dron.

El dron parte de una base sin tener visión del robot Khepera líder. En primer lugar, el dron asciende verticalmente hasta contactar visualmente con el robot Khepera. En ese momento, la posición del dron en el plano horizontal es controlada de forma visual mediante la técnica de Visual Servoing. Además, mientras el dron detecta visualmente su objetivo, se reduce su altura para realizar el seguimiento desde un punto más próximo.

El servo visual (Visual-servoing), también conocido como control de robot basado en visión, es una técnica que utiliza la información extraída de un sensor de visión para controlar el movimiento de un robot. Las técnicas de control de Servoing Visual se clasifican a grandes rasgos en los siguientes tipos:[2][3]

- **IBVS (Image-based visual servoing)** fue propuesto por Weiss y Sanderson [4]. Para la ley de control, se emplea el error existente entre las características actuales y las deseadas en el plano de la imagen, y no implica ninguna estimación de la posición del objetivo. El robot debe moverse hasta que las características actuales de la imagen coincidan con las deseadas.

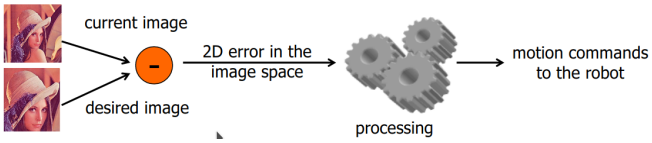

Figure 3: Esquema de control IBVS.

- **PBVS (Position-based visual servoing)** es una técnica basada en modelos. La posición del objeto de interés se estima con respecto a la cámara y luego se emite un comando de control del robot en el espacio cartesiano (control del error en posición). En este caso también se extraen las características de la imagen, pero se utilizan para estimar la información 3D (pose del objeto en el espacio cartesiano).

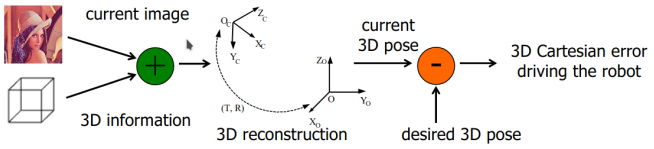

Figure 4: Esquema de control PBVS.

En el caso que estamos tratando, se ha optado por implementar un esquema IBVS. Esta implementación es más simple, ya que no requiere del conocimiento de los parámetros de calibración de la cámara, además resuelve perfectamente nuestro problema: controlar el movimiento horizontal del dron para que el robot Khepera esté centrado en la imagen. Para ello, la característica extraída de la imagen es la posición que ocupa el robot Khepera.

En este caso, el dron solo hace seguimiento del robot Khepera maestro. El error en formación del dron no es tenida en cuenta en el modo cooperativo.

3.3 Programación en Lua e integración con V-rep

El desarrollo de esta se ha realizado en el proyecto V-rep llamado *khepera-sim-3-drone.ttt*.

Se ha creado un nuevo módulo llamado "quadcopter.lua" en el que se ha implementado una nueva clase llamada quadcopter. Esta contiene los métodos empleados para el procesamiento de la imagen y las leyes de control de posición del dron.

Por otro lado, en el child script del dron (drone.lua), se implementan las llamadas a los métodos de la clase quadcopter, a partir de los cuales podemos calcular las velocidades de cada motor del dron. En este script se generan también los efectos visuales del vuelo del dron.

Por último, en el child script del sensor de visión montado en el dron, se ha implementado el pre-procesamiento de la imagen, es decir, el uso de las funciones que implementa la detección de la región de color rojo (Khepera maestro).

A continuación se listan todos los métodos implementados en el módulo *"quadcopter"*.

quadcopter:cameraDetectKhepera(cam)

Esta función se encarga de procesar la información de la cámara de abordo del dron. Para poder identificar el Khepera maestro con la cámara, se ha cambiado el color de la carcasa a rojo. Empleando los algoritmos de Vision que integra V-rep para la detección de regiones (child scripr del sensor de visión), se logra detectar perfectamente la posición relativa del robot en el frame de la imagen.

El primer parámetro que devuelve la función es una variable booleana que indica si se ha detectado el robot Khepera, y el segundo parámetro es un array con la posición del centro del robot en la imagen.

quadcopter:calculateAltitude(detection)

Esta función realiza el control en altitud del dron. Toma como argumento de entrada la variable de detección del robot Khepera maestro. Si no se ha detectado, se establece una altura objetivo del dron de 4m, de forma que al alcanzar esta altura pueda tener visión de toda el campo y así observar al dron maestro.

En el momento en que el dron está dentro del campo de visión y es detectado, la altura de referencia de seguimiento es de 0.4 m.

quadcopter:calculateRotation(detection, rTocenter)

Esta función es la encargada de realizar el control de rotación del dron. El objetivo es que el dron gire en el eje Z hasta alinearse con su propia dirección de avance, para ello, filtrando el valor de los incrementos de posición, se obtiene una estimación de la dirección de avance. Este valor se emplea como consigna para el ángulo yaw del dron.

quadcopter:calculatMovement(detection, imgPos)

Este método calcula el movimiento en el plano x,y que debe realizar el dron una vez se ha detectado el objetivo, para aproximarse y colocarse encima de este.

Implementa el algoritmo de servo visual basado en imagen (IBVS) donde la característica objetivo es que la posición del dron ocupe el centro de la imagen. Sobre esta referencia se calcula el error y se genera el desplazamiento deseado en el plano horizontal.

quadcopter:updateThrust(altitude, targetAlt, l)

Esta función implementa las reglas de control de altura en base a la consigna establecida y a la altura real del dron. Devuelve el factor de empuje de los motores.

quadcopter:updateHorizontalCorrection(m, sp

Implementa el control en bucle cerrado de la posición real del dron y la posición objetivo en el plano horizontal. Devuelve el factor de corrección para el control de los motores.

quadcopter:updateRotation(targetRot)

Implementa las reglas de control para la rotación del dron en base a su posición actual y la posición objetivo. Devuelve el factor de corrección de rotación para el control de los motores.

4 Tarea 4: Simultaneous localization and mapping (SLAM)

En esta tarea varios robots deben navegar por el entorno y construir un mapa del mismo, que debe ser mostrado y almacenado en memoria.

Para la construcción del mapa virtual del entorno se ha empleado la técnica basada en mapa de rejillas probabilístico. Esta consiste en dividir el entorno en una serie de celdas uniformes cuyo contenido es un valor de probabilidad de ocupación o probabilidad de existencia de un obstáculo. Su actualización se lleva a cabo aplicando la teoría bayesiana (ecuación 9), que nos permite fusionar la información percibida por los sensores con la existente previamente en el mapa.

$$P(m_{ij}|z_{1:t}) = \frac{P(m_{ij}|z_t) \cdot P(m_{ij}z_{1:t-1})}{P(m_{ij}|z_t) \cdot P(m_{ij}z_{1:t-1}) + (1 - P(m_{ij}|z_t)) \cdot (1 - P(m_{ij}z_{1:t-1}))} \tag{9}$$

Las probabilidades que tenemos son:

- $P(m_{ij}|z_t)$.- Probabilidad de que las celdas m_{ij} estén ocupadas cuando el sensor realiza una medida en el instante t.
- $P(m_{ij}|z_{1:t-1})$.- Probabilidad de que estén ocupadas las celdas m_{ij} sabiendo las medidas realizadas desde el instante 1 al t-1 (es la calculada en el paso anterior).

Aplicamos la fórmula de Bayes de modo recursivo suponiendo que la probabilidad a priori es la que tenemos antes de la nueva medida, y para el caso inicial en el que no hay medida $P(m_{ij}|z_0) = P(m_{ij}) = 0.5$.

La mayor ventaja de este tipo de mapas es que son fáciles de construir y de mantener, incluso en entornos grandes. Por otro lado, puesto que la geometría de las celdillas se corresponde con la geometría del terreno, es fácil para un robot determinar su posición dentro del mapa tan sólo conociendo su posición y orientación en el mundo real. Además, sobre este tipo de mapas se pueden implementar algoritmos para el cálculo de trayectorias óptimas y evitación de los obstáculos conocidos.

Sin embargo, adolece de un problema básico, la gran cantidad de memoria necesaria para el almacenamiento de la información y la dificultad de extraer información elaborada. Esto es debido a que la resolución del mapa (A), debe ser lo suficientemente fina para capturar la información importante del entorno.

En esta tarea se ha usado un mapa de rejilla de 64x64 cuadrículas, dado que las dimensiones de la superficie son 5.2x5.2m, cada celda corresponde a una dimensión aproximada de 81x81mm.

Hay un mapa de rejilla único que es actualizado con la información de los 5 sensores de cada robot Khepera. En cada nueva medida se aplica la ecuación 9 para fusionar los datos del sensor con la última información de mapa.

En cuanto a la navegación, los robots se mueven de forma independiente, a una velocidad lineal constante. Al topar con un obstáculo realizan las maniobras de evitación determinadas por el algoritmo de Braiteberg, y continúa la exploración avanzando en la última dirección dada al rebasar el obstáculo.

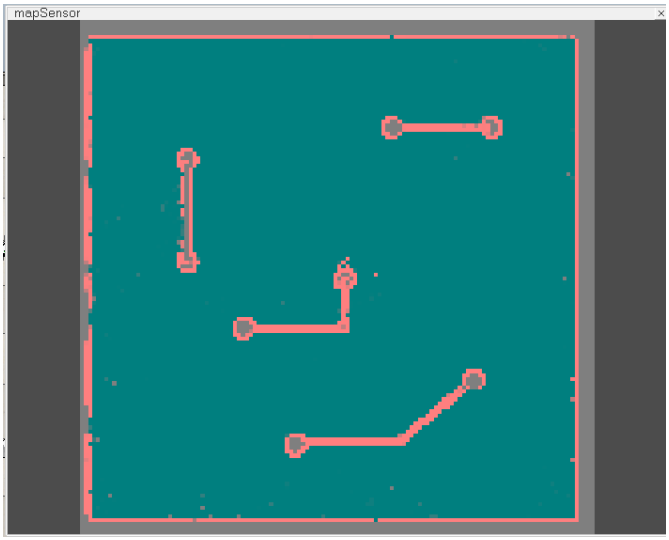

Figure 5: Mapa de rejilla obtenido con una resolución de 128x128.

Este método de exploración es el más simple, sin embargo, es poco eficiente, ya que dejamos al azar la exploración de las zonas. Una forma más eficiente de exploración se podría implementar enviando al robot a zonas no exploradas, es decir, mediante la monitorización del mapa se detecta una zona no explorada y en el centro de esa zona se coloca un punto de destino para uno de los robots. Mediante el algoritmo de point stabilization, el robot se dirige a ese punto

mapeando el ambiente hasta llegar a él. Una vez alcanzado, se busca una nueva zona a explorar y se repite el mismo procedimiento.

4.1 Programación en Lua e integración con V-rep

El desarrollo de esta se ha realizado en el proyecto V-rep llamado *khepera-sim-4.ttt*.

Para esta tarea se ha desarrollado un nuevo módulo llamado "slam.lua", el cual implementa los métodos encargados de crear el mapa de rejilla y actualizarlo a partir de la información proporcionada por los sensores de cada uno de los robots.

Los métodos relacionados con el algoritmo slam son llamados desde la rutina implementada en el child script de mapSensor. En esta rutina se recibe la información de los sensores de cada uno de los robots, y a partir de esta, se actualiza el mapa de rejilla en cada ciclo de ejecución de la simulación.

Para poder visualizar el mapa se ha empleado un sensor de visión, el cual se ha configura para recibir una entrada externa, es decir, la información que recibe el sensor es la imagen obtenida a partir del mapa de rejilla. De esta forma, si se desea cambiar la resolución del mapa de rejilla, se puede hacer desde la propiedades del sensor de visión (mapSensor).

slam:new(mapDimensions, numCells)

Este es el método constructor de la clase slam. Se le pasa como argumentos un array con las dimensiones reales del terreno y el número de celdas (resolución) del mapa de rejilla.

slam:transformCoordinatesToGrid(coord)

Este método se encarga de transformar las coordenadas cartesianas (x,y) de un punto real del mapa a los índices de la matriz que contiene la información del mapa de rejilla.

slam:updateGridMap(sensorIDs)

Esta función se encarga de actualizar el mapa de rejilla a partir del conjunto de datos obtenidos de los sensores de cada robot. Internamente el método itera sobre cada robot Khepera y sobre cada sensor IR. En el caso de detectar un obstáculo, se obtienen las coordenadas cartesianas de dicho punto y se transforman a las coordenadas del mapa de rejilla. Si hay detección la probabilidad que se le asigna a la medida es $0.5 + 0.5/Kd$, siendo el valor de Kd = 2. En el caso de no detección el valor de probabilidad asignado es de 0.1.

Con estos valores se actualiza el mapa de rejilla mediante la aplicación de la teoría bayesiana, ecuación 9.

slam:getGrid ( )

Esta función retorna la matriz que contiene el mapa de rejilla.

5 Bibliografía

[1] Jonathan R. T. Lawton, Randal W. Beard, Brett J. Young. A Decentralized Approach to Formation Maneuvers. IEEE Transactions on Robotics and Automation, vol. 19, no. 6, 2003.

[2] S. A. Hutchinson, G. D. Hager, and P. I. Corke. A tutorial on visual servo control. IEEE Trans. Robot. Automat., 12(5):651–670, Oct. 1996.

[3] F. Chaumette, S. Hutchinson. Visual Servo Control, Part I: Basic Approaches. IEEE Robotics and Automation Magazine, 13(4):82-90, December 2006

[4] A. C. Sanderson and L. E. Weiss. Adaptive visual servo control of robots. In A. Pugh, editor, Robot Vision, pages 107–116. IFS, 1983