uc3m | Universidad Carlos III de Madrid

Máster Universitario en Robótica y Automatización
Curso académico 2018-2019

Práctica Opcional Robots Móviles

SLAM: Localización de robots y construcción simultánea de mapas

---

Laura Martín Gallardo (100330673)

Profesor: Ramón Ignacio Barber Castaño
Luis Santiago Garrido Bullon

Leganés, 12 de abril de 2019

ABSTRACT

En este trabajo se muestran los resultados obtenidos de la ejecución de distintos proyectos o *toolboxes*, cada uno de ellos con implementaciones alternativas para solucionar el problema de la localización y mapeo simultáneos de un entorno desconocido.

Una vez obtenidos los mapas finales de dichos mapeos se realiza un análisis para poder determinar cuales de ellos resuelven con mayor eficiencia dicho problema. Teniendo en cuenta en todo momento las condiciones bajo las que trabaja cada uno de los algoritmos, así como el entorno sobre el que se va a realizar el mapeo, ya que no es lo mismo un entorno sencillo sin obstáculos, que uno con multitud de obstáculos y recovecos.

Palabras clave: *SLAM, Nearest Neighbour, Joint Compatibility Branch and Bound, SINGLES, FastSLAM, Filtro de Kalman*.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

1. INTRODUCCIÓN

En el ámbito de los robots móviles, el SLAM es un problema computacional que consiste en la creación y actualización incremental del mapa de un entorno desconocido, mientras que utiliza dicho mapa para determinar su posición en el entorno.

Este problema utiliza como herramienta principal un robot móvil que extrae la información del entorno a través de unos sensores (láseres, sonsares, cámaras ...) e interactúa con él mediante unos actuadores.

Sin embargo, el ruido captado por los sensores, los errores producidos tanto por dichos sensores como por las aproximaciones que se realizan y la dificultad de representar entornos complejos, suponen que resolver este problema sea una tarea ardua.

Existen diversas tendencias a la hora de tratar de solucionar el problema de SLAM, entre ellas destacan los algoritmos basados en el filtro de Kalman, el filtro de Kalman extendido y el filtro de partículas, que son precisamente los que utilizan los algoritmos que cuyos resultados se comentan en este trabajo.

Además cabe destacar que aunque todos los algoritmos que se muestran en esta práctica se basan en el uso de *landmarks* para poder localizarse en el espacio, hay otros algoritmos de SLAM que utilizan las distancias a las paredes, suelos, etc, para realizar dicha localización.

En los siguientes capítulos se exponen los resultados obtenidos de la ejecución de los algoritmos de cinco proyectos diferentes, cada uno de ellos creado por distintos autores.

2. SLAM SALVI

En este capítulo se exponen los resultados obtenidos al ejecutar cuatro de los programas disponibles en la carpeta *Slam-Salvi*. Concretamente, los que se han seleccionado son aquellos que conllevan el uso de un robot móvil con 2 grados de libertad y que, por tanto, se mueve en un entorno de 2 dimensiones.

Este robot se mueve a lo largo de un *path* definido, intentando detectar los *landmarks* establecidos. Mediante SLAM se calculan tanto la posición y velocidad del robot como la posición de los landmarks. Y para el mapeado se utiliza el Filtro de Kalman Lineal.

En cada paso de este algoritmo se realizan dos cosas: localizarse y mejorar la estimación de donde se encuentran los *landmarks*, de ahí que al principio haya mucho más error y a medida que se avanza, dicho error se va reduciendo.

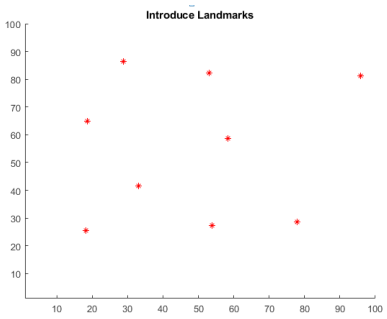
En las siguientes secciones se explica cual es la característica que diferencia cada uno de los cuatro programas seleccionados, así como los resultados obtenidos de cada uno de ellos.

2.1. Slam2D1

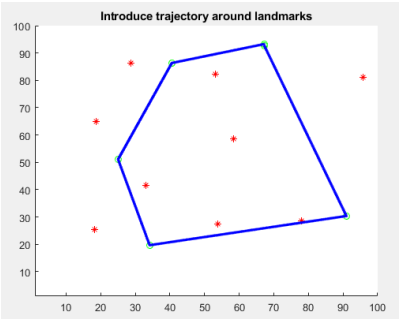
Este primer programa se caracteriza por tener un rango de medida del sensor ilimitado.

Para ejecutar este programa, lo primero que se debe hacer es seleccionar los *landmarks* que se desean establecer, es decir, que se debe hacer clic izquierdo en distintas posiciones de la figura para definir cada uno de los *landmarks*, como se puede observar en la Figura 2.1a. Una vez definidos los puntos se pasa al siguiente paso haciendo clic derecho, lo que le indica al programa que ya no se desean poner más *landmarks*.

A continuación, se debe establecer la trayectoria que va a seguir el robot móvil durante el mapeado, para ello se van estableciendo distintos puntos en la figura, y el propio programa se encarga de ir uniéndolos para crear dicha trayectoria, como se puede ver en la Figura 2.1b.

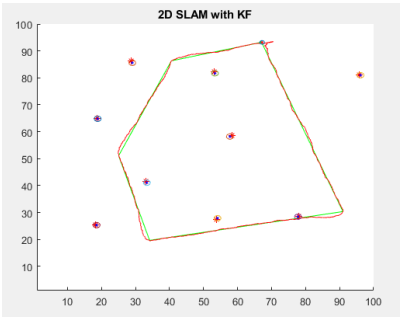


(a) Landmarks

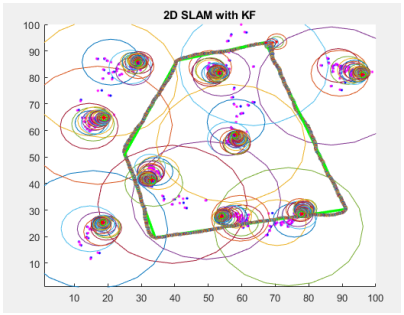


(b) Trayectoria

Hechos esos dos pasos, el programa comienza a mover el robot siguiendo la trayectoria indicada, tratando a su vez de localizar los *landmarks* más cercanos al punto actual para poder así realizar el mapeado del entorno, como se vería en las Figuras 2.1c y 2.1d.



(c) Mapeado



(d) Mapa Final

Fig. 2.1. Resultado de la ejecución del programa Slam2D1.

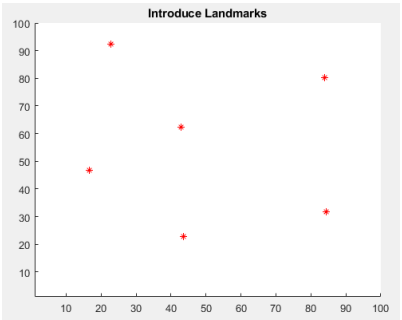
Durante la ejecución de este algoritmo se puede ver como se van reduciendo las elipses a medida que el robot va avanzando, lo cual implica una mejora en la localización.

En este caso además, se puede ver como el mapa obtenido se ajusta bastante a la trayectoria establecida (Figura 2.1d).

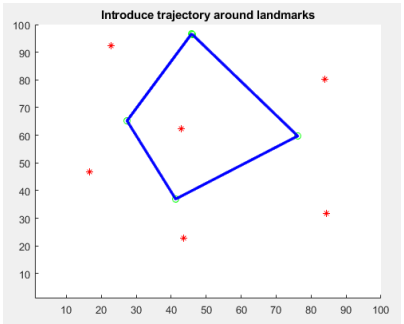
2.2. Slam2D2

Este segundo programa se caracteriza por tener un rango de distancias del sensor limitado a una distancia predefinida, a diferencia del caso anterior. Esto puede suponer bastantes problemas, sobretodo en el caso de que dicho rango sea muy pequeño.

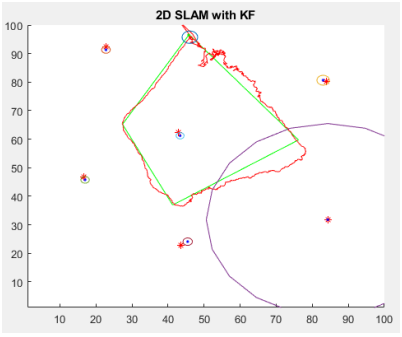
El procedimiento a seguir para ejecutar este segundo programa es el mismo que en el caso anterior y los resultados son los que se pueden ver en las Figuras 2.2a, 2.2b, 2.2c y 2.2d.



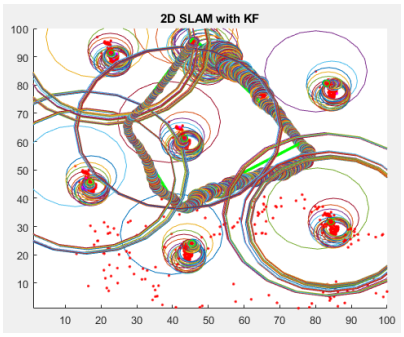
(a) Landmarks



(b) Trayectoria



(c) Mapeado



(d) Mapa Final

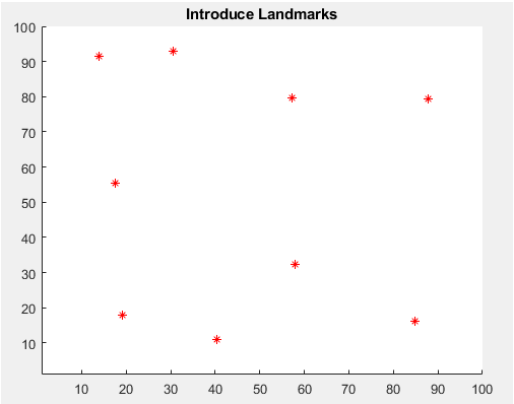
Fig. 2.2. Resultado de la ejecución del programa Slam2D2.

En este caso se puede ver como al establecer un límite en el rango de distancias que puede detectar el sensor, el robot se desvía más de la trayectoria establecida, dando lugar a un mapa menos exacto, como se puede observar si se comparan las Figuras 2.1c y 2.1d con las Figuras 2.2c y 2.2d.

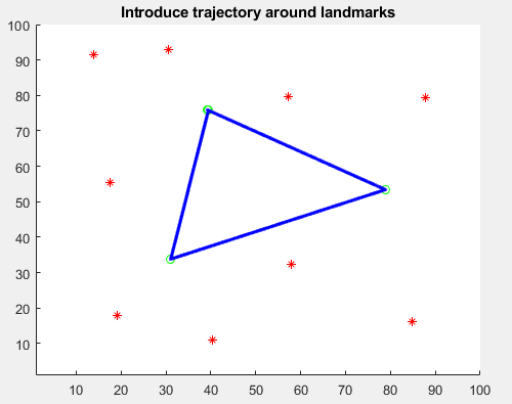
2.3. Slam2D3

En este tercer programa a diferencia de los dos anteriores, solo se tienen en cuenta las observaciones de la velocidad del robot y los *landmarks* establecidos, es decir que se deja de obtener información acerca de la trayectoria que debe seguir. De forma que ahora necesitamos tener un observador y las medidas pasan a ser relativas a la posición del robot.

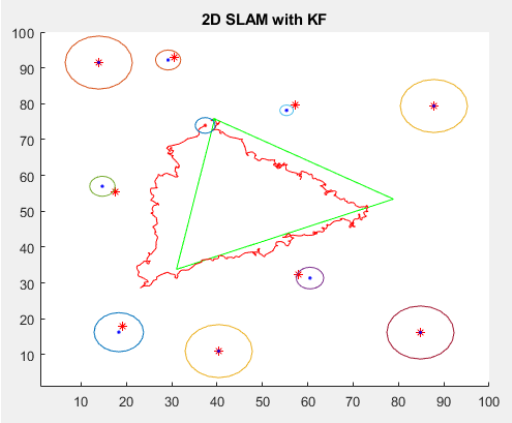
El procedimiento a seguir para ejecutar este tercer programa es el mismo que en los dos casos anteriores y los resultados son los que se pueden observar en las Figuras 2.3a, 2.3b, 2.3c y 2.3d.



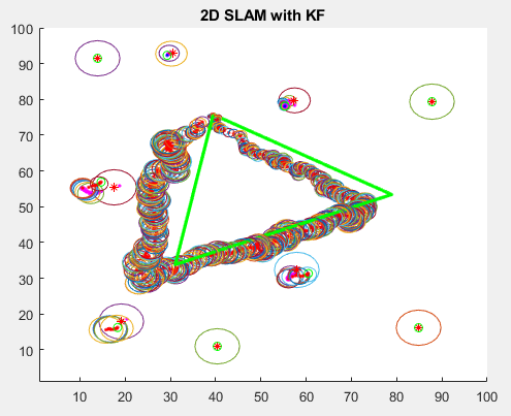
(a) Landmarks



(b) Trayectoria



(c) Mapeado



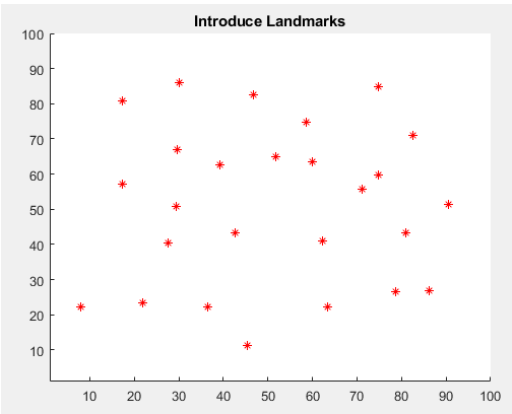
(d) Mapa Final

Fig. 2.3. Resultado de la ejecución del programa Slam2D3.

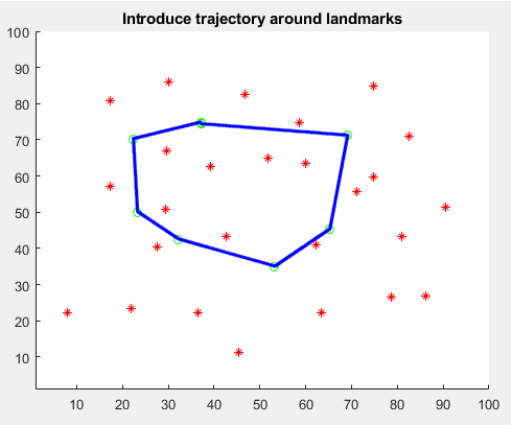
En este caso se puede ver como a pesar de que la línea roja, que corresponde con el recorrido que hace el robot, tiene más o menos la misma forma que la trayectoria establecida, se encuentra desplazada a la izquierda (Figura 2.3b). Este hecho provoca que el mapa final generado esté también desplazado y por tanto, la solución producida por este programa es peor que las obtenidas de los dos programas anteriores.

2.4. Slam2D4

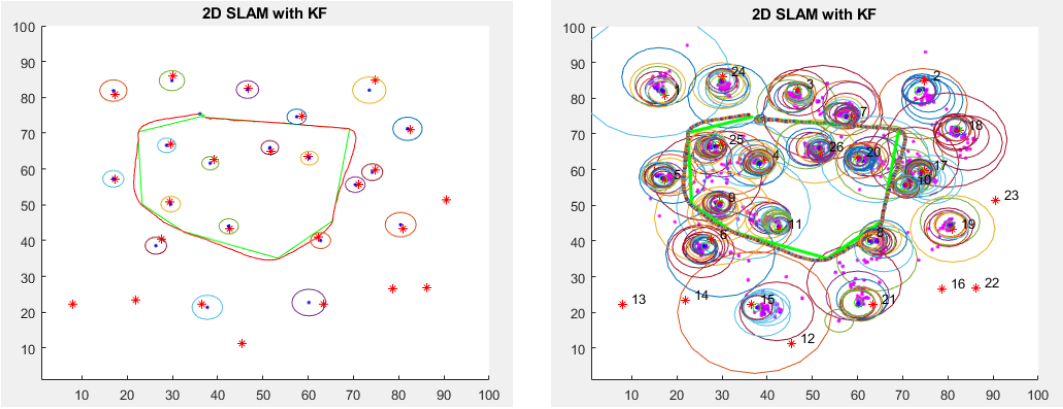
Por último, este cuarto programa, es muy similar al segundo salvo por que los *landmarks* observados por el robot mientras va recorriendo el *path* se van actualizando a medida que se detectan.



(a) Landmarks



(b) Trayectoria



(c) Mapeado (d) Mapa Final

Fig. 2.4. Resultado de la ejecución del programa Slam2D4.

En este caso como se puede observar en las Figuras 2.4c y 2.4d, tanto la trayectoria seguida por el robot como el mapa resultante se ajustan bastante bien a la trayectoria establecida. De forma que este programa obtiene resultados equivalentes al primer programa pero sin condiciones ideales, ya que este sí que tiene la limitación del rango del sensor. Esto quiere decir que el hecho de que los *landmarks* se vayan actualizando a medida que son observados por el robot atenúa el error producido por la limitación del rango del sensor.

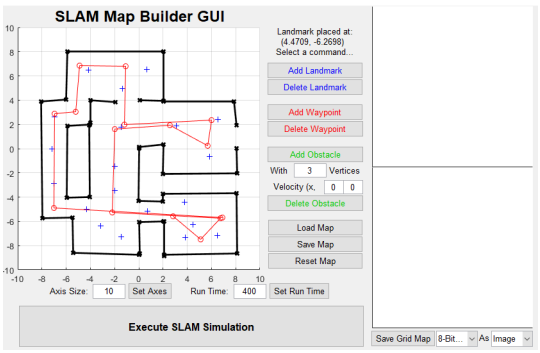
3. EFK SLAM

En este capítulo se exponen los resultados obtenidos al ejecutar el programa de la carpeta *efk-slam-matlab-master*. Para ello se abre el *mapMakerGUI* poniendo en línea de comandos *setup*. Una vez abierto, se selecciona de la carpeta *sample-maps* el entorno que se desea mapear, en este caso se ha escogido *map_largefloor*, como se puede ver en la Figura 3.1a.

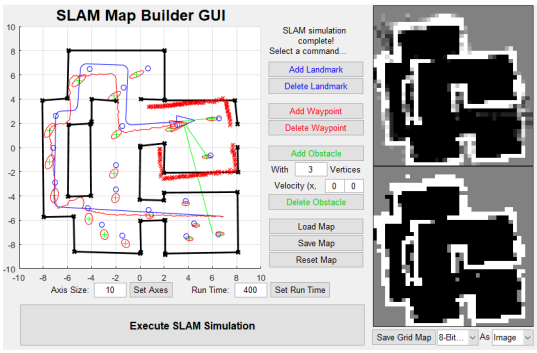
Seleccionado el entorno, se pueden añadir nuevos *landmarks* (cruces azules), *waypoints* (círculos y líneas rojas) u obstáculos (cruces y líneas negras) a los que viene por defecto en el entorno. Para ello simplemente se debe seleccionar el botón correspondiente de la parte derecha del *mapMakerGUI* y posteriormente hacer clic en la localización del mapa en la que se desea establecer.

Una vez establecidos los *landmarks, waypoints* y obstáculos adicionales o en el caso de que no se quieran añadir, se pasa a la creación del mapa, para lo que se debe hacer clic en el botón situado en la parte inferior (Execute SLAM Simulation).

Durante la ejecución, el robot se mueve siguiendo la trayectoria formada por los *waypoints* tomando como referencias los *landmarks* establecidos y tratando de evitar los obstáculos para conseguir crear el mapa del entorno. Una vez finalizada dicha ejecución se obtiene el mapa resultante, que es el que se puede observar en la Figura 3.1b.



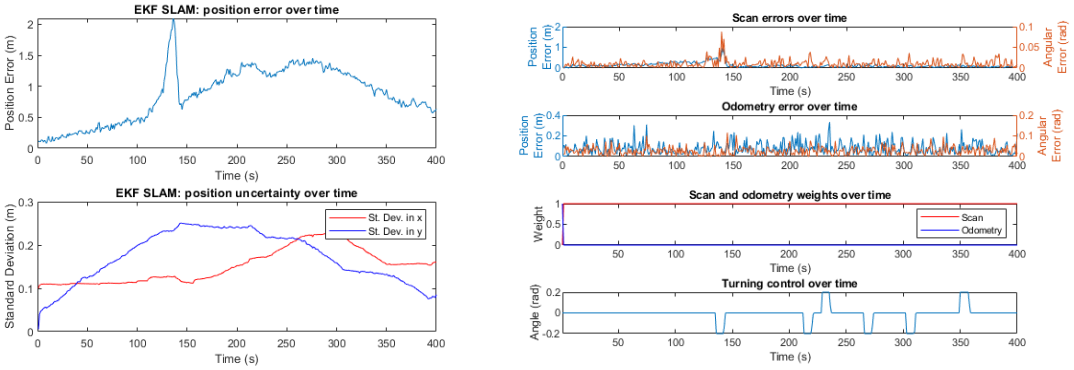
(a) Entorno



(b) Mapa Final

Fig. 3.1. Resultado de la ejecución del programa EFK Slam.

Además de los resultados ya mostrados, el programa también devuelve una serie de gráficas de los errores obtenidos durante la ejecución en cuanto a la posición, el escáner y la odometría, que son las que se pueden observar en las Figuras 3.2a y 3.2b.



(a) Entorno (b) Mapa Final

Fig. 3.2. Errores obtenidos de la ejecución del programa EFK Slam.

4. SLAM BAILEY

En este capítulo se exponen los resultados obtenidos de la ejecución de tres de los algoritmos disponibles en la carpeta *Slam Bailey*. En esta *toolbox* podemos encontrar dos algoritmos basados en el filtro extendido de Kalman, otro basado en *Unscented Kalman Filter* y tres algoritmos basados en el filtro de partículas. Sin embargo, los algoritmos seleccionados son los de FastSLAM, que son los que resuelven el problema de SLAM basándose en el filtro de partículas.

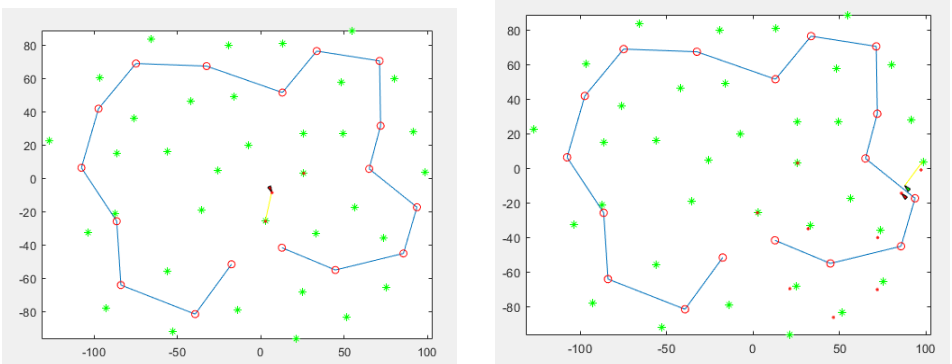
En las siguientes secciones se explica la diferencia entre los tres algoritmos y los resultados obtenidos con cada uno de ellos.

4.1. fastslam1_sim

El FastSLAM 1.0 fue la primera versión que se desarrolló para el problema de SLAM, aunque tiene algunas carencias.

Para lanzar este algoritmo se pueden seguir dos procedimientos: definiendo los *landmarks* y *waypoints* de forma manual, para poder seleccionar los que se quieran o cargando la matriz *example_webmap* que ya los tiene definidos (lm, serían los *landmarks* y wp serían los *waypoints*).

A continuación se muestran los resultados obtenidos de la ejecución del algoritmo *fastslam1_sim*.

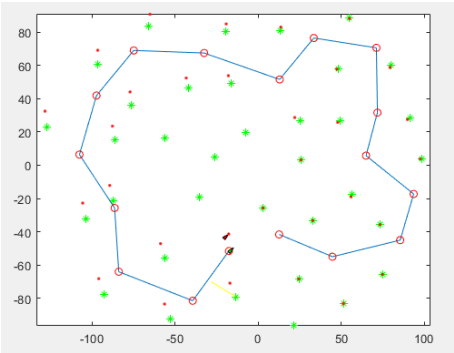


(a) Entorno (b) Proceso

En la Figura 4.1a, se muestra el entorno que se va a mapear, en el que se pueden distinguir los *landmarks* que va a usar el robot móvil para localizarse, la trayectoria que debe seguir y la localización inicial del propio robot.

En la Figura 4.1b, se muestra como el robot móvil (triangulo rojo) trata de seguir la trayectoria apoyándose en los dos *landmarks* más cercanos. Sin embargo, tiene un

pequeño error ya que el piensa que va por la línea azul, pero en realidad está ligeramente desplazado.



(c) Entorno tras la ejecución

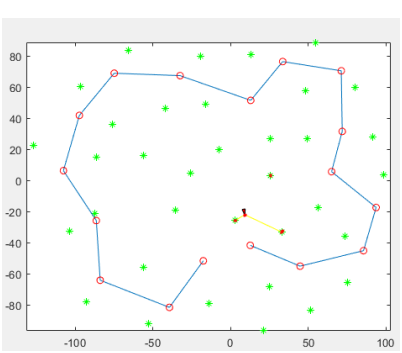
Fig. 4.1. Resultado obtenido de la ejecución del algoritmo FastSlam1.

Y por último en la Figura 4.1c, se puede observar el entorno tras realizar la ejecución completa del algoritmo. Además se pueden ver los *landmarks* que ha observado el robot móvil, ya que han quedado marcados con un punto rojo. En esta se aprecia como a medida que va avanzando el robot, se va separando cada vez más de la línea azul, ya que acumula error, y lo mismo pasa con los *landmarks*, la posición real de estos son las estrellas verdes y la posición en la que el robot piensa que están son los puntos rojos. Como se puede ver al principio esos puntos rojos estaban prácticamente encima de la estrella verde, pero a medida que va llegando al final se van separando cada vez más de la posición real.

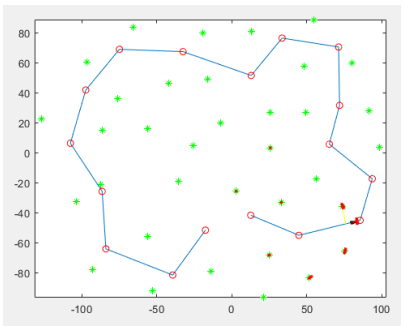
4.2. fastslam2_sim

El FastSLAM 2.0 fue la segunda versión que se sacó del FastSLAM 1.0, a la que se le incorporaron una serie de cambios que permitían solucionar algunos de los errores que se daban en la primera versión. En este se puede ver realmente el funcionamiento de FastSLAM ya que se muestra la nube de puntos, es decir, el filtro de partículas.

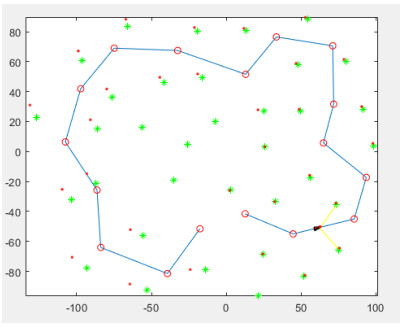
A continuación se muestran los resultados obtenidos de la ejecución del algoritmo *fastslam2_sim*.



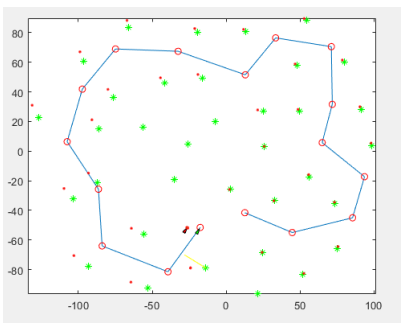
(a) Entorno



(b) Proceso



(c) Proceso vuelta 2



(d) Entorno tras la ejecución

Fig. 4.2. Resultado obtenido de la ejecución del algoritmo FastSlam2.

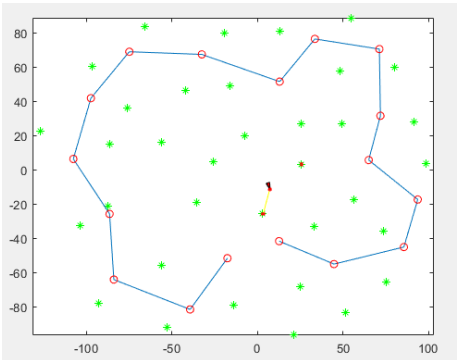
En la Figura 4.2a, se muestra el entorno que se va a mapear, en el que se pueden distinguir los *landmarks* que va a usar el robot móvil para localizarse, la trayectoria que debe seguir y la localización inicial del propio robot. Además cabe destacar que este entorno es el mismo que en el caso anterior (Figura 4.1a).

En las Figuras 4.2b y 4.2c, se muestra como el robot móvil trata de seguir la trayectoria apoyándose en los dos *landmarks* más cercanos.

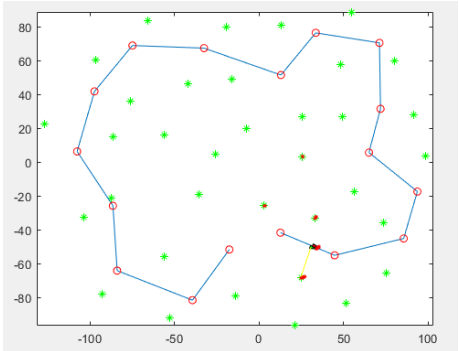
Y por último en la Figura 4.2d, se puede observar el entorno tras realizar la ejecución completa del algoritmo. Además se pueden ver los *landmarks* que ha observado el robot móvil, ya que han quedado marcados con un punto rojo. Y en este caso pasa lo mismo que en el anterior, ya que como se puede ver en dicha figura, los puntos rojos que determinan la posición de los *landmarks* para el robot, se encuentran desplazados de la posición real debido al error acumulado.

4.3. fastslam2r_sim

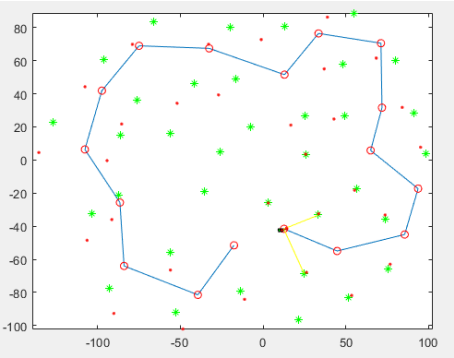
Este algoritmo es una ligera variación del anterior, al que se le ha añadido "*reverse sampling*", es decir que se calculan los pesos de las muestras y se vuelve a realizar el muestreo antes de obtener la distribución de la propuesta.



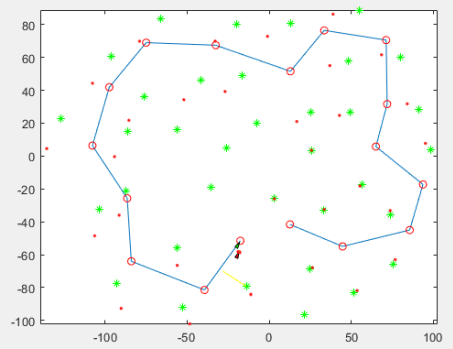
(a) Entorno



(b) Proceso



(c) Proceso vuelta 2



(d) Entorno tras la ejecución

Fig. 4.3. Resultado obtenido de la ejecución del algoritmo FastSlam2r.

En la Figura 4.3a, se muestra el entorno que se va a mapear, en el que se pueden distinguir los *landmarks* que va a usar el robot móvil para localizarse, la trayectoria que debe seguir y la localización inicial del propio robot. Además cabe destacar que este entorno es el mismo que en los dos casos anteriores (Figuras 4.1a y 4.2a).

En las Figuras 4.3b y 4.3c, se muestra como el robot móvil trata de seguir la trayectoria apoyándose en los dos *landmarks* más cercanos, marcados por una línea amarilla que va midiendo las distancias.

Y por último en la Figura 4.3d, se puede observar el entorno tras realizar la ejecución completa del algoritmo. Además se pueden ver los *landmarks* que ha observado el robot móvil, ya que han quedado marcados con un punto rojo.

4.4. Comparativa

En este caso, solo con los resultados obtenidos de la ejecución de los tres algoritmos no se puede determinar cual es mejor, puesto que dichos resultados y la ejecución mostrada es bastante similar.

Aunque basándose en la literatura consultada, al ser FastSLAM 2.0 una versión mejorada de FastSLAM 1.0 que permite solucionar ciertos errores que se producían en la primera versión, se podría considerar cualquiera de las dos variaciones de FastSLAM 2.0 como el mejor algoritmo en este caso.

5. SLAM ZARAGOZA

En este capítulo se exponen los resultados obtenidos de la ejecución de tres algoritmos diferentes: Nearest Neighbour (*NN*), Joint Compatibility Branch and Bound (*JCBB*) y SINGLES.

Una vez obtenidos dichos resultados se realiza una comparativa en función del mapa resultante de cada algoritmo y su nivel de semejanza con el entorno real a mapear. Además se analizan los errores y las matrices de confusión obtenidas de la ejecución de cada uno de los tres algoritmos ya mencionados, pudiendo así observar la cantidad de falsos positivos y negativos que se han producido.

Y finalmente, se determina a través de dicho análisis, que algoritmo de los tres estudiados tiene un mejor resultado para este caso concreto.

5.1. Nearest Neighbour (NN)

Para la ejecución de este primer algoritmo solo es necesario establecer la llamada a la función *NN()* como se puede observar en la Figura 5.1.

```
H = NN (prediction, observations, compatibility);
H = JCBB (prediction, observations, compatibility);
H = SINGLES (prediction, observations, compatibility);
```

Fig. 5.1. Llamada a la función NN.

Dicha llamada ejecuta el código que se puede ver en la Figura 5.2.

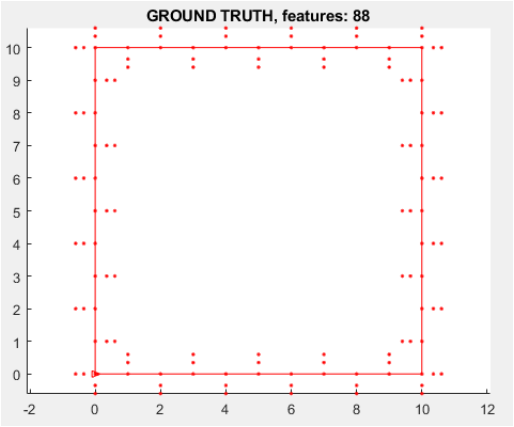
```
global chi2;
global configuration;

for i = 1:observations.m,
    D2min = compatibility.d2 (i, 1);
    nearest = 1;
    for j = 2:prediction.n,
        Dij2 = compatibility.d2 (i, j);
        if Dij2 < D2min
            nearest = j;
            D2min = Dij2;
        end
    end
    if D2min <= chi2(2)
        H(i) = nearest;
    else
        H(i) = 0;
    end
end

configuration.name = 'NEAREST NEIGHBOUR';
```

Fig. 5.2. Implementación del algoritmo Nearest Neighbour.

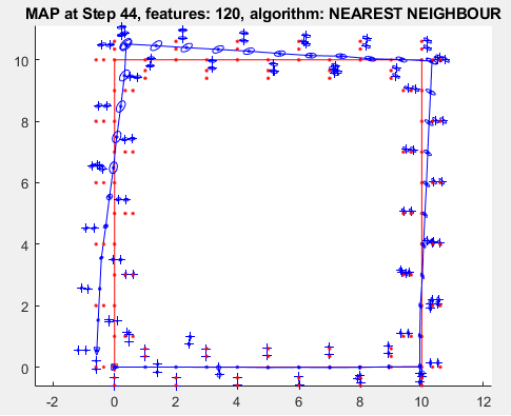
Este algoritmo itera por cada una de las observaciones i, y para cada una de ellas, en primer lugar establece como distancia mínima la distancia entre dicha observación i y la primera predicción. A continuación, se itera por cada una de las predicciones j, y para cada par i,j se obtiene la distancia entre ellas. Esa distancia obtenida se compara con la mínima guardada, de forma que sí esta es menor se establece la nueva distancia como la mínima y se establece como el vecino más cercano la j actual.

Una vez se han comprobado todas las combinaciones posibles para una observación i concreta, se establece el valor de H(i), que será igual a la j con menor distancia en el caso de que dicha menor distancia sea menor o igual que el chi_square y en caso contrario se le asigna el valor 0.

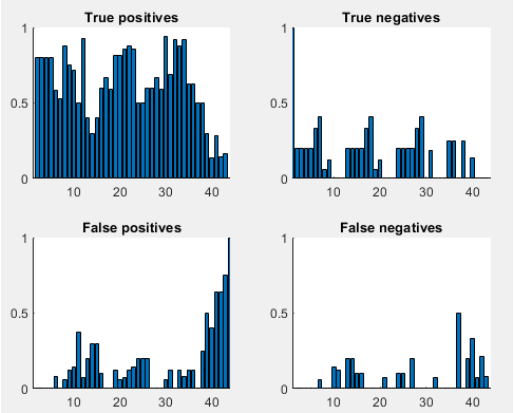
A continuación se muestran los resultados obtenidos de la ejecución de este algoritmo.



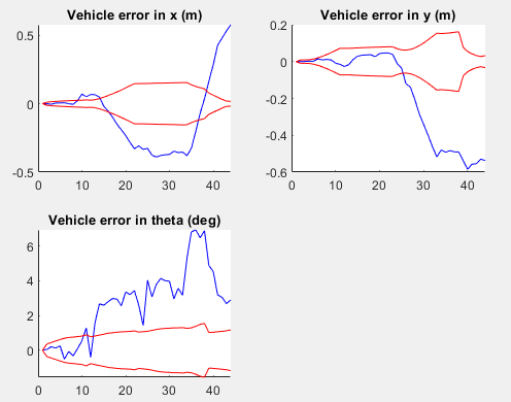
(a) Entorno



(b) Mapa Final



(c) Matriz de confusión



(d) Errores

Fig. 5.3. Resultado obtenido de la ejecución del algoritmo NN.

En la Figura 5.3a, se muestra el entorno a mapear y en la Figura 5.3b, el mapa resultante de aplicar el algoritmo *Nearest Neighbour* sobre dicho entorno.

Por otro lado, en las Figuras 5.3c y 5.3d se muestran los errores y aciertos que se han producido durante la ejecución.

Como se puede ver en la Figura 5.3c, hay muchos más verdaderos positivos que falsos positivos. Además se puede observar como a medida que se va avanzando en el tiempo aumentan el número de falsos positivos, lo cual concuerda con el mapa resultante obtenido ya que al principio se ajusta bastante bien al entorno establecido pero a medida que avanza se va desviando del camino correcto. En el caso de los verdaderos negativos pasa lo mismo, hay mayor número que falsos negativos y el número de falsos negativos se ve incrementado al final del proceso de mapeo.

En cuanto a los errores, como se puede observar en la Figura 5.3d, al principio de la ejecución el error se mantiene en el rango aceptable (el que se encuentra entre las líneas de color rojo), sin embargo al llegar a más o menos a la mitad del proceso se comienzan a acumular más errores haciendo que la línea de error se salga del rango definido como aceptable.

5.2. SINGLES

Para la ejecución de este segundo algoritmo no basta con establecer la llamada a la función *SINGLES()* como se puede observar en la Figura 5.4. Sino que también ha sido necesario implementar el código del algoritmo *SINGLES*, que es el que se puede observar en la Figura 5.5.

```
H = NN (prediction, observations, compatibility);
H = JCBB (prediction, observations, compatibility);
H = SINGLES (prediction, observations, compatibility);
```

Fig. 5.4. Llamada a la función SINGLES.

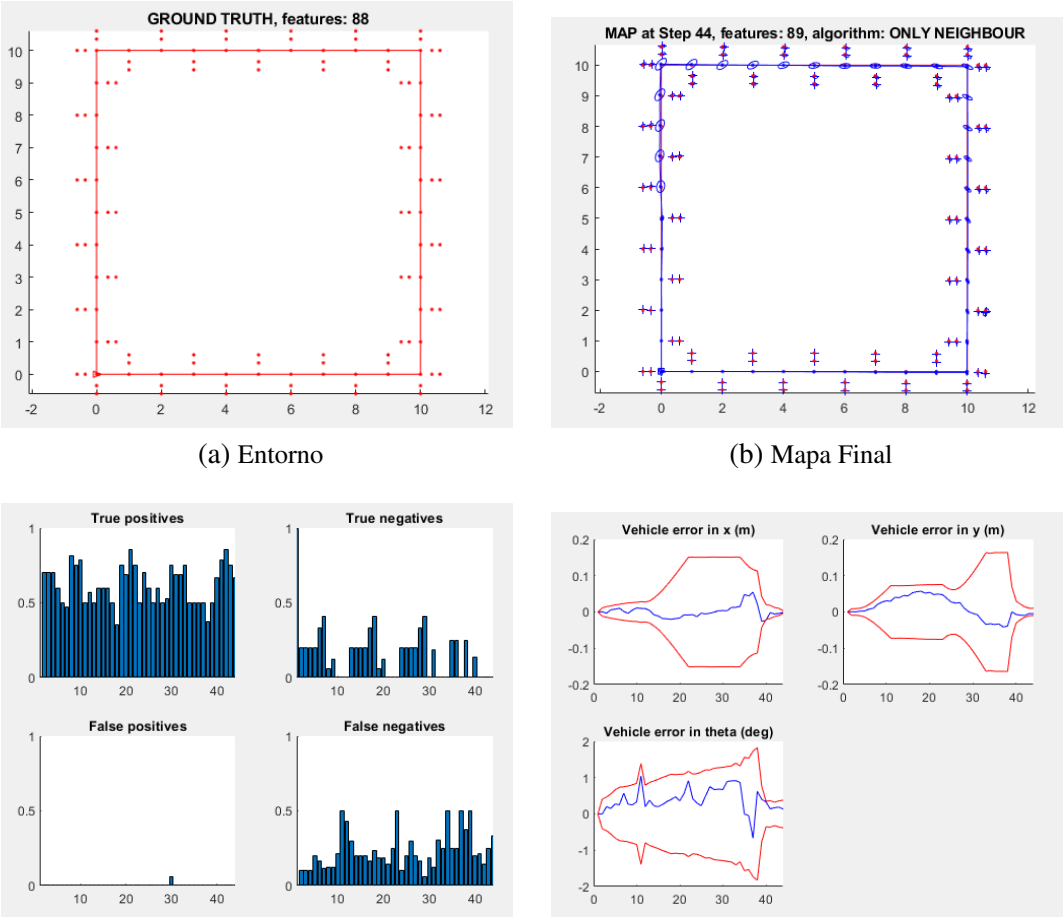
```
global chi2;
global configuration;

H = zeros(1, observations.m);

% You have observations.m observations, and prediction.n predicted features.
%
% For every observation i, check whether it has only one neighbour,
% feature, and whether that feature j has only that one neighbour
% observation i.  If so, H(i) = j.
%
% You will need to check the compatibility.ic matrix for this:
%   compatibility.ic(i,j) = 1 if observation i is a neighbour of feature j.
for i = 1:observations.m
    aux_i = find(compatibility.ic(i,:));
    [~,c] = size(aux_i);
    if c == 1
        aux_j = find(compatibility.ic(:, aux_i(1)));
        [~,c2] = size(aux_j);
        if c2 == 1
            H(i) = aux_i(1);
        end
    end
end
configuration.name = 'ONLY NEIGHBOUR';
```

Fig. 5.5. Implementación del algoritmo SINGLES.

Este algoritmo itera por cada una de las observaciones i y para cada una encuentra sus

predicciones vecinas. Una vez obtenidas las predicciones vecinas se comprueba si dicha observación solo posee una predicción vecina, en caso de cumplirse dicha condición, se obtienen las observaciones vecinas de dicha predicción j. Y si se da que dicha predicción j también tiene una única observación i vecina, se establece que el valor de H(i) es igual a dicha predicción j.

A continuación se muestran los resultados obtenidos de la ejecución de este algoritmo.



(a) Entorno

(b) Mapa Final

(c) Matriz de confusión

(d) Errores

Fig. 5.6. Resultado obtenido de la ejecución del algoritmo SINGLES.

En la Figura 5.6a, se muestra el entorno a mapear y en la Figura 5.6b, el mapa resultante de aplicar el algoritmo *SINGLES* sobre dicho entorno.

Por otro lado, en las Figuras 5.6c y 5.6d se muestran los errores y aciertos que se han producido durante la ejecución.

Como se puede ver en la Figura 5.6c, no hay apenas falsos positivos, lo que explica que el mapa resultante de la Figura 5.6b se ajuste tan bien al entorno establecido. A pesar de no haber falsos positivos, sí que se dan falsos negativos y en este caso en mayor medida que verdaderos negativos.

En cuanto a los errores, como se puede observar en la Figura 5.6d, la línea azul, que

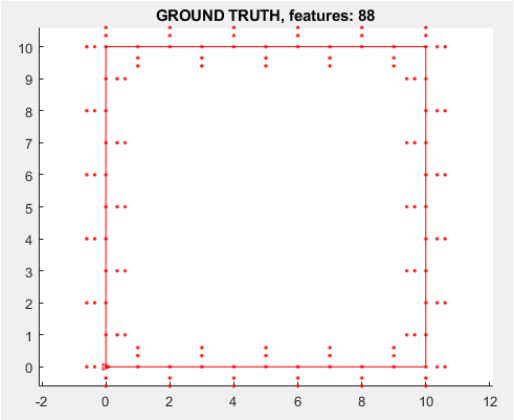
corresponde como los errores extraídos de la ejecución, se mantiene en todo momento entre las dos líneas rojas, que delimitan el rango aceptable de error.

Una vez visto el funcionamiento del algoritmo sin ningún tipo de oclusión se cambia el valor de la variable de configuración *people* a 1 (Figura 5.7), lo cual permite la aparición aleatoria de personas en el entorno mientras se realiza el mapeo.

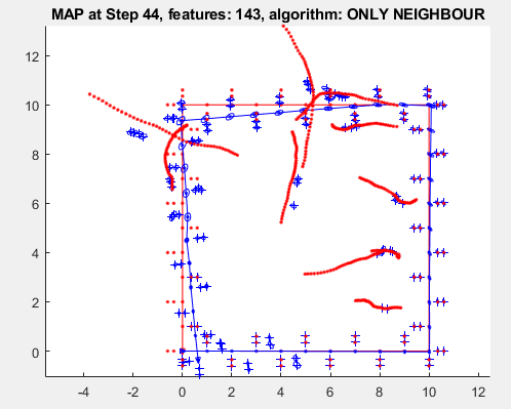
```
% determines execution and display modes
global configuration;

configuration.ellipses = 1;
configuration.tags = 0;
configuration.odometry = 1;
configuration.noise = 1;
configuration.alpha = 0.99;
configuration.step_by_step = 0;
configuration.people = 1;
```

Fig. 5.7. Variable de configuración People.

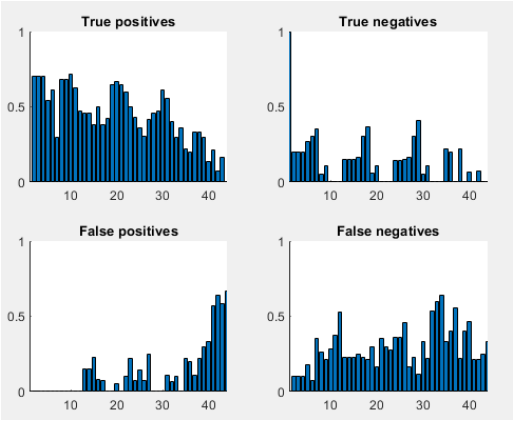
A continuación se muestran los resultados obtenidos de la ejecución con este valor de la variable de configuración *people*.



(a) Entorno



(b) Mapa Final



(c) Matriz de confusión

(d) Errores

Fig. 5.8. Resultado obtenido de la ejecución del algoritmo SINGLES.

En este caso como se puede ver si comparamos las Figuras 5.6b y 5.8b, el mapeo se empieza a desviar de la trayectoria establecida en el punto en el que empiezan a cruzarse las personas (líneas de puntos rojas), ya que provocan errores en los sensores del robot, dando lugar a ese desvío en la trayectoria. Esto además provoca que el número tanto de falsos positivos como de falsos negativos aumente (Figura 5.8c) y que el error se salga del rango aceptable (Figura 5.8d).

5.3. Joint Compatibility Branch and Bound (JCBB)

Para la ejecución de este último algoritmo solo es necesario establecer la llamada a la función *JCBB()* como se puede observar en la Figura 5.9.

```
H = NN (prediction, observations, compatibility);
H = JCBB (prediction, observations, compatibility);
H = SINGLES (prediction, observations, compatibility);
```

Fig. 5.9. Llamada a la función JCBB.

Dicha llamada ejecutará el código que se puede ver en la Figura 5.10.

```
%-------------------------------------------------------
function H = JCBB (prediction, observations, compatibility)
global Best;
global configuration;
Best = zeros(1, observations.m);
JCBB_R (prediction, observations, compatibility, [], 1);
H = Best;
configuration.name = 'JOINT COMPATIBILITY B & B';
%-------------------------------------------------------
function JCBB_R (prediction, observations, compatibility, H, i)
global Best;
global configuration;

if i > observations.m % leaf node?
    if pairings(H) > pairings(Best) % did better?
        Best = H;
    end
else
    individually_compatible = find(compatibility.ic(i,:));
    for j = individually_compatible
        if jointly_compatible(prediction, observations, [H j])
            JCBB_R(prediction, observations, compatibility, [H j], i + 1);
        end
    end
    if pairings(H) + pairings(compatibility.AL(i+1:end)) >= pairings(Best)
        JCBB_R(prediction, observations, compatibility, [H 0], i + 1);
    end
end
%-------------------------------------------------------
function p = pairings(H)
p = length(find(H));
```

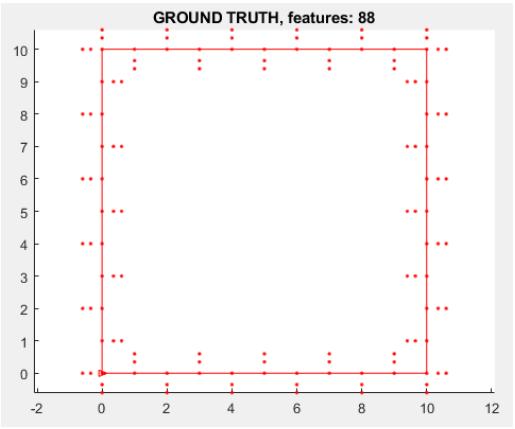
Fig. 5.10. Implementación del algoritmo JCBB.

Este algoritmo empieza estableciendo el valor 1 a la variable i. A continuación se llama a la función *JCBB_R* que comprueba si esa i es mayor que el número de observaciones, en el caso de que se cumpla se comprueba si la cantidad de parejas en H es mayor que la cantidad de parejas almacenadas como la mejor solución, si es así se establece H como la mejor solución.

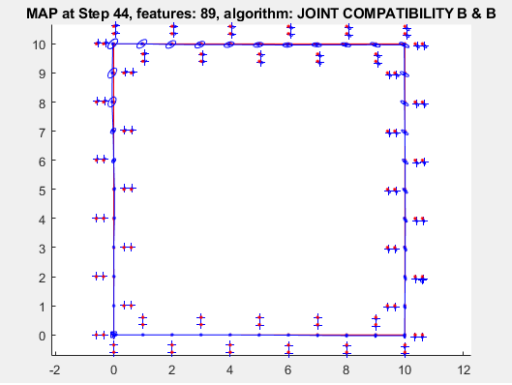
Si por el contrario i es menor que el número de observaciones, se encuentran las predicciones que sean vecinas de esa observación i y para cada predicción j se comprueba si es conjuntamente compatible. En el caso de que se cumpla dicha condición se llama a la función *JCBB_R* con el siguiente valor de i y se almacena en una lista la H actual y la j actual.

Una vez se ha iterado por todas las predicciones vecinas de i, se comprueba si la cantidad de parejas de H y la cantidad de parejas del resto de observaciones es mayor o igual que la cantidad de parejas de la mejor solución almacenada y en el caso de cumplirse la condición de llama a la función *JCBB_R* con el siguiente valor de i y se almacena en una lista la H actual y un 0.

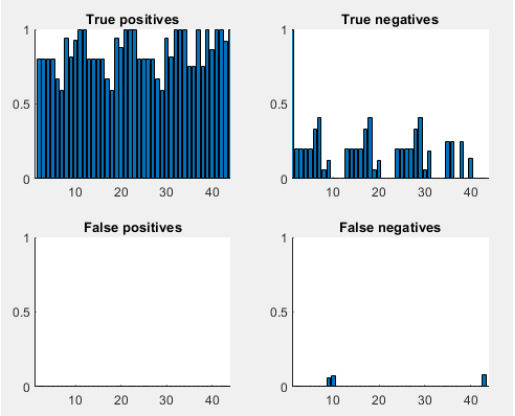
A continuación se muestran los resultados obtenidos de la ejecución de este algoritmo.



(a) Entorno



(b) Mapa Final



(c) Matriz de confusión

(d) Errores

Fig. 5.11. Resultado obtenido de la ejecución del algoritmo Joint Compatibility Branch and Bound.

En la Figura 5.11a, se muestra el entorno a mapear y en la Figura 5.11b, el mapa resultante de aplicar el algoritmo *SINGLES* sobre dicho entorno.

Por otro lado, en las Figuras 5.11c y 5.11d se muestran los errores y aciertos que se han producido durante la ejecución.

Como se puede ver en la Figura 5.11c, no se han producido falsos positivos y el número de falsos negativos es muy bajo, lo cual explica la alta similitud entre el mapa resultante del mapeo y el entorno establecido.

En cuanto a los errores, como se puede observar en la Figura 5.11d, la línea azul, correspondiente con los errores que se han producido durante la ejecución del algoritmo, se ha mantenido en todo momento dentro de los límites establecidos (líneas rojas), de forma que el error extraído se puede considerar aceptable.

5.4. Comparativa

Una vez se han expuesto los mapas resultantes del mapeo del mismo entorno usando cada uno de los tres algoritmos, se puede proceder a realizar un análisis de los propios algoritmos. Para ello se va a realizar una comparativa teniendo en cuenta los resultados obtenidos, es decir, tanto de los mapas resultantes como de las matrices de confusión y los errores extraídos, para así determinar cual ha sido el algoritmo más eficiente en este caso.

En primer lugar, si analizamos los mapas resultantes de las Figuras 5.3b, 5.6b y 5.11b, se puede ver como tanto el algoritmo *SINGLES*, como *JCBB* tienen unos resultados bastante buenos, mientras que *NN* a partir de la mitad del proceso comienza a desviarse de la trayectoria establecida, de forma que podríamos descartar este algoritmo.

Si analizamos los errore de los dos algorimos restantes (Figuras 5.6d y 5.11d), se puede observar que en ambos casos la línea de error se mantiene dentro de los límites establecidos por las líneas rojas de forma bastante similar, con lo cual no es posible descartar ninguno de los dos algoritmos.

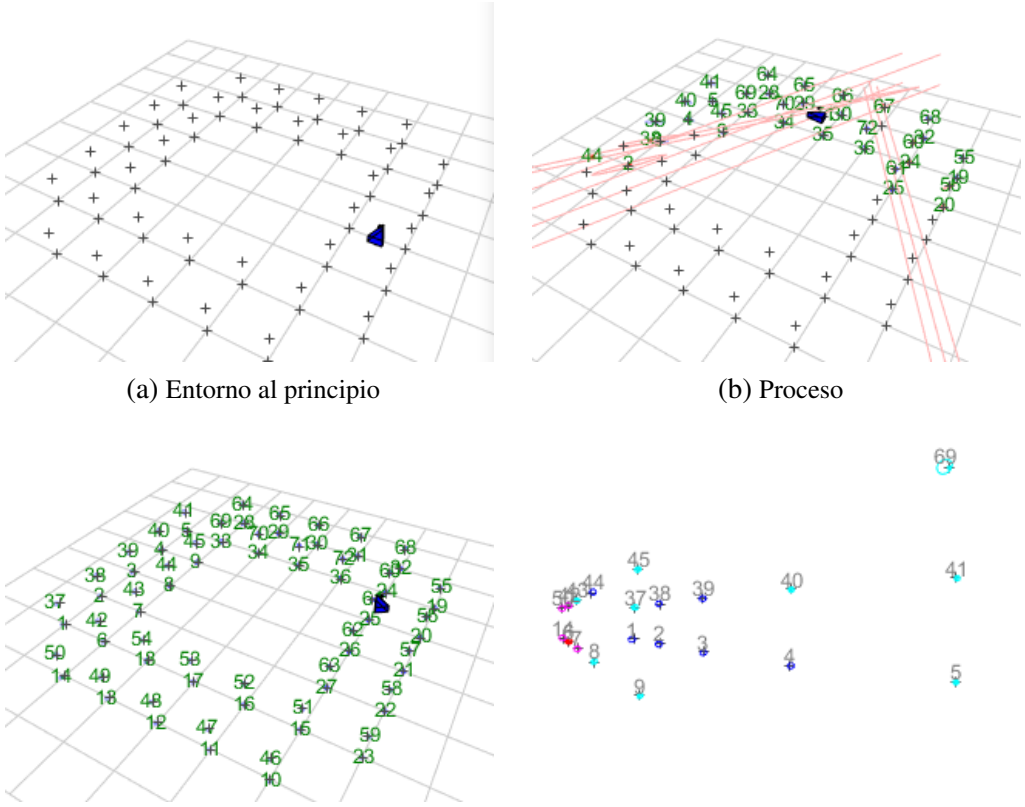
Por último, analizamos las matrices de confusión (Figuras 5.6c y 5.11c), podemos ver que en cuanto a falsos positivos y verdaderos negativos ambas son bastante similares, sin embargo, si nos fijamos en los verdaderos positivos y falsos negativos, se puede ver como el algoritmo *JCBB* tiene un mayor número de verdaderos positivos menor que en el caso de *SINGLES* y el número de falsos negativos y practicamente nulo, de forma que en este caso se puede ver que el algoritmo *JCBB* es el que mejor funciona.

6. SLAMTB SOLA

En este capítulo se exponen los resultados obtenidos al ejecutar el programa de la carpeta *Slamtb-Sola*. Para ello se ejecutan los archivos *slamrc.m* y *slamtb*.

Este proyecto permite ejecutar un algoritmo de SLAM basado en el filtro de Kalman extendido, usando diversos robots, sensores y *landmarks*.

A continuación se muestran los resultados obtenidos de la ejecución de los archivos ya mencionados.



(a) Entorno al principio

(b) Proceso

(c) Entorno al final

(d) Datos del sensor

Fig. 6.1. Resultado obtenido de la ejecución de Slamtb.

En la Figura 6.1a, se muestra el entorno antes de comenzar a procesarlo, en el que cada una de las cruces es un *landmark*.

En las Figuras 6.1b y 6.1d, se muestra el proceso que se va realizando y como a medida que se va moviendo el robot van variando los *landmarks* visibles y que por tanto aparecen en la figura de datos del sensor.

Por último en la Figura 6.1c, se muestra el entorno al finalizar el proceso de mapeo, en la que se puede ver como todos los *landmarks* han aparecido en el sensor del robot y por tanto se han marcado en verde.

7. CONCLUSIONES

Con esta práctica se ha podido ver que hay múltiples formas de resolver un mismo problema, en este caso el problema de SLAM. Todas las variantes utilizadas en este trabajo se basan en el Filtro de Kalman, ya sea el estándar o el extendido o en el filtro de partículas, sin embargo existen otros algoritmos que se están utilizando actualmente para resolver este mismo problema como es el caso de *Parallel Tracking and Mapping* (PTAM), *Patch-based Multi-View Stereo* (PMVS) ...

Teniendo en cuenta la cantidad de algoritmos diferentes que existen y que se sigue investigando en este ámbito, no se puede considerar que ninguno de los algoritmos existentes no va a fallar en ningún caso, siempre puede haber una situación en la que el algoritmo no funcione, de ahí que se siga investigando para obtener nuevos algoritmos y poder elegir el más adecuado en cada situación.