Mapeado:

Vamos a utilizar dos técnicas de mapeado. Explicaremos la forma de utilizar cada una de ellas mediante Matlab a partir del código proporcionado. Posteriormente las compararemos indicando las diferencias observada respecto de la calidad de los resultados obtenidos.

Mapeado con posiciones conocidas

Clase Robotics.OccupancyGrid

Esta clase se encarga de crear una cuadricula de ocupación 2D, cada casilla de la cuadricula posee un valor numérico entre 0 y 1 que representa la probabilidad de que dicha cuadricula este ocupada. Los valores cercanos a 0 representarán que la casilla esta vacía y los cercanos a 1 que este ocupada. Los valores de probabilidad estarán almacenados por un filtro e Bayes binario que estimará la ocupación de cada celda de la rejilla.

El mapa puede ser construido de 5 formas distintas:

- map = robotics.OccupancyGrid(width,height):representa la rejilla según el valor de altura y anchura dados, la resolución será por defecto de 1 rejilla por metro.
- map = robotics.OccupancyGrid(width,height,resolution): similar a la construcción anterior, pero con a diferencia de que se añade la resolución elegida cuyo valor será de celda por metro.
- map = robotics.OccupancyGrid(rows,cols,resolution,"grid"): al añadir al final la opción “grid” los valores dados ya o serán los metros de ancho y largo de la rejilla, sino que será el numero de filas y el número de columnas.
- map = robotics.OccupancyGrid(p):crea la rejilla en función de la matriz p dada, tendrá una resolución por defecto de 1 celda por metro.
- map = robotics.OccupancyGrid(p,resolution): es similar a la construcción anterior solo que añade un valor para la resolución de la rejilla, como antes será de celda por metro.

El mapa además tendrá unas determinadas propiedades:

- FreeThreshold: es un umbral necesario para poder considerar una celda como libre o no libre de obstáculos.
- OccupiedThreshold: umbral para considerar una celda como ocupada o no ocupada.
- ProbabilitySaturation: añade unos límites de saturación e probabilidad añadiendo un máximo y un mínimo a los posibles valores de probabilidad.
- Gridsize: es el numero de filas y de columnas de la rejilla.
- Resolution: es la resolución de la rejilla.
- XWorldLimits: son los valores mínimos y máximos del eje x del rango de coordenadas del mundo.
- YWorldLimits: son los valores mínimos y máximos del eje y del rango de coordenadas del mundo.
- GridLocationInWorld: son las coordenadas mundiales de la rejilla.

Resultados obtenidos sobre el simulador

Hemos utilizado esta técnica de mapeado sobre el simulador stdr. Primero hemos hecho esto sin tener activado el error en la odometría. Es decir, la percepción de la posición del robot es exacta, lo cual nos es algo que pueda darse en un entorno real. Posteriormente hemos repetido el proceso de mapeado en el simulador de nuevo, pero en este caso con la odometría activada de modo que la simulación se asemeje más a un caso real.

Mapeado sin error en la odometría

Cuando no tenemos error en la odometría obtendremos mapas demasiado precisos, alejados de lo que podríamos llegar a obtener sobre un entorno real.

A pesar de esto podemos observar que el mapa seguirá teniendo zonas con ligeras imperfecciones producto de haber pasado demasiado rápido por ellas.

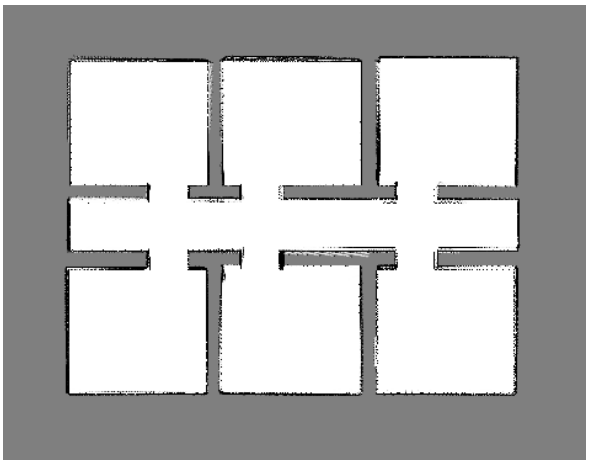


Ilustración 1 Mapeado sobre el simulador sin ruido en la odometría

Mapeado con error en la odometría

Cuando activamos el error en la odometría vemos que el mapa se deforma. Podemos observar que los giros rápidos aumentan considerablemente la distorsión producida en el mapa respecto de la que otros movimientos producen.

También pude observarse que la esquina inferior izquierda es donde el mapa es más preciso siendo este cada vez de menor calidad debido a la acumulación de error.

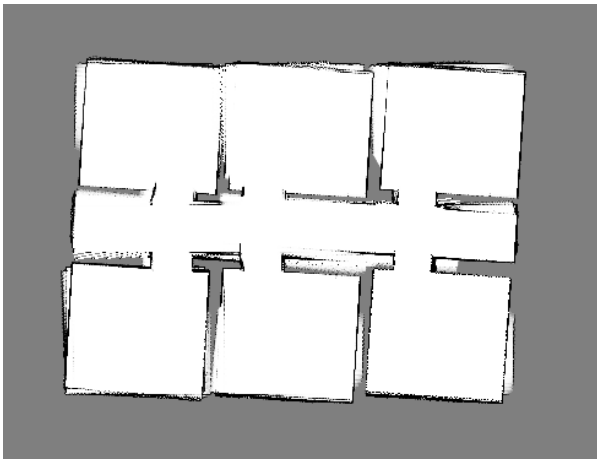


Ilustración 2 Mapeado sobre el simulador con ruido en la odometría

Comparación de ambos mapas

Podemos ver que tal y cabe esperar el mapa con error en la odometría es de peor calidad que cuando este no está presente. Haber realizado el mapeado sobre el simulador nos ha permitido atisbar lo difícil que es realizar mapeado cobre un entorno real donde el error será mayor y la acumulación de este más notable pues el mapa será de mayor tamaño y el robot deberá estar funcionando por más tiempo.

A parte de la acumulación de error de la odometría hemos observado otro problema de aplicar este tipo de algoritmo, el tamaño de mapa es fijo. Esto es problemático ya que si nos salimos de mapa se producirá un error en tiempo de ejecución. Adicionalmente conocer el tamaño del mapa implica conocer el entorno antes de lanzar el robot sobre él lo cual no es ideal. Por último, un tamaño fijo de mapa no es óptimo especialmente en entornos grandes e implica un gasto de memoria y tiempo de cómputo incensario.

Como observación adicional incorporamos una foto de lo que sucede cuando el robot se choca con una pared en el simulador. Cuando esto sucede el error en la odometría se dispara y el mapa se vuelve completamente inservible a partir de ese punto.

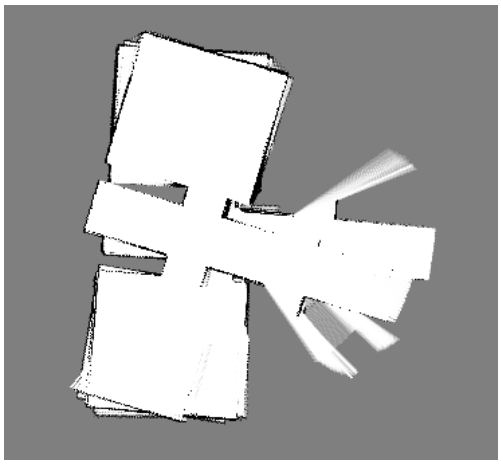


Ilustración 3 Mapeado sobre el simulador después de haber chocado con una pared

Resultados obtenidos sobre el robot real

Para poder utilizar el programa anterior en el robot original solo ha habido que cambiar unos valores de variables. Lo primero que cambiamos en el programa es el tamaño del mapa para adecuarlo al nuevo mapa que es más grande que el de la simulación. Además, cambiamos los nombres de los topics de los que se nutrirá el programa, para saber que topics son los que necesitaremos, los buscaremos con el comando rqt. En este caso necesitaremos “odom” y “base_link”. Lo rangos del láser también habrá que modificarlos para que coincidan con el del láser del robot. Tras estos cambios procederemos a ejecutar el programa a la vez que conectamos el robot. El resultado es que el mapa de de baja calidad, debido a que nos estamos fiando por completo de que la odometría no tenga ningún tipo de error, cuando realmente el error en la odometría es bastante elevado.

Localización y mapeado simultáneos (SLAM)

Vamos a aplicar ahora la técnica de SLAM mediante la cual se realiza el mapeado y la localización simultánea del robot. En la implementación de Matlab el SLAM se realiza sin atender a la odometría ya que en esta se acumula error y se utiliza solo la percepción del entorno captada mediante el lidar.

Clase Robotics.LidarSLAM

Esta clase realiza la localización y el mapeo simulataneamiente, o lo que es lo mismo, realiza un proceso de SLAM para la entrada del sensor de escaneo LiDAR. El algoritmo tomará los escaneos y los adjuntará a un nodo en un grafico pose subyacente. Además, buscará cierres de lazo en el lugar en el que los escaneos se superpongan en regiones previamente asignadas.

Esta clase tiene 3 tipos de construcción diferentes:

- slamObj = robotics.LidarSLAM: crea un objeto LiDAR SLAM con todos los valores dados por defecto, el tamaño del mapa de ocupación será de 20 celdas por metro, y el rango de escaneo máximo será de 8 metros.
- slamObj = robotics.LidarSLAM(mapResolution,maxLidarRange): crea un objeto LiDAR SLAM con los valores de resolución del mapa y de rango máximo de escaneo que se introduzcan.
- slamObj = robotics.LidarSLAM(mapResolution,maxLidarRange,maxNumScans): es similar a la construcción anterior, pero añadiendo un número máximo de exploraciones aceptadas en la generación de código.

Este objeto tendrá las siguientes propiedades:

- PoseGraph: es el gráfico de pose que conecta los escaneos.
- MapResolution: es la resolución de la rejilla de ocupación, dada en celdas por metro.
- MaxLidarRange: es el rango máximo que leerá el sensor lidar.
- OptimizationFcn: es la función de optimización de gráficos de pose.
- LoopClosureThreshold: es el umbral para la aceptación de cierres de bucle.
- LoopClosureMaxAttempts: es el numero de intentos que hará el algoritmo para cerrar el bucle en un determinado punto, aumentar este valor disminuirá la optimización del algoritmo, ya que le tomará más tiempo este paso de cierre de bucle.
- LoopClosureAutoRollback: permite la reversión automática de cierres de lazo
- OptimizationInterval: es el número de cierres de lazo aceptados para poder desencadenar la optimización.
- MovementThreshold: es el cambio mínimo en la pose para que se procesen los escaneos.

Resultados obtenidos sobre el simulador

Hemos aplicado la técnica de SLAM en el simulador. Podemos comprobar como el error final obtenido en el mapa comparado con el mapeado asumiendo que la odometría era exacta. Esto se debe a que con SLAM podemos reducir el error de la posición del robot a partir de las observaciones, especialmente cuando se produce el cierre de lazos.

Es por esto por lo que hemos intentado pasar dos veces por algunas zonas del mapa, pudiendo apreciarse en estas una calidad de mapa mayor. En las que solo hemos pasado una vez se ve como el mapa generado es de una calidad ligeramente inferior, aunque siempre mayor que con el algoritmo de mapeado anterior.

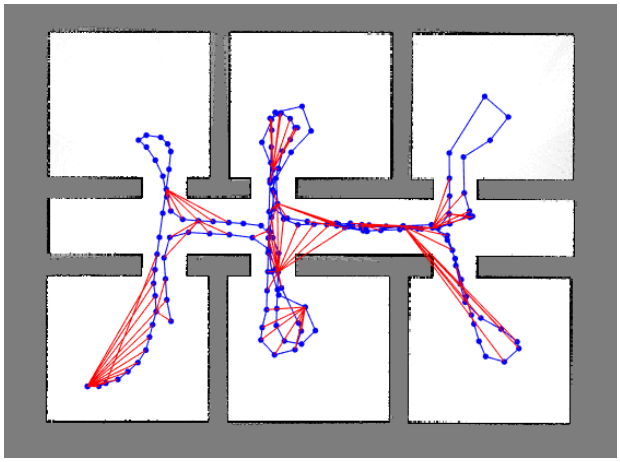


Ilustración 4 SLAM sobre el simulador

Resultados obtenidos sobre el robot real

Aplicaremos ahora la técnica de SLAM sobre el robot real en el pasillo en el que se encuentra el laboratorio.

Para hacerlo hemos conducido el robot por el entorno como ya hicimos en el apartado de mapeado. Hemos procurado ir despacio y haciendo pequeños ciclos al recorrer el entorno de modo que forcemos la corrección de errores mediante el cierre de lazos.

Se debe destacar que durante el recorrido del entorno había otros grupos haciendo la práctica, así como personas de paso siendo todo ese ruido filtrado correctamente por el algoritmo.

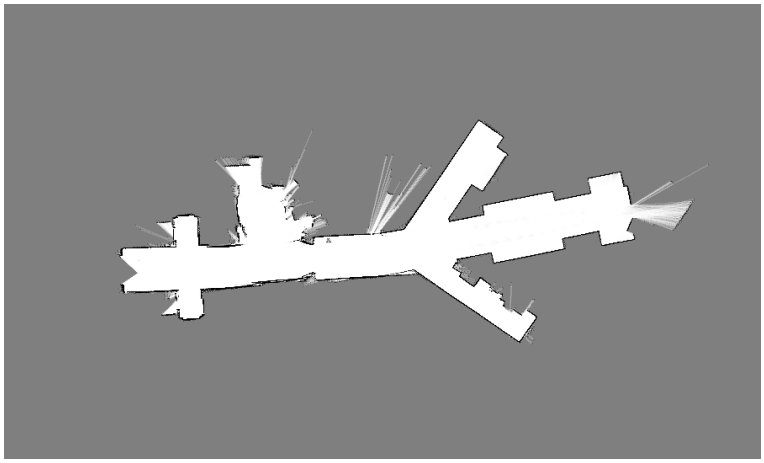


Ilustración 5 SLAM sobre el robot real

El mapa que hemos obtenido mediante SLAM lo hemos editado para eliminar pequeñas imperfecciones y engrosar las paredes. Este mapa será el que utilicemos para la siguiente parte de la práctica.

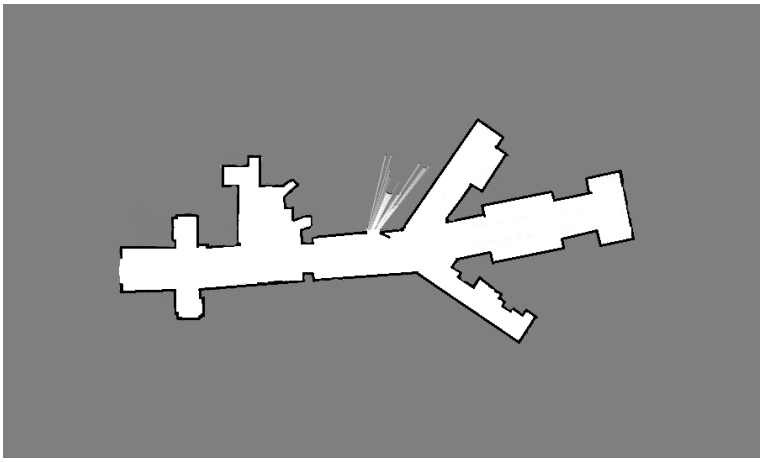


Ilustración 6 Mapa obtenido con SLAM retocado

LOCALIZACIÓN CON “AMCL”

Utilizaremos el algoritmo de Montecarlo para localizarnos sobre un mapa conocido. Dichos mapas serán los que se obtuvieron en el apartado anterior mediante SLAM. Este algoritmo discretiza la creencia y no los estados como sucedía en el algoritmo anterior.

Clase Robotics.MonteCarloLocalization

Esta clase crea un objeto de la clase monteCarloLocalization. Este algoritmo se utiliza para estimar la posición y orientación de un determinado objeto en su entorno utilizando un mapa conocido, un escáner lidar y el sensor de odometría. Para la localización del vehículo utilizará un filtro de partículas que estimará la posición, estas partículas en el mapa representaran lugares en los que puede que se encuentre el robot en ese momento. A lo largo de la ejecución del algoritmo las partículas irán convergiendo en un único punto, que será la posición del robot en ese momento.

El objeto monteCarloLocalization creado tiene dos formas de construcción:

- mcl = monteCarloLocalization: retorna el objeto monteCarloLocalization, con los valores de construcción por defecto, el mapa asignado será uno vacio, asi que antes de poder utilizar el objeto habrá que asignarle un mapa válido.
- mcl = monteCarloLocalization(Name,Value): crea el objeto con unos determinados valores para sus características. “Name” será el valor de la característica a modificar, y “Value” será el nuevo valor que se le dará.

El objeto monteCarloLocalization tiene las siguientes propiedades:

- InitialPose: es la posición inicial del robot dentro de la rejilla.
- InitialCovariance: es la covarianza gaussiana para la posición inicial, está especificada como una matriz diagonal.
- GlobalLocalization: es un flag para decidir si se hará localización global o local, “false” hará que se haga local (es la asignada por defecto), y “true” hará que se haga global.
- ParticleLimits: aquí se define cual será el valor mínimo y máximo del número de partículas que se utilizarán y crearán.
- SensorModel: asigna el modelo del sensor del campo de probabilidad.
- MotionModel: asigna el modelo de odometría para la conducción diferencial.
- UpdateThresholds: es el umbral mínimo en el cambio del estado que hace que se active la siguiente iteración del algoritmo.
- ResamplingInterval: es el numero de actualizaciones del filtro entre cambios en las partículas.
- UseLidarScan: es una flag para decidir si se usa o no el objeto lidarScan, por defecto esta a “false” que significa que no lo utiliza, asignarle un “true” hará que si lo utilice.

Además, el objeto tiene las siguientes funciones:

- getParticles: función que devuelve las partículas actuales del algoritmo de localización.

Este objeto se puede utilizar de dos formas:

- [isUpdated,pose,covariance] = mcl(odomPose,scan): estima la pose y la covarianza del robot con el algoritmo de Montecarlo, los parámetros de entrada son la odometría del robot y los datos del sensor lidar.

- [isUpdated,pose,covariance] = mcl(odomPose,ranges,angles): similar a la construcción anterior pero en lugar de utilizar el escáner del láser lidar, especifica el escaneado agregando directamente los rangos y ángulos.

La utilización del objeto da los siguientes parámetros de salida:

- isUpdated: es un flag para la actualización del pose.
- pose: es la estimación actual del pose.
- covariance: es la covarianza estimada para el pose actual.

Resultados obtenidos sobre el simulador

Aplicaremos el algoritmo de AMCL sobre el simulador en las dos posibles configuraciones que tiene, local y global.

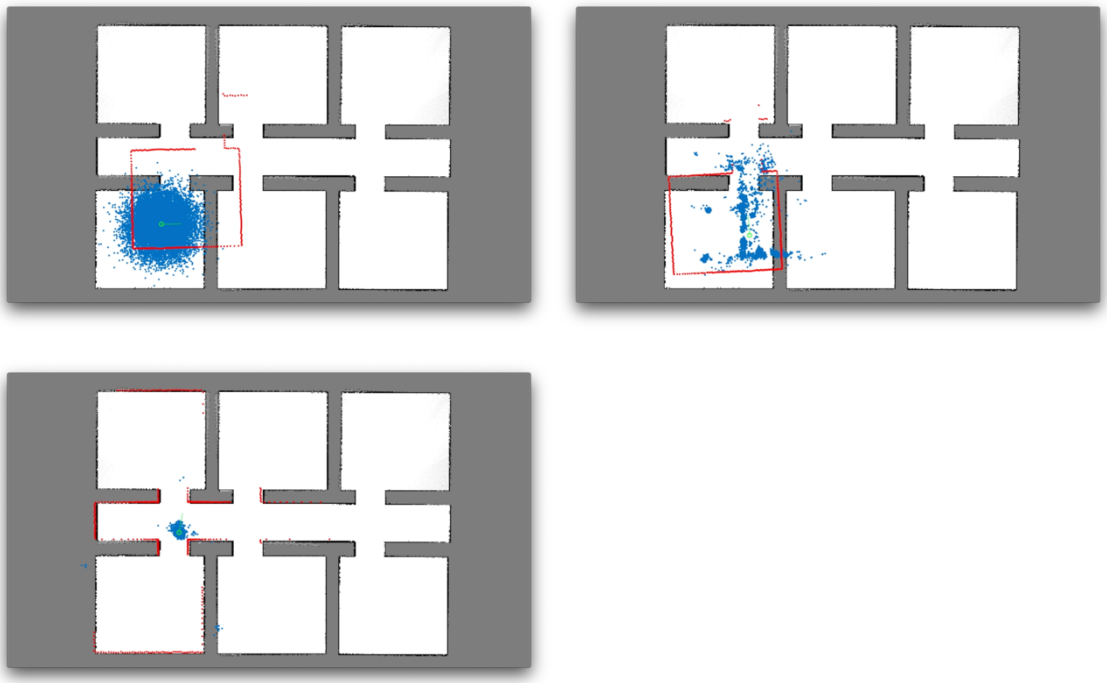
De forma general hemos observado que cuando el número de puntos sobre los que se calcula la creencia es mayor la precisión del algoritmo aumenta y cuando este es menor cae llegando incluso a no converger a la posición real del robot en el entorno. Ya que no hay implementado un mecanismo para detectar cuando la convergencia no se ha producido en el momento en el que en la posición real del robot desaparecen las partículas no podremos llegar a localizarnos.

AMCL local

Para aplicar este algoritmo de forma local se ubicarán puntos en una distribución entorno a la posición que indiquemos que el robot tiene al iniciarse. Este método requiere ese conocimiento adicional del entorno antes de poder empezar a funcionar, no obstante, nos permite obtener mejores resultados ya que incorporamos dicho conocimiento previo del entorno. Adicionalmente ya que la posición inicial del robot está acortada a una región podremos utilizar muchos menos puntos sobre los que calcular la creencia que en la localización global.

https://youtu.be/tx39zU2HHXA

El robot no se localiza debido a las simetrías dentro de la habitación, en cuanto salimos al pasillo, es capaz de localizarse.

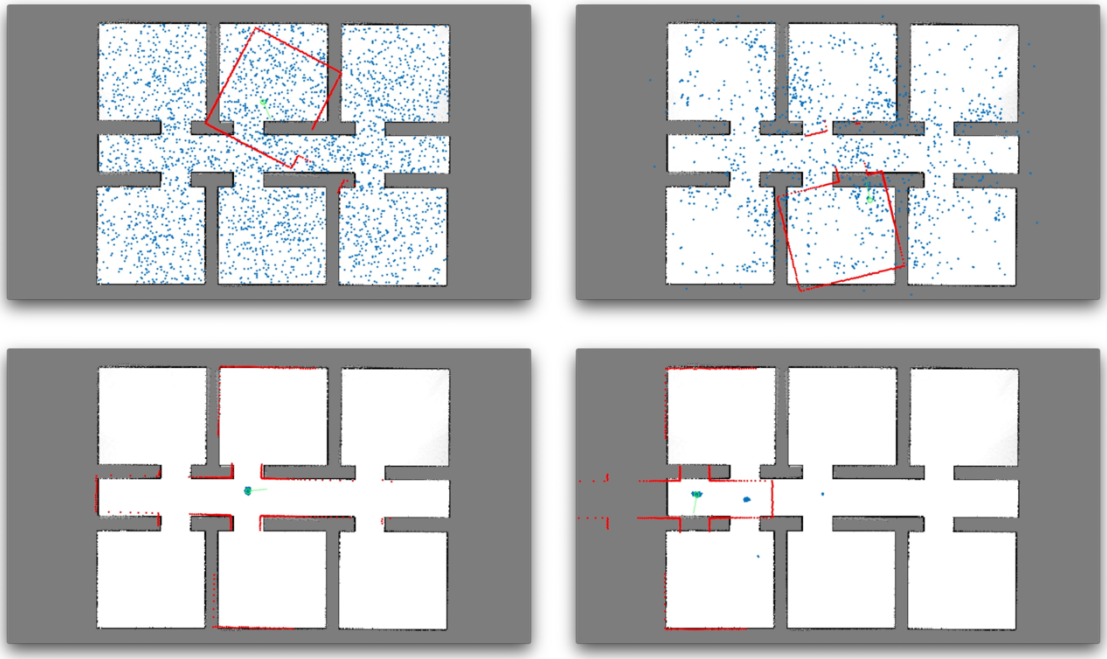


AMCL global

Cuando realizamos la localización de Montecarlo sin especificar una posición inicial como hacíamos en la localización local se deberán ubicar los puntos sobre los que calculamos la creencia por todo el entorno. Debido a eso necesitaremos tener una mayor cantidad de puntos debido a que si no corremos el riesgo de que el algoritmo no converja porque allá donde el robot se encuentra dichos puntos desaparezcan.

https://youtu.be/rahT4VorvMI

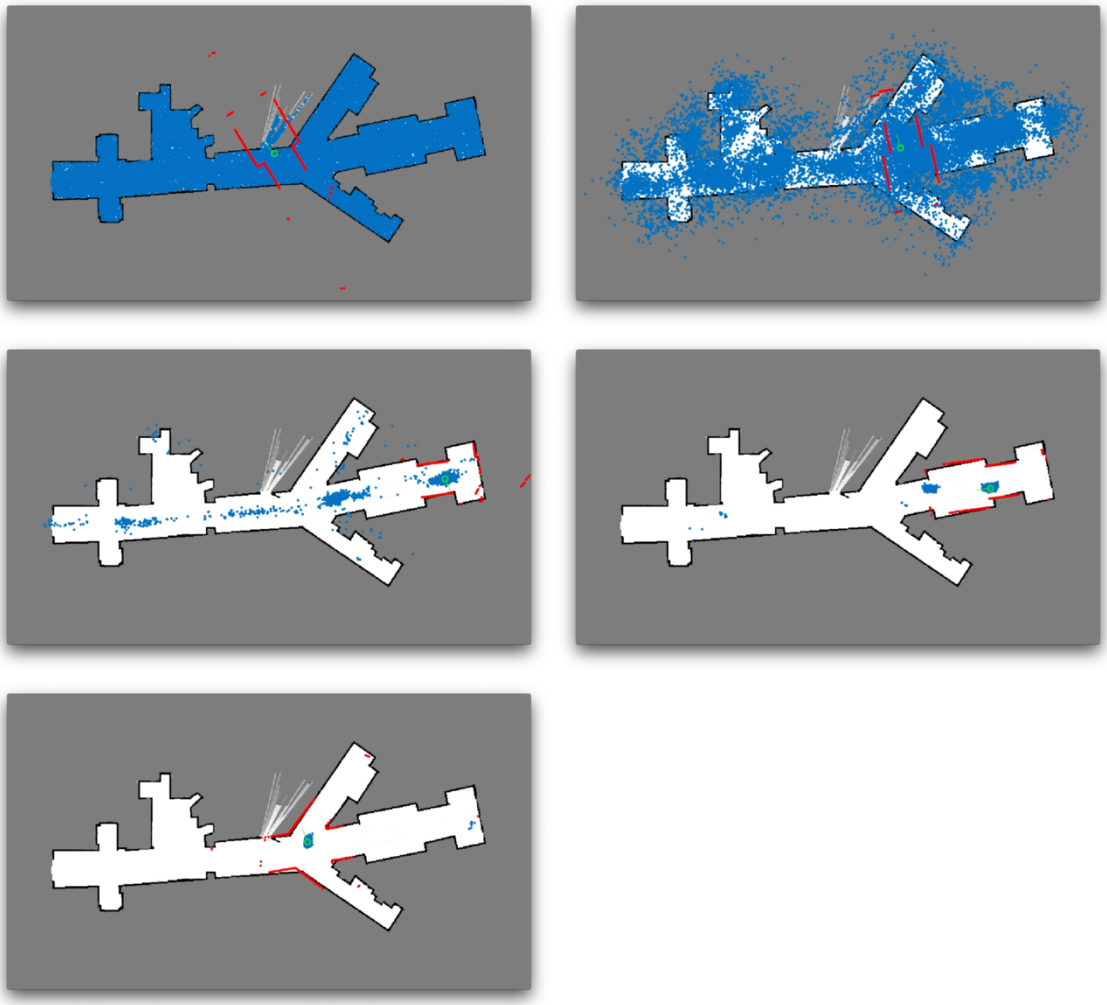
El robot no sabe en qué habitación está ya que hay simetrías entre ellas. Finalmente, cuando salimos al pasillo logra localizarse. Tarda más en converger que la localización local.



Resultados obtenidos sobre el robot real

Muchas partículas: https://youtu.be/pX2NzS_RohY

Como hemos establecido una cantidad de partículas muy elevada tarda más en converger. Podemos observar como al principio sabe que está en un pasillo, pero no en cual y posteriormente acota esto a dos puntos sobre una zona simétrica. Finalmente, de esos dos puntos se queda con el correcto cuando llega a una zona que deja de ser simétrica.



Pocas partículas: https://youtu.be/1_6483fu_ZA

Con pocas partículas no llega a converger correctamente, se queda muy próximo a su localización correcta, pero finalmente las partículas entorno a esta desaparecen y el robot se pierde.s