Instituto Tecnológico y de Estudios Superiores de Monterrey Campus Estado de México

Escuela de Ingeniería y Ciencias

Integración de Robótica y Sistemas Inteligentes

Grupo 501
Equipo 11 - La Voluntad

Evidencia:
Reporte Filtro de Kalman Extendido aplicado a Puzzlebot

Profesor:
Dr. Alejandro Aceves López
Dr. Aldo Iván Aguilar Aldecoa
Dr. Miguel Ángel Galvez Zuñiga
Dr. Alf Kjartan Halvorsen
Dr. Francisco Javier Ortiz Cerecedo
Dr. Arturo Vargas Olivares

Alumno

Bruno Sánchez García	A01378960
Carlos Antonio Pazos Reyes	A01378262
Manuel Agustín Díaz Vivanco	A01379673

Atizapán de Zaragoza, México a 5 de junio de 2023

Reto: Exploración Autónoma

El reto se trata de lograr la programación y configuración del robot móvil *Puzzlebot*, para navegación autónoma y reactiva para exploración de terreno inexplorado. Consiste en dos partes:

- Primero la implementación de un nodo de localización con algoritmo Filtro de Kalman Extendido usando odometría del robot y marcadores ArUco. Con base en estos elementos, el robot debe tener una estimación de localización mucho más precisa que con únicamente odometría desde los *encoders*, y simultáneamente debe poder detectar y obtener mediciones de los ArUco mediante visión computacional.
- La segunda parte es la unión entre localización del robot y la implementación de navegación reactiva para realizar exportación desconocida, donde dentro de un mapa, el robot debe recorrer una trayectoria de mínimo 4 puntos (el último punto es la vuelta a su origen) con un obstáculo en cada trayectoria entre cada par de puntos. Para la evasión de obstáculos se puede usar los algoritmos bug0, bug1, bug2 o alguno otro bug desarrollado en el mini reto de navegación reactiva con lidar.

Los resultados deben poderse observar en los simuladores de Gazebo y Rviz. Particularmente, en Gazebo, la evasión de obstáculos, recorrido de la trayectoria y detección de marcadores ArUco, y en Rviz debe enfatizarse la visualización de la covarianza y la estimación de la localización mediante algoritmo de Filtro Kalman Extendido.

Solución

Para la resolución del reto se optó por el algoritmo de bug2, ya que este algoritmo necesita de alta precisión de odometría pues se basa en evasión de obstáculos desde una línea que cruza el objeto, por lo que la condición de salida necesita mucha precisión de su localización para saber cuando cruza. El bug0 necesita precisión en su angulación para salir del estado seguir la pared, pero la incertidumbre de la angulación es más sensible pues es un robot *differential drive*. Confirmamos esta sensibilidad en las pruebas realizadas, aunque no era mucho, tenía un mayor error en la orientación.

Por otra parte, para el modelo de observación elegimos usar el propuesto por Manchester Robotics de estimar la distancia euclidiana y diferencial de ángulo del robot a los ArUcos. Esto se eligió así ya que al tener el tiempo en nuestra contra, obtener un modelo nuevo de observación, en nuestro caso, no justificaba el tiempo que se iba a invertir con una mejora de resultados. De la misma manera, al analizar los resultados con el Filtro, vemos que el modelo ofrece resultados lo suficientemente precisos para cumplir una trayectoria de mínimo 4 puntos y obstáculos en cada trayecto. Con mayor disponibilidad de tiempo, valdría la pena proponer, implementar y principalmente evaluar el desempeño de otro modelo de observación.

Para las mediciones mediante sensores exteroceptivos, decidimos usar la cámara y la librería OpenCV principalmente debido a la extensa documentación y amplia discusión en los foros. Dentro de los primeros resultados, y principalmente en simulación, el detector de ArUcos de OpenCV ofrece una robustez muy alta, incluso para este proyecto algo excesiva. En la pruebas físicas la robustez fue similar, por lo que es muy satisfactorio el desempeño del

detector, por lo que decidimos no utilizar otro método de sensado, que en un futuro valdría la pena implementar y evaluar un método con lidar, o usando la librería de ArUcos de ROS.
El sensor que sí evaluamos y comparamos es el sensor simulado: entrega la distancia euclidiana y diferencial de ángulo a partir de la odometría sin ruido del robot. Pero en la fase de corrección, al tener en el modelo de observación y el sensado dependientes de la odometría, la corrección no es tan precisa en comparación con sensado desde la cámara, por eso optamos por este método de sensado.

Lógica de Programación
La solución del reto se plantea de la siguiente forma:

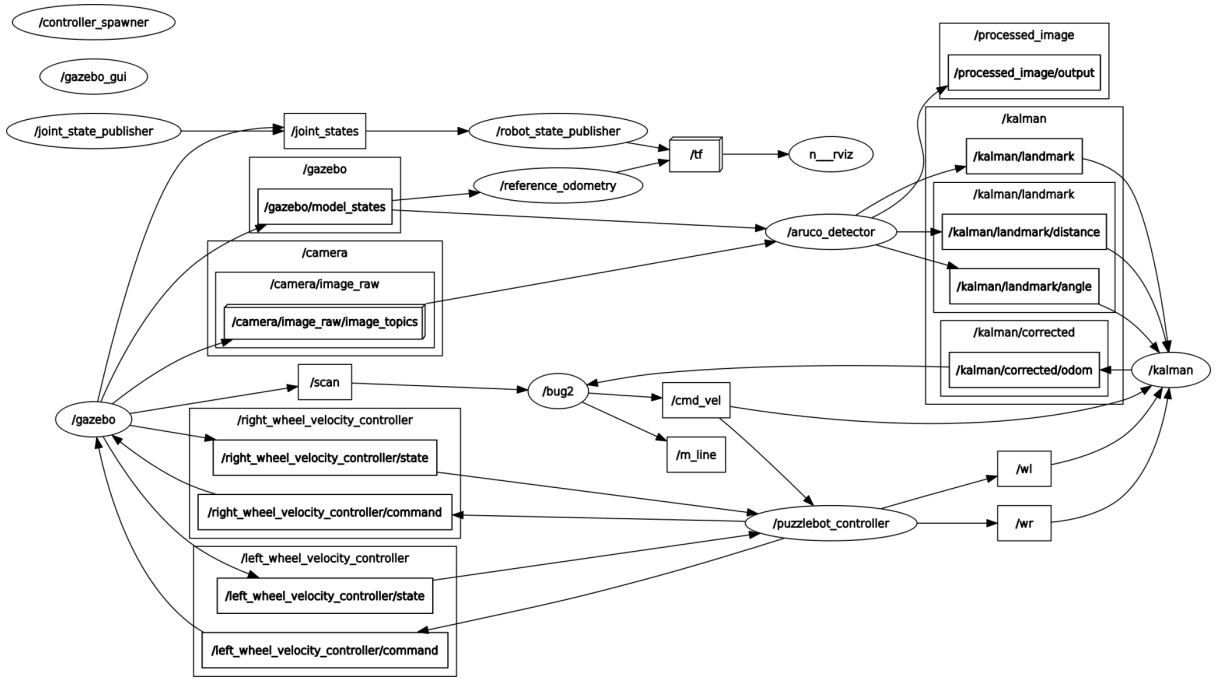

Imagen 1: Diagrama rqt de los nodos y tópicos activos en la solución simulada del reto

La solución se compone esencialmente de 3 nodos:

- kalman: la implementación del algoritmo de Filtro Kalman Extendido. Publica la odometría con la correción del estado localización del robot.
- aruco_detector: toma la imagen e implementa el sensado para el modelo de observación a partir de la detección de ArUcos. Publica las coordenadas del punto (x,y,z) del ArUco detectado, la distancia euclidiana del robot al ArUco y el diferencial de ángulo entre el robot y el ArUco para la corrección de odometría del Filtro Kalman.
- bug2: la implementación del algoritmo bug2. Publica para el control del robot en */cmd_vel.*

Los demás tópicos y nodos son los pertenecientes a las simulaciones Gazebo y Rviz, así como toda la información del robot para resolver el reto, como el lidar, información proporcionada en simulador por el socio formador Manchester Robotics mediante su repositorio de Github:

https://github.com/ManchesterRoboticsLtd/TE3003B_Integration_of_Robotics_and_Intelligent_Systems.git.

Detección de ArUcos

La detección de ArUcos y obtención de mediciones para el modelo de observación se hace a partir de la cámara, en particular de la librería de ArUco de OpenCV.

La esencia del sensado por cámara reside en las funciones:

- opencv.aruco.detectMarkers: recibe una imagen, diccionario y parámetros del detector. Devuelve las esquinas, ids y candidatos rechazados de cada ArUco perteneciente al diccionario usado dentro de la imagen. El procesamiento se hace en dos partes principalmente:
    a. detección de candidatos: filtrar formas cuadradas en la imagen con base en segmentación por umbrales adaptativos y luego por bordes, descartando los bordes más pequeños, lejanos, más pegados entre ellos, no convexos, etcétera que hacen que no asemejen a un cuadrado.
    b. lectura de codificación: a cada candidato primero aplica una transformación de perspectiva para obtener la forma original del ArUco, luego aplica segmentación Otsu para binarizar y determinar/contabilizar bits blanco y negros y finalmente analiza si corresponden al diccionario usado. [2]
- cv2.solvePnP: estiman la pose (rotación y traslación) de un conjunto de puntos objeto. Recibe la imagen, los puntos objeto, los puntos en la imagen, la matriz intrínseca de la cámara, el vector de coeficientes de distorsión de la cámara, y opcionalmente la bandera de Adivinanza Extrínseca, y el método de solución PnP (*Perspective-n-Point*). Devuelve el vector de rotación, el vector de traslación y un *retval*. Dentro de este contexto, el *frame* de la cámara tiene su eje z como aquel que sale de la lente de la cámara, el eje x hacia su derecha, y eje y hacia abajo. Se basa en la proyección de los puntos en un sistema coordinado de mundo a un plano de imagen usando el modelo de proyección II y los parámetros de la matriz intrínseca de la cámara. dispone de varios métodos para obtener la traslación y rotación, como lo son el método Iterativo, P3P o IPEE. [3]

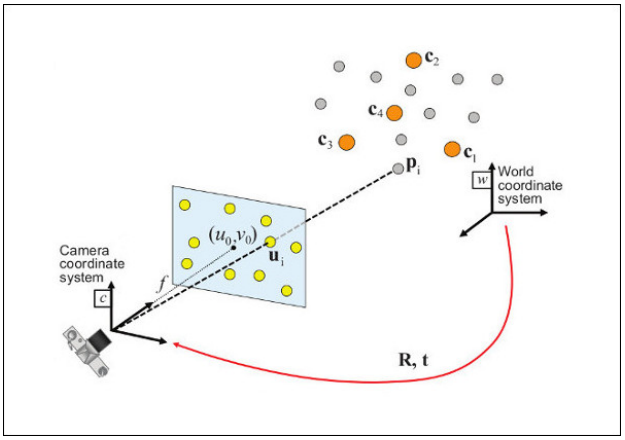

Imagen 2: método solvePnP. (OpenCV, docs.opencv.org/)

Nosotros decidimos utilizar el diccionario 5X5-50 pues no necesitamos más de 20 ArUcos, y cada uno con un tamaño de impresión razonable para pruebas físicas, y para un tamaño equivalente en simulación.

El nodo encargado de la obtención de la distancia euclidiana y diferencial de ángulo se hace de la siguiente manera:

1. Detectar ArUcos
2. Si existe alguno, sino devolver distancia euclidiana de -1
3. Para cada ArUco detectado, obtener su vector rotación y traslación, calcular la distancia euclidiana (en metros) y diferencial de ángulo (en radianes) y guardar dentro de diccionario con llave su id
4. Encontrar el ArUco con menor distancia euclidiana: el más cercano al robot
5. Si la distancia es menor a 4 metros, publicar el punto donde se encuentra el ArUco, su distancia al robot y su diferencial de ángulo correspondientes.

Para facilidad y precisión en la simulación, para la publicación del punto coordenado de cada ArUco en el mundo, nos suscribimos al tópico de los modelos de Gazebo: */gazebo/model_states*, buscamos de entre los modelos, el ArUco encontrado y publicamos su posición. Esto permite facilidad y rapidez para editar los elementos en el mundo simulado.
En la vida real, cada ArUco debe estar especificado con su id en un diccionario, y acorde al id detectado, se publican sus coordenadas.

El mundo simulado también nos proporciona la facilidad de la matriz intrínseca de la cámara, en el tópico */camera/camera_info*, siendo la matriz el parámetro K, y el vector de coeficientes de distorsión siendo el parámetro D del mensaje del tópico. En físico obtuvimos estos parámetros calibrando la cámara con una imagen de tablero de ajedrez y mediante un tablero CharUco.

Imagen 3 y 4: obtención de distancia euclidiana y diferencial de ángulo. Perspectiva detección de ArUco con OpenCV

Con ello, el nodo siempre manda la información necesaria para que el Filtro Kalman pueda hacer la corrección. Publica la distancia al ArUco, las coordenadas del ArUco detectado, y el

diferencial de ángulo del ArUco con el robot, cada uno en un tópico respectivo; y manda -1 cuando no hay detección de ArUco, para que el filtro tenga una señal que le indique no hacer la corrección debido a falta de *landmark*.

Ocurre en simulación, que la cámara detecta básicamente todos los ArUcos dentro del rango de visión siempre y cuando se encuentren completos, sin importar el tamaño, la orientación y la lejanía, por lo que es importante mantener el condicional de el ArUco más cercano, y de una distancia menor a 4 metros. Aunque en la realidad puede suceder que se vean más de 1 ArUco a una distancia y orientación razonable, pero el modelo de observación no contempla más de 1 ArUco, por lo que es importante mantenerlo también para pruebas físicas.

Finalmente, tanto en pruebas físicas como en simulación, la detección de ArUco nunca es infalible, es decir que nunca detecta el ArUco en todas las iteraciones aún cuando existe un ArUco válido dentro del rango de visión, por lo que el publicador de distancia tiende a hacer envíos intermitentes entre la distancia calculada en la detección del ArUco, y -1 a pesar de tener el ArUco y el robot en las exactas mismas posiciones. Por ello, el nodo tiene un contendor que asegura que el detector efectivamente no detecte ArUco por al menos 15 iteraciones para publicar -1.

Filtro de Kalman Extendido

El nodo encargado de implementar el algoritmo Filtro de Kalman Extendido aplicado al *Puzzlebot*. Sigue el algoritmo mostrado a continuación:

Algorithm 1 EKF Localisation with known data association

1. **function** EKF Localisation ($M, \boldsymbol{\mu}_{k-1}, \boldsymbol{\Sigma}_{k-1}, \mathbf{u}_k, \mathbf{z}_{i,k}, \mathbf{Q}_k, \boldsymbol{R}_k$)
2. $\hat{\boldsymbol{\mu}}_k \leftarrow \mathbf{h}(\boldsymbol{\mu}_{k-1}, \mathbf{u}_k)$
3. $\mathbf{H}_k \leftarrow \nabla_{\mathbf{s}_{k-1}} \mathbf{h}(\mathbf{s}_{k-1}, \mathbf{u}_k)|_{\mathbf{s}_{k-1}=\boldsymbol{\mu}_{k-1}}$
4. $\hat{\boldsymbol{\Sigma}}_k \leftarrow \mathbf{H}_k \boldsymbol{\Sigma}_{k-1} \mathbf{H}_k^T + \mathbf{Q}_k$
5. **if** $\mathbf{z}_{i,k}$ **corresponds to landmark** $\mathbf{m}_i \in \boldsymbol{M}$
6. $\hat{\mathbf{z}}_{i,k} \leftarrow \mathbf{g}(\mathbf{m}_i, \hat{\boldsymbol{\mu}}_k)$
7. $\mathbf{G}_k \leftarrow \nabla_{\mathbf{s}_k} \mathbf{g}(\mathbf{m}_i, \mathbf{s}_k)|_{\mathbf{s}_k=\hat{\boldsymbol{\mu}}_k}$
8. $\mathbf{Z}_k \leftarrow \mathbf{G}_k \hat{\boldsymbol{\Sigma}}_k \mathbf{G}_k^T + \mathbf{R}_k$
9. $\mathbf{K}_k \leftarrow \hat{\boldsymbol{\Sigma}}_k \mathbf{G}_k^T \mathbf{Z}_k^{-1}$
10. $\boldsymbol{\mu}_k \leftarrow \hat{\boldsymbol{\mu}}_k + \mathbf{K}_k(\mathbf{z}_{i,k} - \hat{\mathbf{z}}_{i,k})$
11. $\boldsymbol{\Sigma}_k \leftarrow (\mathbf{I} - \mathbf{K}_k \mathbf{G}_k)\hat{\boldsymbol{\Sigma}}_k$
12. **return** $\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k$

Imagen 5: pseudo-codigo algoritmo Filtro de Kalman Extendido. (MCR, mayo 2023)

La Imagen 5 explica paso a paso las ecuaciones a implementar. Los pasos 1 a 4 corresponden a la predicción en el algoritmo de Kalman, en donde se obtiene la mejor estimación del estado y su covarianza, mientras los pasos 5 a 12 corresponden a la corrección en el algoritmo de Kalman.
En orden:
2. Modelo de transición del estado: una función h que depende del estado anterior y el vector de control.
3. Matriz jacobiana del modelo de transición del estado.
4. Matriz de covarianza del estado del robot: a partir de la matriz jacobiana, la matriz de covarianza anterior, y la matriz no-determinística de error.
6. Modelo de observación para la fase de corrección: una función g que depende de un *landmark* (punto conocido siempre en el sistema coordenado del mundo donde se mueve el robot) y la mejor estimación del estado obtenida en la fase de predicción.
7. Matriz jacobiana del modelo de observación.
8. Matriz de covarianza del modelo de observación: a partir de la matriz jacobiana, la matriz de covarianza del estado del robot, y una matriz R que correlaciona el error en el método de medición mediante sensores exteroceptivos para la fase de corrección del algoritmo.
9. Ganancia de Kalman.
10. Corrección del estado del robot.
11. Corrección de la covarianza del estado del robot.

La implementación dentro de un nodo de ROS es como la siguiente:

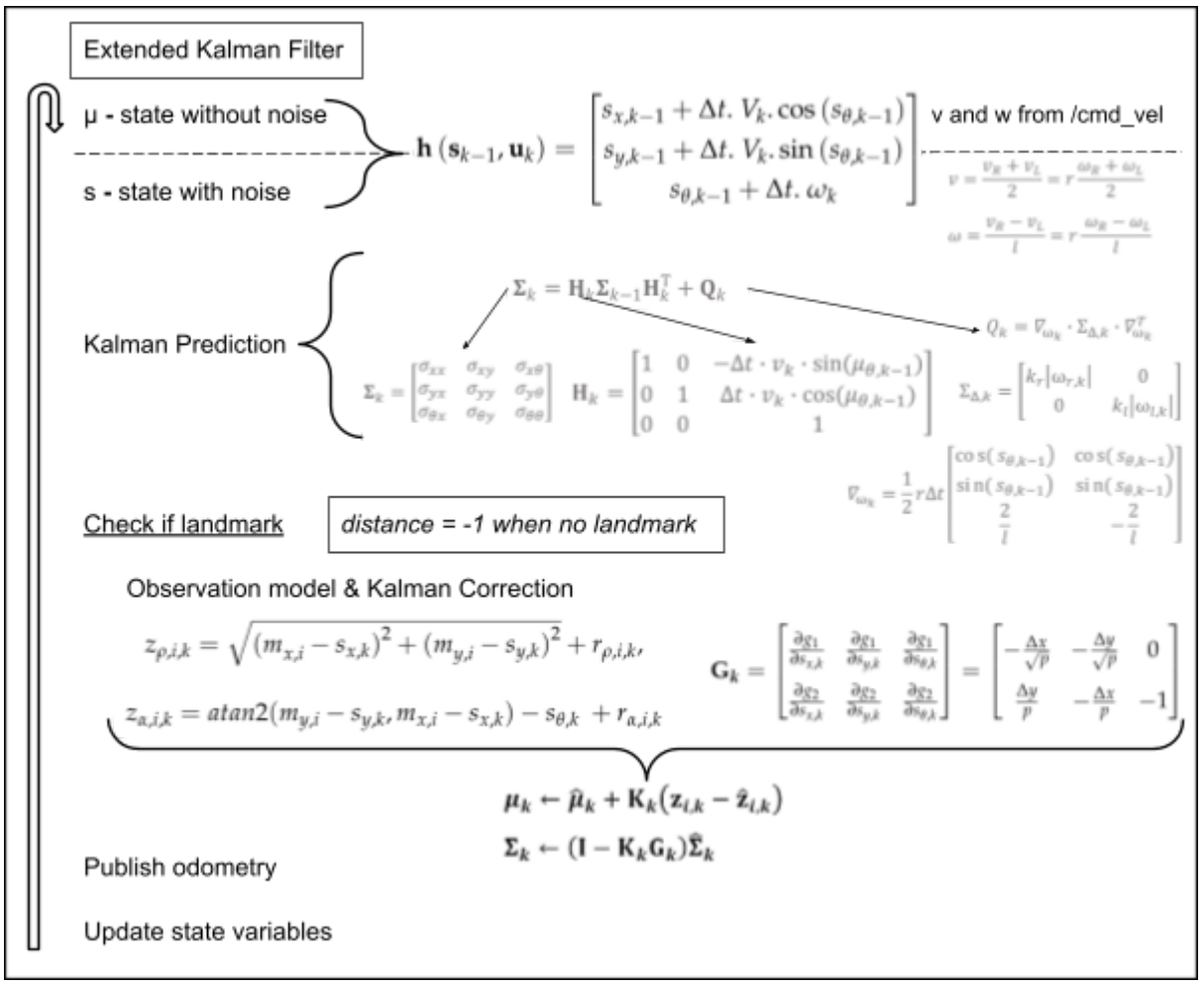

Imagen 6: lógica de implementación del Filtro de Kalman Extendido para Puzzlebot

Se muestra arriba la particularidad de la implementación de Filtro de Kalman Extendido al *Puzzlebot*. Los primeros pasos, correspondientes a la fase de predicción, son aquellos que involucran la odometría y predicción. El modelo de transición del estado se hace con las ecuaciones de odometría del robot, y la predicción utiliza la su matriz jacobiana, covarianza y matriz no determinística. Necesitamos obtener el estado sin ruido, y el estado con ruido para poder obtener la matriz de covarianza en la predicción. La corrección se hace únicamente cuando el nodo detector de ArUco publica una distancia mayor a -1, es decir sí hay detección. El modelo de observación corresponde al propuesto por Manchester Robotics: obtener la distancia euclidiana y el diferencial de ángulo a partir del punto coordenado del ArUco y el estado del robot. El detector de ArUco entrega las mismas variables, pero medidas por la cámara (independiente del estado del robot) para poder hacer la corrección. Calcula la ganancia de Kalman con la covarianza y la matriz Jacobiana del modelo de observación, y de ahí calcula el estado y covarianza corregidos. Esta información se publica para el funcionamiento de bug2 que hará el control para la navegación autónoma.

Bug2

Este algoritmo de navegación reactiva basada en lidar, traza una línea imaginaria entre el origen y el punto meta. El robot se mueve hacia el punto de meta, cuando encuentra un obstáculo lo rodea, y cuando vuelve a encontrar la línea, se redirige hacia la meta. Cuando está moviéndose a la meta, no es necesario que vaya sobre la línea, sino únicamente como criterio para dejar de rodear un obstáculo.

Para la implementación del algoritmo bug2 se construyó una máquina de 3 Estados:

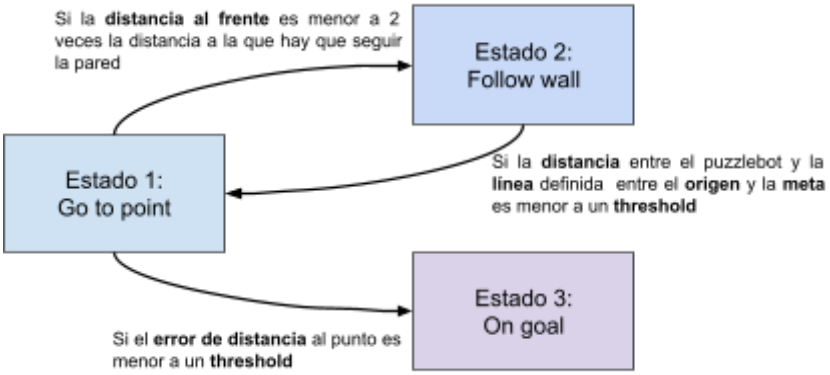

Imagen 7: Diagrama de máquina de Estados del algoritmo bug 2.

De la máquina de Estados destacan en las transiciones las constantes para cambiar de Estado, pues estos valores fijos van a ser clave para sintonizar exitosamente el algoritmo, por ejemplo para disminuir los casos donde se hace intermitente la transición entre Estado 1 y 2 pues ambas condiciones son verdaderas, como cuando que el robot detecta una pared a una distancia menor a 2 veces la de seguir la pared, y al mismo tiempo se encuentra a una distancia a la línea menor al *threshold* establecido. Esta sintonización de constantes es clave para la versatilidad de la solución, sobre todo porque se busca una navegación en terreno desconocido.

Para el Estado 1 de ir hacia la meta, se tiene un pequeño controlador para corregir la angulación del robot hacia el punto meta, donde se obtiene el diferencial de ángulo y se multiplica por una ganancia proporcional que se manda a la velocidad angular del robot. Con velocidad lineal constante. Alternativamente se puede hacer un control similar al diferencial angular pero con la distancia euclidiana del robot al punto meta para la velocidad lineal.

El controlador es el siguiente:

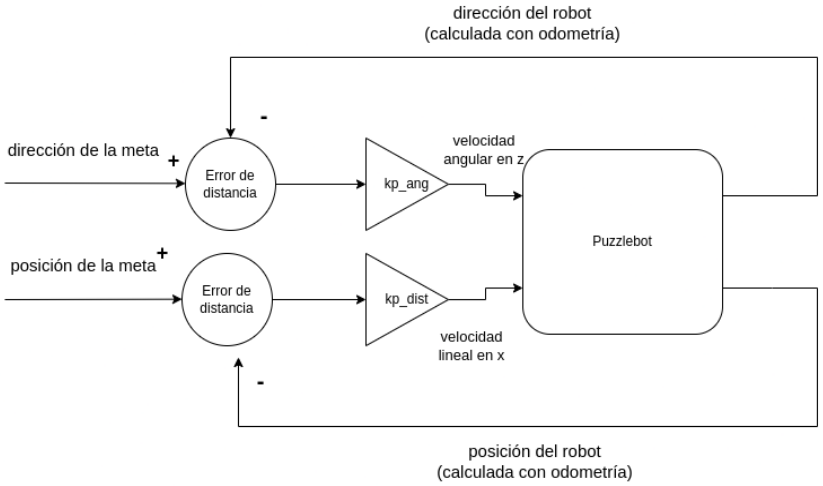

Imagen 8: Diagrama de la ley de control go to point para mover el robot hacia el punto de meta.

El Estado 1 es el que depende de la odometría corregida del Filtro de Kalman, pues determina si ha llegado a la meta o no a partir de su localización y de la del punto objetivo.

Para el Estado 2 de seguir la pared se implementó un controlador según el lado que se desee (derecha o izquierda), la distancia de la pared, y las velocidades lineal y angular específicas. El controlador se especifica así:

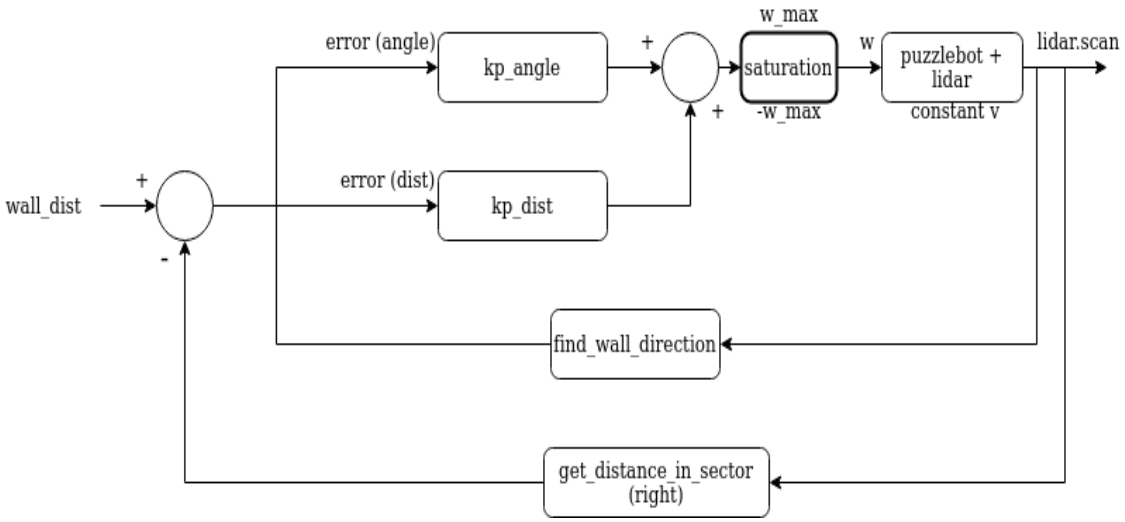

Imagen 9: Diagrama de la ley de control follow_wall para seguir la pared a la derecha.

De la Imagen 9 se describe un controlador de velocidad angular del robot para ir en paralelo a la pared de un obstáculo y a cierta distancia, a partir de dos variables: el ángulo de orientación de la pared con respecto al robot (obtenido en *find_wall_direction*), y el diferencial de la distancia deseada a la pared (obtenida en *get_disstance_in_sector*). Sumadas y multiplicadas por su respectiva ganancia proporcional, controlan la velocidad angular del robot, mientras avanza a una velocidad constante.

La Imagen 9 describe el controlador para el lado derecho, más para seguir la pared del lado izquierdo, hay que multiplicar el resultado antes de la saturación por -1, y en las funciones *find_wall_direction* y *get_distance_in_sector* especificar *left* como argumento, que se encarga de el mismo procesamiento pero en espejo.

Para los giros cuando no hay pared (por ejemplo para rodear un cubo), al encontrar el ángulo de la pared en *find_wall_direction,* ésta devuelve NaN cuando calcula con infinitos (medición que sobrepasa el rango máximo del lidar), se manda al robot girar con radio igual a la distancia a la pared mediante la proporción $r = \frac{v}{w}$.

En este Estado se encuentran las otras constantes a sintonizar más importantes que son las ganancias para seguir la pared. Las otras son las velocidades lineal y angular máximas, particularmente esta última es importante para poder darle al robot la velocidad suficiente para ajustarse a los giros más abruptos en obstáculos.

Finalmente, en el Estado 3, el robot llega al fin de una trayectoria, por lo que el waypoint cambia al siguiente. En caso de que sea el último, se publica 0.0 en todas las velocidades para detener el robot en el punto. Para cambiar al siguiente punto objetivo, debemos marcar una bandera que se ha llegado al objetivo por primera vez, ya que la condición de llegada depende de un *threshold* de distancia a la meta, por lo que será verdadera en cada iteración dentro de este radio.

El bug2 y su correcta sintonización son claves para una implementación robusta del algoritmo, principalmente porque el algoritmo de Kalman no contempla situaciones como cuando el robot choca una pared y se atora mientras los encoders continúan avanzando. De forma similar no podemos depender del lidar ya que a distancias demasiado cortas, como una colisión no garantiza una medida lo suficientemente fidedigna para asegurarnos que ha chocado el robot.

Descripción de los Experimentos de Desempeño

Para probar la robustez de nuestro robot, diseñamos una ruta con 6 puntos, donde para hacer la trayectoria a otro punto se tienen que hacer giros de hasta 90 grados. Además se añadieron cuatro cajas que se encuentran a la mitad de 5 puntos, por lo que el *Puzzlebot* debe de utilizar la navegación reactiva para evadir los obstáculos. Finalmente, en las pruebas se fueron eliminando arucos para ver cómo se ve afectado el filtro de Kalman.

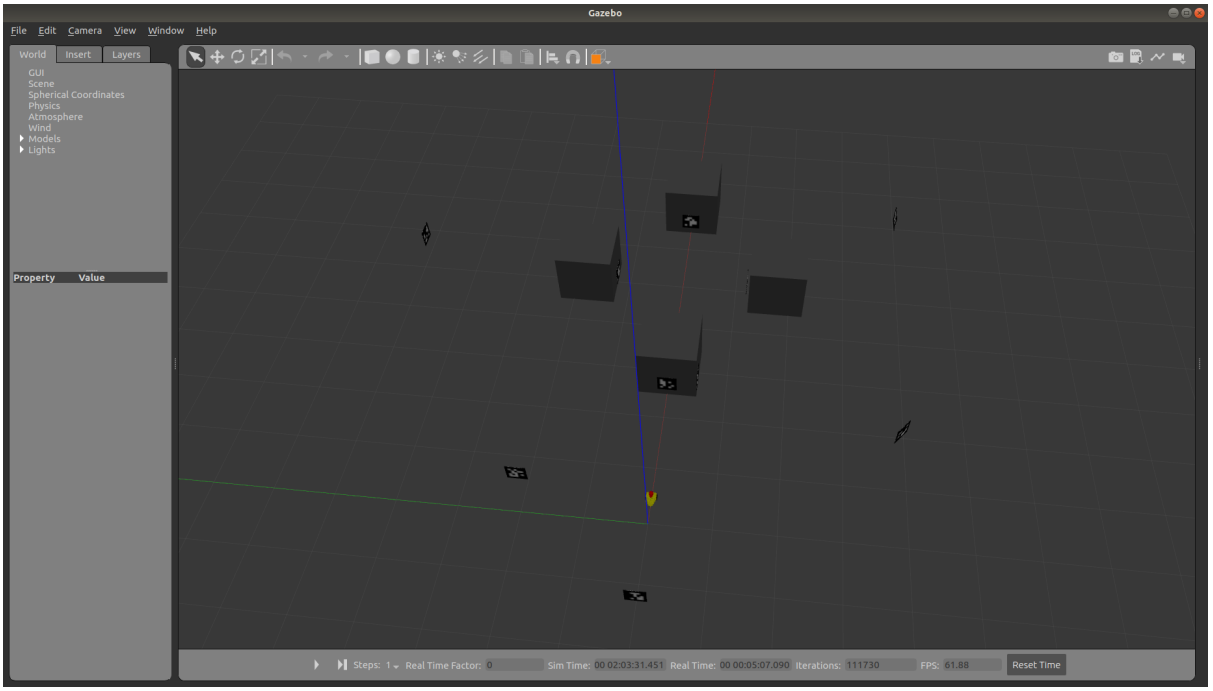

Imagen 10: captura de simulación Gazebo del mundo utilizado

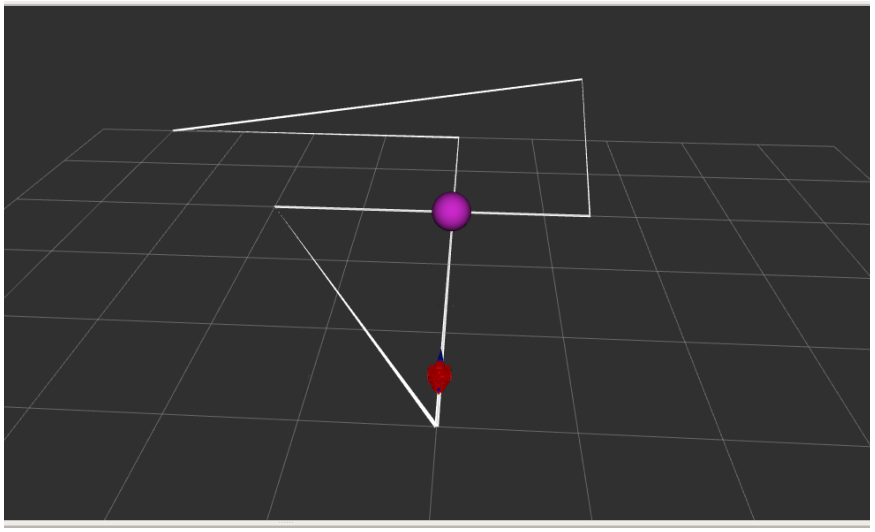

Imagen 11: captura de simulación Rviz de la trayectoria utilizada

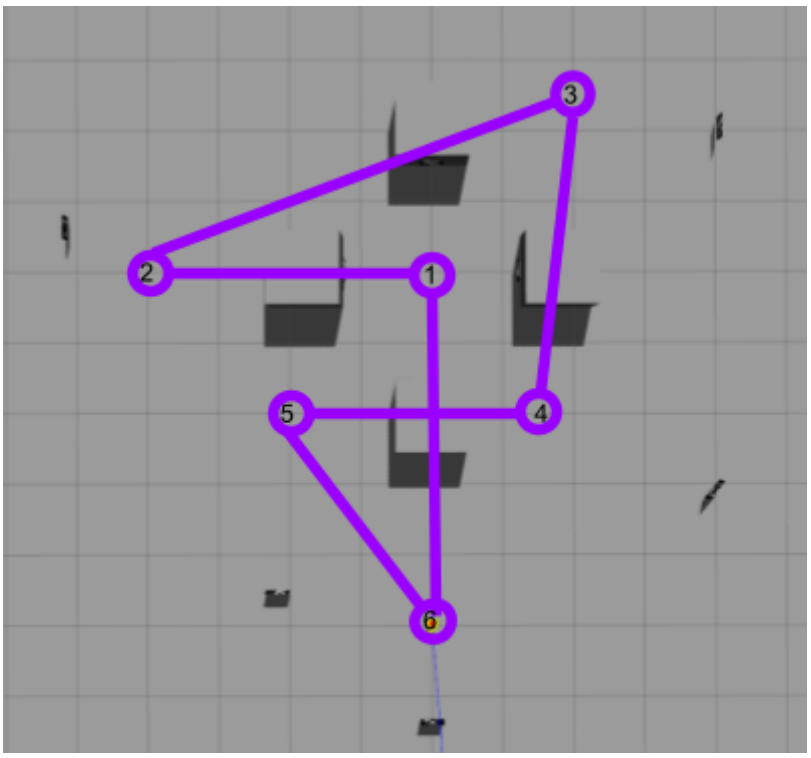

Imagen 12: trayectoria utilizada dibujada en el mapa de gazebo

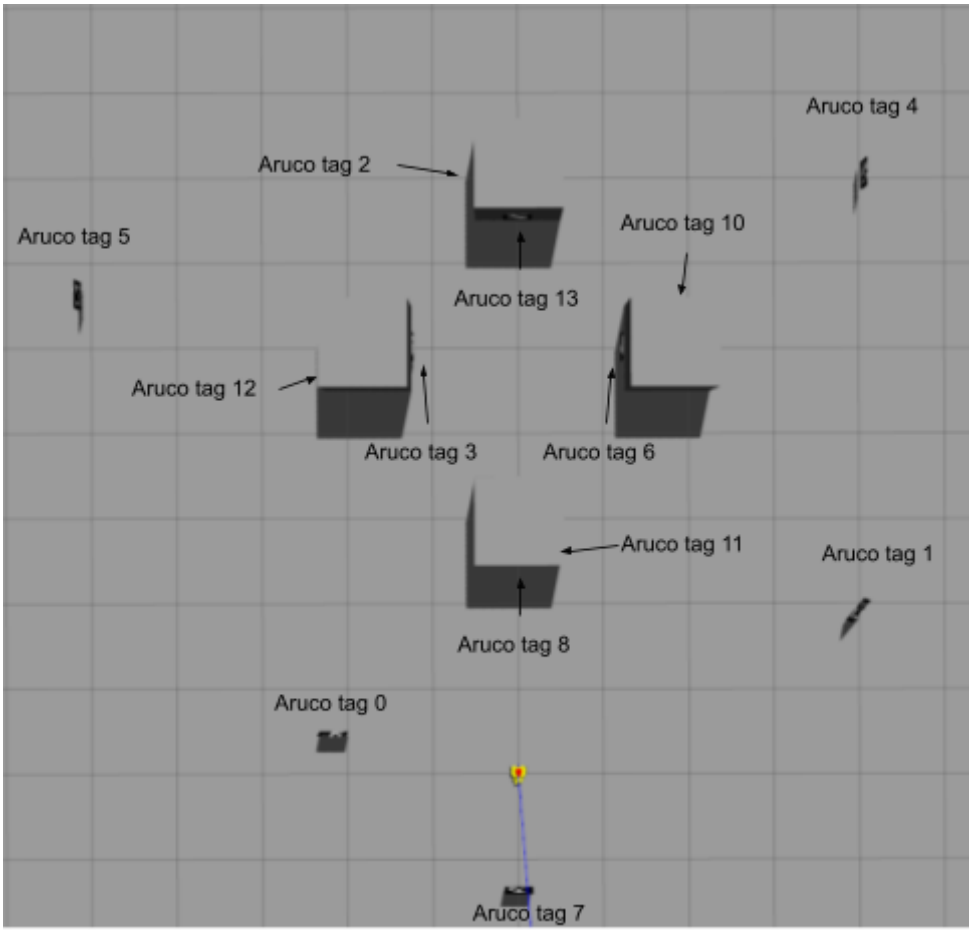

Imagen 13: referencias de ArUcos en el mundo Gazebo

Tabla 1: resultados cuantitativos de las pruebas de desemepeño

N°	Número de arucos	Error en x para punto final	Error en y para punto final	Observaciones
Todos los arucos presentes				
1	13	4.5 cm	2.3 cm	• La mayor parte del recorrido el puzzlebot se encontraba dentro de la covarianza, en caso de que no, solamente salía por unos 5 cm aproximadamente
2	13	6.6 cm	-2.0 cm	• La mayor parte del recorrido el puzzlebot se encontraba dentro de la covarianza, en caso de que no, solamente salía por unos 10 cm aproximadamente
3	13	4.4 cm	0.6 cm	• La mayor parte del recorrido el puzzlebot se encontraba dentro de la covarianza, en caso de que no, solamente salía por unos 5 cm aproximadamente
Aruco eliminado: aruco tag 7				
4	12	47 cm	-15 cm	• Al intentar llegar al punto y al no haber un aruco cerca de la meta, la covarianza del puzzlebot comenzó a crecer rápidamente y el robot se desvió en gran medida de la trayectoria, sin embargo, logró detectar un aruco que lo ayudó a retomar un poco la trayectoria.
5	12	86 cm	-41 cm	• En múltiples ocasiones el puzzlebot salió de la covarianza y demostró un comportamiento inesperado al ir al punto 3. El robot se movió muy rápido en una dirección equivocada, pero al detectar un aruco se recuperó
6	12	48 cm	-14 cm	• En la trayectoria al último punto, la covarianza del puzzlebot creció bastante y el robot se salió de ella por unos 10 cm. • El puzzlebot tuvo un comportamiento similar a la prueba 4
Arucos eliminados: aruco tag 5, aruco tag 4 y aruco tag 1				
7	10	X	X	• Recorrido fallido, al no tener arucos en los waypoints, en el punto 3, la covarianza creció demasiado, lo que provocó que el puzzlebot se saliera de la trayectoria, sin esperanza de volver a ver un aruco
8	10	2 cm	-4cm	• Se llegó a separar de la covarianza por aproximadamente 20 cm • llega a los puntos, menos al final que si tiene un aruco cerca, con bastante error, pero dentro de la covarianza

9	10	2 cm	-2 cm	• Las covarianzas crecen mucho, llega a los puntos intermedios con mucho error, pero al punto final, al tener un aruco, se llega con un error mínimo
Arucos eliminados: aruco tag 8, aruco tag 12 y aruco tag 13				
10	10	18 cm	-15 cm	• En el último aruco, el puzzlebot se salió por aproximadamente 20 cm de la covarianza
11	10	-17 cm	-20 cm	• Comportamiento similar a la prueba 10 • Al no tener suficientes arucos, en la trayectoria al punto tres, la covarianza creció en gran medida, haciendo que el puzzlebot recorriera una ruta inesperada, sin embargo logró detectar otro arauco por el centro y logró corregirse
12	10	X	X	• Trayectoria no completada, al no tener un aruco inicial, la covarianza creció mucho y el puzzlebot se desvió bastante de la trayectoria, llegó a la esquina del obstáculo, donde detectó la m-line, ocasionando que el robots alineará al punto y no le diera tiempo suficiente para corregir la distancia a la pared, lo que ocasionó la colisión

Resultados

La implementación de la solución se puede ver en:
Simulación:
https://drive.google.com/file/d/1yNpWKjWp_Juj0GfuQFMgNc_AfnBCehhI/view?usp=sharing

Fisico:
https://drive.google.com/file/d/1Z1MowJo9i2Rx8oNItYspimTw0rAMvuCk/view?usp=sharing

Código de la solución en:
https://github.com/RoboticaInteligente8voTecCEM2023/advanced_navigation_mobile_robots.git

Video presentación para Manchester Robotics:
https://drive.google.com/file/d/1LXe50Rd2nwZUiwYz4AS2irvtAeMWoyaP/view

Discusión Ventajas y Limitaciones

Con las pruebas de desempeño vemos principalmente que la precisión de localización aumenta muy significativamente comparado con odometría sin Filtro Kalman. Llega con mucha precisión a todos los puntos en la trayectoria y se mantiene siempre cerca del robot

real. El robot se encuentra dentro de la covarianza conforme se mueve. Cuando se observa un ArUco, la covarianza disminuye y el ajuste de localización se acerca más al robot.
El bug2 es un algoritmo que necesita una precisión considerable en la posición del robot, por lo que si no hay ArUcos suficientes, el robot podría entrar a un ciclo infinito en el estado follow wall, donde la posición estimada nunca llegue a cruzar la m-line. Por esto mismo, valdría la pena explorar la alternativa del bug0, para poder comparar y obtener lo mejor de ambos algoritmos. Un punto que es importante mencionar es que el Filtro de Kalman y la simulación en Rviz tienen que tener un valor igual en el frame rate por parte del Rviz y el rospy Rate de Kalman para poder sintonizarlos y que se pueda visualizar de manera correcta el comportamiento.
Una particularidad de nuestro modelo de observación es que el resultado y la trayectoria se ve altamente afectada por el número de ArUcos. El error de cada punto aumenta considerablemente si no cuenta con un ArUco cerca para hacer la corrección y la trayectoria se ve afectada por los ArUcos intermedios que se encuentran entre los puntos.
Dadas estas particularidades, valdría la pena también explorar un algoritmo de navegación reactiva que considere choques con obstáculos, pues ni esta implementación de Filtro Kalman, ni algoritmos bug, consideran colisiones con obstáculos, por lo que algún choque resulta fatal para la localización del robot.
Finalmente, el objetivo del reto es poder obtener una localización precisa en terrenos desconocidos, por lo que la robustez es vital. La posición de los ArUcos es vital en este reto, por lo que es muy importante saber donde ponerlos y que efectivamente ayuden a la corrección del robot. Esencialmente, hay que agregar un ArUco cada cuando la covarianza sea demasiado grande y la localización muy alejada del robot real para que pueda corregirse y no perder la trayectoria o chocar al momento de evadir un obstáculo.
El reto se logró implementarse en el *Puzzlebot* físico, y precisamente nos enfrentamos a las cuestiones de sintonización y ubicación de los ArUcos, pues a pesar de funcionar en simulación, en la realidad funciona distinto, similar, y la localización efectivamente es más precisa, pero sí hubo de hacer varias escalaciones, sintonización y equivalencias de manera empírica para ello.

Conclusiones

El Filtro de Kalman es un algoritmo altamente robusto para la localización de un un robot pues mejora la precisión en la odometría a cuando se hace sin el Filtro. Presenta errores lo suficientemente pequeños como para considerarlos despreciables en ciertos casos. Entre sus ventajas se pudo observar que permite la repetibilidad de las trayectorias al reducir el ruido y las desviaciones que pudiesen surgir debido a emplear únicamente los modelos de observación y predicción por separado. Al ser un algoritmo recursivo, está constantemente retroalimentandose, lo que permitirá que siempre se tenga la mejor estimación con los datos medidos y estimados hasta cada instante de tiempo.
En este proyecto se ha visto como el Filtro es un método efectivo para manejar el ruido inevitable en un robot *differential drive*.

Referencias

[1] Díaz Vivanco, M. A. Pazos Reyes, C. A. Sánchez García B. (2023). Navegación Reactiva: bug0 y bug2 con sensor lidar[Unpublished]. Instituto Tecnológico y de Estudios Superiores de Monterrey.

[2] Bradski, G. (2000). The OpenCV Library. *Dr. Dobb's Journal of Software Tools*. https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html

[3] Bradski, G. (2000). The OpenCV Library. *Dr. Dobb's Journal of Software Tools*. https://docs.opencv.org/4.x/d5/d1f/calib3d_solvePnP.html

[4] Ogata, K. (2010). *Modern Control Engineering*. Prentice Hall.

[5] Alexandru Stancu, Autonomous Mobile Robots, Lecture Notes, The University of Manchester, 2023.