UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

LOCALIZACIÓN DE UN ROBOT MÓVIL UTILIZANDO FILTRO DE KALMAN Y REDES NEURONALES CONVOLUCIONALES

TESIS
QUE PARA OPTAR POR EL GRADO DE
MAESTRÍA EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

PRESENTA:

JULIO CÉSAR MARTÍNEZ CASTILLO

TUTORES

DR. JESÚS SAVAGE CARMONA
FACULTAD DE INGENIERÍA, UNAM

DR. JOSÉ DAVID FLORES PEÑALOZA
FACULTAD DE CIENCIAS, UNAM

CIUDAD UNIVERSITARIA , CIUDAD DE MÉXICO, FEBRERO 2023

Al Posgrado en Ciencias e Ingeniería en Computación de la UNAM, por la formación que me han dado durante mi estancia.
A Ana María Paula, mi abuelita que me cuidas desde el cielo.
En verdad, gracias.

Julio César Martínez Castillo

Reconocimientos

Agradezco sinceramente ...

... a Eulalia y Galdino, mis padres, por se la fuente principal de apoyo y por haberme puesto en este camino.

... a Anabhel, mi hermana, quien me ha acompañado y dado motivación durante este tiempo en el posgrado.

... a Ana Maria Paula, que desde el cielo, me resguarda y vió mi progreso durante este camino.

... a la vida, por permitirme conocer a Rosario, quien me brindo su apoyo y cariño incondicional en todo momento.

... a CONACYT, a través de la beca 782096, y a DGAPA-UNAM por el financiamiento, a través del proyecto PAPIIT AG101721 "Modelos computacionales para el comportamiento adaptable de robots de servicio y de humanos".

... a mis mentorores, el Dr. Jesús Savage Carmona y el Dr. Jose David Flores Peñaloza, por compartir sus conocimientos, por guiar mi avance académico durante el posgrado y sobre todo, por gran paciencia.

... a todos los integrantes que conforman el Laborotario de Biorobótica de la UNAM.

... y a los miembros de mi jurado, Dr. Marco Negrete, Dr. Miguel Ángel Padilla, y Dr. Ulises Olivares, por sus valiosas recomendaciones.

Resumen

Dentro del área de la robótica móvil, la localización es uno de los componentes mas importantes cuando de navegación se refiere. Para ello, existen diferentes métodos aplicables para localizar a un robot por medio de la información recibida por medio de sus sensores. Cabe mencionar que cada método es diferente del otro, por ende, tiene sus ventajas y desventajas. Para cuestiones del presente trabajo, abordaremos el método de filtro de Kalman extendido (EKF por sus siglas en inglés), el cual es un algoritmo que permite conocer el estado oculto dentro de modelos no lineales. El método requiere de componentes externos que le den información para llevar acabo sus dos etapas, las cuales son la predicción y actualización del modelo.

Para este trabajo, los componentes que fueron utilizados son marcas naturales fijas dentro de un entorno limitado, que en este caso es un laboratorio de investigación. Para la detección de estas marcas naturales, fueron implementadas técnicas de visión computacional en conjunto de técnicas de reconocimiento de patrones (en este caso redes neuronales convolucionales) que le permiten al robot detectar y obtener sus coordenadas de referencia. Este sistema de visión se encuentra alojado dentro de una GPU embebida, ya que el modelo de la red neuronal requiere un alto poder de procesamiento para ejecutarse.

En este trabajo se detalla una metodología propuesta basada en la unión de ambos sistemas, el filtro de Kalman extendido y la detección de marcas naturales para localización de un robot móvil. También es mencionado el análisis de los resultados obtenidos al poner a prueba el sistema conjunto con marcas previamente entrenadas por la red neuronal.

Abstract

In the field of mobile robotics, localization is one of the most important components when it comes to navigation. To achieve this, there are different methods that can be applied to locate a robot using information received through its sensors. Each method is different from the others, therefore, it has its advantages and disadvantages. For the purposes of this work, we will discuss the extended Kalman filter (EKF) method, which is an algorithm that allows knowing the hidden state within nonlinear models. The method requires external components that provide information for its two stages, which are prediction and model update.

For this work, the components that were used were fixed natural landmarks within a limited environment, which in this case is a research laboratory. For the detection of these natural landmarks, computer vision techniques were implemented along with pattern recognition techniques (in this case, convolutional neural networks) that allow the robot to detect and obtain its reference coordinates. This vision system is hosted within an embedded GPU, as the neural network model requires high processing power to run.

This work describes a proposed methodology based on the combination of both systems, the extended Kalman filter and the detection of natural landmarks for the localization of a mobile robot. It also mentions the analysis of the results obtained when testing the combined system with landmarks previously trained by the neural network.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Se conoce a la robótica como la ciencia de percibir y manipular el mundo físico a través de dispositivos controlados por computadora. La representación física de la robótica la podemos ver aplicada en los robots, los cuales están constituidos por componentes como motores sensores y la unidad de procesamiento.

Los robots están diseñados para desempeñar diferentes tareas dentro de áreas, las cuales por lo general son áreas cerradas. Un ejemplo de ello son los robots de servicio llamados ROOMBA, los cuales son robots cuya tarea es limpiar el piso mientras estos navegan. Para cumplir esta tarea, los robot tienen integrados sensores que les indican donde se encuentran las paredes para no quedar bloqueados y puedan seguir su labor. Otro tipo de robots que se desplazan para realizar una tarea son los seguidores de línea, los cuales tienen como objetivo seguir una ruta previamente trazada.

También existen otro tipo de robots, los cuales están diseñados para hacer manipulación de objetos de un lugar a otro, conocidos como brazos robóticos. Estos brazos están conformados por un conjunto de articulaciones que le permiten manipular objetos con gran precisión. Comúnmente, los brazos robóticos se utilizan dentro de la lineas de producción para diferentes tareas que requieren manipular. Ejemplo de estas tareas son soldar piezas metálicas, empacar productos y clasificar componentes, entre otras. Por lo tanto, dependiendo de cual sea el objetivo o la tarea a cumplir, son los requerimientos que el robot debe tener para realizar la tarea para la cual esta destinado.

Una gran parte de los robots móviles necesita saber su localización dentro de un entorno para poder navegar hasta un punto determinado y realizar una acción. El robot puede identificar objetos por medio de una cámara, pero la cantidad de objetos a entrenar está limitado por el almacenamiento interno del robot. Por lo cual, se busca tener un pre-entrenamiento de objetos pertenecientes al lugar (por ejemplo si detecta una cama, el robot sabe que está en una recamara) y este pueda navegar sabiendo en que parte del espacio se encuentra. La detección de los objetos clave para entrenar se obtiene por medio de un algoritmo de visión artificial para obtener las características del ambiente en el que se encontrará el robot.

1.1. Justificación

EL propósito del presente escrito busca continuar el trabajo desarrollado en una tesis previa. Está tesis consistió en tener un robot móvil que se localiza en un ambiente controlado por medio del filtro de Kalman y la identificación de puntos de referencia. Para estos puntos fueron utilizados códigos ArUco (Augmented Reality marker of the Univeristy of Cordoba por sus siglas en inglés).

Dentro de una situación real no se tienen señas o imágenes de referencia como lo son los códigos QR (Quick Response code por sus siglas en íngles), se busca que ahora el robot tenga un conjunto de objetos pre-entrenados los cuales tienen una categoría asociada a la habitación en la que se encuentre y esto le ayude al robot a localizarse.

Este conjunto de imágenes de los objetos entrenados tiene diferentes muestras capturadas de varias perspectivas. Es decir, se tienen imágenes de los objetos clave de diferentes vistas para que el robot pueda identificarlos y localizarse. Una vez teniendo un conjunto de datos para ser entrenados, estos son entrenados y clasificados por una de las tantas técnicas de clasificación como lo son redes neuronales convolucionales.

Cabe mencionar que la arquitectura de la red neuronal convolucional influye mucho en el entrenamiento del conjunto de datos. Existen redes neuronales convolucionales que se especializan en realizar entrenamientos y detección de imágenes etiquetadas con las regiones de interés dentro de una escena. Para cuestiones del presente trabajo, escogeremos una arquitectura la cual nos pueda ayudar a entrenar y detectar marcas naturales provenientes de una cámara instalada dentro del robot.

1.2. Hipótesis

Se puede replicar el comportamiento humano de orientación y ubicación utilizando la vista dentro de un entorno. Este comportamiento se puede reproducir en un robot de servicio por medio de un sistema principal compuesto de un sub-sistema de localización basado en el filtro de Kalman y un sub-sistema de detección por visión computacional. El sistema desarrollado consiste en reconocer objetos propios del ambiente que brinden información al robot del espacio por el cual navega.

1.3. Objetivo

1.3.1. Objetivo general

Desarrollar una metodología la cual estima la localización de un robot móvil implementando el filtro de Kalman e identificando y clasificando dichos objetos clave por medio de técnicas de clasificación y de visión computacional.

1.3.2. Objetivos específicos

- Implementar un sistema de detección y de clasificación basado en redes neuronales convolucionales para que el robot móvil identifique las marcas naturales por medio de su visión.
- Utilizar el filtro de Kalman que se trabajó previamente para estimar la posición del robot para que en conjunto con la localización de cada objeto que sea detectado, se pueda estimar la posición dentro del entorno.
- Realizar pruebas del sistema propuesto dentro de una GPU que pertenece al robot verificando su rendimiento.

1.4. Estructura de la tesis

En base a los puntos ya anteriormente descritos de los que se fundamentan los propósitos de este escrito, la estructura del mismo consiste en los siguientes capítulos:

- Para el segundo capítulo se abarca el marco teórico, el cual contiene las bases de la arquitectura de los robots de servicio así como su funcionamiento.
- Dentro del tercer capítulo se detalla el aspecto de como los robots de servicio se localizan dentro de un ambiente. Además, se detalla como es que se utiliza el filtro de Kalman como método de localización dentro de un entorno.
- En el cuarto capítulo se explican las técnicas de visión que el robot implementa para extraer la información del ambiente y este a su vez pase por un preprocesamiento para trabajar con los datos obtenidos.
- En el quinto capítulo se mencionan las técnicas de reconocimiento que se desarrollaron para ser implementadas dentro del robot, identificando las marcas naturales dentro de una imagen.
- Para el sexto capítulo se detalla la metodología que se siguió para implementar el sistema reconocedor de imágenes con cada una de las formas de reconocimiento de patrones, y conjuntarlo con la localización por medio del filtro de Kalman.
- Dentro del séptimo capítulo se registran los resultados que se obtuvieron al implementar estos tipos de reconocedores de patrones, así como la efectividad de cada uno para determinar cual de estos cumple mejor la tarea de localización.
- Por ultimo, para el octavo capítulo se presentan conclusiones a las que se llegaron a partir de las pruebas realizadas con los sistemas dentro del robot. De igual manera, se detallan las áreas de oportunidad que se presentaron para ser considerados para trabajo futuro.

Capítulo 2

Marco teórico

Dentro de este capitulo se explica qué son los robot móviles y se describe el como están constituidos los robots de servicio: las partes que lo conforman (en este caso la arquitectura VirBot vista en la figura 2.1), en que consiste cada una y como se encuentran implementadas dentro de un robot.

2.1. Robots Móviles

Los robots móviles son los robots que tienen la capacidad de desplazarse dentro de un medio predeterminado. La forma en la que estos pueden desplazarse puede ser de manera teleoperada por un usuario o de manera automática. Dado que el propósito del presente trabajo consiste en que un robot de servicio se localice dentro de una habitación para llevar acabo una tarea precargada, nos enfocaremos en los robots que trabajan en forma autónoma.

Los robots móviles están constituidos con una arquitectura, la cual esta conformada por varios bloques. Estos bloques tienen asignada una tarea, la cual puede ser obtener información del medio en el que se desplaza, otro modulo encargado del procesamiento que interpreta la información de los sensores para que posteriormente lleve acabo una toma de decisiones para llevar acabo una acción y otro modulo de actuadores los cuales se encargan de realizar movimientos.

Dentro del laboratorio de bio-robótica de la Facultad de ingeniería de la UNAM, se propuso una arquitectura llamada ViRBot (por sus siglas en inglés Virtual Real Robot). Esta arquitectura fue creada para poder diseñar, organizar, integrar y probar software orientado a los robots de servicio autónomos (30). Esta arquitectura está conformada por cuatro bloques representados en la figura 2.1, los cuales son: un bloque de entrada, otro bloque de planeación, otro bloque que maneja el conocimiento y un bloque de ejecución. Estos bloques, a su vez, están conformados de otros sub-bloques específicos relacionados a su bloque principal. Para la navegación y localización de un robot de servicio, nos podemos apoyar de el bloque de planeación y del bloque de manejo del conocimiento.

Para motivos de la navegación del robot, vamos a considerar el sistema locomotor que implementa para desplazarse en varias direcciones dentro de un entorno. También vamos a considerar como sensor una cámara para obtener información del entorno por el cual navega y le ayude a localizarse.

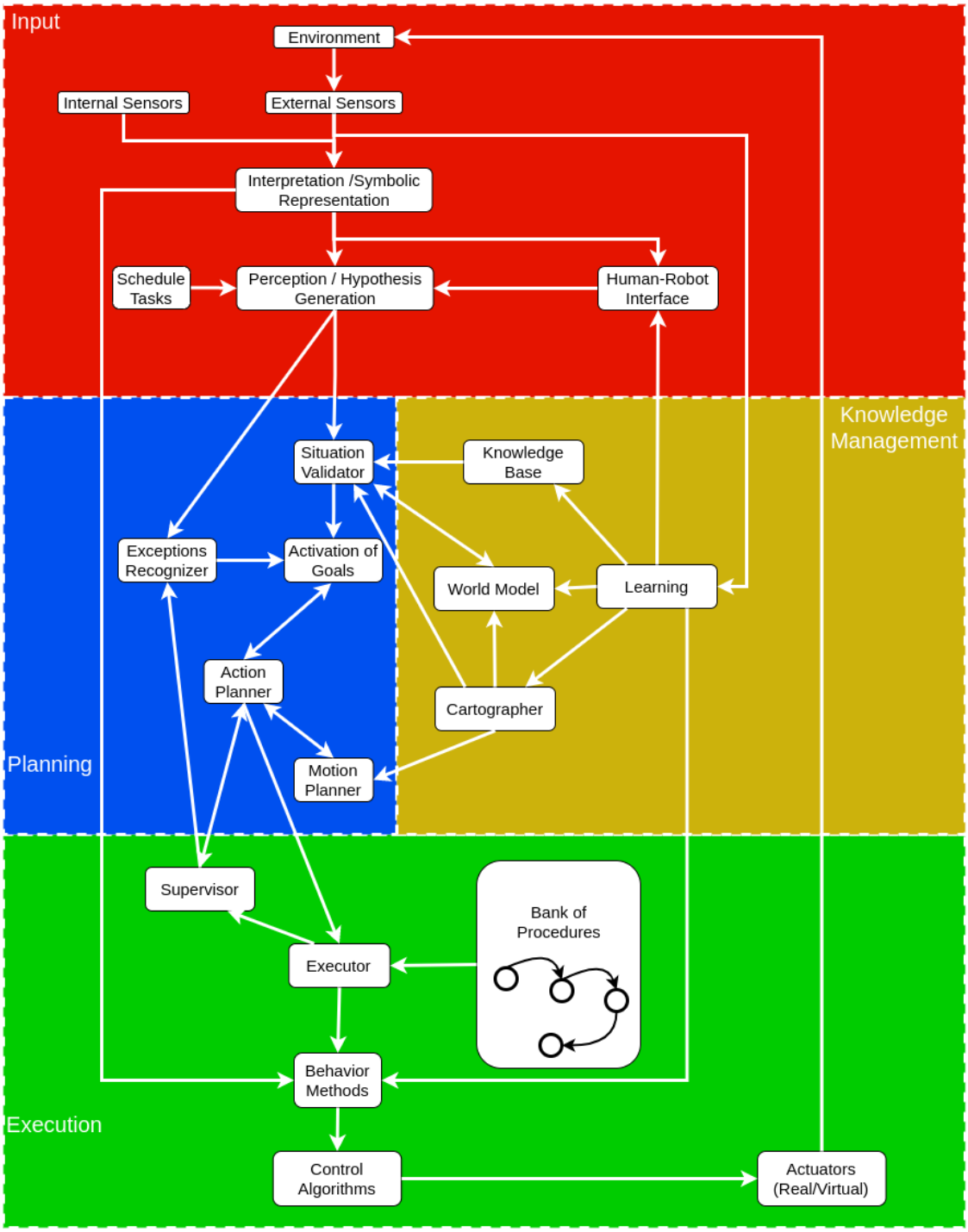

Figura 2.1: Representación del esquema VirBot. Tomado de (30).

2.2. Posición y orientación (pose)

Un robot de servicio debe conocer su posición dentro del entorno donde navega. Para ello, el robot maneja un sistema de coordenadas dentro de un plano para ir de un punto A a un punto B. Aparte de la posición, también necesita conocer su orientación, puesto que es una referencia que le indica donde girar para orientar su navegación.

Podemos nombrar a este concepto de la posición el nombre de *pose*, con el cual haremos referencia a las coordenadas de la posición de un robot respecto a su eje. A este concepto también se le puede agregar el término de la orientación del robot. Para cuestiones propias de éste trabajo, y considerando que implementamos robots móviles de servicio, los cuales de desplazan en una casa, podemos manejar nuestro sistema de referencia dentro de un plano.

El uso del termino *pose* hace referencia a la tripleta de componentes (x, y, θ), en donde las componentes x y y (haciendo referencia a la posición) son valores reales asociados a un punto fijo en el plano y la componente θ (haciendo referencia a la orientación) hace referencia al ángulo en el que se encuentra orientado el robot respecto al eje x del sistema coordenado.

En la figura 2.2 podemos ver de manera gráfica el como esta conformada la *pose* del robot. Dentro de la gráfica tenemos una referencia global dada por G y la posición del robot desde el dentro de un plano generado por los ejes x y y, así como la orientación representada con el ángulo θ que se forma a partir de los ejes.

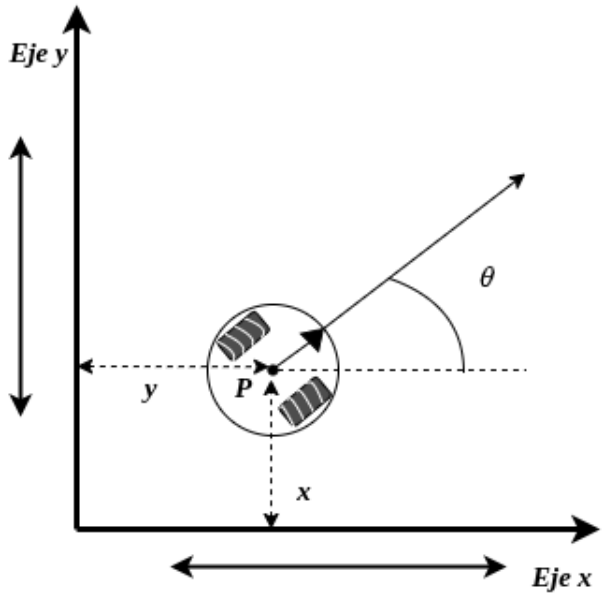

Figura 2.2: Representación de la pose en coordenadas (x, y, θ).

2.3. Cinemática en un robot móvil

La cinemática dentro de un robot móvil es importante, porque es la parte en la que se implementa un actuador, que en este caso es un sistema mecánico. Este sistema hace que el robot pueda desplazarse dentro de un entorno predefinido. En algunos casos específicos, el sistema tiene implementado un control dedicado a seguir una ruta en específico. Estas consideraciones del control de movimiento se realizan dentro del robot porque dependiendo de su locomoción, el robot móvil tendrá restricciones en sus movimientos. Para este caso, consideramos movimientos de traslación y/o movimientos de rotación sobre su propio eje.

A partir de la cinemática del robot móvil, podemos establecer un sistema de referencia (localización) asociado a los movimientos del robot. Este sistema es necesario porque nos ayuda a conocer la posición de un robot móvil dentro de su entorno. Este sistema debe ser constante, es decir, debe ir obteniendo los movimientos a lo largo del tiempo y calculando la posición actual del robot.

Se pueden presentar factores externos que afecten a los movimientos del robot haciendo que al cabo de varios movimientos, el robot no se encuentre exactamente donde se esperaría estar. Estos factores ocurren en cada movimiento, por lo cual, el sistema debe ser lo más robusto posible para que con ayuda de otros métodos estadísticos (mencionados más adelante), puedan ir corrigiendo la ruta hacia un punto determinado.

2.3.1. Cinemática de un robot diferencial

Para cuestiones de este trabajo, vamos a considerar un robot móvil, cuya locomoción esta constituida por un par diferencial, como se muestra en la figura 2.3. En esta figura se tiene un marco de referencia L, el cual se encuentra asociado al centro del robot de manera local. También se tiene un segundo marco de referencia en general G que se encuentra fijo como referencia. con estas dos referencias, vamos a considerar un ángulo θr, el cual se forma entre marco general y el marco local del robot. Y por último, consideramos dentro del robot un punto de referencia P, el cual, es el punto donde parte el marco de referencia local del robot y a su vez, es el eje central de rotación que tiene el robot móvil.

Con está representación, podemos determinar los tres componentes que describen la posición de un robot móvil, que de forma vectorizada resultan en la ecuación 2.1:

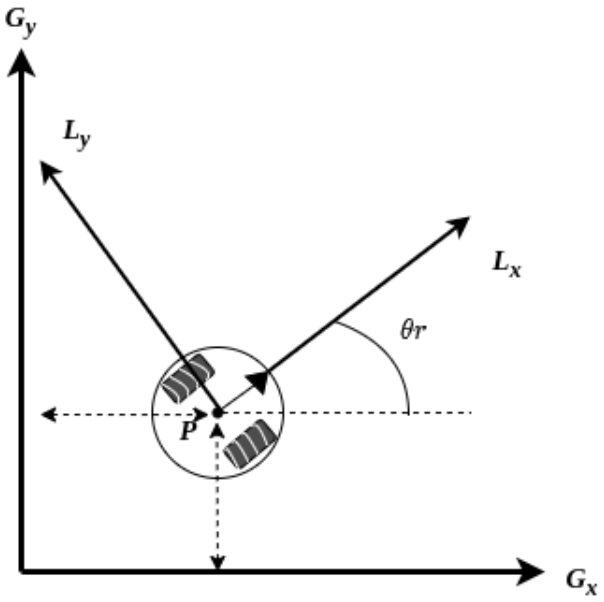

Figura 2.3: Representación de los movimientos de un robot móvil respecto a un marco general.

$$\epsilon\vec{_G} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \tag{2.1}$$

Ahora, para poder describir un movimiento por medio de sus componentes, se requiere hacer un mapeo de los movimientos que se realizan sobre el eje local del robot. Este mapeo lo podemos observar como una matriz de rotación ortogonal como se muestra en la ecuación 2.2:

$$R(\theta) = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

Teniendo la ecuación 2.1 y la ecuación 2.2 previamente definidos, podemos obtener una relación entre el marco de referencia general respecto al marco local del robot dadas las velocidades, quedando en la ecuación 2.3:

$$\vec{\epsilon_R} = R(\theta)\vec{\epsilon_G} \tag{2.3}$$

Podemos poner de ejemplo el resultado que se tiene cuando el robot hace un giro con un ángulo de $\frac{\pi}{2}$ (figura 2.4).

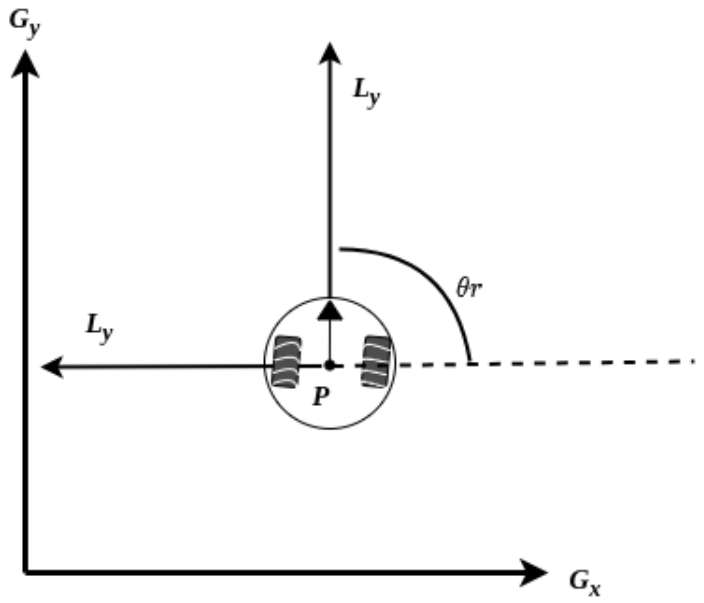

Figura 2.4: Ejemplo del movimiento del robot móvil con un marco de referencia local L y un marco de referencia general G formando un ángulo θ entre ellos de $\frac{\pi}{2}$.

Entonces, dada la información que tenemos de $\frac{\pi}{2}$, podemos plantear la ecuación 2.3 teniendo las componentes de la posición (x, y, θ), resultando en la ecuación 2.4:

$$\vec{\epsilon_R} = R(\frac{\pi}{2})\vec{\epsilon_G} = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} = \begin{bmatrix} x \\ -y \\ \theta \end{bmatrix} \tag{2.4}$$

Por lo cual, la ecuación 2.4 nos dice que el robot móvil está realizando un movimiento sobre el eje L_x cuyo equivalente está en y y el movimiento del eje L_y corresponde a x.

2.3.2. Relación Rueda-movimiento del robot

Para que el robot móvil pueda desplazarse en su entorno, es necesario saber que relación tienen las ruedas con las que realiza movimientos. Es importante considerar esta relación porque puede afectar al movimiento respecto al chasis que contiene el robot. Cada una de éstas características que tienen las ruedas se deben considerar dentro de las ecuaciones de movimiento.

El par diferencial del robot tiene una rueda de cada lado con un diámetro r, separadas entre sí a una distancia l, con un punto P respecto al centro del robot y una velocidad angular del giro de la rueda w. Con estas características podemos obtener una ecuación, la cual calcula la velocidad general del robot dentro del marco general antes mencionado, resultando en la ecuación 2.5:

$$\dot{\vec{\epsilon_G}} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = f(l, r, \theta, \dot{w}_1, \dot{w}_2) \tag{2.5}$$

Si tomamos la ecuación 2.4 cuyo ángulo de giro que se forma es de $\frac{\pi}{2}$, se puede encontrar el movimiento ahora con respecto al marco general desde el marco local, generando así la relación de la ecuación 2.6:

$$\dot{\vec{\epsilon_G}} = R(\frac{\pi}{2})^{-1}\dot{\vec{\epsilon_L}} \tag{2.6}$$

Vamos a considerar que el marco de la referencia local del robot que se uso previamente, en cada una de las llantas que tiene el robot, tienen una velocidad diferente. En una de las llantas realiza una traslación desde el punto P en dirección de L_x mientras que la otra se queda quieta. Esto causará que dado que P es punto medio, el robot se moverá a la mitad de la velocidad, resultando en dos dos ecuaciones 2.7 y 2.8, que al sumarlas dan la componente en L_x de $\dot{\vec{\epsilon_L}}$:

$$\dot{x}_{r_1} = (\frac{1}{2})r\dot{w}_1 \tag{2.7}$$

$$\dot{x}_{r_2} = (\frac{1}{2})r\dot{w}_2 \tag{2.8}$$

El cálculo de los movimientos en el eje L_y es igual a cero, porque tenemos las restricciones que se usaron anteriormente en el robot.

Una vez teniendo las componentes independientes respecto a los ejes L_x y L_y, podemos calcular la velocidad angular del robot ($\dot{\vec{\epsilon_L}}$) con el movimiento que tiene cada llanta y después sumarlas. Si tenemos que la llanta derecha genera una velocidad angular diferente de cero, pero la llanta izquierda genera una velocidad angular igual a cero, el robot genera un movimiento en el marco general en sentido anti-horario. La velocidad angular pero respecto al punto P puede calcularse dado el movimiento que se realiza a lo largo del arco que proviene de un círculo de radio $2l$, resultando en la ecuación 2.9:

$$\vec{\epsilon_G} = R(\theta)^{-1} \begin{bmatrix} \frac{r\dot{w}_1}{2} + \frac{r\dot{w}_2}{2} \\ 0 \\ \frac{r\dot{w}_1}{2l} + \frac{-r\dot{w}_2}{2l} \end{bmatrix} \tag{2.9}$$

En uno de los parámetros de la ecuación 2.9, una de las fracciones contiene un signo menos en la suma de la velocidad angular. Esto establece que la llanta derecha esta detenida mientras que la llanta izquierda realiza un giro positivo, haciendo que el robot gire en sentido horario.

Sabiendo esto, podemos calcular una matriz inversa $R(\theta)$ (ecuación 2.10) y darle los siguientes valores para ejemplificar el funcionamiento: $\theta = \frac{\pi}{2}$, $r = 1$, $l = 1$ y las velocidades $w_1 = 4$ y $w_2 = 2$ de cada llanta. como resultado, obtenemos la velocidad respecto al marco general de tal manera como se muestra en la ecuación 2.11.

$$R(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.10}$$

$$\dot{\vec{\epsilon}}_R = R(\frac{\pi}{2})^{-1}\dot{\vec{\epsilon}}_G = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 3 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 3 \\ 1 \end{bmatrix} \tag{2.11}$$

Con este análisis podemos obtener la posición del robot en un instante en el tiempo calculándolo desde una de las llantas. Se debe considerar que existen movimientos que se vean restringidos que al sobrepasar ese limite, causando fallos a futuro en su mecanismo y termine en un accidente. Es por esto que es importante éste análisis a fondo para poder comprender con mayor precisión los movimientos y restricciones del robot y en todos los movimientos para desplazarse.

2.4. Sensores

Otro tipo de elementos que tienen los robots móviles para interactuar con su entorno son los sensores. Los sensores son el medio por el cual los robots móviles obtienen información del ambiente, transformándola a una medición que el sistema de procesamiento identifica para la toma de decisiones. Los sensores pueden medir diferentes detalles del ambiente como puede ser la cantidad de luminosidad del entorno, la distancia a la que se encuentra de una habitación, saber cuanto avanzo el robot, entre otros aspectos.

Un ejemplo de ello lo podemos ver en las cámaras, ya que para algunos robots móviles funcionan como sensores. Esto se debe a que genera una imagen del ambiente y por medio de técnicas de procesamiento de imágenes, el robot móvil puede extraer características que sea relevantes para realizar las tareas para las que fue diseñado.

2.5. Cámaras

Las cámaras como sensores ayudan a los robots móviles cuando se necesita de trabajar con visión computacional. Con ellas obtenemos imágenes las cuales contienen información identificando los objetos clave (como lo pueden ser muebles o artículos) y extraer sus características relevantes para el robot móvil. Estas características pueden ser el color, tamaño. En caso de personas, posturas o gestos que sean necesarios reconocer para llevar acabo decisiones que conduzcan a una acción del robot.

La implementación de cámaras dentro de robots móviles ha sido de gran importancia puesto que conjuntar los avances en visión computacional en el ámbito de la robótica permite el tratar de imitar el comportamiento humano con respecto a la vista. Para el presente trabajo, utilizamos la información de ciertos objetos para estimar la localización de un robot móvil en un entorno predeterminado.

2.5.1. Cámaras RGB

Existen un tipo de cámaras, las cuales son conocidas como cámaras RGB (por sus siglas en ingles Red, Green y Blue). Estas cámaras contienen un sensor construido por millones de celdas fotosensibles que detectan las diferentes longitudes de onda dando lugar a los colores de la imagen. Estas generan imágenes recolectadas por luz visible para luego convertirlas en señales eléctricas, y posteriormente, organizar esas señales que tienen la información para representar imágenes y secuencias de vídeo. La longitud de onda de la luz con la cual trabajan las cámaras es de 400 a 700 nm, que es el mismo espectro de color que el ojo humano puede detectar.

Este tipo de cámaras buscan recrear las imágenes que que normalmente una persona percibe por medio de la visión humana, captando la luz en longitudes de onda (RGB) roja, verde y azul para una representación precisa del color. Una forma de ejemplificar su funcionamiento puede ver en la figura 2.5. En ella vemos como la cámara RGB toma al objeto y por medio de los lentes que tiene refracta la figura para que los sensores internos RGB detecten las longitudes de onda y de manera matricial se genere el conjunto de píxeles. Un sector de la matriz que se ve en la figura 2.5 vemos que representa la cabeza del sujeto ya en representación de píxeles que para este caso es color negro.

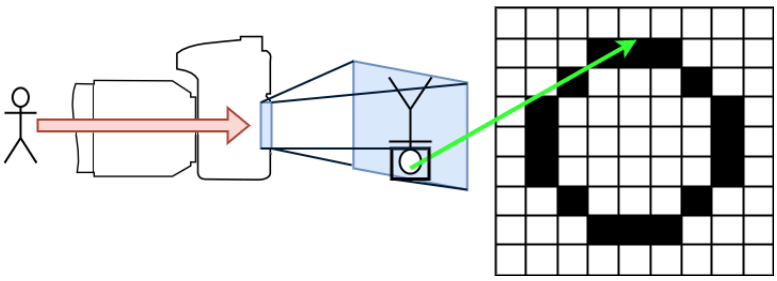

Figura 2.5: Representación del funcionamiento de una cámara RGB.

Las cámaras RGB se ven afectadas de igual forma que la vista humana por la iluminación en la que se encuentra, ya que son afectadas en su rendimiento. Estas variantes a la iluminación se pueden ver pueden ver alteradas debido a las condiciones atmosféricas como la niebla, la bruma, el humo, las olas de calor y el smog.

Por los motivos antes mencionados, las cámaras a menudo deben combinarse con aditamentos para ajustar la iluminación. También existen aditamentos infrarrojos para trabajar en ambientes nocturnos de luz baja, o en entornos con poca visibilidad a causa de factores externos (como ejemplos son la niebla, la neblina, el humo o tormentas de arena).

2.5.2. Cámaras estereoscópicas

Las cámaras estereoscópicas son cámaras que pueden generar imágenes en tres dimensiones. Éstas cámaras buscan replicar la visión binocular (figura 2.6) del ser humano. Esta visión consiste en producir dos imágenes (una para cada ojo) que luego el cerebro interpreta y mezcla creando la imagen 3D. Son utilizadas comúnmente para hacer detección de ciertos objetos que se encuentran sobre superficies (por ejemplo repisas, mesas, objetos en un piso, colgados sobre la pared) aún estando en el interior o en el exterior de una habitación.

Las cámaras replican este comportamiento por medio de la implementación de dos cámaras internas, las cuales toman una imagen de la misma escena para reconstruir la escena por medio de un mapa de paridad. Este mapa de paridad toma los píxeles que se encuentran más lejos se consideran de una baja disparidad, mientras que los píxeles que están más cerca se consideran de una alta disparidad.

Figura 2.6: Ejemplo de una cámara de profundidad. Tomada de (24).

2.5.3. Cámaras con sensor RGB-D

Una cámara RGB-D tiene una tarea más específica, la cual es hacer detección de profundidad dentro de una imagen capturada. Esta imagen se puede aumentar con

información con respecto de la profundidad (relacionada con la distancia del sensor) por cada píxel.

El funcionamiento básico de la cámara RGB-D se lleva a acabo utilizando sensores que detectan la luz infrarroja que recibe de varias direcciones. Estos sensores realizan un cálculo por triangulación para saber a que distancia se encuentra cada píxel generando un mapa de profundidad de la imagen obtenida. La implementación de estos sensores dentro de las cámaras RGB-D puede variar según el fabricante, dado que cada uno de ellos puede agregar y/o configurar su funcionamiento interno.

Una de las cámaras RGB-D más conocidas a nivel comercial es el Kinect de Xbox 360 (Figura 2.7). El Kinect de Xbox 360 es un controlador de juego dedicado al entretenimiento creado por Alex Kipman, desarrollado por Microsoft para la videoconsola Xbox 360, y desde junio de 2011 para PC a través de Windows 7 y Windows 8 (20).

Los componentes básicos que tiene el sensor Kinect son los siguientes (figura 2.7):

- Un sensor láser.
- Una cámara RGB.
- Una cámara de profundidad.
- Un arreglo de micrófonos.
- Un eje de movimiento

El Kinect utiliza los sensores IR (infrarrojos por sus siglas en inglés) de la siguiente forma:

- El láser ilumina a todos los objetos que están dentro de su campo de visión.
- Esta luz rebota en los cuerpos de la escena que se captura y viajan de nuevo hasta el receptor del Kinect.
- El receptor captura la información de la distancia recorrida por la luz por cada píxel.
- Se genera una malla cartesiana que regresa la información de cada píxel en RGB con su respectiva profundidad.

Adicionalmente, el Kinect implementa un sistema de reconocimiento de voz, un sistema reconocedor de poses con las manos y un sistema de reconocimiento facial para la identificación automática de personas. También permite reconstruir de manera virtual ambientes en 3D, tanto muebles como objetos que se encuentren.

El uso de este sistema a llevado que numerosos desarrolladores implementen al sensor para sus investigaciones, dándoles posibles aplicaciones al Kinect que van más allá del propósito desarrollado inicialmente. Para ello, Microsoft desarrollo un software llamado de Kinect SDK, que permite manejar esos sistemas por medio de una computadora (35).

Algunos proyectos donde han sido implementado el Kinect para ciertas tareas en específico son:

Figura 2.7: Foto muestra de Kinect.

- La generación del mapa virtual de una habitación en 3D y hacer que el robot responda a los gestos humanos (34).
- Implementar una interfaz basada en la extensión de JavaScript para Google Chrome llamada depthJS que permite a los usuarios para controlar el navegador con gestos con las manos (6).
- Por medio de módulos desarrollados en código abierto, se han implementado herramientas para hacer renderizado 3D en tiempo real, creando elementos básicos en 3D (21).

2.6. GPU

Los robots de servicio pueden realizar el procesamiento de datos que provienen de los sensores por medio de las unidades de procesamiento (dígase CPU, microcontroladores, microprocesadores, etc.). Pero en algunos casos, cuando la información que obtenemos es de gran tamaño como las imágenes, podemos recurrir a implementar un GPU para que nos ayude a ir manejando la cantidad de información en forma de píxeles que se puede obtener por medio del sensor.

La GPU (por sus siglas de unidad de procesamiento de gráficos) es un circuito el cual esta dirigido a la modificación y a la manipulación rápida de la memoria para la creación de imágenes. Esta información viaja por un buffer de fotogramas hasta llegar a un dispositivo que los pueda visualizar.

Existe un tipo de GPU especiales que es una clase más poderosa, ya que suelen interactuar con la placa base por medio de una ranura de expansión como el PCI Express o por el puerto de gráficos acelerado. Por lo general, se pueden reemplazar o actualizar con relativa facilidad, asumiendo que la placa base es capaz de soportar la

actualización. Algunas tarjetas gráficas en la actualidad usan ranuras de interconexión de componentes periféricos, pero su ancho de banda es tan limitado que generalmente se usan sólo cuando una ranura PCIe (interconexión de componentes periféricos express por sus siglas en ingles) o AGP (puerto de gráficos acelerado por sus siglas en inglés) no está disponible.

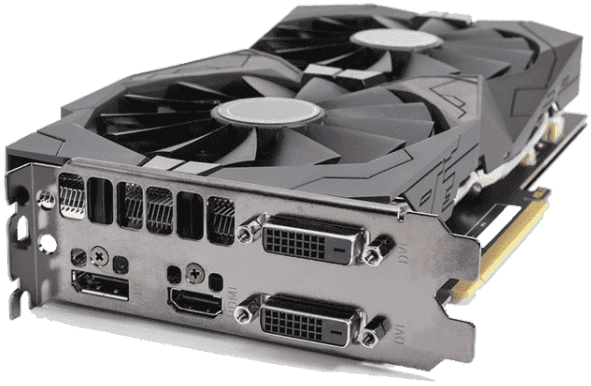

Figura 2.8: Foto muestra de un GPU. Tomado de (8).

2.6.1. GPU y la inteligencia artificial

El uso de GPU por sus características de procesar grandes cantidades de información en forma de píxel, es utilizado en desarrollo de técnicas en inteligencia artificial y en computo paralelo. El uso en computo paralelo ha sido de gran importancia en el campo de investigación a tal grado que se han buscado herramientas para llevar a cabo procesos en paralelo. Es aquí donde el GPU entra, ya que es posible paralelizar cálculos haciendo más eficientes los procesos. Cabe mencionar que un CPU normalmente cuenta con 2 a 8 núcleos dentro de su estructura y que pueden trabajar como si fuesen dobles haciendo llegar hasta 16, en cambio un GPU tiene cientos o miles de núcleos en su composición (figura 2.9).

El CPU al contar con una estructura con circuitos de control de tamaño grande le permite ejecutar de manera eficientemente instrucciones a bajo nivel (o también conocido como código máquina), tiene un funcionamiento a latencia baja pero con grandes frecuencias de reloj, le permite gestionar los puertos de entrada/salida y manejar el acceso a la memoria principal. En cambio el GPU tiene una estructura grande en cuestión de su unidad lógica-aritmética que le permite hacer sus cálculos de manera paralela y a una velocidad mayor a la de un CPU (figura 2.10).

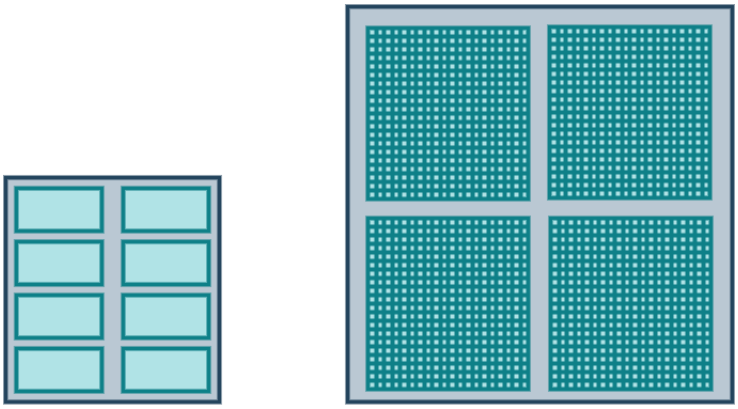

Figura 2.9: Comparación entre los núcleos de un CPU y los núcleos de un GPU.

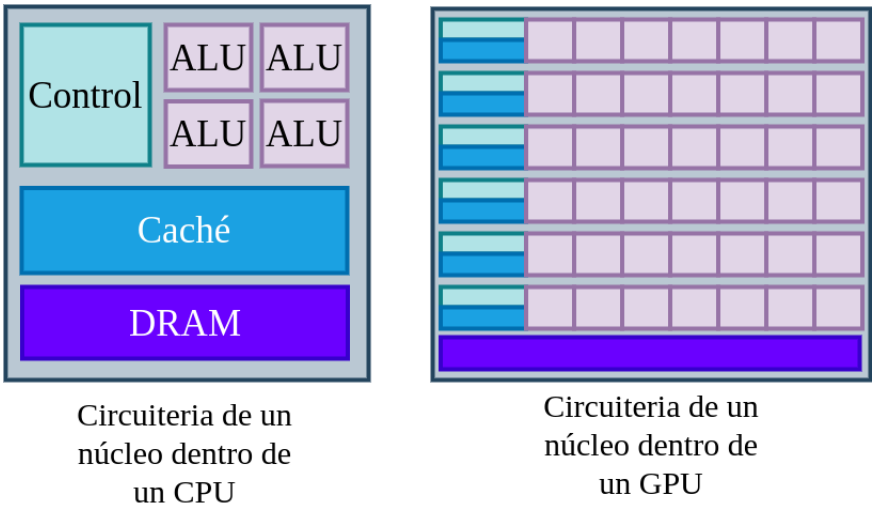

Figura 2.10: Comparación entre los componentes de un núcleo de un CPU y un núcleo de un GPU. Tomado de (28).

Actualmente, para manejar una GPU a nivel código es común encontrarse con código desarrollado en lenguaje de programación C ó $C++$. Una de las más importantes y conocida es CUDA (Compute Unified Device Architecture por sus siglas en inglés) creada por NVIDIA. CUDA fue desarrollada para que trabaje bajo los siguientes términos (4):

- CUDA permite al usuario definir funciones globales o funciones de dispositivos, las cuales se separan en funciones que se ejecuten por el CPU y funciones ejecutadas en paralelo por el GPU por medio de etiquetas.

- CUDA transfiere las funciones creadas en código máquina a la tarjeta de vídeo (GPU) para almacenarlas dentro de su memoria, todo esto antes de que comience a ejecutar cualquier programa.
- El código principal es ejecutado dentro del CPU, y este a su vez obtenga las instrucciones de ejecución al GPU. Cuando son ejecutadas varias copias de un algoritmo estas pasan al GPU para que por medios de hilos se puedan realizar. Al final, una vez concluido esos hilos pasan los resultados al CPU.

2.6.2. Arquitectura y funcionamiento

La arquitectura de un GPU es semejante a la de un CPU. El GPU se diferencia en que es una arquitectura que trabaja en paralelo. Esta arquitectura tiene la cualidad de manejar procesos y atenderlos de forma eficiente. En la figura 2.11 vemos una arquitectura de una GPU, la cual es compatible con CUDA. Esta arquitectura esta constituida de la siguiente forma:

- *Host:* Es un bloque que representa al CPU, el cual envía datos e instrucciones al GPU para que este los ejecute.
- *Ensamblador de entrada (Input assemble en inglés):* Este bloque se encarga de encolar los procesos provenientes del CPU para que sean atendidos de manera secuencial.
- *Administrador de ejecución de subprocesos (Thread Execution Manager en inglés):* Este bloque se dedica a tomar los procesos y asignarlos a los hilos de GPU que estén disponibles para ejecutarlos de forma paralela.
- *Load/Store (carga/almacenamiento en inglés):* En este bloque se encuentran las memorias secundarias del GPU, donde son almacenados los datos resultantes y diferentes variables propias del GPU.

El caso de la memoria global (glbal memory en inglés), forma parte de un conjunto de memorias del GPU, y por ende, se explicará más adelante en el escrito con detalle.

2.6.2.1. Arquitectura de CUDA

CUDA también cuenta con una arquitectura propia, la cual se conforma de tres partes que son básicas para su funcionamiento. El uso de esta arquitectura hace que el usuario aproveche la capacidad de cómputo por completo del GPU con CUDA.

La arquitectura divide al GPU en cuadriculas, bloques e hilos (grids, blocks and threads en inglés) dentro de una jerarquía (15), como se muestra en la figura 2.12. La arquitectura debe ser robusta, puesto que existen subprocesos dentro de un bloque, varios bloques existen en una cuadricula y varias cuadriculas se encuentran alojadas dentro del GPU.

Estos tres componentes los vamos a detallar a continuación:

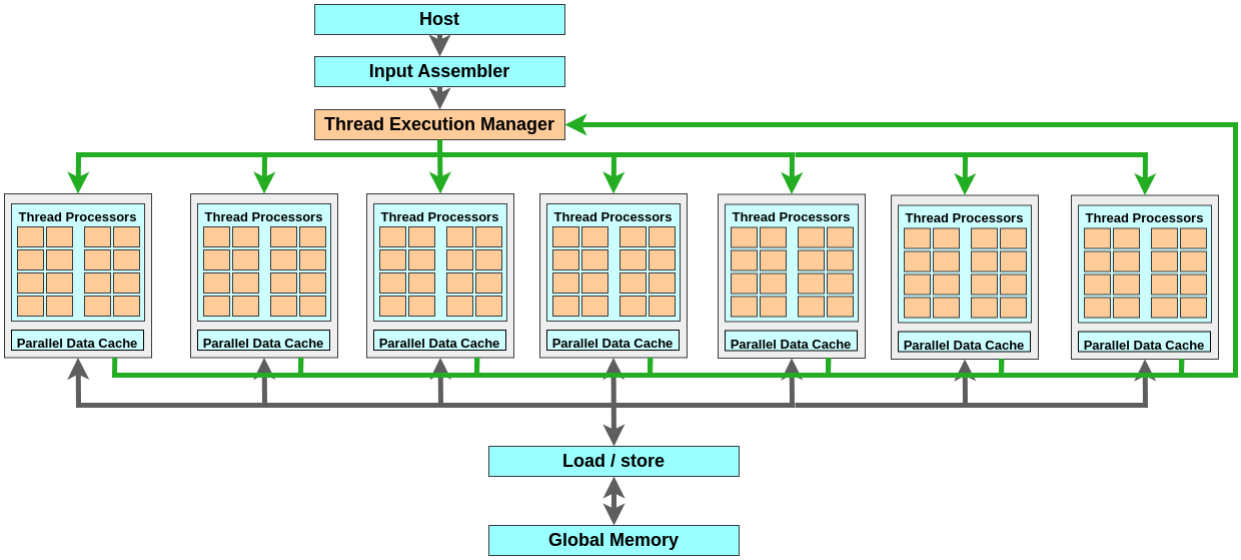

Figura 2.11: Arquitectura de un GPU. Tomada de (15).

- *Cuadricula:* Es un conjunto de subprocesos que son ejecutados dentro del mismo kernel. Cuando el CPU hace una llamada a CUDA para realizar una tarea, lo hace a través de la cuadrícula. Estas pueden ejecutarse varias de manera sincrónica.

  Cabe aclarar que en sistemas con más de un GPU, las cuadriculas no pueden ser compartidas entre sí, porque se usan de varias cuadriculas en un solo GPU para que puedan tener una eficiencia del 100 %.

- *Bloque:* Es una unidad lógica que contiene dentro varios hilos que se coordinan para acceder a una memoria compartida. Esto hace que cuando ejecuten un programa, varios bloques lo puedan hacer.

  Para identificar cada bloque, se tiene una variable asociada, llamada **"blockIdx"**, y cada identificador de los bloques puede estar en 1D o 2D.

- *Hilos:* Dentro de los bloques, existen los hilos, los cuales se encargan de ejecutar subprocesos dentro de sus núcleos individuales de los multiprocesadores. Esto hace que se diferencien de las cuadriculas y de los bloques, ya que los hilos no están limitados a utilizar un solo núcleo.

  Los subprocesos de los hilos tienen una variable para identificarse entre sí, llamado **"threadIdx"**. Estos pueden ser de 1D, 2D o 3D dependiendo de la dimensión del bloque. La identificación de los subprocesos son de forma relativa a la identificación del bloque.

Dentro de la arquitectura de CUDA es necesario el almacenamiento de información, por lo cual, el uso de memorias dentro de está es necesario. Para una mejor gestión del almacenamiento de datos, se tienen diferentes tipos de memorias para separar los

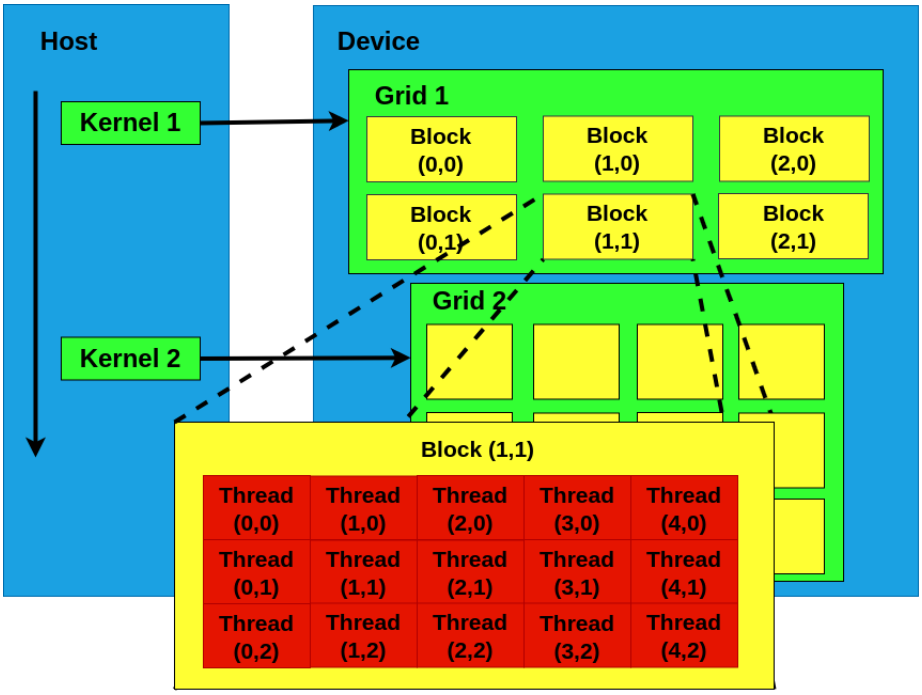

Figura 2.12: Arquitectura de CUDA. Tomada de (15).

datos para ciertos procesos. Las memorias que maneja según Anthony Lippert (15) en la figura 2.13 son:

- *Memoria global* (Global memory en inglés): Es una memoria dedicada a la lectura y la escritura. Esta escritura y lectura es de forma secuencial y está conformada de 16 bytes para que su funcionamiento sea rápido. Cabe mencionar que dentro de esta memoria no se almacena caché.
- *Memoria de textura* (Texture memory en inglés): Esta memoria solo tiene modo lectura. Cuenta con un caché optimizado encargado de bosquejar el espacio en 2D.
- *Memoria constante* (Constant memory en inglés): Dentro de esta memoria son almacenadas las constantes y argumentos del núcleo.
- *Memoria compartida* (Shared memory en inglés): Como se mencionó anteriormente, los subprocesos de un bloque utilizan una memoria compartida donde hacen operaciones de lectura y escritura. El número de subprocesos que pueden ejecutar de manera simultánea esta delimitado por la capacidad de esta memoria.
- *Memoria local* (Local memory en inglés): Funciona como memoria auxiliar, la cual almacena la información que no cabe en los registros. Esta memoria permite realizar lecturas y escrituras en conjunto con los registros funcionando como si fuese un sólo registro.

- *Registros* (Registers en inglés): Esta memoria es de las más rápidas, porque ella se realiza el almacenamiento y recuperación rápida de datos utilizados frecuentemente por los subprocesos.

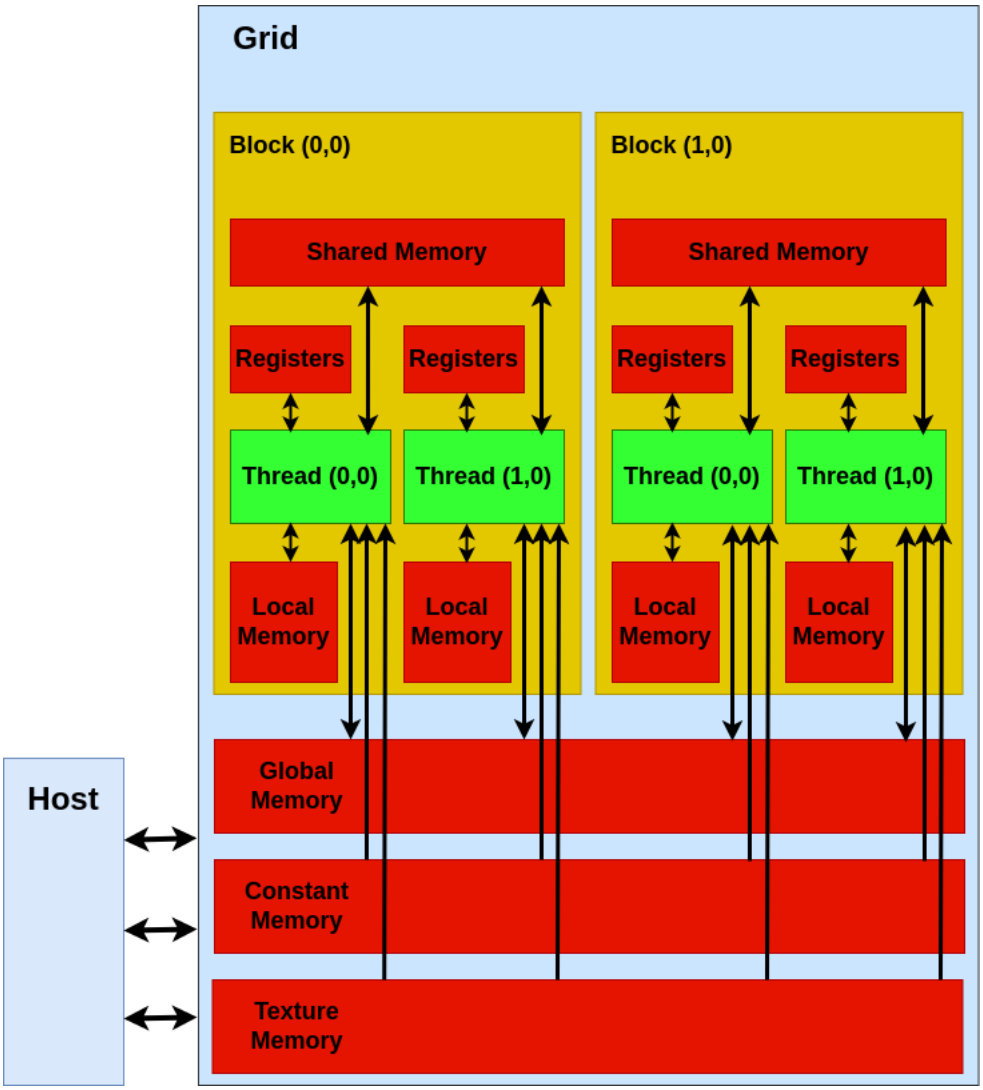

Figura 2.13: Arquitectura de las memorias en CUDA. Tomada de (15).

2.7. ROS

Dado que manejamos un sistema complejo como lo es un robot móvil compuesto de sensores y actuadores, necesitamos una herramienta a nivel software el cual nos pueda

ayudar a gestionar cada uno de estos componentes. Una de las propuestas más utilizadas en el ámbito de la robótica es el uso de ROS. ROS (Robotics Operated System por sus siglas en inglés) (25) son bibliotecas en código que ayudan a crear aplicaciones dentro de un robot.

ROS es código abierto, lo que hace que sea usado para varios proyectos de Robótica. Asimismo, la misma comunidad de ROS lo utiliza para implementar módulos que sean compatibles en diferentes proyectos. Esto hace que existan algunas ventajas que nos brinda ROS para gestionar los componentes de un robot móvil a nivel software. Por ejemplo:

- Es compatible con varios lenguajes de programación (entre los más utilizados C/C++ y Python), lo cual lo hace que al momento de desarrollar algoritmos para el control e interacción del robot sea fácil comunicar procesos entre sí.
- La vía de comunicación entre diferentes procesos lo puede realizar por medio de punto a punto (figura 2.14). Este consiste en tener un proceso que publique resultados y uno que se suscriba a ese proceso para obtener dicho resultado. También se puede realizar por medio de un servicio (figura 2.15) que consiste en solicitar información al servicio y este lo atiende. La ventaja de los servicios es que uno o más procesos pueden acceder a él simultáneamente.
- Dado que en un mismo código se pueden implementar varios procesos (o como los maneja ROS tópicos), un mismo código puede manejar varios tópicos y/o servicios dentro de su ejecución.

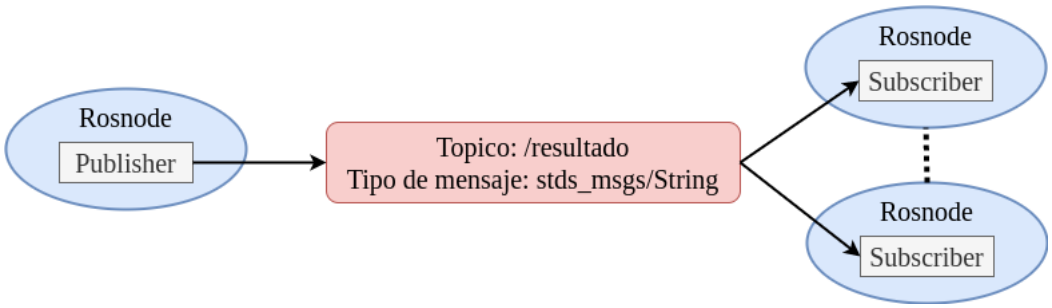

Figura 2.14: Ejemplo de un tópico en ROS. Tomado de (17).

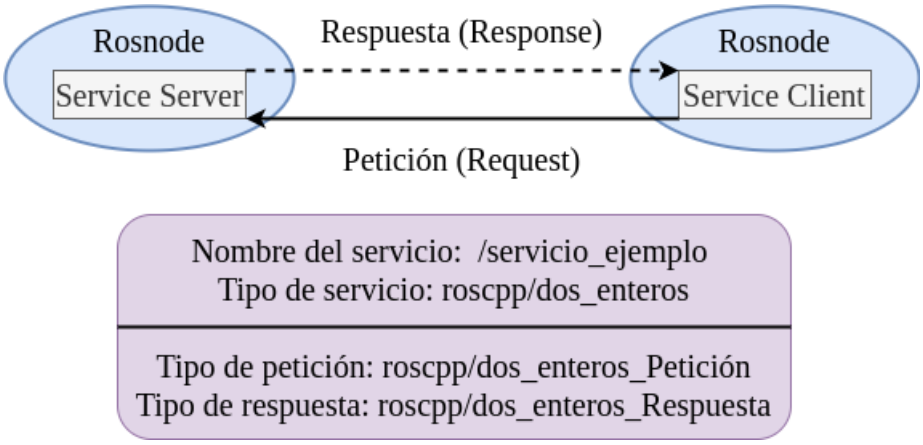

Figura 2.15: Ejemplo de un servicio en ROS. Tomado de (16).

Capítulo 3

Localización con el filtro de Kalman

Dentro de este capitulo, se explica como un robot móvil se puede localizar dentro de un entorno. De igual manera, se detalla como es que se implementa el algoritmo de filtro de Kalman como forma de localización; así como la variante del filtro para sistemas no lineales para su implementación.

3.1. Localización

Un robot móvil debe ser capaz de poder localizarse por medio de obtener la estimación respecto a la pose sobre un eje de referencia. Actualmente existen diferentes maneras para que el robot pueda obtener un aproximado de su localización dentro de un entorno. Una de las primeras técnicas que se implementaron para localizarse es llevar un conteo de cada uno de los movimientos realizados por las ruedas del robot desde un punto de referencia inicial.

Idealmente, esto funcionaria en la teoría, pero experimentalmente esto no es viable. Dado que existen muchos factores externos que influyen al momento de que las ruedas se empiecen a mover. Ejemplo de ello lo pueden ser el material con el que están hechas las ruedas, el tipo y textura de la superficie por donde se desplace, diseño de construcción, limitaciones físicas, entre otras más.

Podemos suponer entonces que el robot sólo tiene movimientos de avance, giro a la derecha de 90^{o} y giro a la izquierda de 90^{o}. Además, consideremos que tiene que ir de un punto A a un punto B (figura 3.1), entonces:

- Podemos tener que el punto A y el punto B están en diferentes partes del espacio, lo que haría que el robot al momento de trasladarse tenga pequeños errores y no llegue exactamente al punto B.
- Podemos tener que el punto A y el punto B coincidan en el mismo lugar. En este caso el robot puede que no haga un recorrido y quedarse en el mismo punto. La única manera en el que el robot haga un recorrido y llegue a la misma posición, es cuando se presentan más puntos entre A y B.

Respecto al segundo punto, si realizamos varias veces el recorrido hacia un mismo punto, podemos ver claramente que el error se acumula, haciendo que cada que realice el recorrido no llegue al punto del cual parte el robot.

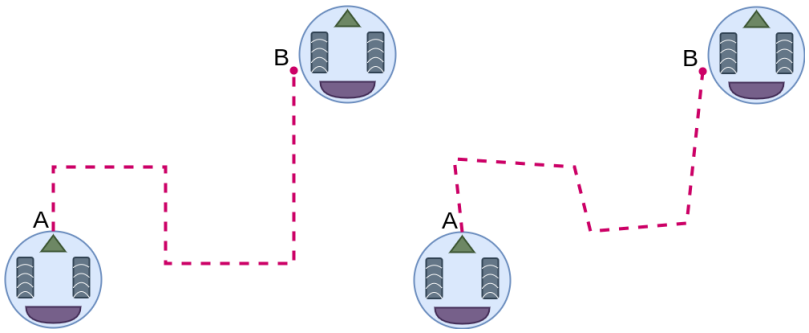

Figura 3.1: Representación de errores en la trayectoria de un robot móvil.

3.1.1. Localización con odometría

El término de odometría se refiere al estudio que se realiza para estimar la posición de vehículos con ruedas mientras estos se encuentran navegando. Esta estimación se obtiene por medio de los giros que realiza las ruedas, para de esa manera poder aproximar su posición a lo largo del tiempo.

La odometría para un robot móvil se refiere a este segundo término mencionado, el cual implementa una técnica para calcular la posición del robot a partir de los giros que realizaron sus llantas durante un periodo de tiempo. Dado que utilizamos un robot móvil, se usan sensores como los encoders que nos ayudan a ir llevando la cuenta de cuanto avanzo el robot para estimar su posición. Pero si nos basamos solamente en obtener la posición por medio del movimiento de las ruedas del robot, podemos acumular los errores que alteren el resultado en la distancia calculada. Algunos de los factores que pueden provocar alteraciones son:

- Factores internos del robot: Estos pueden ser cuestiones de las ruedas que afecten su movimiento (diferentes diámetros entre ellas, diferente material, mala colocación), cuestiones de los sensores (la discretización de los valores obtenidos de los encoders, la resolución de la medición del encoder, defecto de fabrica), entre otros.
- Factores externos del robot: Estos son todos los factores presentes en el ambiente por el cual navega el robot, como pueden ser el suelo (que este muy liso, tenga algún desperfecto donde se atore), como lo puede ser los objetos del lugar (que no llegue a ciertos lugares por tamaños de muebles), entre otros.

Dicho lo anterior, las técnicas de localización se manejan no solo de una odometría, sino que también se utilizan otros tipos de localización para que en conjunto se pueda estimar de mejor manera la posición del robot.

3.1.2. Localización por detección de marcas naturales

Una forma en la que los robots móviles pueden realizar cálculos de estimación con respecto a la posición y orientación es usando cámaras RGB-D. Por medio de su campo de visión son capaces de detectar marcas naturales que se encuentren en el ambiente, tratando de recrear el comportamiento humano de ubicación en un entorno.

Existen varias técnicas de visión con las que se pueden detectar marcas naturales dentro de una escena, como lo es el implementar algoritmos descriptores de puntos clave con SIFT (Scale-invariant feature transform por sus siglas en inglés) o con BRIEF (Binary Robust Independent Elementary Features por sus siglas en inglés). Otra forma de detectar marcas naturales es por medio de implementar una red neuronal convolucional previamente entrenada.

Estas marcas naturales pueden ser de cualquier tipo, sin embargo cuando una técnica de visión se implementa, esta marca natural debe ser visible en todo campo de visión del robot. Esto es importante porque al reconocer dicha marca natural en el ambiente, el sistema debe hacer cálculos de posición de la cámara con respecto a la marca y de esta forma el robot móvil puede estimar una pose sabiendo las posición de cada marca natural a detectar (Figura 3.2).

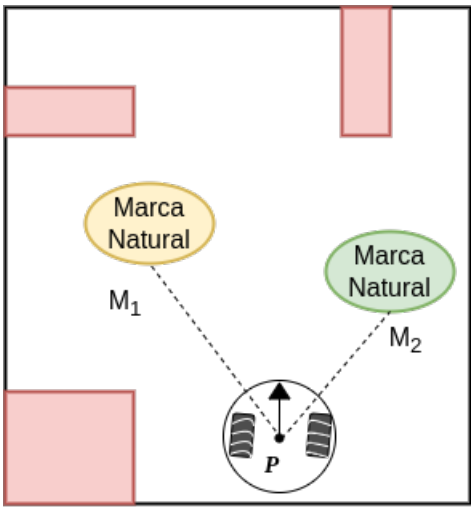

Figura 3.2: Localización de un robot móvil por medio de marcas naturales conocidas.

Para realizar la estimación de la posición de un robot móvil por medio de las marcas naturales, el Dr. Yukijiro Minami y el Dr. Jesús Savage proponen una solución. Esta solución implementa circunferencias de tal manera que si se conoce tanto la posición de las marcas dentro del mapa, como la distancia del robot a la marca se puede encontrar la posición del robot (12). Si la distancia que existe entre el robot y las marca naturales es equivalente al radio de una circunferencias, entonces podemos encontrar un punto de intersección entre circunferencias para considerarlo el equivalente a la posición del robot dentro del mapa (siendo este un caso ideal).

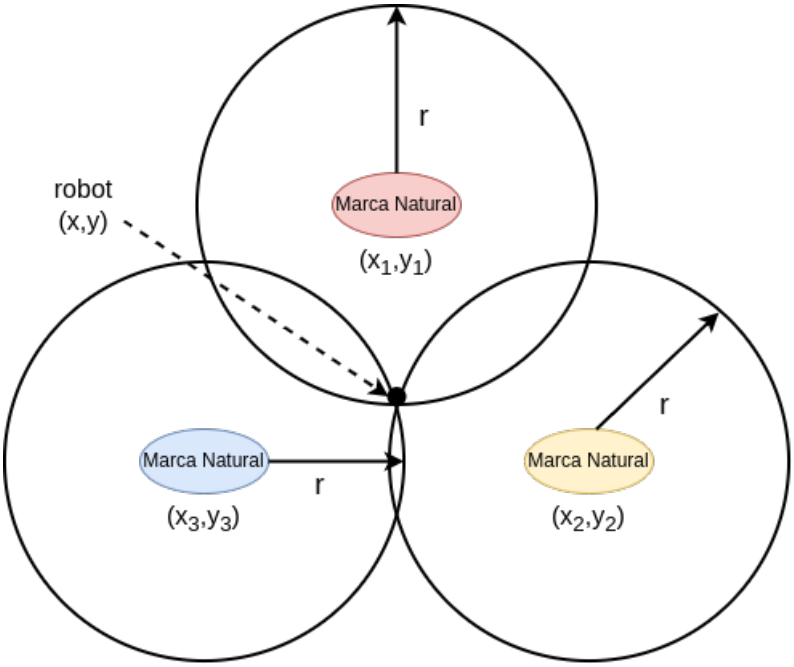

Figura 3.3: Localización de un robot móvil por medio de la intersección de las circunferencias descritas por la distancia entre el robot y cada marca natural detectada dentro de un entorno.

En la Figura 3.3 podemos ver una representación gráfica en la que podemos ver la posición del robot a partir de las marcas naturales. La posición del robot la tenemos en coordenadas (x, y) y las marcas naturales en coordenadas (x_i, y_i) en donde i representa la posición i-ésima marca natural que el robot detecta. Visto esto en ecuaciones, se bosquejan de la siguiente manera:

$$(x - x_1)^2 + (y - y_1)^2 = r_1^2 \tag{3.1}$$

$$(x - x_2)^2 + (y - y_2)^2 = r_2^2 \tag{3.2}$$

$$(x - x_3)^2 + (y - y_3)^2 = r_3^2 \tag{3.3}$$

Estas tres representaciones en ecuaciones de las marcas naturales las podemos trabajar con ellas de tal manera que podemos obtener dos ecuaciones, las cuales son ecuaciones de una recta en un plano cartesiano. Estas rectas son desarrolladas reescribiendo los binomios y restando la ecuación 3.1 de la ecuación 3.2 y de la ecuación 3.3; se forman las siguientes dos rectas:

$$2x(x_2 - x_1) + x_1^2 - x_2^2 + 2y(y_2 - y_1) + y_1^2 - y_2^2 = r_1^2 - x_2^2 \tag{3.4}$$

$$2x(x_2 - x_3) + x_3^2 - x_2^2 + 2y(y_2 - y_3) + y_3^2 - y_2^2 = r_3^2 - x_2^2 \tag{3.5}$$

Existen variables las cuales cuando se presentan pueden afectar al sistema y/o al entorno donde se encuentran las marcas naturales. Esto no garantiza que las circunferencias coincidan en un mismo punto. Por ende, cuando se generan las dos rectas nos asegura que cruzan por dos puntos de intersección de las circunferencias o que son perpendiculares a la recta que pasa por los centros pasando por el espacio medio que existe entre las circunferencias de las marcas naturales, como se muestra en la Figura 3.4. Las coordenadas (x, y) están dadas por las ecuaciones 3.4 y 3.5; dan la posición del robot y están conformados de la siguiente manera:

$$x = \frac{(y_2 - y_1)(r_2^2 - r_3^2 - x_2^2 + x_3^2 - y_2^2 + y_3^2) - (y_3 - y_2)(r_1^2 - r_2^2 + x_1^2 + x_2^2 - y_1^2 + y_2^2)}{2(x_3 - x_2)(y_2 - y_1) - 2(x_3 - x_2)(y_3 - y_2)} \tag{3.6}$$

$$y = \frac{(x_2 - x_1)(r_2^2 - r_3^2 - x_2^2 + x_3^2 - y_2^2 + y_3^2) - (x_3 - x_2)(r_1^2 - r_2^2 + x_1^2 + x_2^2 - y_1^2 + y_2^2)}{2(x_2 - x_1)(y_3 - y_2) - 2(x_3 - x_2)(y_2 - y_1)} \tag{3.7}$$

3.1.3. Representación gráfica del ambiente

Los robots móviles requieren un entorno o un mapa en el cual se puedan localizar al momento de realizar ciertas tareas. Por lo cual se tiene que generar una representación gráfica la cual sea la guía interna del robot para realizar algoritmos necesarios para navegar.

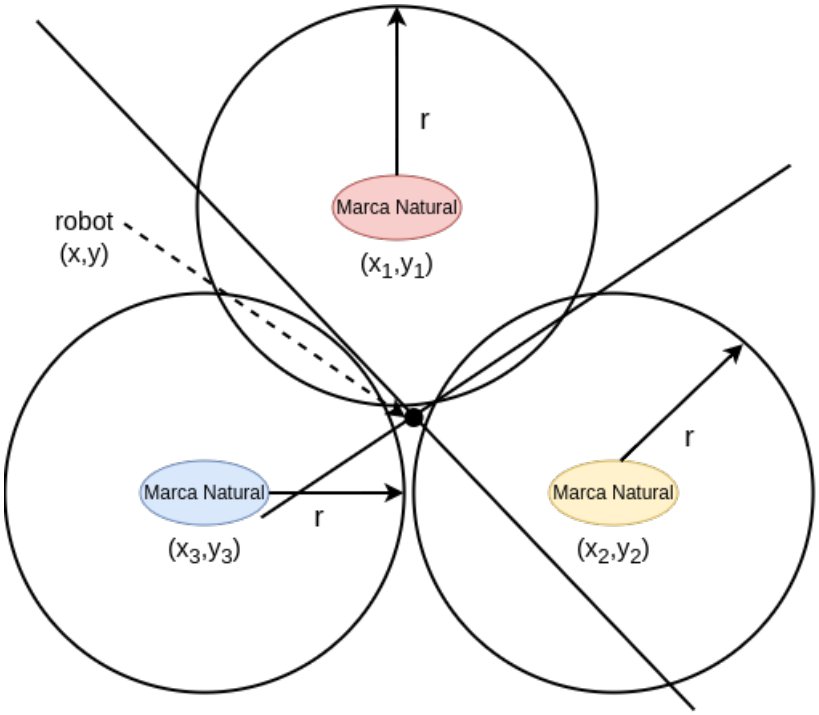

Figura 3.4: Localización de un robot móvil por medio de de las rectas en donde cruzan las circunferencias descritas por la distancia entre el robot y cada marca natural detectada dentro de un entorno.

En la construcción del ambiente virtual las marcas naturales son de importancia, ya que al momento de generar el mapa debemos considerar información de donde se encuentra la marca en el entorno real para saber su posición. Un ejemplo de esto es si utilizáramos señalamientos de precaución como marcas a detectar y por medio de su posición poder localizarse. Otro ejemplo es utilizar objetos como muebles o sillas, y paredes detectados por un escáner 2D para recrear el área virtualmente en un mapa y localizarse, como se muestra en la Figura 3.5. Cabe aclarar que el tamaño para cada celda en el mapa, está delimitado por una relación de una celda igual a un metro para gestionar mejor los recursos computacionales.

Un algoritmo conocido para la creación de mapas sintéticamente es el uso de SLAM (18) (Simultaneous Localization and Mapping por sus siglas en inglés). Este método ha sido implementado en los vehículos autónomos para crear un mapa y a la vez localizar el vehículo en el mismo. Este algoritmo también es implementado por robot móviles para el mismo propósito de localizarse y crear mapas.

SLAM funciona por medio de dos bloques de procesamiento (Figura 3.6). El primero es el procesamiento de la señal del sensor, el cual los brinda información de donde se encuentran los elemento del ambiente en el que se encuentra el robot. El segundo bloque realiza la optimización de gráficos de poses, el cual va recreando de manera virtual el entorno dependiendo de los valores obtenidos por los sensores y ajustando el tamaño de

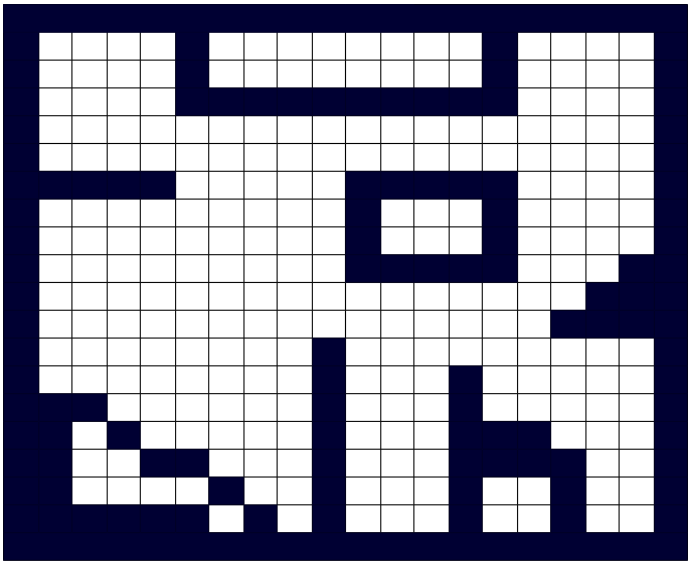

Figura 3.5: Representación discreta de un entorno real en un mapa virtual.

las celdas para optimizar los gráficos acorde a los componentes del robot.

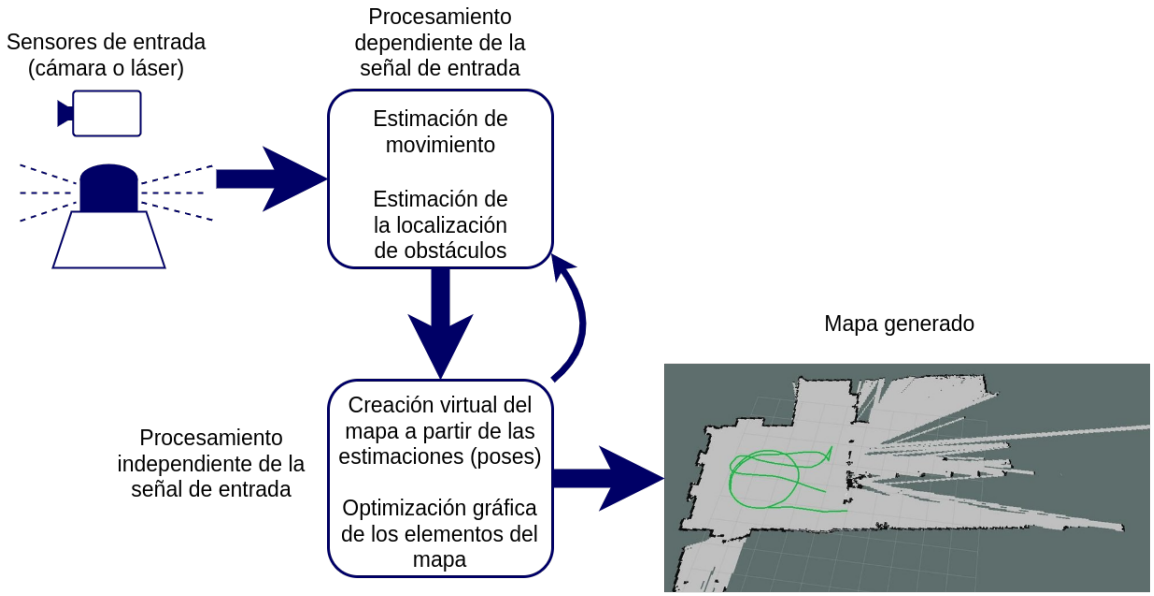

Figura 3.6: Funcionamiento de SLAM

El algoritmo de SLAM puede ser implementado de 2 maneras diferentes:

- SLAM Visual (o también conocido como vSLAM), el cual calcula la posición y la orientación de un robot móvil con respecto al entorno en el que se encuentra y al mismo tiempo mapeando sólo utilizando las entrada visuales de una cámara.

- SLAM por LiDAR funciona por medio de un sensor láser para generar un mapa en 2 dimensiones. Se puede utilizar cualquier sensor de tipo láser que retorne la distancia de los objetos que detecte para construir el mapa.

3.1.4. Consideraciones del ambiente de un robot

Dentro de los ambientes en los que puede navegar un robot móvil se tienen que contemplar muchos factores que pueden o no influir al momento de crear un mapa. No es lo mismo que un robot navegue por un ambiente solo con elementos fijos a un ambiente en el cual el flujo de personas esta presente. Estos ambientes los podemos catalogar en dos tipos: los ambientes estáticos y los ambientes dinámicos.

Los ambientes estáticos o casi estáticos son los ambientes que sufren de pocos cambios en su ambiente real al momento en el que un robot se localiza. Dado que los sensores son los que miden el ambiente en el que se encuentra el robot, el ambiente debe quedar lo mas intacto posible para que, al momento de crear el mapa sean lo mas parecidos. También se presenta el caso en el que al momento de crear el mapa algunos elementos del ambiente no sean registrados por el sensor y por ende queden descartados siendo catalogados como ruido.

En cambio, un ambiente dinámico se presenta cuando existe un constante cambio de elementos de su lugar de origen. Normalmente estos ambientes se presentan cuando existe interacción humana constante de por medio (por ejemplo personas caminando, moviendo objetos como muebles, estar sentado en ciertos lugares que se interpongan, etc.). Este tipo de perturbaciones pueden afectar de tal manera que el robot no pueda llegar a su punto destino, o que no pueda identificar su punto inicial en el mapa.

El sistema de navegación de un robot móvil requiere inicializar con un punto inicial del cual parta a otro punto destino dentro de un mapa. Esto se debe porque a menos que los sensores sean capaces de reconocer o de ubicar su punto origen, el robot al momento de estar navegando dentro del ambiente se localiza constantemente para llegar a su destino de manera exitosa, y si el punto de inicio presenta errores eso afecta el trayecto. Es por eso que el sistema de navegación debe contener un sistema de corrección el cual vaya rectificando las trayectorias del robot para que pueda cumplir con navegar a su punto destino.

Supongamos que el ambiente a mapear es un laboratorio, cuyo entorno se encuentra la mayor parte del tiempo de manera estática y que los cambios que llega a sufrir son por el cambio de lugar de las sillas. Entonces podemos utilizar un sensor 2D como lo es el láser que tiene el robot para escanear las paredes, algunos muebles como lo son las mesas de trabajo y archiveros creando una replica 2D del ambiente (como se observa en la figura 3.7). En el caso de las sillas se pueden considerar como ruido a las lecturas al momento de localizarse.

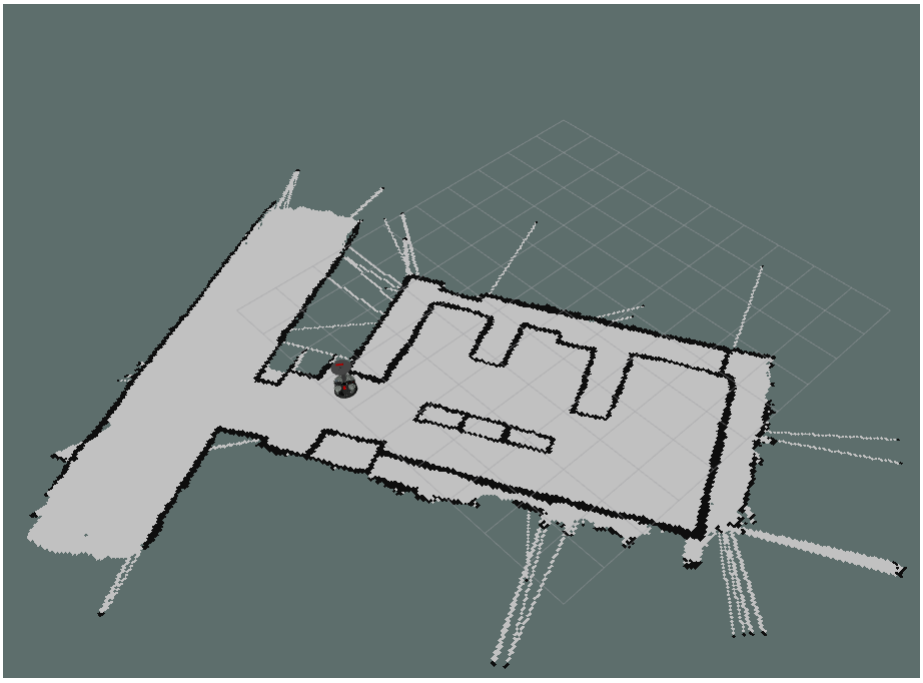

Figura 3.7: Ejemplo de un mapa creado por SLAM basado de un ambiente real.

3.2. Filtro de Kalman

Un método probabilístico conocido dentro del ámbito de la robótica que se ha implementado en muchos procesos es el filtro de Kalman, el cuál es un algoritmo desarrollado en el año 1960 por Rudolf E. Kalman (10). Este algoritmo consiste en identificar el estado oculto con respecto a un sistema dinámico lineal. Este algoritmo esta constituido de dos etapas: una es la predicción del modelo y la otra es la actualización.

Para explicar el funcionamiento del filtro de Kalman se propone como ejemplo estimar la posición de un robot móvil que se traslada sobre una linea recta coincidente con el eje x. También vamos a tomar a x_k como una variable que representa la posición del robot móvil sobre el eje x en un momento k y a v_k como la representación de la velocidad del robot v en el mismo momento k. Ambas variables de posición x_k y velocidad v_k serán utilizadas en un vector de estado $\vec{x}_k$ las vamos a considerar con condiciones iniciales desde el origen iguales a cero (Figura 3.8).

$$\vec{x}_k = \begin{bmatrix} x_k \\ v_k \end{bmatrix} \tag{3.8}$$

Posteriormente, los valores que toman las variables de posición y velocidad son de

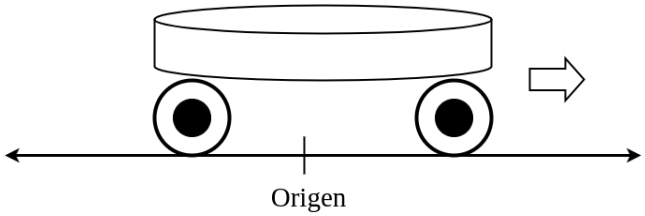

Figura 3.8: Modelo de desplazamiento de un robot móvil.

manera aleatoria considerando una distribución normal, con una media μ para cada variable (siendo el centro de la distribución) y una varianza σ^2 que representa la incertidumbre.

3.2.1. Predicción

Para llevar a cabo la predicción del algoritmo vamos a partir de la correlación que existe entre la posición y la velocidad del robot móvil. Está correlación se establece de la siguiente manera: Si la velocidad del robot va en aumento, se recorre una mayor distancia; pero si la velocidad disminuye, la distancia que recorre el robot es menor (figura 3.9). La correlación la podemos ver representada por una matriz de correlaciones $\sum_k$, cuyo grado esta dado por las variables del vector de estado, resultando en una matriz de correlación cuadrada de tamaño $n \times n$ siendo n el tamaño del vector. Y la distribución de la matriz es generado a partir de los vectores y de los valores característicos de la correlación y centrando la media en cada una de las variables.

Considerando lo anterior, se desconoce la posición y la velocidad exacta del robot, sin embargo, sí podemos estimar la posibilidades que sean más probables que sucedan (que se encuentren más cerca de la media).

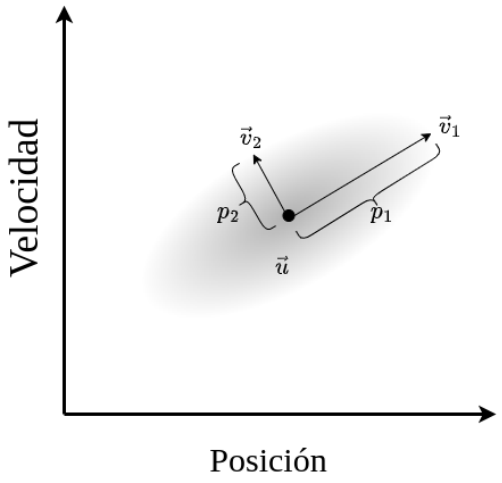

Figura 3.9: Correlación Posición-Velocidad.

$$\vec{x}_k = \begin{bmatrix} x_k \\ v_k \end{bmatrix} \tag{3.9}$$

Para realizar la predicción del modelo, debemos considerar un estado $k-1$, el cual representa el estado anterior en el que se encuentra el robot (un punto desde el cual parte el robot) y un estado k que representa el estado siguiente del robot (punto destino al cual llega el robot). Basando el modelo con el ejemplo del robot móvil utilizaremos las ecuaciones del MRU (por sus siglas en español movimiento rectilíneo uniforme), en x_k y en v_k y posteriormente, representar a nuestro vector en forma matricial, resultando en la ecuación 3.12.

$$x_k = x_{k-1} + \Delta t v_{k-1} \tag{3.10}$$

$$v_k = v_{k-1} \tag{3.11}$$

$$\vec{x}_k = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \vec{x}_{k-1} \tag{3.12}$$

La implementación del filtro del Kalman es robusto ante variantes que puedan existir y que sean de origen desconocido (como lo pueden ser la fricción de las ruedas contra el suelo, la textura del mismo, el viento del ambiente donde se encuentra el robot, entre otros) representados en $\vec{\gamma}_{k-1}$. Por lo cual, al vector $\vec{x}_k$ vamos a considerar un ruido gaussiano $\vec{\gamma}_{k-1}$ de las mismas dimensiones de la matriz dentro del vector resultando en la formula 3.13.

Una de las variantes que vamos a considerar como conocidas son las instrucciones que recibe el robot para desplazarse. Supongamos que la reacción del sistema (la aceleración visto de manera física) ejecuta la siguiente instrucción de movimiento del robot, esto hace que resulte en una x_k y v_k representadas en la ecuaciones 3.14 y 3.15 respectivamente:

$$\vec{x}_k = F_{k-1}\vec{x}_{k-1} + \vec{\gamma}_{k-1} \tag{3.13}$$

$$x_k = x_{k-1} + \Delta t v_{k-1} + \frac{1}{2}\alpha \Delta t^2 \tag{3.14}$$

$$v_k = v_{k-1} + \alpha \Delta t \tag{3.15}$$

Una vez planteado las instrucciones del robot, podemos reescribir la ecuación del vector $\vec{x}_k$ en su forma matricial, resultando en la ecuación 3.16 o en su ecuación reescrita 3.17:

$$\vec{x}_k = F_{k-1}\vec{x}_{k-1} + \begin{bmatrix} \frac{\Delta t^2}{2} \\ \Delta t \end{bmatrix} \alpha \tag{3.16}$$

$$\vec{x}_k = F_{k-1}\vec{x}_{k-1} + B_{k-1}\vec{u}_{k-1} + \vec{\gamma}_{k-1} \tag{3.17}$$

Otro aspecto también a considerar es el planteamiento de una probabilidad relacionada con la transición de estado, la cual se utiliza para predecir el siguiente estado de un modelo a partir de sus variables. Para plantear esto en nuestro vector $\vec{x}_k$, vamos a tomar una transición de estado $p(x_k|u_t, x_{k-1})$, en donde u_t representa el comando de movimiento que el robot va a ejecutar. El vector $\vec{x}_k$ debe considerar fuerzas externas, como lo vemos en la ecuación 3.13, en donde la matriz B_k representa la matriz de control y $\vec{u}_k$ representa el vector de control. Pero en el caso de nuestro sistema, dado que es sencillo no serán considerados.

Partiendo de la formula 3.13 se puede predecir el nuevo estado $\vec{x}_k$ y realizar la actualización en la matriz Σ_k, resultando en la formula 3.18. Para la predicción del siguiente estado debemos considerar una matriz de incertidumbre Q que nos ayuda a predecir a la nueva posición del robot (ecuación 3.19).

$$\Sigma_k = F_{k-1}\Sigma_{k-1}F_{k-1}^T \tag{3.18}$$

$$\Sigma_k = F_{k-1}\Sigma_{k-1}F_{k-1}^T + Q \tag{3.19}$$

Para finalizar, la predicción del estado $\vec{x}_k$ es una estimación calculada a partir del estado anterior $\vec{x}_{k-1}$. A esta estimación se añade una corrección de fuerzas conocidas, (como lo es $B_{k-1}u_{k-1}$) ya sea que este considerada o no, dependiendo del sistema en cuestión. También la matriz de incertidumbre Σ_k del estado a predecir depende de la matriz anterior Σ_{k-1} y de una matriz de incertidumbre Q que existe en el entorno del robot.

3.2.2. Actualización

Para esta parte del algoritmo del filtro de Kalman, el robot móvil por medio de la lectura de los sensores puede hacer la corrección del movimiento y llegar al punto destino. Primero debemos obtener lecturas del sensor que nos brinden información del entorno, es decir, las lecturas que se suponen se predijeron en la etapa anterior y nos

confirmen que el robot llego a su punto destino. Matemáticamente, podemos bosquejar los valores esperados de los sensores similarmente a la matriz de incertidumbre que usamos para la predicción, a la cual nombraremos como C_k y el μ_e como el resultado (ecuación 3.20).

En la figura 3.10 podemos ejemplificar como los sensores nos ayudan a hacer la actualización del algoritmo. En la primer gráfica podemos observar la correlación Posición-Velocidad, la cual la podemos representar como la pose del robot deseada conforme a las instrucciones de movimiento. Mientras que la otra gráfica representa la pose del robot que nos brinda las lecturas de los sensores. Esta transformación de una gráfica a otra esta dada al aplicar la matriz C_k a la correlación Posición-Velocidad.

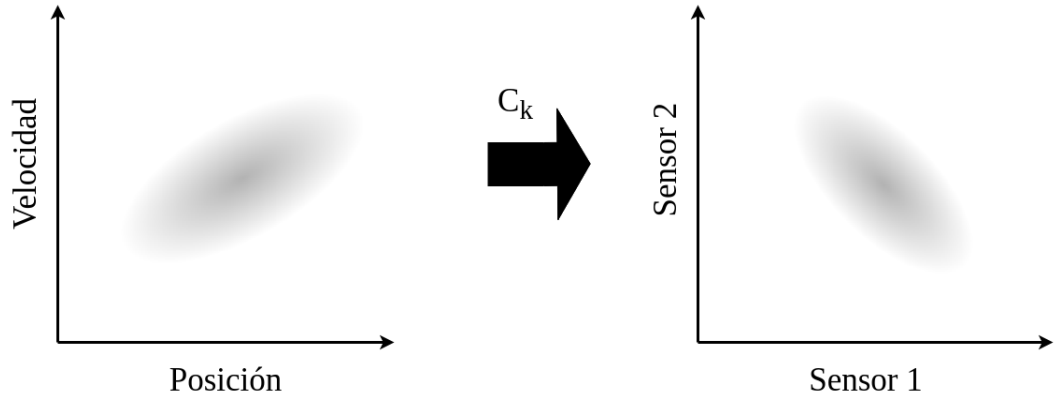

Figura 3.10: Lecturas de los sensores modeladas a partir de la matriz C_k.

Para el caso de la actualización, también tenemos que considerar el ruido que se llegue a presentar. Para ello, utilizaremos una matriz de correlación Σ_e y calcular dichos valores de una forma parecida a la anterior matriz (ecuación 3.21). Se debe tener en cuenta que las unidades para los valores que resulten de las operaciones y poder recrear las lecturas que los sensores deben tener partiendo de la predicción, es decir, tratar de obtener valores semejantes entre los predichos con los calculados.

$$\vec{\mu}_e = C_k \vec{x}_k \tag{3.20}$$

$$\Sigma_e = C_k \Sigma_k C_k^T \tag{3.21}$$

Para recabar los valores que provienen de los sensores del robot, los almacenamos dentro del vector $\vec{z}_k$, como se puede observar en la figura 3.11 que representa el punto central de una distribución del estado que proviene de los sensores. Además, se considera una matriz R_k que representa la covarianza de la misma distribución. Esta se obtiene de forma experimental representando la incertidumbre del conjunto de lecturas obtenidas.

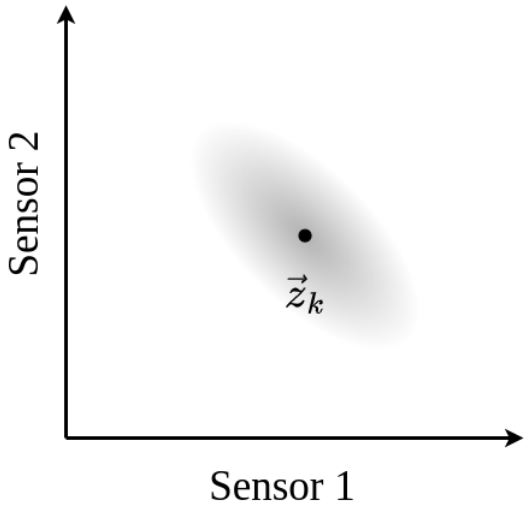

Figura 3.11: Vector $\vec{z}_k$ localizado en un punto medio de lecturas de los sensores.

El siguiente paso es el combinar las distribuciones que provienen de los sensores reales con los esperados a obtener (ecuación 3.22). Para ello se utiliza la multiplicación entre distribuciones gaussianas (de dimensión uno), y posteriormente se pueda extrapolar en otras distribuciones de ámbito multivariado (5). La distribución la obtendremos por medio una campana de Gauss utilizando una ecuación que obtiene una media μ y una varianza σ^2 (ecuación 3.23). Luego, se busca encontrar una distribución intermedia μ' resultante del producto entre dos distribuciones gaussianas μ_0 y μ_1 dadas, como se observa en la figura 3.12.

$$N(x, \mu', \sigma') = N(x, \mu_0, \sigma_0) * N(x, \mu_1, \sigma_1) \tag{3.22}$$

$$N(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{x-\mu^2}{2\sigma^2}} \tag{3.23}$$

Para calcular nuestras variables σ' y μ' vamos a sustituir en la ecuación 3.23 la ecuación 3.22, resultando en las ecuaciones 3.24 y 3.25. A partir de estas dos ecuaciones, podemos obtener una ganancia, que para el algoritmo es conocido como la ganancia de Kalman. Esta ganancia es usada como peso relativo que proviene de las mediciones y la estimación del estado actual, y es usado para lograr "$ajustar$" un rendimiento particular dentro del algoritmo. El termino de K de la ecuación 3.26 representa la ganancia y es obtenido de la simplificación de las dos ecuaciones anteriores.

$$\sigma' = \sigma_0^2 - K\sigma_0^2 \tag{3.24}$$

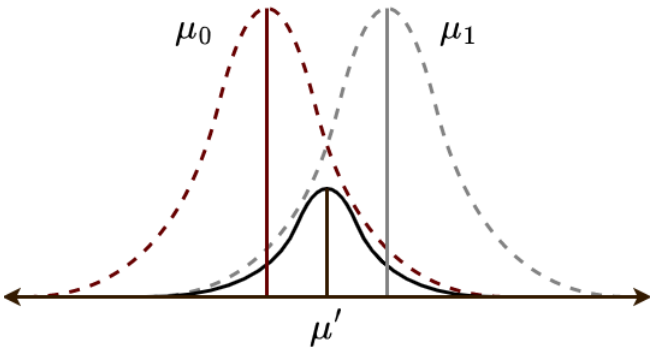

Figura 3.12: Distribución intermedia resultante del producto de dos distribuciones dadas.

$$\mu' = \mu_0 + K(\mu_1 - \mu_0) \tag{3.25}$$

$$K = \frac{\sigma_0^2}{\sigma_0^2 + \sigma_1^2} \tag{3.26}$$

Dado que las ecuaciones que utilizamos para las lecturas de los sensores están de forma matricial, reescribimos las ecuaciones anteriores a una forma matricial, resultando en las ecuaciones 3.27, 3.28 y 3.29 respectivamente:

$$\Sigma' = \Sigma_0 - K\Sigma_0 \tag{3.27}$$

$$\vec{\mu'} = \vec{\mu_0} + K(\vec{\mu_1} - \vec{\mu_0}) \tag{3.28}$$

$$K = \Sigma_0(\Sigma_0 + \Sigma_1)^{-1} \tag{3.29}$$

Considerando el ejemplo del robot desplazándose en linea recta que se propuso desde el inicio del tema, las dos distribuciones que se consideran para obtener la intersección entre ellas. Estas distribuciones son: la distribución observada $(\vec{z}_k, R_k)$ y la distribución esperada $(C_k\vec{x}_k, C_k\Sigma_k C_k^T)$. Sustituyendo la distribución esperada en nuestras ecuaciones matriciales 3.27 y 3.28 resulta en las ecuaciones 3.30 y 3.31:

$$C_k\vec{x}_k = C_k\vec{x}_k - K(\vec{z}_k - C_k\vec{x}_k) \tag{3.30}$$

$$C_k\Sigma_k C_k^T = C_k\Sigma_k C_k^T - KC_k\Sigma_k C_k^T \tag{3.31}$$

Para obtener la ganancia de Kalman en términos de matrices, vamos a reescribir la ecuación 3.29, resultando en la ecuación 3.32:

$$K = C_k \Sigma_k C_k^T (C_k \Sigma_k C_k^T + R_k)^{-1} \tag{3.32}$$

Por ultimo, para obtener la predicción final en $\vec{x}_k$ y Σ_k, utilizaremos las ecuaciones reescritas. Cada que el robot reciba una nueva instrucción de movimiento, se tiene que volver a implementar el algoritmo para calcular un nuevo $\vec{x}_k$ y Σ_k (ecuaciones 3.33 y 3.34) partiendo de los obtenidos previamente, tomando el lugar de $\vec{x}_{k-1}$ y Σ_{k-1} (como lo menciona Jesús Savage en su presentación (2)). En la figura 3.13 podemos ver un diagrama en el cual se explica mejor el funcionamiento del algoritmo.

$$\vec{x}_k = \vec{x}_k + K(\vec{z}_k - C_k \vec{x}_k) \tag{3.33}$$

$$\Sigma_k = \Sigma_k - K C_k \Sigma_k \tag{3.34}$$

$$K = \Sigma_k C_k^T (C_k \Sigma_k C_k^T + R_k)^{-1} \tag{3.35}$$

Otra representación que ejemplifica el filtro de Kalman lo podemos ver en el algoritmo 1 en un pseudocódigo en donde vemos como se obtienen las variables $\vec{x}_k$ y Σ_k durante la etapa de predicción, así como su corrección y el cálculo de de la ganancia de Kalman en la etapa de actualización. Este pseudocódigo se repite las iteraciones que sean necesarias hasta llegar al punto deseado.

Algorithm 1 Pseudocódigo del Filtro de Kalman.

procedure KF ALGORITHM($\vec{x}_{k-1}, \Sigma_{k-1}, \vec{u}_k, \vec{z}_k$)
  Predicción ▷ Obtenemos $\vec{x}_k$ y Σ_k de una iteración anterior.
  $\vec{x}_k = F_{k-1}\vec{x}_{k-1} + B_{k-1}\vec{u}_{k-1}$
  $\Sigma_k = F_{k-1}\Sigma_{k-1}F_{k-1}^T + Q$
  ▷ $\vec{x}_{k-1}, \Sigma_{k-1} = 0$ si es el inicio
  Actualización
  $K = \Sigma_k C_k^T (C_k \Sigma_k C_k^T + R_k)^{-1}$ ▷ Ganancia de Kalman
  $\vec{x}_k = \vec{x}_k + K(\vec{z}_k - C_k \vec{x}_k)$
  $\Sigma_k = \Sigma_k - K C_k \Sigma_k$
  return $(\vec{x}_k, \Sigma_k)$
end procedure

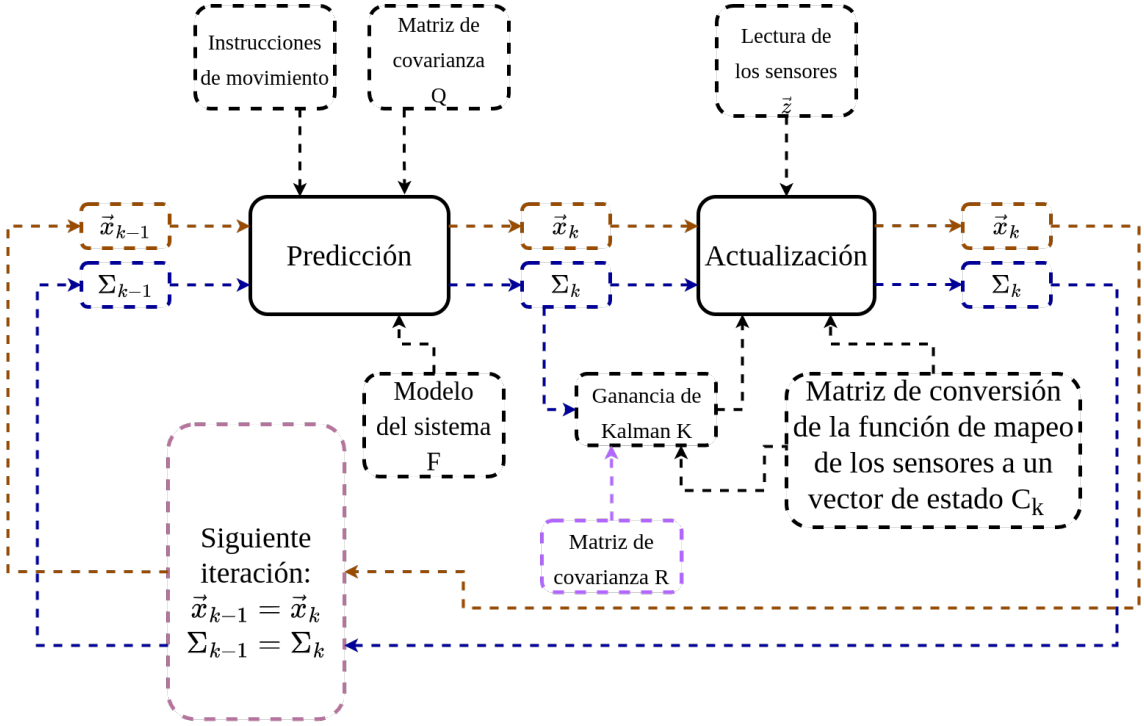

Figura 3.13: Diagrama de flujo del filtro de Kalman.

3.3. Filtro de Kalman extendido

Como hemos visto hasta el momento, el funcionamiento del filtro de Kalman es implementado en sistemas que son lineales (recordando el ejemplo del movimiento recto del robot móvil), pero en un entorno de la vida cotidiana los sistemas son no lineales. Esto implica que de debe ajustar el algoritmo para trabajar en sistemas no lineales. Como lo menciona Julier y Uhlmann (9), el EKF (Filtro de Kalman Extendido por sus siglas en inglés) busca acercar a la función lineal a un punto medio de una recta tangente respecto a la función. Dicho punto se considera el punto más probable que se encuentre en una distribución normal, para este caso de μ.

Cuando se buscar obtener la predicción del EKF, tenemos que considerar ciertas diferencias dentro de sus ecuaciones o formulas. Para ello vamos a tomar como punto de partida la ecuación 3.13, que al momento de reescribirla nos queda la ecuación 3.36, donde la función $g(u_t, \vec{x}_{k-1})$ representa una función no lineal.

$$\vec{x} = g(u_t, \vec{x}_{k-1}) + \gamma_k \tag{3.36}$$

Posteriormente, tenemos que linealizar a la ecuación por medio de una matriz Jacobiana resultando en:

$$G = J_{g(\vec{u}_t, \vec{x}_{k-1})} \tag{3.37}$$

De igual forma, retomaremos la ecuación 3.20 para reescribirla en términos de una función no lineal 3.38, para luego aplicar una matriz Jacobiana resultando en la ecuación 3.39.

$$\mu_e = h(\vec{x}_k) + \delta_k \tag{3.38}$$

$$H = J_{h(\vec{x}_k)} \tag{3.39}$$

Para obtener las matrices Jacobianas de las ecuaciones 3.37 y 3.39 son generadas a partir de derivadas parciales de primer orden. Estas matrices tiene la función de acercar en forma lineal una función evaluada en un punto. Para este caso, consideramos la matriz Jacobiana como la representación de la derivada respecto a una función multivariable. Retomando el ejemplo anterior para obtener la ganancia de Kalman, se manejaban dos matrices: la C y la F que formaban nuestro algoritmo. Para el EKF vamos a sustituir la Jacobiana G por F y la Jacobiana H por C, resultando en el algoritmo 2:

Algorithm 2 Pseudocódigo del Filtro de Kalman Extendido.

procedure EKF ALGORITHM($\vec{x}_{k-1}, \Sigma_{k-1}, \vec{u}_k, \vec{z}_k$)
  Predicción ▷ Obtenemos $\vec{x}_k$ y Σ_k de una iteración anterior.
  $\vec{x}_k = g(u_t, \vec{x}_{k-1})$
  $\Sigma_k = G_k \Sigma_k G_k^T + Q$
  ▷ $\vec{x}_{k-1}, \Sigma_{k-1} = 0$ si es el inicio
  Actualización
  $K = \Sigma_k H_k^T (H_k \Sigma_k H_k^T + R_k)^{-1}$ ▷ Ganancia de Kalman
  $\vec{x}_k = \vec{x}_k + K(\vec{z}_k - h(\vec{x}_k))$
  $\Sigma_k = \Sigma_k - K H_k \Sigma_k$
  return $(\vec{x}_k, \Sigma_k)$
end procedure

3.4. Localización de un robot móvil utilizando el filtro de Kalman extendido

Para implementar nuestro EKF utilizaremos como base la implementación de este algoritmo con el simulador que menciona la tesis de Diego Cordero (3). Este simulador fue diseñado como una herramienta para los alumnos al momento de implementar algoritmos de navegación con robots móviles.

El funcionamiento del simulador (3) se basa en un robot móvil, cuyas características que se consideran son: un par diferencial para el avance, uso de instrucciones de giro y avance en coordenadas de (θ, d), y los valores obtenidos por los sensores. Estos sensores tienen tareas que incluyen detectar obstáculos, luminosidad del ambiente, entre otros más. El sensor que nos compete es el sensor Kinect, porque el sensor le da información al robot de donde se encuentran las marcas naturales.

Dentro de la simulación de robot, son recreados los movimientos que este realiza en su entorno real. Simulando la detección de las marcas naturales de la etapa de actualización del EKF, podemos ver virtualmente la posición del robot en cada momento. Tanto para la simulación, como para la localización en ambiente real, se utilizó el algoritmo 3.

Este algoritmo requiere cuatro parámetros para su funcionamiento, los cuales son:

- La pose de iteración anterior $\vec{x}_{k-1}$. En dado caso que sea el primer avance del robot, el valor de $\vec{x}_{k-1}$ es la pose con la que inicia.
- La matriz de covarianza respecto a una iteración anterior Σ_{k-1}. Si en dado caso es el primer avance del robot, los valores de la matriz equivalen a cero.
- La instrucción $\vec{u}$ que el robot ejecuta, el cual se compone de un valor de avance u_d y uno de giro u_θ (ecuación 3.40).
- Una matriz S (ecuación 3.41) la cual contiene los landmarks o marcas naturales que reconoce el robot.

$$\vec{u} = \begin{bmatrix} u_d \\ u_\theta \end{bmatrix} \tag{3.40}$$

Con respecto a la matriz S, esta conformada de un tamaño de $n \times 5$ (ecuación 3.41), donde los n renglones se interpretan como el número de marcas naturales que fueron detectadas, mientras que las 5 columnas son las características de cada marca. Estas características son: x_i y y_i los cuales son la posición de la marca natural, r_i la distancia y la orientación que existe entre la marca y el robot, ϕ_i representa los valores que provienen de los sensores y por último, m_i equivale al identificador de la marca.

$$S = \begin{bmatrix} x_1 & y_1 & r_1 & \phi_1 & m_1 \\ x_2 & y_2 & r_2 & \phi_2 & m_2 \\ x_3 & y_3 & r_3 & \phi_3 & m_3 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{n-1} & y_{n-1} & r_{n-1} & \phi_{n-1} & m_{n-1} \\ x_n & y_n & r_n & \phi_n & m_n \end{bmatrix} \tag{3.41}$$

Como lo vimos anteriormente, el algoritmo EKF utiliza un vector de estado para realizar la predicción y la actualización. Este vector lo vemos conformado en la ecuación 3.42 donde tomamos en cuenta a la posición (x, y) del robot, así como la θ de su orientación. Podemos observar que el vector $\vec{u}$ contiene la instrucción que predice el lugar donde se encontrará el robot. Este vector lo vamos a reescribir en la ecuación 3.42 para manejar a g como una función que depende tanto de la siguiente instrucción $\vec{u}_k$ y del estado anterior $\vec{x}_{k-1}$.

$$\begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix} = \begin{bmatrix} x_{k-1} \\ y_{k-1} \\ \theta_{k-1} \end{bmatrix} + \begin{bmatrix} u_d * cos(\theta_{k-1} + u_\theta) \\ u_d * sin(\theta_{k-1} + u_\theta) \\ \theta_{k-1} + u_\theta \end{bmatrix} \tag{3.42}$$

$$\vec{x} = g(\vec{u}_k, \vec{x}_{k-1}) \tag{3.43}$$

Para la función g de la ecuación 3.43 se obtiene una matriz Jacobiana como se muestra en la ecuación 3.44. Esta operación se realiza puesto que g es una función no lineal.

$$G = \frac{\partial g(\vec{u}_k, \vec{x}_{k-1})}{\partial \vec{x}_{k-1}} = \begin{bmatrix} \frac{\partial x_k}{\partial x_{k-1}} & \frac{\partial x_k}{\partial x_{y-1}} & \frac{\partial x_k}{\partial \theta_{k-1}} \\ \frac{\partial y_k}{\partial x_{k-1}} & \frac{\partial y_k}{\partial x_{y-1}} & \frac{\partial y_k}{\partial \theta_{k-1}} \\ \frac{\partial \theta_k}{\partial x_{k-1}} & \frac{\partial \theta_k}{\partial x_{y-1}} & \frac{\partial \theta_k}{\partial \theta_{k-1}} \end{bmatrix} \tag{3.44}$$

Para llevar acabo la actualización del algoritmo EKF vamos a considerar a cada una de las marca naturales detectadas por los sensores del robot. Estas marcas las podemos bosquejar dentro de un vector $\widehat{z}$, el cual contiene las componentes de distancia y orientación esperados que se obtuvieron durante la predicción. Estas componentes, idealmente, deben contener los valores desde la posición del robot donde se detectaron, pero como consideramos que no es un modelo lineal, consideramos el ruido provocado por

el ambiente. Como parte del algoritmo, los componentes se obtienen como se muestran en la ecuación 3.45:

$$\widehat{z}_i = h(S_i, *, \vec{x}_k) = \begin{bmatrix} r \\ \phi \\ m \end{bmatrix} = \begin{bmatrix} \sqrt{(S_{i,x} - x_k)^2 + (S_{i,y} - y_k)^2} \\ \text{atan2}\,(S_{i,y} - y_k, S_{i,x} - x_k) - \theta_k \\ S_{i,m} \end{bmatrix} \tag{3.45}$$

Como en el caso anterior de la función $g(\vec{u}_k, \vec{x}_{k-1})$, vamos a obtener la matriz Jacobiana de $h(S_i, *, \vec{x}_k)$ para los cálculos dentro del algoritmo, lo cual resulta en la ecuación 3.4:

$$H_i = \frac{\partial h(S_i, *, \vec{x}_k)}{\partial \vec{x}_{k-1}} = \begin{bmatrix} \frac{\partial r}{\partial x_k} & \frac{\partial r}{\partial x_y} & \frac{\partial r}{\partial \theta_k} \\ \frac{\partial \phi}{\partial x_k} & \frac{\partial \phi}{\partial x_y} & \frac{\partial \phi}{\partial \theta_k} \\ \frac{\partial m}{\partial x_k} & \frac{\partial m}{\partial x_y} & \frac{\partial m}{\partial \theta_k} \end{bmatrix} =$$

$$\begin{bmatrix} \frac{-(S_{i,x} - x_k)}{\sqrt{(S_{i,x} - x_k)^2 + (S_{i,y} - y_k)^2}} & \frac{-(S_{i,y} - y_k)}{\sqrt{(S_{i,x} - x_k)^2 + (S_{i,y} - y_k)^2}} & 0 \\ \frac{(S_{i,y} - y_k)}{\sqrt{(S_{i,x} - x_k)^2 + (S_{i,y} - y_k)^2}} & \frac{-(S_{i,x} - x_k)}{\sqrt{(S_{i,x} - x_k)^2 + (S_{i,y} - y_k)^2}} & -1 \\ 0 & 0 & 0 \end{bmatrix} \tag{3.46}$$

Durante este proceso, la actualización se ejecutará tantas veces como el número de marcas naturales sean detectados por el Kinect del robot. De igual manera que en el caso anterior, vamos a ejemplificar el proceso de localización por medio de un pseudocódigo (algoritmo 3) el cual es más fácil de interpretar este proceso y poder implementarlo en código para que el robot móvil se localice.

Asimismo, por medio del diagrama de bloques que se muestra en la figura 3.14, vemos de manera gráfica como se ejecuta el algoritmo, sobretodo en el proceso de actualización iterativo por medio de las marcas naturales.

Que el algoritmo de EKF sea más exacto y más preciso es depende del numero de marcas naturales reconozca. Es decir, entre más marcas naturales detecte y reconozca el robot, su localización es más precisa. En cambio, si reconoce pocas marcas naturales, aumenta la incertidumbre de poder localizarse en el entorno haciéndolo menos preciso.

En el caso de que el robot no detecte ninguna marca natural, el robot solo navegará con las instrucciones que vaya recibiendo con la predicción. Para estos casos, la localización con EKF puede funcionar siempre y cuando los momentos en los que no detecte una marca sean breves, o de lo contrario el robot navegaría por estimaciones.

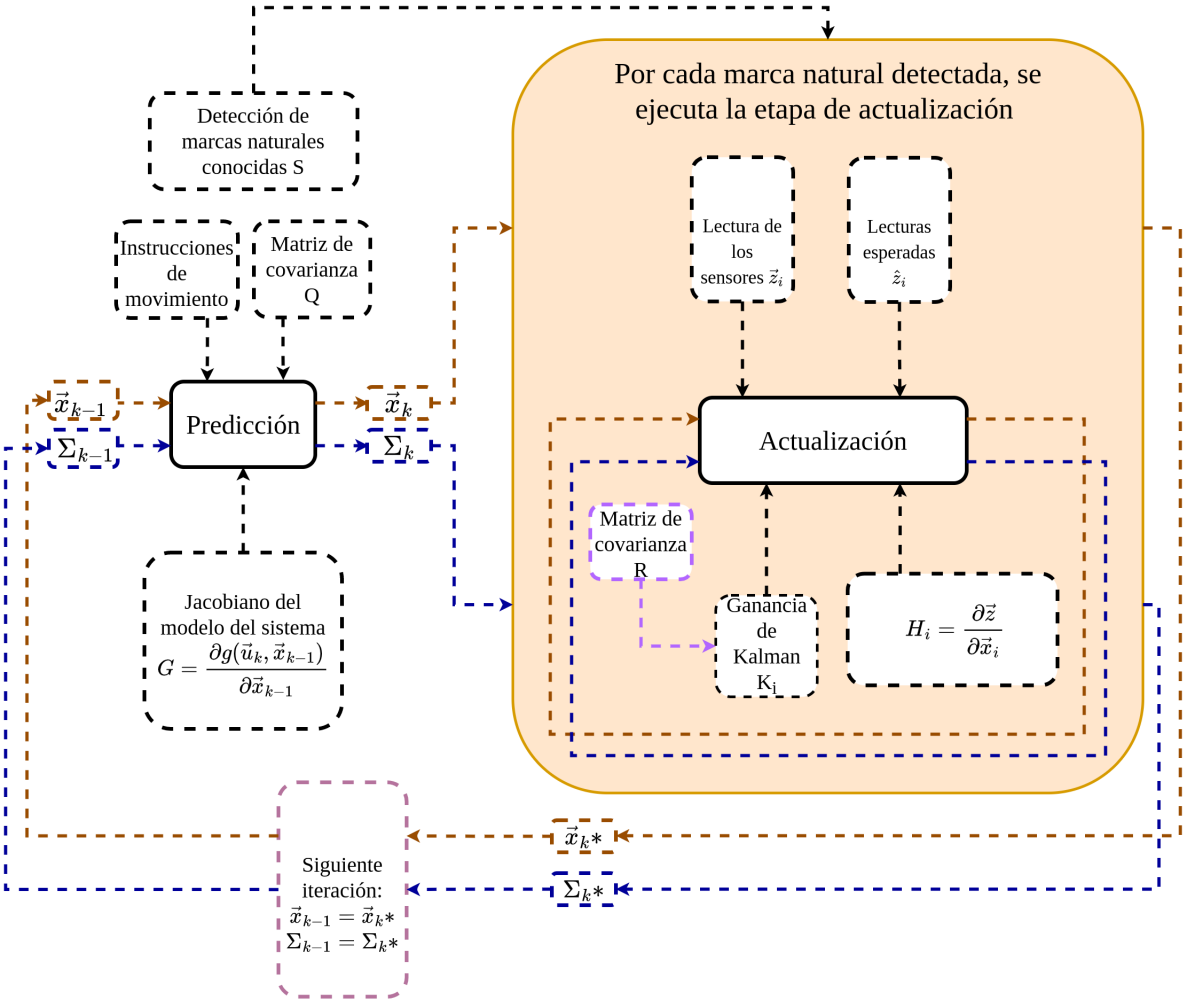

Figura 3.14: Diagrama de flujo del localización por EKF y marcas naturales.

Algorithm 3 Pseudocódigo de la localización con EKF.

procedure EKF ALGORITHM($\vec{x}_{k-1}, \Sigma_{k-1}, \vec{u}, S$)

Predicción ▷ Calculamos x_k, y_k y θ_k de una iteración anterior.

$x_k = x_{k-1} + u_d * cos(\theta_k)$

$y_k = y_{k-1} + u_d * sin(\theta_k$

$\theta_k = \theta_{k-1} + \theta_k$

▷ $x_{k-1}, y_{k-1}, \theta_{k-1} = 0$ si es el inicio

$$G = \begin{bmatrix} 1 & 0 & -u_d * sin(\theta_k) \\ 0 & 1 & u_d * cos(\theta_k) \\ 0 & 0 & 1 \end{bmatrix}$$

$\Sigma_k = G\Sigma_{k-1}G^T + Q$

Actualización

for `(i=0;i<n;i++)` **do**

$$\widehat{z} = \begin{bmatrix} \sqrt{(S_{i,x} - x_k)^2 + (S_{i,y} - y_k)^2} \\ \text{atan2}\,(S_{i,y} - y_k, S_{i,x} - x_k) - \theta_k \\ S_{i,m} \end{bmatrix}$$

$$z = \begin{bmatrix} S_{i,r} \\ S_{i,\phi} \\ S_{i,m} \end{bmatrix}$$

$$H = \begin{bmatrix} \frac{-(S_{i,x}-x_k)}{\sqrt{(S_{i,x}-x_k)^2+(S_{i,y}-y_k)^2}} & \frac{-(S_{i,y}-y_k)}{\sqrt{(S_{i,x}-x_k)^2+(S_{i,y}-y_k)^2}} & 0 \\ \frac{(S_{i,y}-y_k)}{\sqrt{(S_{i,x}-x_k)^2+(S_{i,y}-y_k)^2}} & \frac{-(S_{i,x}-x_k)}{\sqrt{(S_{i,x}-x_k)^2+(S_{i,y}-y_k)^2}} & -1 \\ 0 & 0 & 0 \end{bmatrix}$$

$I = (H * \Sigma_k * H^T) + R$

$K = \Sigma_k * H^T I^{-1}$ ▷ Ganancia de Kalman

$Z' = z - \widehat{z}$

$\vec{x}_k = \vec{x}_k + K * Z'$

$\Sigma_k = (M_I - K * H) * \Sigma_k$

▷ M_I es la matriz identidad

end for

return $(\vec{x}_k, \Sigma_k)$

end procedure

Capítulo 4

Técnicas de aprendizaje

Dentro de este capitulo se detalla la implementación de una técnica de aprendizaje, como lo es una red neuronal. La implementación de esta red neuronal para el presente trabajo se debe a que facilita la detección de marcas naturales dentro de un entorno. La detección de marcas por esta técnica se usa en conjunto con el filtro de Kalman extendido para hacer localización de un robot móvil.

4.1. Aprendizaje profundo

Se le conoce como aprendizaje automatizado (también conocido como aprendizaje de máquina) al estudio de los programas que aprenden a partir de ejemplos de manera autónoma en lugar de ser manualmente programados.

Durante las últimas décadas, la implementación del aprendizaje profundo (deep learning en inglés) ha tomado un impacto significativo dentro del desarrollo de sistemas basados en tareas que se encargan de realizar reconocimiento de voz, clasificación y/o segmentación de imágenes, entre otras más.

El aprendizaje profundo es un tipo de aprendizaje automatizado el cual consiste en la implementación de redes neuronales artificiales, la cual utiliza una estructura jerárquica, en la cual en cada nivel se obtienen las características significativas de un nivel anterior. El uso de modelos neuronales facilita el procesamiento de la información dentro de los sistemas de cómputo.

Goodfellow menciona en su libro un ejemplo (7). Este menciona que el uso del modelos neuronales es difícil para una computadora por el hecho de darle significado a un conjunto de píxeles de una imagen de entrada. Implementar el aprendizaje profundo para resolver este tipo de problemas es factible, porque en el momento, se plantea en el modelo neuronal para la imagen, se pueden obtener patrones representativos en cada capa del modelo.

Se puede ejemplificar un modelo neuronal para extraer características de una imagen como entrada. En este modelo lo que se realiza es: dado un conjunto de píxeles de entrada, determinar en cada nivel ciertas características, y con dicha información

podemos utilizarla como se necesite, ya sea para clasificar y/o reconocer. Este nivel del modelo es conocido como la capa visible porque se conocen los datos de entrada (en este caso los píxeles de una imagen). En las siguientes niveles, cada capa se dedica a extraer características abstractas, las cuales le indican al modelo si lo que obtiene pueden ser contornos, bordes o esquina.

Estas capas son conocidas como capas ocultas, puesto que las variables que manejan son propias del modelo y no son son vistas por el desarrollador. Al final del modelo, podemos encontrar una capa de salida, la cual da como resultado una descripción a partir de los patrones que fueron obtenidos.

4.2. Algoritmos de aprendizaje automatizado

Podemos implementar las técnicas de aprendizaje automatizado por medio de algoritmos, lo cual hace que el sistema pueda aprender partiendo de la información que recibe como entrada. Estos algoritmos adaptan el proceso de aprendizaje, haciendo que una red neuronal artificial sea capaz de cambiar sus parámetros y lograr un resultado deseado. Esta modificación se realiza teniendo un conjunto de muestras de entrada y salida, para posteriormente, ejecutar iterativamente procesos de corrección de errores hasta llegar a un resultado objetivo (26).

Los algoritmos de aprendizaje se pueden clasificar en dos tipos, los cuales son el aprendizaje automatizado supervisado y el no supervisado:

- El aprendizaje automatizado supervisado es un método de análisis de datos el cual implementa algoritmos que aprenden iterativamente de los datos para permitir que las computadoras puedan encontrar información que se encuentre oculta para la máquina. El implementar el aprendizaje supervisado nos ayudar a resolver problemas por medio de conjuntos de datos etiquetados para que posteriormente sean entrenados para un algoritmo de propósito especifico.
- En cambio, el aprendizaje automatizado no supervisado es un método implementado para identificar patrones nuevos y/o detección de anomalías dentro de los datos. Estos datos cuando entran al algoritmo no se encuentran etiquetados, por lo cual el algoritmo trata de relacionarlos obteniendo sus características o patrones.

4.3. Redes Neuronales Artificiales

Una red neuronal artificial es un modelo computacional representativo que busca recrear los procesos biológicos que lleva acabo el sistema nervioso y la forma en la que el cerebro lleva a acabo el almacenamiento, aprendizaje y reconocimiento de patrones provenientes de una fuente de información. El término de red neuronal tiene su origen desde los primeros intentos para poder dar una representación matemática del procesamiento de la información en los sistemas biológicos.

4.3.1. Estructura

Una red neuronal artificial esta constituida (figura 4.1) por un conjunto de neuronas que están estrechamente interconectadas y organizadas en capas diferentes. Estas capas son la capa de entrada que recibe la entrada, la capa de salida produce el resultado final; normalmente, se tienen una o más capas ocultas que están intercaladas entre la entrada y la salida. Esta estructura hace que sea imposible predecir o conocer el caudal exacto de los datos.

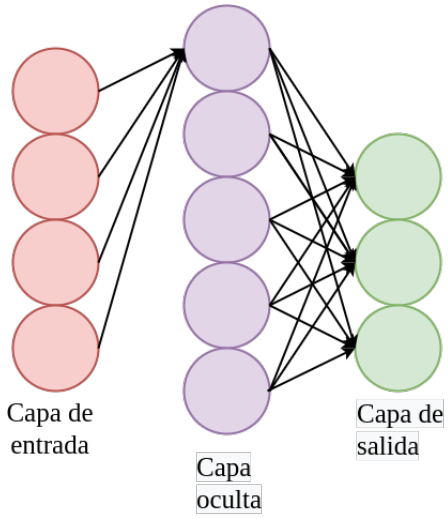

Figura 4.1: Representación de una red neuronal.

Cada capa esta constituida por un conjunto de nodos, los cuales representan una neurona artificial. Para cada capa se tienen las siguientes características:

- **Capa de entrada:** En esta capa se toman los datos de entrada. Esta información es obtenida por medio de sensores o de algún sistema que trabaje a bajo nivel.
- **Capas ocultas:** En estas capas se obtienen las características abstractas provenientes de los datos de entrada, y posteriormente se genera una descripción de los patrones más significativos. Pueden contener una o varias capas ocultas dependiendo de los patrones que se necesiten identificar.
- **Capa de salida:** En esta capa se muestran los resultados representados en forma de una detección, de una interpretación o de una clasificación que relacione a la información de entrada.

4.3.2. Perceptrón

Entre todos los modelos neuronales, existe un modelo simple el cual se le conoce como perceptrón. El perceptrón es el modelo matemático mas simple de una neurona.

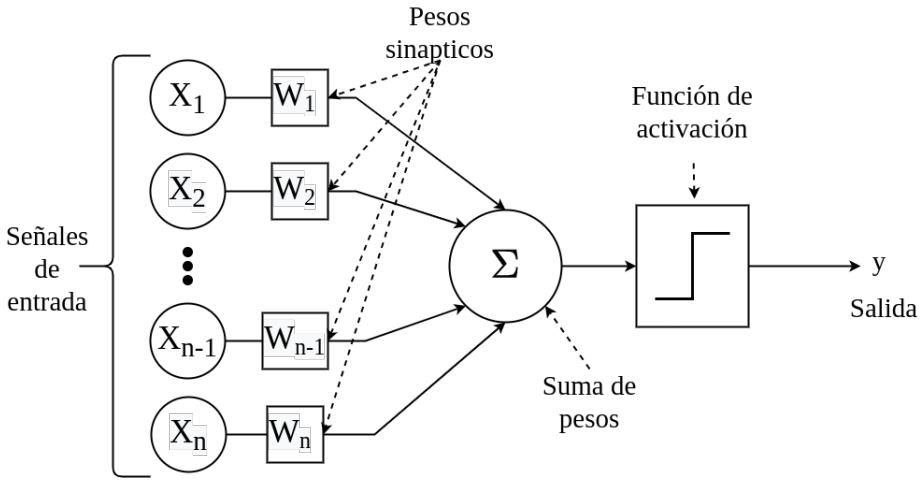

Figura 4.2: Representación de un perceptrón.

Este modelo fue implementado por primera vez por Frank Rosenblatt en 1957 (27), el cual dio importantes aportes introduciendo variables numéricas conocidas como pesos y la implementación de una función de activación que se encuentra en cada neurona, la cual se dedica a dar una toma de decisión de transmitir o no los datos a una capa posterior.

Dicho modelo propuesto del percerptrón de Rosenblatt (como se muestra en la figura 4.2) está constituido por los siguientes elementos (27):

- Se tiene un vector de entrada $\vec{x} = (x_1, x_2, \ldots, x_n)$, el cual representa la entrada de los datos a la neurona para ser analizados.
- Para cada entrada se tiene un vector de pesos asociado $\vec{w} = (w_1, w_2, \ldots, w_n)$ a cada una de las entradas. Los pesos son coeficientes que se adaptan dentro de la red neuronal y determinan que intensidad tendrá la señal de entrada.
- Además del vector $\vec{x}$, se considera una variable Θ llamada sesgo, la cual funciona como una variable que determina el qué tan predispuesta está la neurona a disparar un 1 o un 0 independientemente de los pesos.
- Una suma ponderada dada por $\Sigma w_j x_j$, la cual hace la suma de cada entrada con su respectivo peso y el sesgo θ para determinar el resultado.
- Una función de activación que en este caso es una función escalón de Heaviside, que esta definida como en la ecuación 4.1:

$$f(x) = \begin{cases} 1 \; si \; \sum_i^n w_j x_j + \theta > 0 \\ 0 \; otro \; caso \end{cases} \tag{4.1}$$

Cabe mencionar que las entradas del vector de pesos $\vec{w}$ son valores que están ya predeterminados, pero en caso de no obtener los pesos $\vec{w}$, se pueden implementar los algoritmos de aprendizaje automatizado que se mencionaron previamente.

4.4. Redes Neuronales Convolucionales

Una red neuronal convolucional es un conjunto de neuronas que buscan tratar de replicar el comportamiento de las neuronas que el ser humano tiene en la corteza visual primaria (V1), las cuales nos ayudan a identificar la posición y la orientación de un objeto el cual se este observando. Esta red convolucional artificial es una variante de un perceptron multicapa que es muy efectiva para implementarse en tareas de visión computacional artificial como lo es la segmentación o la clasificación de imágenes.

Cabe mencionar que la convolución consiste en una operación matemática en la cual se forma una función a partir de otras dos. Esta nueva función tiene como cualidad que representa la magnitud superpuesta de sus dos funciones origen. Suponiendo que la función f es una función entrada y g una función núcleo, obtenemos la convolución $f * g$ definida como la integral del producto entre las dos funciones considerando un desfase t dentro de una de las funciones, como se muestra en la ecuación 4.2. De la misma forma, podemos calcular la convolución en su forma discreta como se muestra en la ecuación 4.3.

$$(f * g)(t) = \int_{-\infty}^{\infty} f(x)g(x-t)dx \tag{4.2}$$

$$v(t) = (f * g)(t) = \sum_{n} f(n)g(n-t) \tag{4.3}$$

La convolución vista desde la perspectiva de aprendizaje profundo, utiliza a la función entrada como un arreglo multidimensional con los datos, y la función núcleo como un arreglo multidimensional de parámetros adaptables por medio de un algoritmo de aprendizaje (7). Podemos ver esto con un ejemplo en donde tomamos una imagen de entrada, la cual es una matriz bidimensional I y una matriz núcleo bidimensional Q, resultando en la ecuación 4.4:

$$v(i,j) = (I * Q)(i,j) = \sum_{m}\sum_{n} I(m,n)Q(i-m, j-n) \tag{4.4}$$

En diferentes fuentes de información con respecto a las redes neuronales, algunas literaturas consideran e implementan una función semejante conocida como correlación

cruzada. Esta es calculada de la misma manera que la convolución sin girar la matriz núcleo Q, quedando en la ecuación 4.5:

$$v(i,j) = (Q * I)(i,j) = \sum_m \sum_n I(i+m, j+n) Q(m,n) \tag{4.5}$$

4.4.1. Construcción y funcionamiento de una red neuronal convolucional

La red convolucional esta constituida de múltiples capas con filtros convolucionales de una o más dimensiones, para realizar un mapeo no lineal que se encuentra añadida entre capa y capa. Cuando una red neuronal convolucional se usa para clasificación de imágenes, estas pasan por una fase de extracción de características. La fase esta conformada de neuronas convolucionales, neuronas de reducción de muestreo las cuales se encargan de procesar la información de la imagen pudiendo reducir su dimensión, y de neuronas de perceptron más sencillas las cuales pueden aprender una orientación del objeto de interés. Entre más neuronas con perceptron se usen, se pueden identificar las diferentes orientaciones del objeto.

4.4.2. Aprendizaje de una red neuronal convolucional

La red convolucional aprende reconociendo varios objetos que se encuentren en una imagen, pero para ello se ocupan una cierta cantidad de muestras para que las neuronas capten las características únicas de cada objeto. Por ejemplo, si dentro de la red una neurona ha visto una misma imagen varias veces, va a inferir de otras imágenes que no haya visto antes buscando similitudes que pudieran existir creando un reconocimiento inteligente.

El proceso de aprendizaje y funcionamiento de la red neuronal convolucional es el siguiente:

- **Neuronas con píxeles:** La red neuronal convolucional toma como entrada los píxeles de una imagen. Por ejemplo, si una imagen de 8 píxeles de alto por 8 píxeles ancho, es equivalente a que se usen 64 neuronas. Ahora, son 784 píxeles si fuera una imagen a escala de grises (como se muestra en la figura 4.3) puesto que se considera que tiene un canal o solo color.

  En cambio, si fuera una imagen a color en RGB se necesitan 3 canales (red, green, blue), por ejemplo, para la figura 4.4 usaríamos 5 píxeles de ancho x 5 píxeles de alto x 3 canales = 75 neuronas de la capa de entrada.

- **Procesamiento previo:** Un paso intermedio antes de que la información de la imagen entre a la red neuronal convolucional es de convertir los 1 y 0 de los píxeles y dividirlos entre 255 porque la escala de los píxeles en RGB va de 0 a 255.

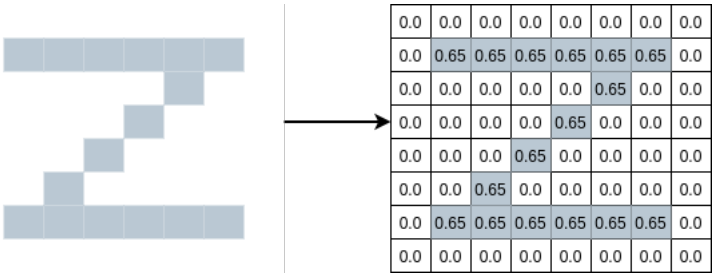

Figura 4.3: Representación en píxeles de una imagen en escala de grises (un canal).

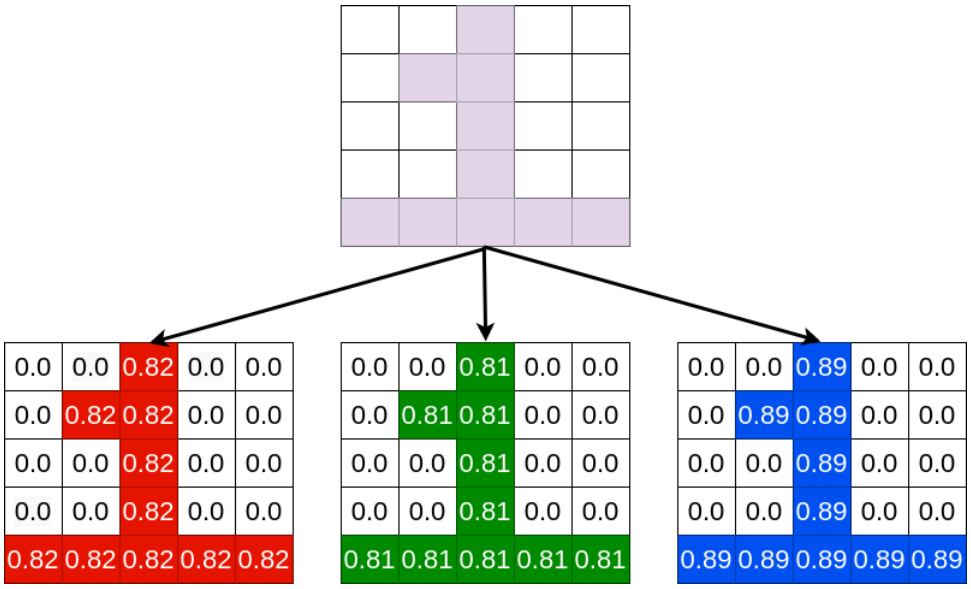

Figura 4.4: Representación en píxeles de una imagen en RGB (tres canales).

Como podemos ver en la figura 4.4 esta separada en escala RGB y a su vez hacer un reescalamiento para que tome valores del 0 al 255. En este caso el canal R representa un 209, el canal G un 206 y finalmente el canal B representa un 227.

- **Convolución:** La etapa de la convolución dentro de las neuronas consiste en tomar la imagen de entrada e ir tomando grupos de píxeles que generan matrices de una cierta dimensión (3x3 para este caso) y vamos a tener un kernel de la misma dimensión para realizar un producto escalar para generar una matriz nueva de salida. En dado caso que sean más de un kernel se vuelve a tomar otro conjunto con el mismo numero de píxeles para formar la matriz.

- **Filtros con el kernel:** Los filtros son un conjunto de kernels los cuales van a estar operando con las matrices, puesto que la operación entre matrices no solamente se realiza una sino varias veces.

  Si se tienen n filtros, vamos a obtener n matrices de salida; por ejemplo para

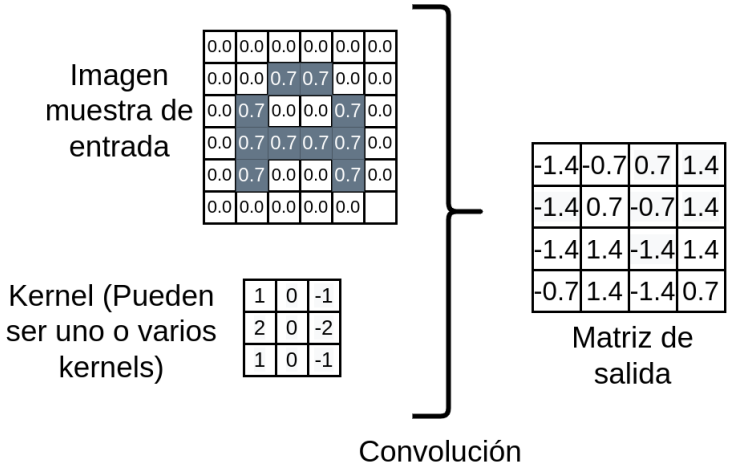

Figura 4.5: Representación de la operación entre matrices dentro de la convolución.

una primer capa oculta, si tenemos 16 filtros tendremos 16 matrices de salida y si tenemos nuestra imagen de 5×5 por 3 canales RGB nos dá como resultado 1200 neuronas de la primera capa.

Por lo tanto, conforme vamos moviendo el kernel vamos generando una imagen nueva filtrada por el kernel. Dentro de estas imágenes nuevas lo que se esta bosquejando son ciertas características de la imagen original, lo cual ayudará en el futuro a poder distinguir un objeto de otro.

En la figura 4.5 podemos ver que en la imagen de entrada en canal de grises los píxeles ya tienen la representación en 255 y el kernel a implementar. Podemos observar que al momento de operar el kernel con las submatrices de 3 x 3, nos generan un nuevo valor. El kernel opera tomando nuevos subconjuntos con un desplazamiento de píxel a píxel dando como resultado de una matriz de 6 x 6 a una de 4 x 4. También se observa que mantiene la mayor parte de las características, que para este caso es el tono de gris.

- **Función de activación:** La función de activación es una función que se encarga de transmitir la información que está generada por la combinación lineal de los pesos y las entradas, es decir, que es una función la cual impone un limite o un umbral al resultado que pase a la siguiente neurona.

  Dado que la red neuronal es capaz de resolver problemas que se vuelven mas complejos y difíciles, las funciones de activación hacen que los modelos que son entrenados no sean lineales. Algunas de las funciones de activación son: La función sigmoide, la función de tangente hiperbólica o la función ReLU siendo esta ultima de las que son mas común implementar en redes convolucionales.

  Una función ReLU (por sus siglas en inglés Rectified Lineal Unit) es una función que convierte los valores que son introducidos dentro de ella haciendo ceros los valores negativos y dejando los positivos tal y como entran. Un ejemplo de ello lo

vemos en la figura 4.6 la cual vemos que aplicando ReLU a la matriz resultante de las operación con el kernel identifica los valores negativos para posteriormente hacerlos cero.

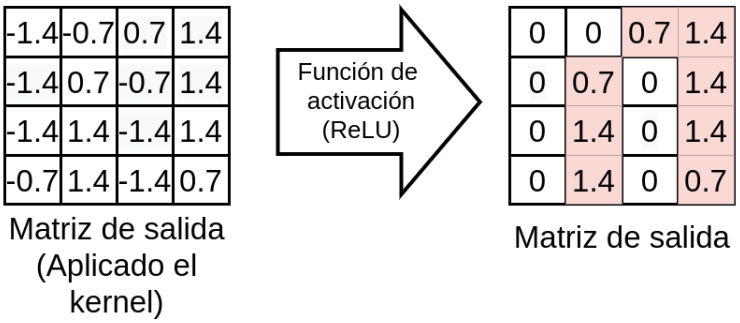

Figura 4.6: Representación de la operación ReLU.

- **Muestreo (Max Pooling):** El motivo por el cual se hace un muestreo después de obtener la matriz de salida por parte del filtro, es porque como se vio anteriormente, una capa oculta puede generar una gran cantidad de salidas que pasarían a la siguiente capa oculta, lo cual haría que se requiera una gran cantidad de cómputo al momento de realizar las convoluciones y las operaciones con los filtros.

  Entonces, lo que se realiza es una reducción del numero de salidas haciendo un muestreo, el cual mantiene las características mas importantes que fueron detectadas por cada uno de los filtros. Una de las técnicas más utilizadas para realizar muestreo es Max-Pooling.

  El Max-Pooling es un proceso de discretización por medio de las muestras. Consiste en reducir las dimensiones de una representación de entrada (ya sea una imagen, matriz de salida de capa oculta, etc.), para que posteriormente se hagan suposiciones sobre las características contenidas en las sub-regiones agrupadas. La idea principal de Max-Pooling es el de ser un filtro que se aplica para generar un nuevo conjunto, el cuál es el valor máximo de todos los elementos que se encuentren en el subconjunto.

  En la figura 4.7 vemos que a la matriz de 4×4 resultante de la función ReLU se le puede aplicar Max-Pooling, tomando subconjuntos de la matriz de 2×2 para redimensionar y generar una nueva celda con el valor máximo que se haya encontrado en cada subconjunto de la matriz.

  Hasta este punto, se considera que se obtuvo una primer convolución, la cual considera que se detectaron características primitivas propias de una imagen (como lo pueden ser lineas rectas o curvas). Al realizar más convoluciones a la imagen podemos extraer características más complejas por cada capa convolucional, hasta tener un conjunto de capas convolucionales.

- **Capa totalmente conectada (fully connected):** Ésta capa tiene la cualidad de ser una sub-red neuronal de arquitectura totalmente conectada, es decir, que todas

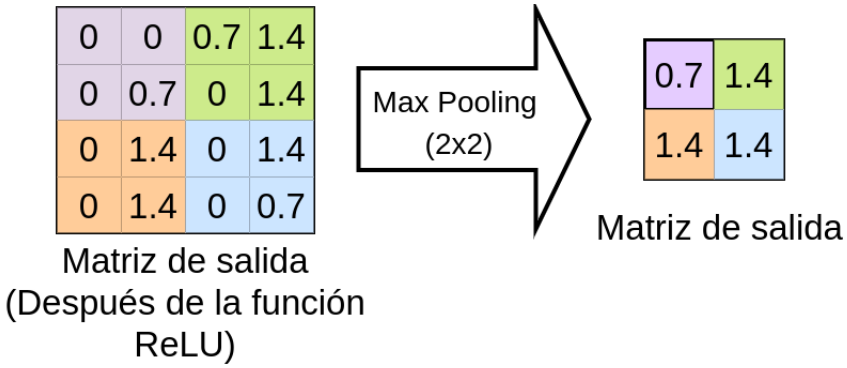

Figura 4.7: Representación del proceso de Max-Pooling.

su neuronas están conectadas a una capa anterior. En nuestro caso es la última capa de convolución en donde se encuentran el conjunto de características de una capa anterior. Esta capa tiene como función principal agrupar la información entrante para poder realizar cálculos para posteriormente hacer una clasificación a la imagen (figura 4.8).

En cuestión de redes convolucionales, es común el uso de varias capas totalmente contactadas en serie para detectar características más especificas. La última de estas contiene el número de clases que encontraron en el conjunto de datos. Estos valores de salida de la capa pasarán a una función probabilística que se encargará de hacer la clasificación.

- **Función de clasificación:** Básicamente, la tarea de esta función es clasificar los resultados de la capa totalmente conectada por medio de una función de probabilidad que nos diga a que clase pertenece la imagen que procesamos. La función más utilizada en redes convoluciones es la función de softmax, la cual nos muestra la distribución de probabilidad de la clase a la que puede pertenecer.

  Para esto nos retorna un vector n con las probabilidades, cuya suma es uno. La ecuación que representa esta función se muestra en la ecuación 4.6.

$$s(y_i) = \frac{e^{y_i}}{\sum_k e^{y_j}} \tag{4.6}$$

En la mayoría de las arquitecturas de redes neuronales, una capa es alimentada por una capa anterior, pero en otras arquitecturas se consideran otro tipo de capas llamadas capas residuales, las cuales son capas que alimentan información a la siguiente capa y a otra capa que se encuentra a n capas de distancia, como se muestra en la figura 4.9.

Esto es para que no se presente el problema de desvanecimiento del gradiente, el cual consiste en que llegara un punto en el que la red no encuentre un gradiente que

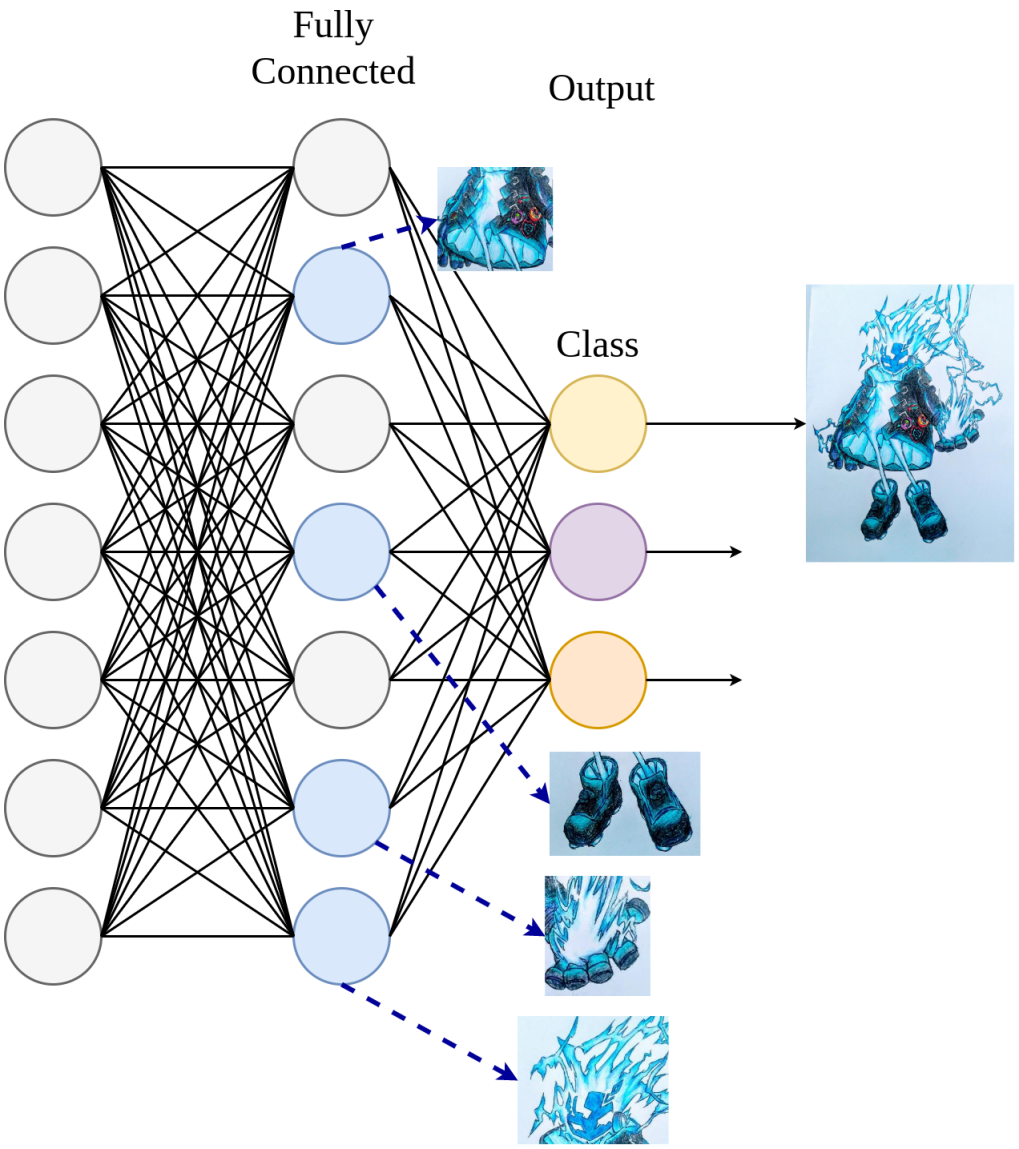

Figura 4.8: Representación de una capa totalmente conectada.

sea óptimo cuando se tienen una gran cantidad de capas en la arquitectura. La red residual propaga gradientes que sean más grandes a las capas del inicio, haciendo que estas últimas puedan aprender eficientemente como lo haría una capa final (29).

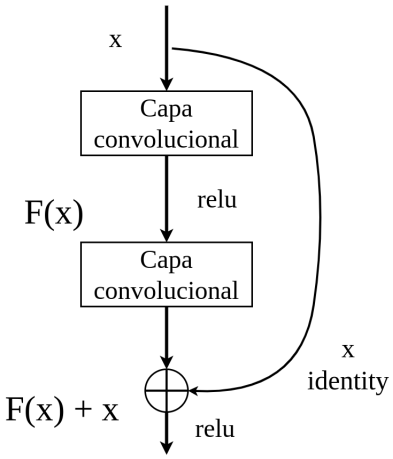

Figura 4.9: Representación de una capa residual. Tomado de (29).

4.5. Retro-propagación de una red Neuronal

Cuando una red neuronal implementa dentro de su arquitectura una red previamente alimentada (conocido en inglés como feedforward network), esta funciona recibiendo una entrada dada por a y una salida dada por b. Esto quiere decir que la información se propaga en dirección de a hacia b, de tal forma que pase por todas las capas ocultas que contiene la red neuronal dando un resultado. A este tipo de propagación se le conoce como propagación hacia adelante (conocido en inglés como forward propagation). A partir de la salida de la red, podemos calcular el error que se presentó en la forward propagation con respecto a la base de datos del entrenamiento.

Se le conoce al algoritmo de retro-propagación (back propagation conocido en inglés) a un algoritmo basado en aprendizaje supervisado el cual hace que que el error se propague hacia atrás. Esto se hace para que la función de costo vaya minimizando el error en función del ajuste de parámetros de pesos y sesgos, todo esto usando el algoritmo de descenso por gradiente.

La función de costo dentro de la red neuronal es una una herramienta que mide el desempeño en un modelo de aprendizaje de máquina que tiene como objetivo cuantificar el error que existe entre las predicciones obtenidas respecto a los datos reales del modelo. Implementar esta función nos ayuda a encontrar los parámetros que reduzcan la perdida (11).

Existen diferentes funciones de costo que son aplicables dependiendo del problema, como lo son el error medio absoluto (ecuación 4.7), el error cuadrático medio (ecuación 4.8), la entropía cruzada binaria (ecuación 4.9) o la entropía cruzada categórica (ecuación 4.10). En todas la ecuaciones se comparte las variables; i que es el índice de la muestra, $\widehat{y}$ es el valor que se predice, x el valor esperado, y n el número de clases del conjunto de datos.

$$EMA = \frac{1}{n}\sum_{i=1}^{n}|\widehat{y_i} - x_i| \tag{4.7}$$

$$ECM = \frac{1}{2}\sum_{i=1}^{n}(\widehat{y_i} - x_i)^2 \tag{4.8}$$

$$ECB = -\frac{1}{n}\sum_{i=1}^{n}(x * \log(\widehat{y_i}) + (1-x) * (\log(1-\widehat{y_i})) \tag{4.9}$$

$$ECE = -\frac{1}{n}\sum_{i=1}^{n} x_i * \log(\widehat{y_i} \tag{4.10}$$

4.5.1. Algoritmo por descenso del gradiente

El descenso del gradiente es un algoritmo meta-heurístico incremental, que lo que hace es minimizar el valor de una función de costo. Este se calcula a partir de una primer solución escogida aleatoriamente o definida previamente como punto de partida para que posteriormente el algoritmo optimice la función sin modificar el resultado (**?**).

El algoritmo calcula el gradiente iterativamente en dirección del menor valor en cada repetición, que se encuentra definido por el negativo del gradiente $-\nabla J$. Podemos escribir la relación entre un cualquier peso $w_{i,j}$ y una función de costo J por medio de derivadas parciales y la regla de la cadena, quedando en la ecuación 4.11 donde J es una función de costo, $w_{i,j}$ es el peso i en la capa j, a_j es la salida de la neurona después de la función de activación y z_j es la combinación lineal del resultado de $\sum \vec{x}\vec{w}$.

$$\frac{\partial J}{\partial w_{i,j}} = \frac{\partial J}{\partial a_j}\frac{\partial a_j}{\partial w_{i,j}} = \frac{\partial J}{\partial a_j}\frac{\partial a_j}{\partial z_j}\frac{\partial z_j}{\partial w_{i,j}} \tag{4.11}$$

Con base a esto, podemos calcular un nuevo valor para el peso $w_{i,j}$ por medio de la ecuación 4.12, donde α es la tasa de aprendizaje del modelo.

$$w_{i,j} = w_{i,j} - \alpha\frac{\partial J}{\partial w_{i,j}} \tag{4.12}$$

4.5.2. Transferencia de conocimiento

La transferencia de conocimiento es una técnica a nivel aprendizaje de máquina que implementa un modelo pre-entrenado de una red neuronal que tiene como objetivo hacer un nuevo entrenamiento de un conjunto pequeño de imágenes, pero que tienen una gran cantidad de características que tienen en común con el modelo preentrenado.

Para esta metodología de aprendizaje se utiliza un modelo pre-entrenado que haya sido generado con un conjunto de datos robusto para que posteriormente la red haga un reentrenamiento o un ajuste fino de las últimas capas de la arquitectura (las capas de clasificación). Por lo cual, cuando se genere el nuevo modelo éste aprovechará las características que haya obtenido del modelo pre-entrenado y clasificaría solamente las nuevas clases que el sistema necesita.

4.5.3. Sobre-ajuste y sub-ajuste

Uno de los aspectos importantes a considerar dentro de una red neuronal, es que pueda tener generalización; es decir, que los datos que entren a la red sean distintos a los datos que se utilizaron para el entrenamiento del modelo. La implementación de sobre-ajuste y sub-ajuste en el modelo hace que este se desempeñe de manera correcta al momento de recibir nuevos datos a la red (7).

Se debe tener en cuenta que estos datos nuevos que entran a la red contienen información que puede ser irrelevante, distorsionada, con algún ruido o con muestras que no sean lo suficientemente funcionales al momento de hacer el entrenamiento de la red. Aquí es cuando el modelo pasa un sobre-ajuste, es decir, que el modelo al momento de entrenar puede predecir con ayuda de un rango los datos que sean aceptados y descartar los que no pertenezcan al rango de valores.

Caso contrario es el sub-ajuste, este se presenta cuando el modelo no cuenta con suficientes datos de entrada para el entrenamiento, haciendo una generalización baja y en consecuencia hace predicciones de clases incorrectas. En la figura 4.10 podemos ver los ejemplos de sub-ajuste y de sobre-ajuste.

Algunas consideraciones que debemos tener presentes para evitar la presencia de sobre-ajuste o sub-ajuste en la red neuronal son:

- Tener una variedad y una cantidad equitativas de clases.
- Evitar tener una cantidad excesiva de capas neuronales dentro de la arquitectura de la red.
- Seccionar el conjunto de datos en un conjunto de entrenamiento y en un conjunto de validación.
- Tener una cantidad considerable de muestras para cada una de las clases a entrenarse.

- Realizar técnicas de data augmentation al conjunto de datos; es decir, aplicar técnicas incluyendo filtros que mejoren el aprendizaje del conjunto de datos.

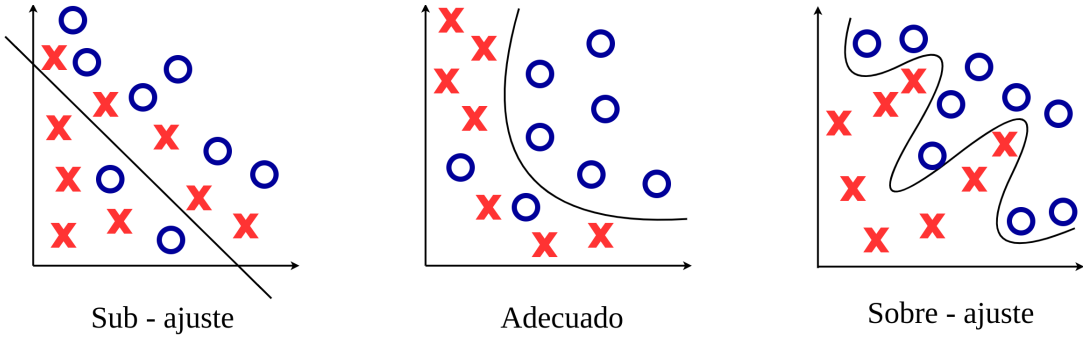

Figura 4.10: Representación sobre ajuste y sub ajuste.

4.6. YOLO

YOLO (You Only Look Once por sus siglas en inglés) es un algoritmo de detección de objetos en tiempo real basado en redes neuronales presentado por Joseph Redmon, Santosh Divvala, Ross Girshick y Ali Farhadi en el año 2015 (22). Ellos mencionan en su articulo que la forma en la que pueden resolver el problema de detectar imágenes es viendo el problema como un problema de regresión y no un problema de clasificación.

La detección se realiza por medio de la implementación de una red neuronal convolucional arrojando como resultado un cuadro delimitador (bounding box en inglés), el cual tiene asociada una probabilidad de que tan cercana está la imagen de entrada a las entrenadas. Cabe mencionar que una escena puede tener varios bounding boxes de las imágenes a detectar, como se muestra en la figura 4.11.

Para este proyecto de tesis, se utilizo la versión 4 de YOLO (YOLOV4). El uso de ésta versión se debe a que la velocidad y precisión al momento de entrenar hace que la detección de objetos sea óptima en comparación a sus versiones anteriores (figura 4.12).

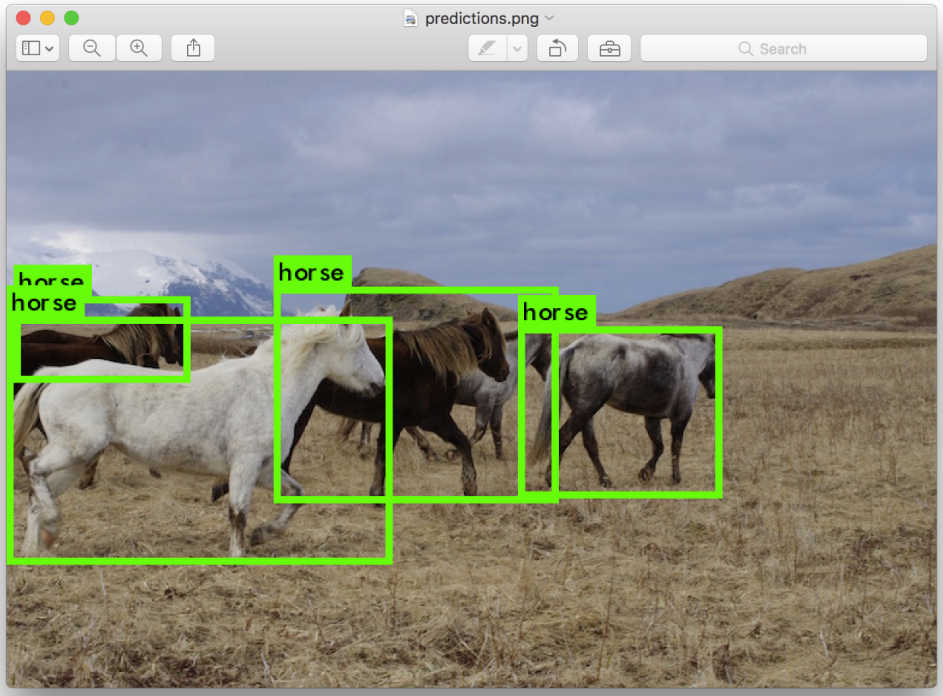

Figura 4.11: Detección de objetos implementando YOLOV4. Tomada de (19)

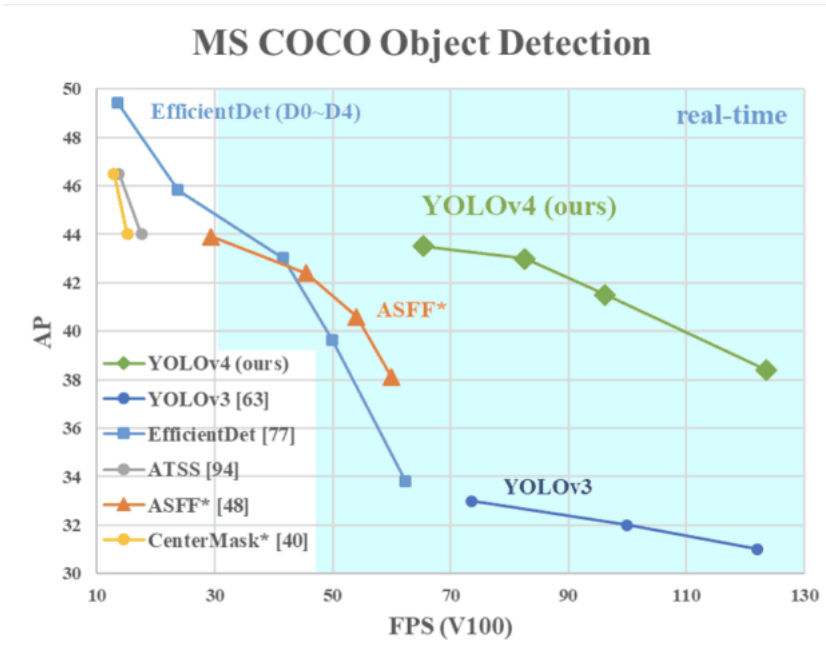

Figura 4.12: Rendimiento de YOLOV4 ante sus versiones anteriores. Tomada de (1).

4.6.1. Arquitectura de YOLO

La arquitectura que contiene YOLOV4 la podemos ver en la figura 4.13, en donde observamos una estructura del algoritmo del cual se basa la estructura de YOLOV4. Está arquitectura esta constituida de varios componentes, los cuales son: un conjunto de imágenes como entrada para alimentar a la red para su entrenamiento, luego se tiene el *Backbone* y el *Neck* que se dedican a hacer la extracción de características de cada imagen, y por último el *Head* que realiza la parte de detección/clasificación del conjunto de datos (1).

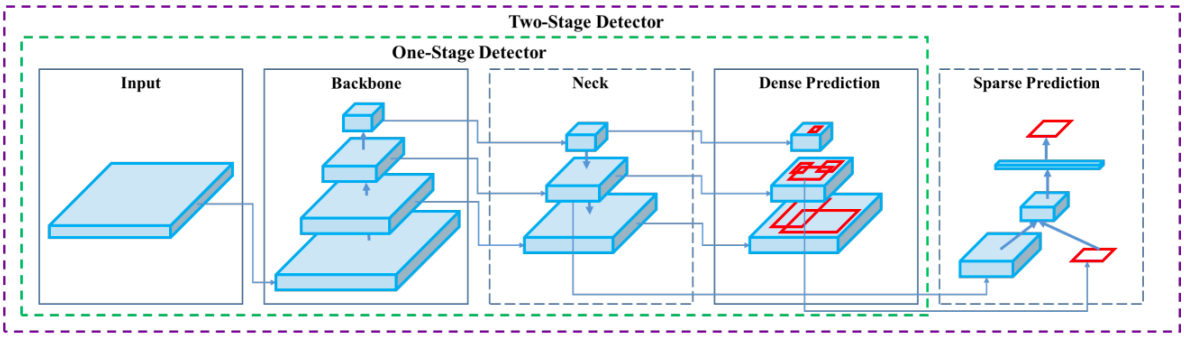

Figura 4.13: Arquitectura de YOLOV4. Tomada de (1).

4.6.2. Backbone

Un *Backbone* es una red neuronal profunda que está formada por capas convolucionales. El objetivo principal de esta red es hacer la extracción de las características esenciales de un conjunto de datos (en este caso imágenes). Es común encontrar un *Backbone* en redes neuronales previamente entrenadas ya que son usadas para hacer entrenamiento y mejorar el rendimiento de detección de los objetos. El *Backbone* está constituido de tres partes:

- **Bag of freebies (BoF):** En esta parte del *Backbone* se aplican métodos al conjunto de datos que faciliten la extracción de características; es decir, busca mejorar o cambiar el costo del entrenamiento para que se pueda realizar siempre y cuando se mantenga un costo de inferencia bajo. Algunos de estos métodos son: *Data augmentation*, *Geometric distorsion*, *IoU loss*, *focal loss*, entre otros.
- **Bag of specials (BoS):** Aquí se tienen conjunots de métodos para aumentar en cantidades pequeñas el costo de la inferencia para mejorar de manera significativa la precisión al momento de detectar los objetos entrenados.
- **CSPDarknet53:** Es una arquitectura basada en DenseNet, que lo que realiza es concatenar entradas previas con las entradas actuales antes de entrar a capas densas dentro de una red.

Las capas dentro de una etapa de DenseNet (figura 4.14) están constituidas de un bloque denso y de una capa de transición. Para cada bloque se tiene una cantidad de n capas densas. Estas capas se conectan de tal manera que la entrada i-ésima de la capa densa se conecta a su salida i-ésima, dando un resultado concatenado generando una nueva entrada (i+1)-ésima en la capa densa, como se muestra en la ecuación 4.13.

$$\begin{aligned} x_1 &= w_1 * x_0 \\ x_2 &= w_2 * (x_1, x_0) \\ x_3 &= w_3 * (x_2, x_1, x_0) \\ &\vdots \\ x_n &= w_n * (x_{n-1}, \ldots, x_2, x_1, x_0) \end{aligned} \tag{4.13}$$

Donde:

- representa el operador de convolución
- $[x_0, x_1, \ldots]$ significa concatenar x_0, x_1, ...
- w_i y x_i son los pesos y la salida de la i-ésima capa densa, respectivamente.

En cambio, la arquitectura **CSP** (Cross Stage Partial por sus siglas en inglés) es una etapa basada en DenseNet, solo que este divide la entrada i-ésima en dos partes las cuales son x'_0, x''_0, donde x''_0 no se concatena, sino que se circula a través de la capa densa x''_0, mientras que x'_0 se concatena a la salida de la capa densa x''_0 (figura 4.14). Esto lo podemos ver en la ecuación 4.14.

$$\begin{aligned} x_n &= w_n * (x_{n-1}, \ldots, x_2, x_1, x''_0) \\ x_o &= w_o * (x_n, \ldots, x_2, x_1, x''_0) \\ x_p &= w_p * (x_o, x'_0) \end{aligned} \tag{4.14}$$

CSP tiene las ventajas de que se conservan las características de los mapas obtenidos para hacer un reenvió de la información más eficiente, hace que la red pueda reutilizar dichas características y reduce la cantidad de parámetros necesarios para la red neuronal.

4.6.3. Neck

Neck es el lugar donde se lleva a cabo la agregación de características a la red neuronal. Su trabajo es hacer la recopilación de mapas de características que se fueron

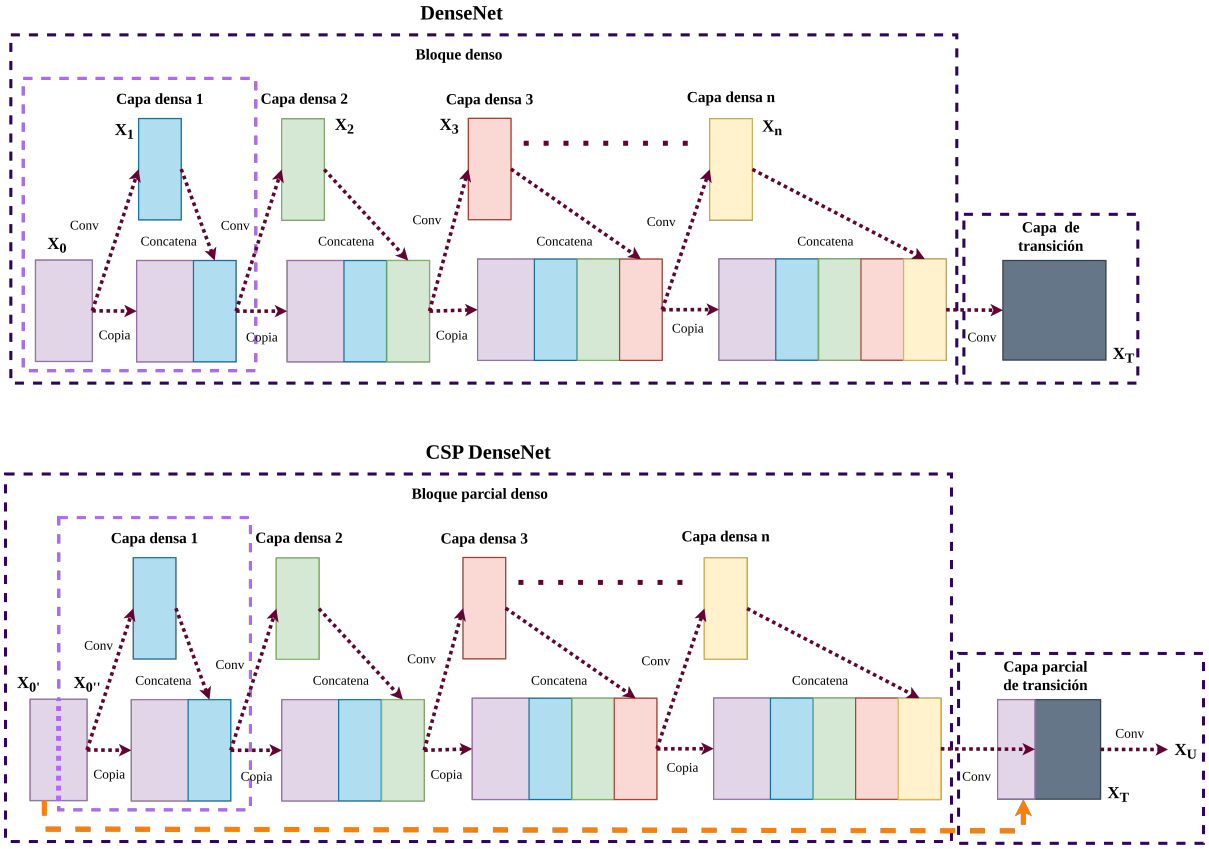

Figura 4.14: Arquitectura de una *DenseNet* vs *CSP*.

generando en las capas convolucionales del *Backbone*. Luego estos mapas son combinados para que sean utilizados por la red consultando las características que se obtuvieron. Es por esto que los flujos de información van en dirección de abajo hacia arriba y/o arriba hacia abajo. *Neck* considera los siguientes elementos dentro de su estructura:

- **SPP:** Es un bloque adicional que se conecta entre las capas finales de las capas densamente convolucionales de la *Backbone CSPDarknet53* y la **PatNet** de lo que se encarga es de aumentar el campo receptivo de la red y de ir separando las características más importantes. Los mapas de características que provienen de las salidas de la convolución son filtradas para detectar formas geométricas circulares. Estas características pueden generar otro mapa, el cual las resalta y manteniendo su ubicación dentro de la imagen.

  Dicho proceso se muestra en la figura 4.15, en el cual vemos un ejemplo de un **SPP** en 3 niveles y en su ultima capa de convolución se encuentra un mapa con 256 características, entonces se procede a realizar los siguientes pasos:

  1. Cada mapa de características es agrupado hasta ser un valor único, resultan-

do en un vector de tamaño 1,256 (la parte de color gris en la figura 4.15).

2. Luego, cada mapa vuelve a ser agrupado para generar 4 valores, resultando en otro vector de tamaño 4,256 (la parte de color verde en la figura 4.15).
3. Después, se vuelven a agrupar los mapas hasta generar 16 valores, resultando en otro vector de tamaño 16,256 (la parte de color azul en la figura 4.15).
4. Por último, los tres vectores que fueron generados son concatenados en un solo vector de tamaño fijo, el cual es la entrada a una red totalmente conectada.

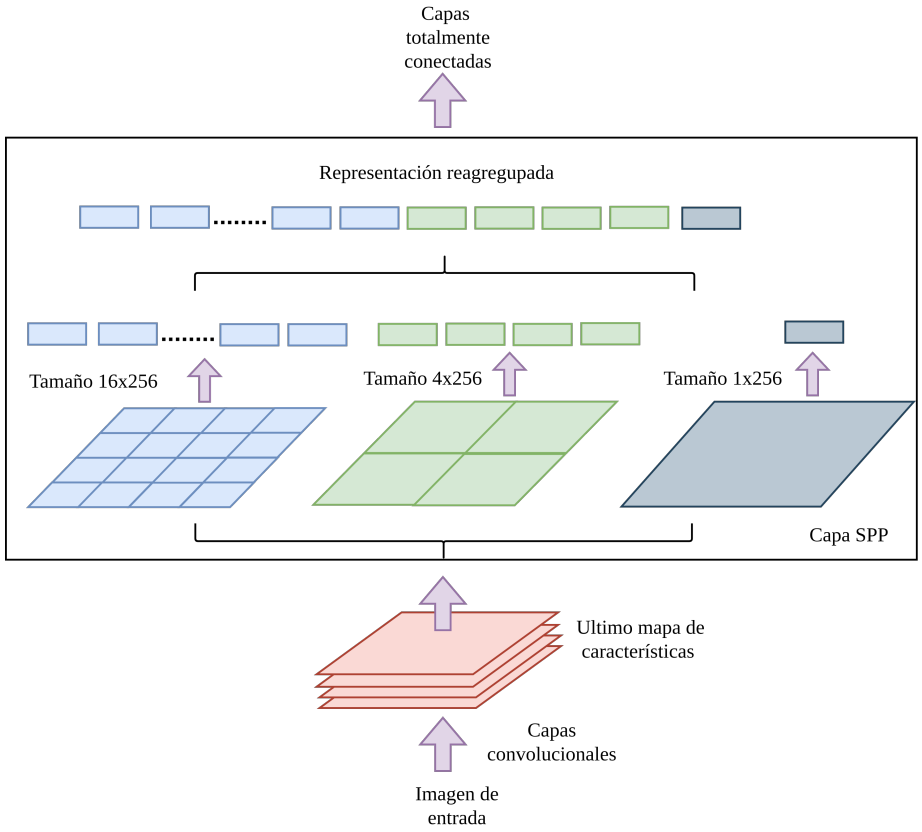

Figura 4.15: Arquitectura de *SPP*.

- **PatNet:** Su función principal es mejorar el proceso de segmentación de las instancias mantiene la información de su distribución espacial, lo cual hace que la localización de los píxeles sea la adecuada cuando se estime su posición por medio de predicciones.

  Sus propiedades más importantes al momento de predecir máscaras de manera precisa son: *Fully-Connected Fusion, Bottom-up Path Augmentation y Adaptive Feature Pooling*. Un **PatNet** modificado que utiliza *Adaptive Feature Pooling*

puede ir concatenando varias capas vecinas en vez de agregarlas, como se muestra en la figura 4.16.

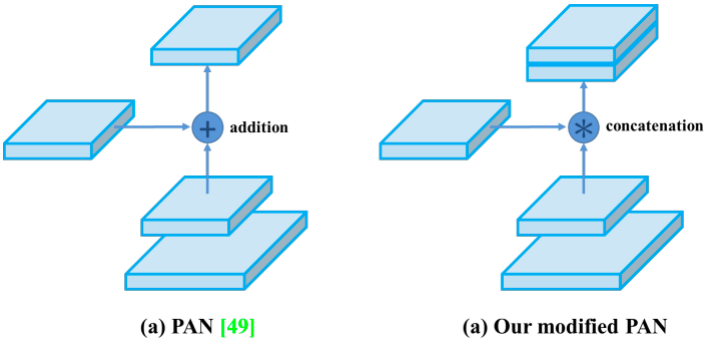

Figura 4.16: Arquitectura de *PatNet*. Tomado de (1)

4.6.4. Head

En esta última etapa es donde se hace la detección, *Head* tiene como función principal el crear y localizar el bounding box delimitando nuestro objeto a detectar, así como clasificarlo. *Head* busca hacer una predicción densa en la imagen, es decir, predecir en forma vectorial las coordenadas del bounding box (alto, ancho, centro), la precisión de la predicción (que tan acertado fue la detección) y la etiqueta con la clase a la que pertenece (como se muestra en la figura 4.17).

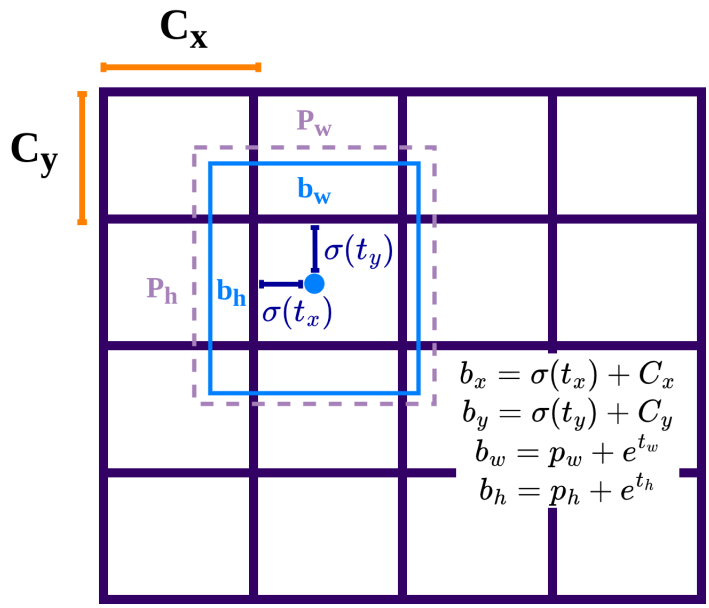

Figura 4.17: Predicción de un Bounding Box en la etapa *Head*. Tomado de (23)

Capítulo 5

Metodología propuesta

En este capítulo se detalla el objetivo principal de la tesis, que consiste en hacer localización de un robot móvil con filtro de Kalman extendido y detectando marcas naturales implementando YOLOV4 y Darknet. Estos dos sistemas están conectados por medio de nodos desarrollados en ROS, en los cuales el modulo de YOLO publica la información del objeto que detectó (la clase y la distancia) y el modulo del filtro de Kalman recibe la información para hacer la actualización de la pose del robot.

5.1. Herramientas

Para realizar la tarea de localización, vamos a implementar tanto un robot móvil en el cual vamos a ver en funcionamiento el algoritmo de EKF y una computadora, en la cual se ejecute tanto la detección de objetos como el EKF. Para ello vamos a utilizar como robot móvil a un Robot del laboratorio de bio-robotica llamado Robotino 3 y como computadora para las pruebas de algoritmos la Jetson TX2 Developer Kit de la marca NVIDIA.

5.1.1. Robotino 3

Robotino es un dispositivo basado en un robot creado por la marca FESTO (figura 5.1), el cual está dirigido para el desarrollo de software, en el nivel educativo. Este robot cuenta con diferentes componentes que hacen que se puedan probar algoritmos de navegación; y su construcción se presta para agregar componentes para realizar más tareas. Los componentes base con los que cuenta son:

- Una base que le permite realizar movimientos omnidireccionales que hacen que el robot se pueda desplazar de manera autónoma o de manera remota. Esta base esta constituida por un conjunto de tres motores con cigüeñal y sus respectiva ruedas omnidireccionales (figura 5.2).

Figura 5.1: Robotino 3 de FESTO.

- Una computadora embebida que puede gestionar componentes y/o sensores ajenos al robot. La computadora tiene un sistema operativo Ubuntu 16.04 LTS, lo que permite utilizar ROS versión kinetic para crear nodos para cada uno de los componentes que tenga el robot.

Los componentes externos agregados que maneja Robotino son:

- Una cámara RGB-D Kinect para que nos dé la información del ambiente. En este caso se usa un Kinect. Lo utilizamos para la detección de las marcas naturales en el ambiente, siendo al entrada del detector de YOLO.
- Un sensor de distancia láser de marca Hokuyo, el cual mide la distancia que existe entre el robot y los objetos que se encuentre al momento de navegar (paredes y/o obstáculos.)
- Un manipulador de la Marca FESTO, el cual se utiliza para tomar objetos.

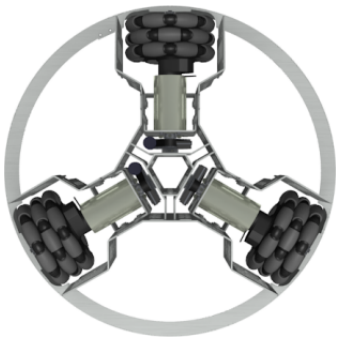

Figura 5.2: Estructura de la base omnidireccional del Robotino 3.

5.1.2. Jetson TX2 Developer Kit

La Jetson TX2 Developer Kit es una placa construida por NVIDIA (figura 5.3), en la cual se puede desarrollar hardware y software de manera sencilla. Una de las ventajas de esta placa es que tiene embebido un sistema llamado Jetson TX2 AI, el cual se presta para desarrollo de algoritmos implementando técnicas de inteligencia artificial. Los componentes que tiene la Jeston TX2 para su funcionamiento son (33):

- Una GPU Pascal™ de 256 núcleos
- Un CPU Denver 2 de doble núcleo de 64 bits + ARM® Cortex®-A57 MPCore de cuatro núcleos.
- Una memoria de 8 GB LPDDR4 de 128 bits.
- Un almacenamiento de 32 GB eMMC 5.1.
- Una interfaz de red dada por:
  - Ethernet: 10/100/1000BASE-T auto-negotiation.
  - Wireless: 802.11ac WIFI + Bluetooth.
- Una cámara de 12 lanes MIPI CSI-2, D-PHY 1.2 (30 Gbps).
- Una interfaz USB USB 3.0 + USB 2.0 (Micro USB).
- Una interfaz de pines para desarrolo, de los cuales son GPIO, I2C, I2S, SPI y UART.
- Una alimentación de 19V.

La placa utiliza un sistema operativo Ubuntu 18.04 LTS adecuado por NVIDIA (NVIDIA Jetpack) el cual es un SDK (paquete de desarrollo de software por sus siglas

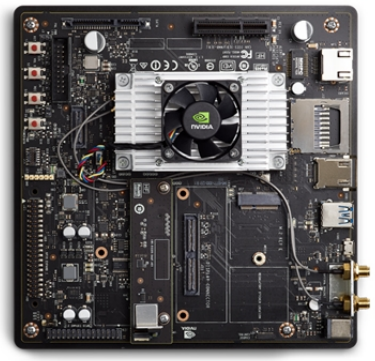

Figura 5.3: Jetson TX2 Developer Kit de NVIDIA.

en inglés) que contiene software completo para el manejo de bibliotecas de software BSP, CUDA, cuDNN y TensorRT dirigidas a aprendizaje profundo, visión artificial, computación GPU y/o procesamiento multimedia.

5.1.3. Software RViz

Como se mencionó en el Capitulo 2, el uso de ROS facilita el control por software del robot para el desarrollo e implementación de algoritmos en robots. ROS tiene una interfaz en la cual podemos simular las acciones que hace un robot en la vida real, llamado RViz (ROS Visualization por sus siglas en inglés).

En RViz podemos recrear por simulación la estructura del robot, y sobretodo los sensores, lo cual nos ayuda mucho para determinar la posición del robot para realizar los cálculos para la localización. Los elementos virtuales que son el robot, el Kinect, el mapa por donde navega y la odometría, tienen su sistema de referencia propio (como se muestra en la figura 5.4).

Estos sistemas de referencia tienen una jerarquía tomando como una referencia principal el mapa por donde navega, y al ser diferentes entre sí, se necesita calcular transformaciones para obtener información dependiendo del punto de referencia. Estos sistemas de referencia, como cualquier otro, están compuestos por ejes y de un origen que surge de la intersección de los mismos.

Por ejemplo, la referencia de la base del robot está relacionada con la referencia del Kinect y él a su vez con el del sensor láser, como se muestra en la figura 5.4; cuando el robot hace un giro sobre su propio eje, modifica la referencia de los sensores.

Otro ejemplo es cuando el sensor Kinect obtiene las coordenadas (x_0, y_0, z_0) de la marca natural respecto al sistema de referencia del Kinect, y se quiere determinar la distancia que existe entre el origen del robot a la marca, se aplica una transformación (en este caso una suma) la cual agrega las coordenadas al origen del robot (x_c, y_c, z_c); resultando en la posición de la marca natural respecto al origen del robot con coordena-

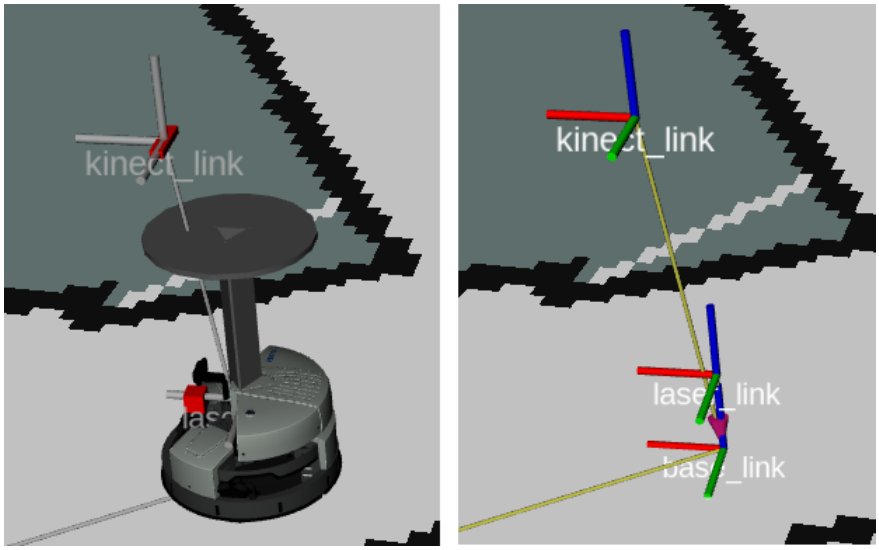

Figura 5.4: Visualización de las transformaciones del robot en RViz.

das $(x_0 + x_c, y_0 + y_c, z_0 + z_c)$. También estas transformaciones se aplican a la localización del robot dentro de un mapa para saber a que distancia se encuentra del origen del mapa al origen del robot.

5.2. Implementación de redes neuronales convolucionales enfocadas a imágenes

La implementación de la localización utilizando el algoritmo EKF utiliza marcas para hacer una estimación de la posición dentro de un área. En este caso, el proyecto de tesis busca reemplazar los códigos ArUco que se usaron como puntos de referencia en la tesis de Diego Cordero (3) y en su lugar utilizar marcas naturales que sirvan de guía para la localización del robot.

Una de las soluciones que se propone para resolver el problema de detectar marcas naturales, es implementando un sistema ocupando una red neuronal convolucional, la cual consiste en tomar la escena a partir del sensor Kinect del robot como entrada y detectar estas marcas naturales. Una vez identificadas las marcas naturales, el Kinect por medio de su sensor de distancia, pase la distancia a la que se encuentra el robot y el EKF puede actualizar la posición. Para ello se debe proveer un conjunto de datos de las marcas naturales para su entrenamiento en YOLO, y posteriormente transferir la información al detector para reconocer las marcas.

5.2.1. Creación del conjunto de datos para el entrenamiento

Para crear nuestro conjunto de datos a detectar con YOLO tenemos que considerar varios aspectos que son importantes. El primero de ellos es la selección de marcas naturales dentro del ambiente por el cual el robot va a estarse localizando, que en este caso es el laboratorio de bio-robótica que se encuentra en la unidad de posgrado de la Facultad de Ingeniería en la UNAM.

Para cuestiones practicas de la tesis, estas marcas naturales deben ser objetos fijos dentro del laboratorio. El motivo de esta decisión es porque se busca replicar la forma en la que nos ubicamos las personas por medio de la vista, observando referencias, como por ejemplo señalizaciones, letreros o edificaciones. Las figuras A.1, A.2, A.3, A.4, A.5, A.6, A.7 y A.8 del Apéndice A se muestran los objetos seleccionados para las pruebas de localización.

Una vez teniendo nuestras marcas naturales a detectar, tenemos que crear un conjunto de muestras por cada una de ellas. Para ello, se grabó a cada una de ellas desde diferentes perspectivas en las que el robot las puede observar desde el sensor Kinect que tiene colocado. Estas grabaciones se realizaron de diferentes tomas para tener muchas muestras mejorando entrenamiento dentro de la red neuronal.

Posteriormente, cada una de las grabaciones fueron convertidas en imágenes por medio del algoritmo 4. Este algoritmo lo que hace es acceder a la carpeta de la marca natural, tomar el vídeo que se encuentra guardado e ir separando el vídeo cuadro por cuadro, quedando la perspectiva del objeto en diferentes imágenes. Luego cada imagen se va guardando en un directorio dentro de la carpeta de la marca natural asociándole un nombre cuyo formato es el siguiente: *ruta del directorio/Images/Nombre_de_la_marca_natural + número_de_frame_generado + .jpg*

Una vez teniendo las imágenes de las perspectivas de nuestra marca natural, estas pasaron por un proceso de etiquetado el cual consiste en tomar un cuadro del vídeo y encerrarla con un *Bounding box*. Para ello existen diferentes software dedicados al etiquetado de imágenes y preparan a las mismas en el formato en el que YOLO las acepte como entradas a la red.

El software utilizado para este proyecto de tesis es YOLO_MARK el cual etiqueta imágenes conforme las clases a reconocer. La forma en que etiqueta YOLO_MARK es conforme al algoritmo 4, por medio de una interfaz gráfica en la cual se puede indicar que clase estamos etiquetando en la imagen y a su vez creando un documento en formato *.txt* el cual contiene las coordenadas de *Bounding box* sobre la marca natural a entrenar en el siguiente formato: *numero_de_la_clase coordenadas(x,y)_del_primer_pixel_de_la_marca coordenadas(x,y)_del_ultimo_pixel_de_la_marca.*

Estas coordenadas las obtiene conforme a la figura 5.5, donde la imagen toma en cuenta la posición $(1, 1)$ como inicio y (m, n) como el fin de la imagen, siendo m, n el tamaño del renglon-columna de la matriz M. También se puede manejar el punto inicial como (x_i, y_i) y el punto final (x_f, y_f) del bounding box.

Algorithm 4 Pseudocódigo CreateBoundingBox

procedure CREATEBOUNDINGBOX(*Name_Image*)
id_image = *set_ID_Class()*

(x_i, y_i) = *locate_init_Point(Name_Image)*

(x_f, y_f) = *locate_final_Point(Name_Image)*

ClassImage = *create_file_class(id_image,* x_i, y_i, x_f, y_f*)*
return ClassImage
end procedure

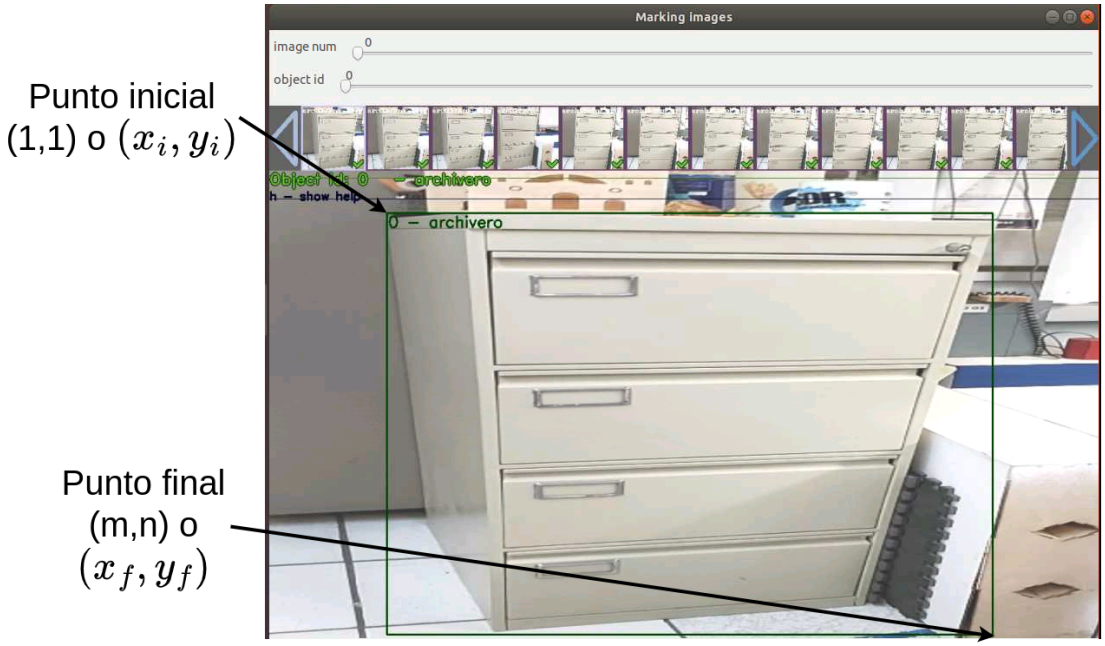

Figura 5.5: Software de etiquetado de imágenes YOLO Mark.

5.2.2. DarkNet-ROS

Como ya hemos mencionado antes, ROS permite crear módulos para desarrollar algoritmos que pueden implementar los robots móviles, y en este caso existe un modulo que maneja YOLO para su uso. DarkNet-ROS es un módulo creado para la detección de objetos a partir de una arquitectura basada en YOLO y sus diferentes versiones cargando la arquitectura con la que se entrenó y los pesos que fueron generados durante el mismo. Este módulo utiliza varios archivos de configuración para hacer la detección de las marcas, los cuales son:

- *yolov4.weights:* En este archivo se encuentran los pesos generados una vez acabado el entrenamiento con las marcas naturales del modelo a detectar.
- *yolov4.cfg:* Este archivo contiene la arquitectura neuronal de YOLOV4 con la que fue entrenado el modelo con las marcas naturales.
- *yolov4.yaml:* En este archivo se tienen los nombres de las clases que fueron entrenados a partir de las marcas naturales.

El nodo de DarkNet funciona por medio de tópicos, de los cuales recibe una imagen que proviene de otro tópico que maneja el Kinect para obtener imágenes a tiempo real de las marcas naturales. Este a su vez devuelve un tópico con la detección de la marca (Clase del objeto, id de la clase, coordenadas de bounding box $(x_i, y_i), (x_f, y_f)$, distancias respecto al Kinect en (x, y, z), la probabilidad de reconocimiento del objeto.) como se ve en la figura 5.6.

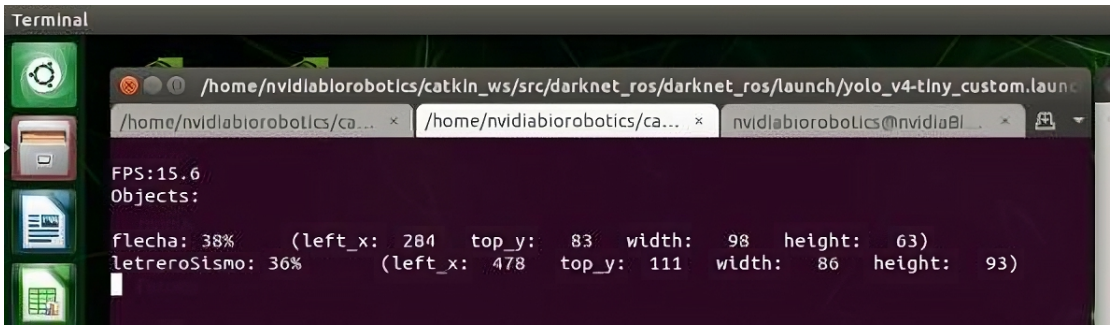

Figura 5.6: Detección de imágenes utilizando DarkNet-ROS.

5.2.3. Entrenamiento de los datos utilizando YOLOV4 y DarkNet

Conforme al funcionamiento de YOLO y DarkNet, el entrenamiento de las marcas naturales fue realizado usando la transferencia de conocimiento sobre el modelo neuronal de YOLOV4 y un conjunto preentrenado COCO (14) para crear al nuevo modelo reentrenado. Este nuevo modelo necesita de los siguientes elementos para su reentrenamiento:

- El conjunto de datos que fue etiquetado previamente.
- Un archivo el cual contiene los pesos preentrenados de COCO (yolov4.conv.137).
- Dos archivos de texto, uno de ellos llamado *train.txt* donde está un conjunto de datos para entrenar, y *test.txt* donde está un conjunto de prueba para entrenar. Cabe aclarar que la separación del conjunto de datos fue 70 % para train y 30 % para test.

- Configurar los archivos *yolov4.cfg*, *yolov4.names* y *yolov4.data* para la arquitectura de YOLOV4.

Para la configuración de los archivos del último punto, se editó cada archivo de la siguiente forma para el reentrenamiento en YOLOV4:

- **Configuración de yolov4.cfg:** Dentro de este archivo se tiene la configuración de la arquitectura neuronal de YOLOV4, las cuales son las capas convolucionales, las capas de reducción, capas residuales y las capas para clasificar. De igual manera en este archivo se encuentran los parámetros que se utilizan para realizar un reentrenamiento, que son:
  - *Batch:* Es el tamaño de lote de entrenamientos que se necesitan para que se actualicen los parámetros internos dentro de la red neuronal.
  - *Subdivisiones:* Es el número de subdivisiones que utilizará el batch. Estas subdivisiones se implementan para gestionar los recursos físicos de una GPU.
  - *Height y width:* Son los parámetros de largo y ancho de las imágenes de entrada.
  - *Channels:* Es el número de canales que contiene la imagen de entrada.
  - *Learning rate:* Es la tasa de aprendizaje que usará la red neuronal.
  - *Max batches:* Es el número máximo de lotes que puede procesar la red neuronal cuando se encuentre haciendo entrenamiento.
- **Configuración de yolov4.names:** Dentro de este archivo se encuentran los nombres de las clases de los objetos a entrenar por la red neuronal.
- **Configuración de yolov4.data:** Dentro de este archivo se encuentran los siguientes parámetro a modificar para un reentrenamiento:
  - *Classes:* Es el número de clases que están en nuestro conjunto de datos.
  - *Train:* Es la ruta del directorio donde está guardado el archivo de train.txt.
  - *Test:* Es la ruta del directorio donde está guardado el archivo de test.txt.
  - *Names:* Es la ruta del directorio donde está guardado el archivo de yolov4.names.
  - *Backup:* Es la ruta del directorio donde se guardan las las copias de los pesos generados.

Posteriormente configurados los parámetros para hacer reentrenamiento, podemos empezar a correr el entrenamiento con darknet usando el comando *./darknet detector train data/obj.data cfg/yolov4-custom.cfg yolov4.conv.137 -dont_show -map* dentro de una terminal. Para este caso, el entrenamiento termina cuando se cumple con el valor máximo de max batches o cuando el promedio de error es muy pequeño (figura 5.7).

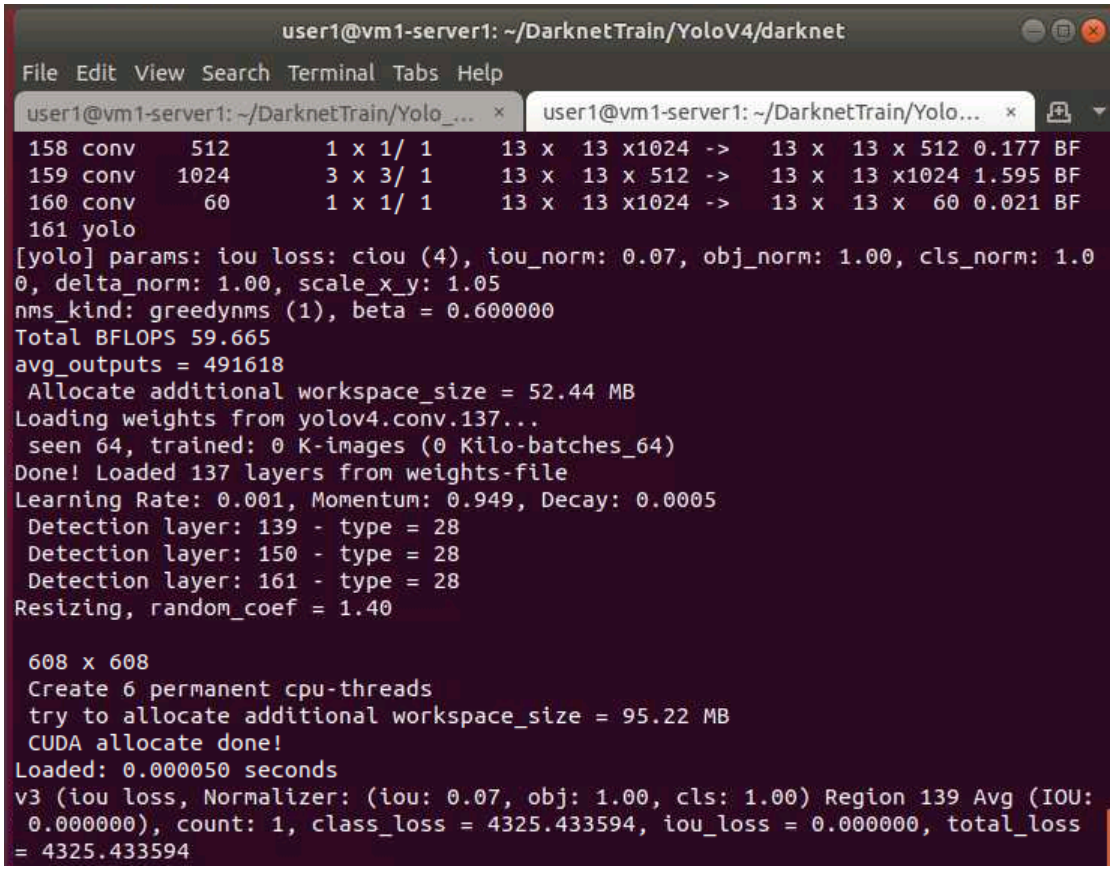

Figura 5.7: Entrenamiento de imágenes utilizando DarkNet-ROS.

5.2.4. Detección de los objetos entrenados utilizando YOLOV4 y Darknet

Una vez teniendo nuestros pesos entrenados a partir de las coordenadas de los bounding boxes, estos fueron cargados al nodo de detección basado en DarkNet-ROS para hacer reconocimiento de nuestras marcas naturales. Este nodo de detección esta basado en el algoritmo 5, el cual lo que hace es tomar una imagen de nuestro entorno por el cual Robotino navegó buscando las marcas naturales. Para ello primero lo que hizo fue buscar en un área de la imagen los puntos (x_i, y_i) y (x_f, y_f) por medio de un rango de valores dado por la red neuronal, los cuales le indican el lugar probable donde esté la marca.

Una vez delimitada el área de búsqueda, la red neuronal extrajo las características dentro del bounding box buscando que coincidan con las que fueron entrenadas para poder clasificar en caso de encontrar un resultado positivo. Por métodos probabilísticos la red neuronal trató de clasificar la imagen de entrada. Por ultimo, la red neuronal devolvió como resultado el nombre de la clase a la cual se asemejó la imagen, así como

la probabilidad de acierto de pertenecer a la clase.

Algorithm 5 Pseudocódigo DetectLandmark

procedure DETECTLANDMARK(*Image*)
area_search = *found_ BoundingBox(Image)*

ID_{class}, P_{class} = *found_ Class(area_ search)*

return ID_{class}, P_{class}
end procedure

Este proceso lo podemos ejemplificar con el diagrama de la figura 5.8, en donde vemos que el nodo del Kinect se ejecuta en tiempo real obteniendo los frames en búsqueda de marcas. Estos frames por medio de tópicos son recibidos por el nodo detector de DarkNet-ROS para reconocer, y si encuentra uno muestra la clase que detectó así como que tanto se parecen a los modelos entrenados.

5.2.5. Implementación de la detección de objetos en la Jetson TX2 Developer Kit

Como se mencionó previamente, esta implementación desarrollada en Linux utilizó un GPU puesto que DarkNet-ROS utiliza CUDA para realizar la red neuronal tanto para entrenar, como para detectar. Siendo este un modulo desarrollado en ROS, puede ser utilizado en otro dispositivo con Linux siempre y cuando contenga CUDA para usarse.

En este caso se utilizó la Jetson TX2 Developer Kit, dado que su arquitectura interna se maneja con CUDA cores y con un sistema basado en Linux, lo que lo hace compatible con este nodo. La tarjeta de desarrollo Jetson TX2 cuenta con un sistema operativo Ubuntu 18.04, así como una versión de ROS Melodic. Teniendo en cuenta esto, los nodos de DarkNet-ROS y el EKF se encuentran en el mismo directorio de trabajo de ROS, y a su vez cuando son ejecutados, estos se comunican por medio de un ROS-Master.

5.3. Integración de DarkNet-ROS con el sistema de localización con el filtro de Kalman extendido

El algoritmo de EKF que se implementó en el trabajo previo fue adaptado anteriormente con códigos ArUco para hacer la detección de marcas, pero para cuestiones de este trabajo se cambio por la detección de objetos por medio de DarkNet-ROS y YOLOV4, retornando las coordenadas del objeto para realizar la etapa de Actualización

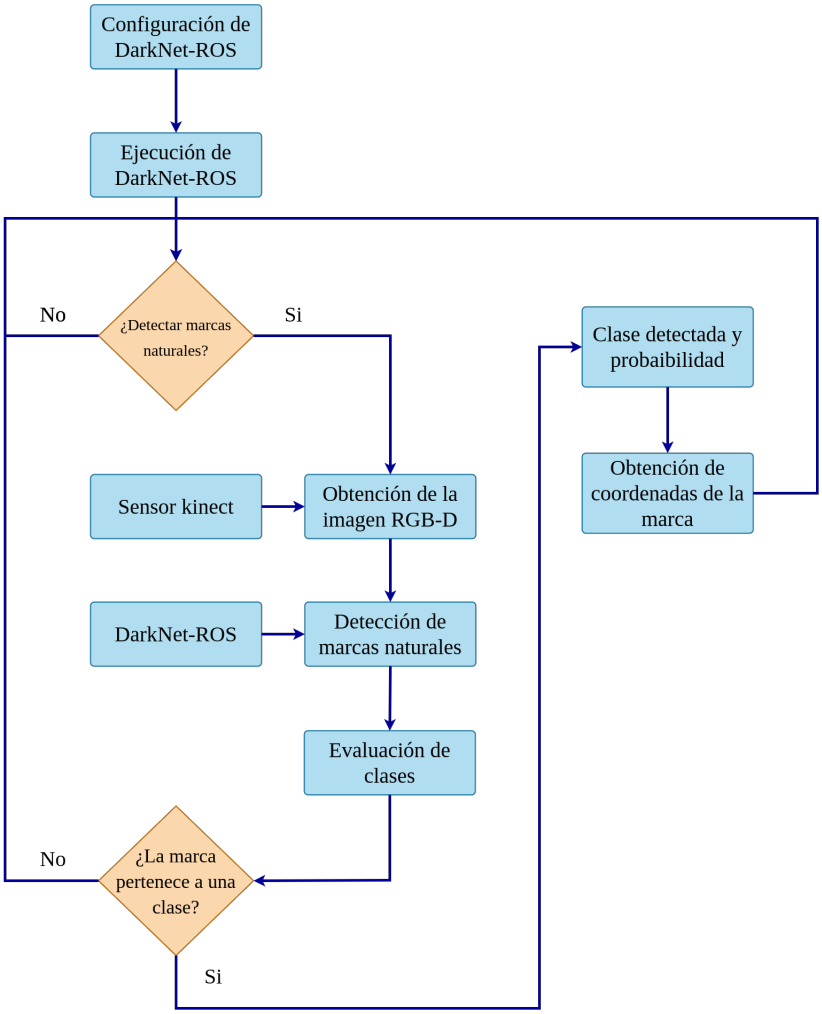

Figura 5.8: Diagrama de detección de marcas utilizando DarkNet-ROS.

del algoritmo.

Para ello, los cambios que se realizaron en este proyecto son almacenar las marcas naturales dentro de un archivo para que posteriormente el algoritmo EKF las consulte para tener la referencia teórica de las marcas y compararlas con las que está detectando el sensor Kinect del Robotino.

5.3.1. Uso del filtro de Kalman para localización

Con base a los capítulos anteriores y a la Tesis de Diego Cordero (3) el algoritmo EKF que se propuso para este sistema fue el que se menciona en el algoritmo 6. Cabe mencionar que para obtener los valores del vector $\vec{x}$ se calcularon utilizando una función llamada *muestreoOdometria()* (visto en el algoritmo 7), el cual lo que hizo es muestrear

los movimientos del robot en instrucciones de avance y giro visto en el capitulo 3.

Algorithm 6 Pseudocódigo de la localización con EKF del sistema

procedure EKF ALGORITHM($\vec{x}_{k-1}, \Sigma_{k-1}, \vec{u}, S$)

Predicción ▷ Calculamos x_k, y_k y θ_k de una iteración anterior.

$\vec{x} = muestreoOdometria(u_k, x_{k-1})$

▷ Se calcula la odometria por medio de la función *muestreoOdometria()*

$$G = \begin{bmatrix} 1 & 0 & -u_d * sin(\theta_k) \\ 0 & 1 & u_d * cos(\theta_k) \\ 0 & 0 & 1 \end{bmatrix}$$

$\Sigma_k = G\Sigma_{k-1}G^T + Q$

Actualización

for `(i=0;i<n;i++)` **do**

$$\widehat{\vec{z}} = \begin{bmatrix} \sqrt{(S_{i,x} - x_k)^2 + (S_{i,y} - y_k)^2} \\ \text{atan2}\,(S_{i,y} - y_k, S_{i,x} - x_k) - \theta_k \\ S_{i,m} \end{bmatrix}$$

$$z = \begin{bmatrix} S_{i,r} \\ S_{i,\phi} \\ S_{i,m} \end{bmatrix}$$

$$H = \begin{bmatrix} \frac{-(S_{i,x}-x_k)}{\sqrt{(S_{i,x}-x_k)^2+(S_{i,y}-y_k)^2}} & \frac{-(S_{i,y}-y_k)}{\sqrt{(S_{i,x}-x_k)^2+(S_{i,y}-y_k)^2}} & 0 \\ \frac{(S_{i,y}-y_k)}{\sqrt{(S_{i,x}-x_k)^2+(S_{i,y}-y_k)^2}} & \frac{-(S_{i,x}-x_k)}{\sqrt{(S_{i,x}-x_k)^2+(S_{i,y}-y_k)^2}} & -1 \\ 0 & 0 & 0 \end{bmatrix}$$

$I = (H * \Sigma_k * H^T) + R$

$K = \Sigma_k * H^T I^{-1}$ ▷ Ganancia de Kalman

$Z' = z - \widehat{z}$

$\vec{x}_k = \vec{x}_k + K * Z'$

$\Sigma_k = (M_I - K * H) * \Sigma_k$

▷ M_I es la matriz identidad

end for

return $(\vec{x}_k, \Sigma_k)$

end procedure

En la figura 5.9 podemos ver mejor la implementación que se hizo del algoritmo tal considerando la actualización por cada marca detectada. En dado caso que el detector de DarkNet-ROS no le devuelva alguna marca recibida por el Kinect (haciendo que no se tengan condiciones para actualizar), el EKF se mantiene en la etapa de predicción.

Algorithm 7 Pseudocódigo MuestreoOdometría. Tomado de (3)

procedure MUESTREOODOMETRIA($\vec{u}_t, \vec{x}_{t-1}$)

$\delta_{rot1} = \text{atan2}\,\bar{y}' - \bar{y}, \bar{x}' - \bar{x} - \bar{\theta}$

$\delta_{tras} = \sqrt{(\bar{x} - \bar{x}')^2 + (\bar{y} - \bar{y}')^2}$

$\delta_{rot2} = \theta' - \theta - \delta_{rot1}$

$\widehat{\delta}_{rot1} = \delta_{rot1} - sample(\alpha_1\delta^2_{rot1} + \alpha_2\delta^2_{tras})$

$\widehat{\delta}_{tras} = \delta_{tras} - sample(\alpha_3\delta^2_{tras} + \alpha_4\delta^2_{rot1} + \alpha_4\delta^2_{rot2})$

$\widehat{\delta}_{rot2} = \delta_{rot2} - sample(\alpha_1\delta^2_{rot2} + \alpha_2\delta^2_{tras})$

$x' = x + \widehat{\delta}_{tras} cos(\theta + \widehat{\delta}_{rot1})$

$y' = y + \widehat{\delta}_{tras} sin(\theta + \widehat{\delta}_{rot1})$

$\theta' = \theta + \widehat{\delta}_{rot1} + \widehat{\delta}_{rot2}$

return $x_t = (x', y', \theta')^T$

end procedure

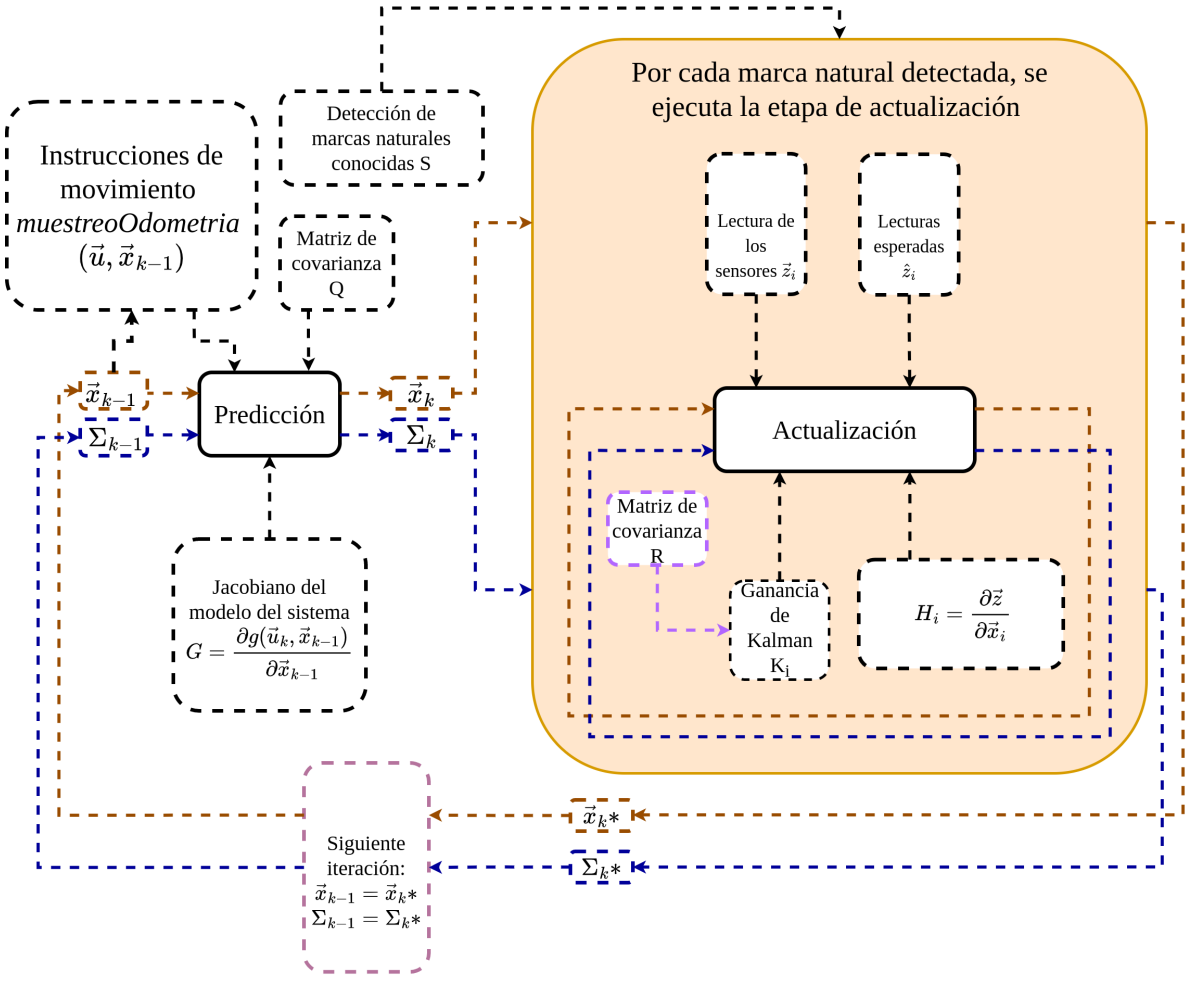

Figura 5.9: Diagrama de flujo del EKF del sistema.

5.3.2. Obtención de distancias entre las marcas naturales y el robot

Cuando se calculan los parámetros del vector $\widehat{\vec{z}}$, son necesarias las coordenadas de las marcas naturales para poder hacer una estimación de distancias dentro del mapa y que el robot móvil pueda realizar la actualización en el EKF. Por lo cual se necesita que DarkNet-ROS le envíe al EKF las coordenadas de cada marca natural detectada.

Para ello, en el nodo de DarkNet-ROS se implementó una función, la cual calcula las coordenadas a partir de la imagen con los bounding boxes y de la imagen de profundidad tomadas al mismo tiempo. Las coordenadas (x, y, z) de las marcas naturales se obtienen considerando dos parámetros importantes, los cuales son:

- La imagen de profundidad que se obtiene del sensor Kinect, también llamado nube de puntos. Cabe mencionar que una nube de puntos es un conjunto de puntos en el espacio que tienen la característica de dar información respecto a las coordenadas (x, y, z) dentro de una imagen de profundidad (13).

  Estas se suelen estar asociadas a los píxeles de la imagen real y así obtener las coordenadas (x, y, z) de un objeto dentro en la imagen. Esto lo podemos ver en la figura 5.10.

- Los valores internos de la cámara son necesarios para hacer los cálculos de las coordenadas de cada punto que se muestre en la imagen de profundidad, ya que son parámetros que hacen la transformación del sensor de distancia del Kinect respecto a la imagen de profundidad. Estos valores son la distancia focal del lente del Kinect (f_x, f_y) y el centro de la imagen (c_x, c_y).

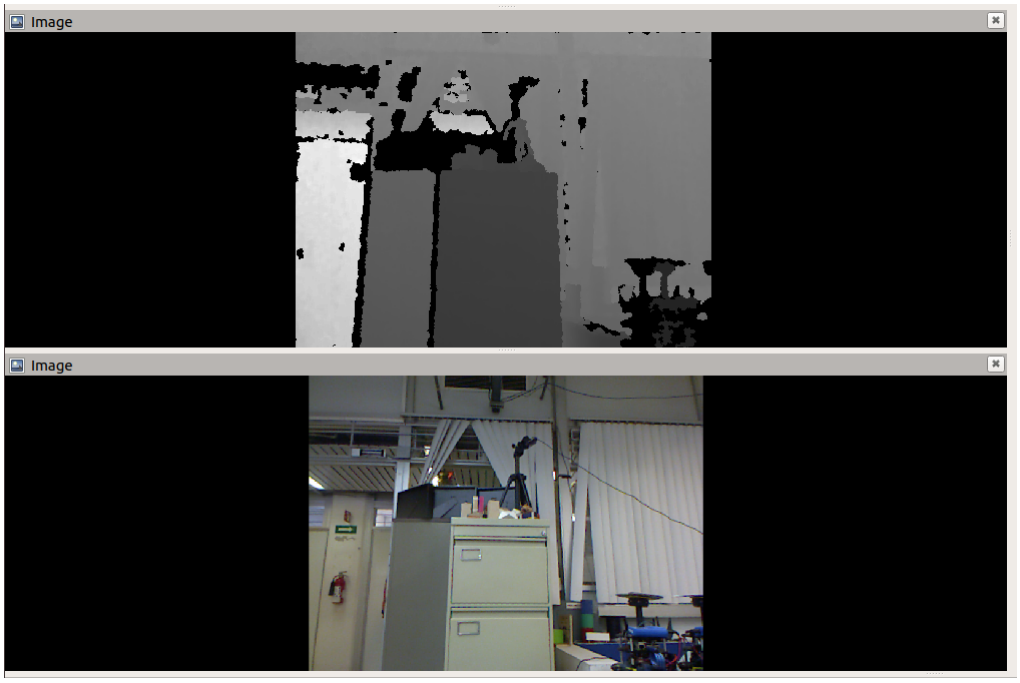

Figura 5.10: PointCloud de una escena con su imagen rgb vistas desde el sensor Kinect.

Tanto la imagen de color como la imagen de profundidad tienen que son obtenidas por el Kinect casi al mismo instante, ya que los puntos que queramos encontrar en la imagen de color, deben coincidir en la imagen de profundidad para obtener nuestras coordenadas.

Una vez considerados estos parámetros, fueron implementados en el algoritmo 8, el cual lo que haces, es obtener las coordenadas (x, y, z) respecto a los bounding boxes que detectó DarkNet-ROS. Este algoritmo lo que hace es utilizar el ID de la clase de la marca natural, así como el punto inicial (x_i, y_i) y el punto final (x_f, y_f) del bounding box con el propósito de calcular el centro de este, como se muestra en las ecuaciones 5.1 y 5.2.

Posteriormente el centro calculado lo vamos a tomar de referencia y lo vamos a buscar en nuestra imagen de profundidad. Teniendo el punto centro en la imagen de profundidad podemos saber cual es la distancia de ese punto al Kinect, lo cual nos da la coordenada z (figura 5.11).

Algorithm 8 Pseudocódigo obtain_Coordenates

procedure OBTAIN_COORDENATES($LandMark_{ID}$, x_{min}, y_{min}, x_{max}, y_{max})
$C_x = 320.5$
$C_y = 240.5$

$f_x = 554.25$
$f_y = 554.25$

$X_c = ((x_{max} - x_{min})/2) + xmin$
$Y_c = ((y_{max} - y_{min})/2) + y_{min}$

Z_{real} = depthDistance(X_c, Y_c)
$X_{real} = ((X_c - C_x) * Z_{real})/focal_x$
$Y_{real} = ((Y_c - C_y) * Z_{real})/focal_y$
return X_{real}, Y_{real}, Z_{real}
end procedure

$$X_c = ((x_{max} - x_{min})/2) + xmin \tag{5.1}$$

$$Y_c = ((y_{max} - y_{min})/2) + y_{min} \tag{5.2}$$

Por ultimo, una vez localizado z podemos encontrar las coordenadas que faltan obtener x,y con ayuda de la distancia focal y el centro de la imagen, siendo estas calculadas por las ecuaciones 5.3 y 5.4:

$$X_{real} = ((X_c - C_x) * Z_{real})/f_x \tag{5.3}$$

$$Y_{real} = ((Y_c - C_y) * Z_{real})/f_y \tag{5.4}$$

Donde:

- X_{real}, Y_{real} son las coordenadas de la marca natural.
- (C_x, C_y) son las coordenadas del centro de la imagen.
- (f_x, f_y) son las coordenadas de la distancia focal.
- (X_c, Y_c) son las coordenadas calculadas en las ecuaciones 5.1 y 5.2.

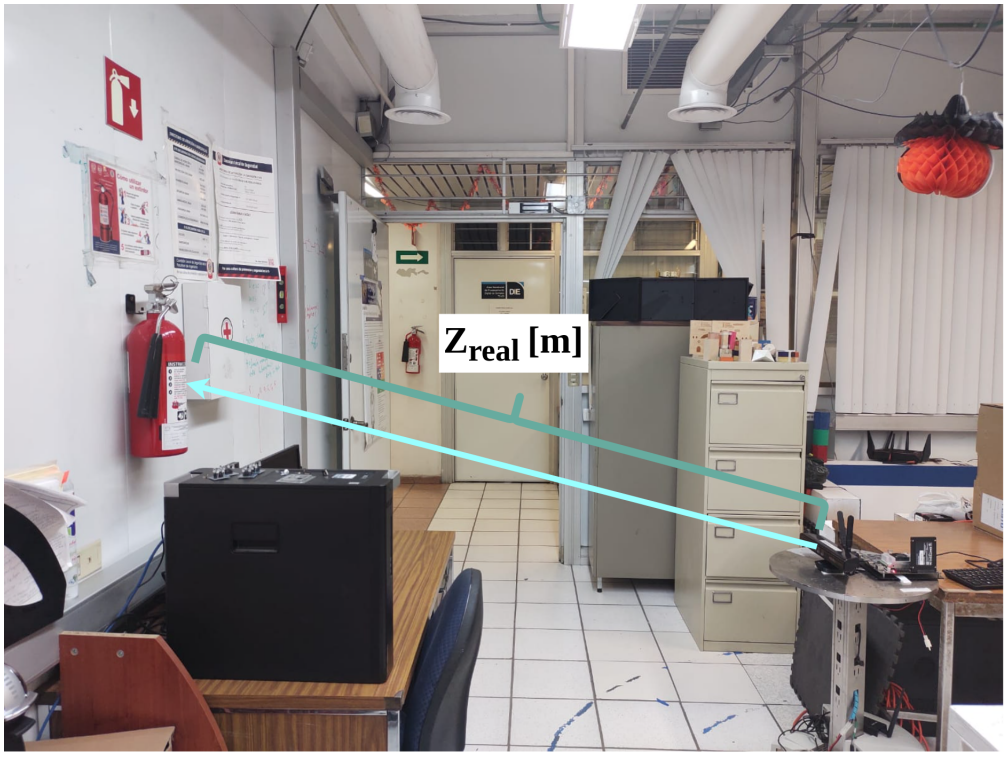

Figura 5.11: Representación de la distancia entre la marca y el Kinect (coordenada z).

Puesto que se implemento el sensor Kinect para este proyecto, las coordenadas (X_c, Y_c) las podemos sustituir por coordenadas de color (U, V) del sensor, haciendo que las ecuaciones 5.1 y 5.2 sean modificadas en las ecuaciones 5.5 y 5.6. De igual manera las ecuaciones 5.3 y 5.4 se ven modificadas, resultando en las ecuaciones 5.7 y 5.8 :

$$U = ((x_{max} - x_{min}) + x_{min}) \tag{5.5}$$

$$V = ((y_{max} - y_{min}) + y_{min}) \tag{5.6}$$

$$X_{real} = ((X_c - C_x) * Z_{real})/f_x \tag{5.7}$$

$$Y_{real} = ((Y_c - C_y) * Z_{real})/f_y \tag{5.8}$$

En donde:

- X_{real}, Y_{real} son las coordenadas de la marca natural.
- (C_x, C_y) son las coordenadas del centro de la imagen.
- (f_x, f_y) son las coordenadas de la distancia focal.
- (U, V) son las coordenadas calculadas en las ecuaciones 5.5 y 5.6.

5.3.3. Representación del ambiente

Como se mencionó en secciones anteriores, para que el robot pueda navegar, se requiere un mapa que le brinde información del área, por lo cual se requiere recrearlo por medio de los datos que le brindan los sensores. El sistema del trabajo anterior cuenta con un nodo el cual recreó el área por donde navega, lo que incluyó también la información de los códigos ArUco.

En este caso, el algoritmo 9 fue implementado para hacer un mapeo del entorno con la detección de marcas con DarkNet-ROS. Lo que hizo este algoritmo es que cada que se detectó una marca natural nueva a una cierta distancia del robot, se obtiene el id, nombre de su clase y la distancia de la marca respecto al origen del mapa y las coordenadas en (x, y, z). Si en dado caso detectó una marca previamente vista por el sensor, solo se actualizan las coordenadas.

Tenemos que considerar que estas coordenadas están calculadas con respecto al Kinect, por lo cual se realizó la transformación de referencias, cambiando como referencia del Kinect al del mapa. Esto se hace para que el robot tenga una referencia fija para localizarse, como lo es el mapa por donde navega el Robotino. Por ultimo, por medio de una bandera se indicó al algoritmo si guarda en un archivo *.txt* los datos para que el algoritmo EKF pueda consultarlos y compararlos.

Algorithm 9 Pseudocódigo MappingLandmarks

procedure MAPPINGLANDMARKS(*Image, max_distance*)
LandMark$_{ID}$, *LandMark*$_{Probability}$, x_{min}, y_{min}, x_{max}, y_{max} = *DetectLandmark(Image)*

$X_{mark}, Y_{mark}, Z_{mark}$ = *obtain_Coordenates(LandMark*$_{ID}$, x_{min}, y_{min}, x_{max}, y_{max}*)*

if $Z_{mark} > max\_distance$ **then**
$X_{LandmarkToMap}, Y_{LandmarkToMap}, Z_{LandmarkToMap}$ = *transform_Coordenates_ToMap*($X_{mark}, Y_{mark}, Z_{mark}$)
else

$X_{LandmarkToMap}, Y_{LandmarkToMap}, Z_{LandmarkToMap}$ = *transform_Coordenates_ToMap*($X_{mark}, Y_{mark}, Z_{mark}$)
filesLandmarks = *save_Landmark*(*LandMark*$_{ID}$, *LandMark*$_{Probability}$, $X_{LandmarkToMap}$, $Y_{LandmarkToMap}$, $Z_{LandmarkToMap}$)
end if
return filesLandmarks
end procedure

.

Capítulo 6

Análisis de Resultados

En este capítulo se muestran y detallan las pruebas que se realizaron en la detección de marcas naturales y de la localización con el EKF. Para ello vamos a utilizar los 15 objetos mencionados en las figuras A.1, A.2, A.3, A.4, A.5, A.6, A.7 y A.8, los cuales son las clases que fueron detectadas con DarkNet-ROS. Cabe mencionar que el área de pruebas fue el laboratorio de bio-robótica del posgrado en ingeniería.

6.1. Pruebas y resultados de detección con DarkNet-ROS

En este caso se realizaron pruebas de detección de las marcas naturales creando dos entrenamientos diferentes pero con el mismo conjunto de datos etiquetados para poder comparar desempeños. Estos entrenamientos fueron realizados con una red YOLOV4 y una red tiny-YOLOV4, la cual se trata de una red neuronal más rápida (al menos 442 % más rápido corriendo a 244 FPS). El tamaño de esta red tiny-YOLOV4 ronda de los 50 MB, lo que lo hace factible de utilizar en dispositivos como la Jetson TX2 Developer Kit. La desventaja de ejecutar un modelo tiny es que puede llegar a ser impreciso al momento de detectar.

6.1.1. Creación del conjunto de datos

Las marcas naturales que fueron seleccionadas para ser etiquetadas pasaron por el proceso de convertir vídeo a imágenes. La duración de todos los vídeos de las marcas naturales fueron de 45 segundos para cada uno, obteniendo un número de imágenes dado por la tabla 6.1, en donde vemos que las imágenes fueron en promedio de 1400 aproximadamente para cada clase. También se consideró que la creación de estos vídeos fueran tomados de diferentes perspectivas a un metro y a dos metros de distancia tratando de considerar todas las vistas posibles para alimentar a la red neuronal.

ID de la clase	Nombre de la clase	Número de tomas por clase
0	Archivero	1406
1	Botiquín	1398
2	Cámara	1368
3	Estante	1429
4	Extintor	1385
5	Flecha	1400
6	Impresora	1367
7	Letrero biorobótica	1400
8	Letrero extintor	1366
9	Letrero sismo	1440
10	Locker	1369
11	Organizador	1398
12	Poster	1400
13	Refrigerador	1368
14	Ventilador	1364

Tabla 6.1: Conjunto de imágenes para cada marca natural.

6.1.2. Entrenamiento del conjunto de imágenes

El entrenamiento del conjunto de datos descrito en el capítulo 5 fue realizado en un servidor con un sistema operativo Ubuntu 18.04 LTS y una tarjeta gráfica NVIDIA RTX 2080. En este caso se implementaron dos entrenamientos diferentes para ser utilizados en la tarjeta Jetson TX2 Developer Kit comparando rendimientos y escogiendo un modelo para hacer las pruebas de localización del Robotino. Estos modelos son un modelo normal de YOLOV4 y un modelo Tiny-YOLOV4.

Para cada uno de los modelos se implementaron parámetros de entrenamiento, los cuales se observan en la tabla 6.2. Para el tiempo de entrenamiento se consideró un tiempo de entrenamiento equivalente a un fin de semana (dos días), puesto que el conjunto de clases son 15 y el conjunto de imágenes siendo aproximadamente 1400 para cada clase.

Parámetro	YOLOv4	Tiny-YOLOV4
batch	64	64
subdivisions	32	32
width	416	416
height	416	416
channels	3	3
learning rate	0.001	0.00261
max batches	28000	28000
steps	22400, 25200	22400, 25200
filters	32	32
classes	15	15

Tabla 6.2: Configuración de una arquitectura YOLOV4 y una arquitectura Tiny-YOLOV4.

Para la evaluación en cada entrenamiento de los modelos fue realizado utilizando la métrica mAP (Mean Average Precision por sus siglas en inglés) que se muestra en la ecuación 6.1, la cual es utilizada por YOLO para evaluar la clasificación - detección de las marcas entrenadas por los modelos. En esta ecuación tenemos que: mAP representa el Mean Average Precision, n_{class} es el número de clases, TP son los positivos verdaderos, FP son los positivos falsos.

$$mAP = \frac{1}{|n_{class}|} \sum_{C \in n_{class}} \frac{n_{TP}(C)}{n_{TP}(C) + n_{FP}(C)} \tag{6.1}$$

Mientras el entrenamiento de ambos modelos neuronales se realizó durante un fin de semana, YOLO fue creando archivos de respaldo los cuales contienen los pesos entrenados de cada 10000 iteraciones. Cada uno de los archivos respaldados se evaluó con la métrica mAP. Estos resultados los vemos reflejados en la figura 6.1 de la arquitectura de YOLOV4 y en la figura 6.2 de la arquitectura Tiny-YOLOV4 donde vemos que se generaron archivos a las 10000, 20000 y 28000 iteraciones que fueron el máximo establecido para el entrenamiento.

Dentro de estas gráficas podemos observar dos sub-gráficas diferentes: la gráfica de mAP y la perdida del modelo entrenado, las cuales fueron resultado del entrenamiento considerando un 70 % del conjunto de datos como entrenamiento y un 30 % del conjunto para validación-Prueba del modelo. Para la figura 6.1, la métrica mAP, vemos que la gráfica converge a 80 % aproximadamente por la iteración número 1000 y llegando a un 99.7 % de mAP aproximadamente en la iteración 2000; mientras que la perdida se observa que se minimiza entre la iteración 1000 y 2000, reduciendo el error promedio a 0.1541. Esto significa que a partir de la iteración 2000, el modelo llegó a su máximo promedio evaluado del modelo entrenado (modelo YOLOV4).

Para la figura 6.2, la métrica mAP, vemos que la gráfica converge de 40 % a 94 % aproximadamente por la iteración número 1000, de 94 % a 98 % en la iteración 2800, para posteriormente llegue a un 99.6 % de mAP aproximadamente de la iteración 4000; mientras que la pérdida se observa que se minimiza entre la iteración 1000 y 2000, reduciendo el error promedio a 0.0178. Esto significa que a partir de la iteración 2000, el modelo llegó a su máximo promedio evaluado del modelo entrenado (modelo tiny-YOLOV4).

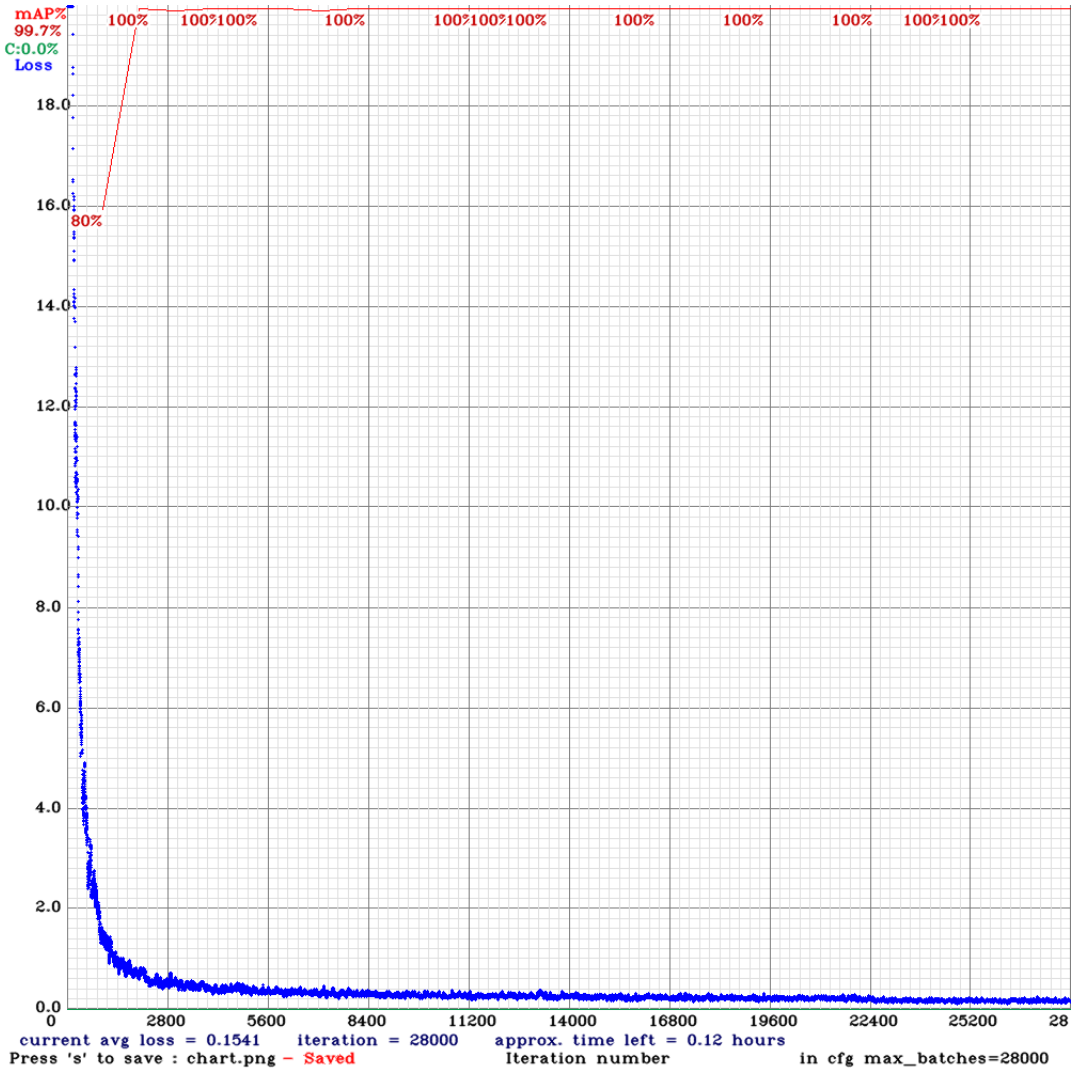

Figura 6.1: Gráfica mAP de la arquitectura YOLOV4.

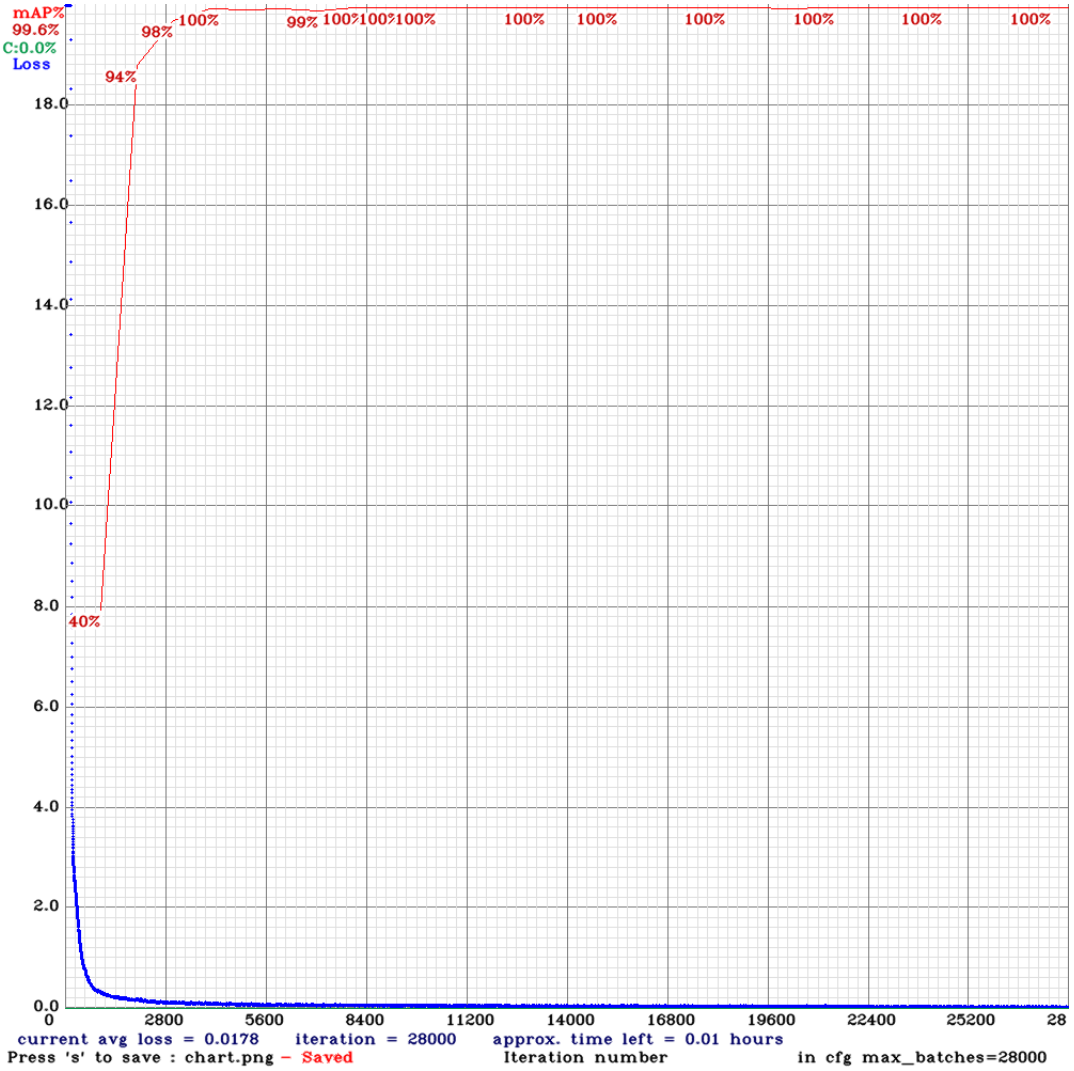

Figura 6.2: Gráfica mAP de la arquitectura Tiny-YOLOV4.

6.1.3. Detección de las marcas naturales

Una vez teniendo los modelos YOLOV4 y tiny-YOLOV4 previamente entrenados con nuestras marcas naturales, se hicieron pruebas de detección para cada modelo siendo ejecutadas dentro de la Jetson TX2 Developer Kit. Está prueba consistió en localizar al robot a ciertas distancias de cada marca natural dentro del laboratorio. Las distancias que se consideraron dadas las consideraciones de espacio dentro del laboratorio fueron a un metro y a un metro y medio entre las marcas naturales y el robot.

En la figura 6.3 y en la tabla 6.3 podemos observar las marcas que fueron detectadas por el modelo de YOLOV4 con su respectiva probabilidad de acierto. En la figura 6.3 vemos de color azul las marcas detectadas a un metro y de color naranja las que fueron

detectados a 1.5 metros de separación. Podemos observar que las marcas a 1.5 metros fueron las que más fueron detectadas en comparación de las detectadas a un metro. También podemos observar que existen ausencias de marcas, que en este caso fueron el estante, la cámara, el refrigerador y el botiquín.

Cabe mencionar que el modelo de YOLOV4, al ejecutarse dentro de la Jetson, corrió a 3 FPS, lo cual hizo que la detección no sea en tiempo real y que tuviera un retraso en cada momento en el que el robot navegó dentro del laboratorio.

Objeto	Probabilidad a 1m	Probabilidad a 1.5m
Archivero	92 %	40 %
Extintor	0 %	97 %
Flecha	0 %	97 %
Impresora	99 %	0 %
Letrero bio-robótica	73 %	93 %
Letrero extintor	84 %	94 %
Letrero sismo	0 %	94 %
Locker	100 %	100 %
Organizador	80 %	80 %
Poster	98 %	98 %
Ventilador	100 %	95 %

Tabla 6.3: Probabilidades de las marcas con YOLOV4.

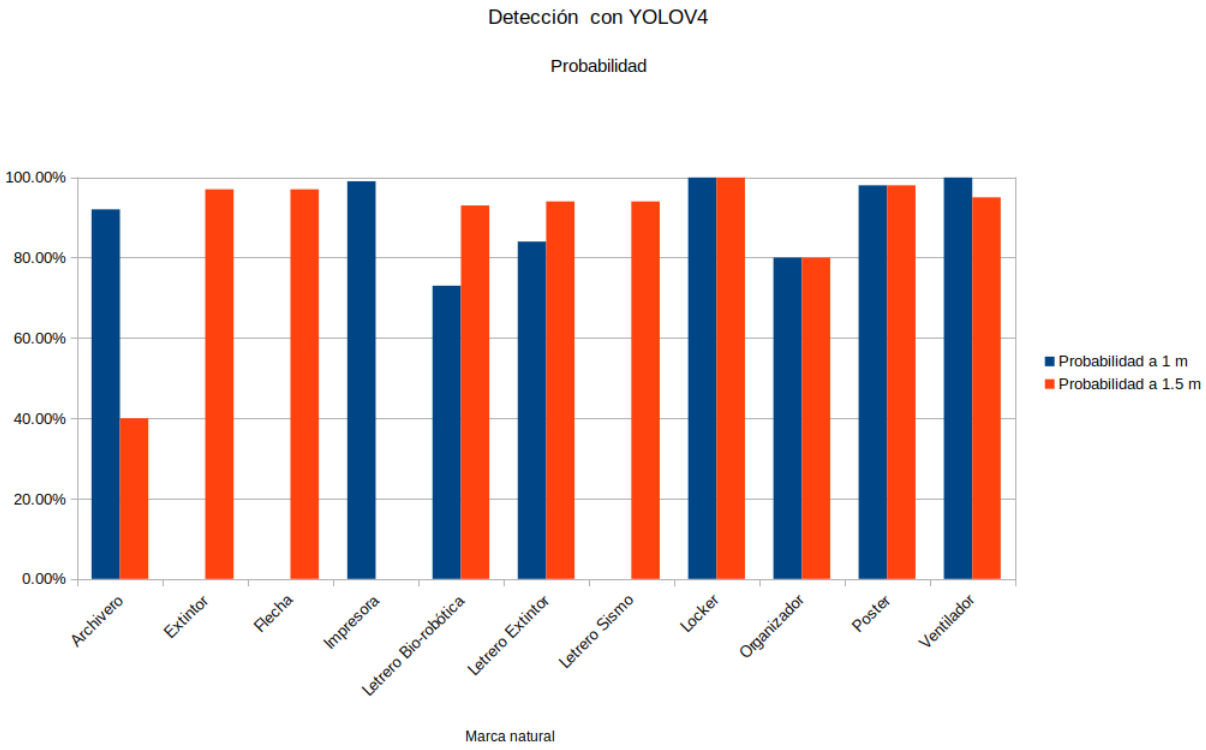

Figura 6.3: Probabilidades de detección con YOLOV4

En la figura 6.4 y en la tabla 6.4 podemos observar las marcas que fueron detectadas por que el modelo de Tiny-YOLOV4 con su respectiva probabilidad de acierto. En la figura 6.4 vemos de color azul las marcas detectadas a un metro y de color naranja las que fueron detectados a 1.5 metros de separación. Podemos observar que las marcas a 1.5 metros fueron las que más fueron detectadas en comparación de las detectadas a un metro. También podemos observar que como en el modelo anterior coinciden en no detectar las mismas marcas previamente mencionadas.

Cabe mencionar que el modelo de Tiny-YOLOV4 al ejecutarse dentro de la Jetson corrió a 29 FPS, lo cual hizo que la detección fue lo más lo más cercano a un tiempo real y sin presentar algún retraso durante el tiempo que el robot navegó dentro del laboratorio.

Objeto	Probabilidad a 1m	Probabilidad a 1.5m
Archivero	70 %	49 %
Extintor	92 %	73 %
Flecha	0 %	69 %
Impresora	93 %	0 %
Letrero bio-robótica	0 %	98 %
Letrero extintor	0 %	49 %
Letrero sismo	0 %	72 %
Locker	100 %	100 %
Organizador	83 %	0 %
Poster	99 %	99 %
Ventilador	100 %	93 %

Tabla 6.4: Probabilidades de las marcas con Tiny-YOLOV4.

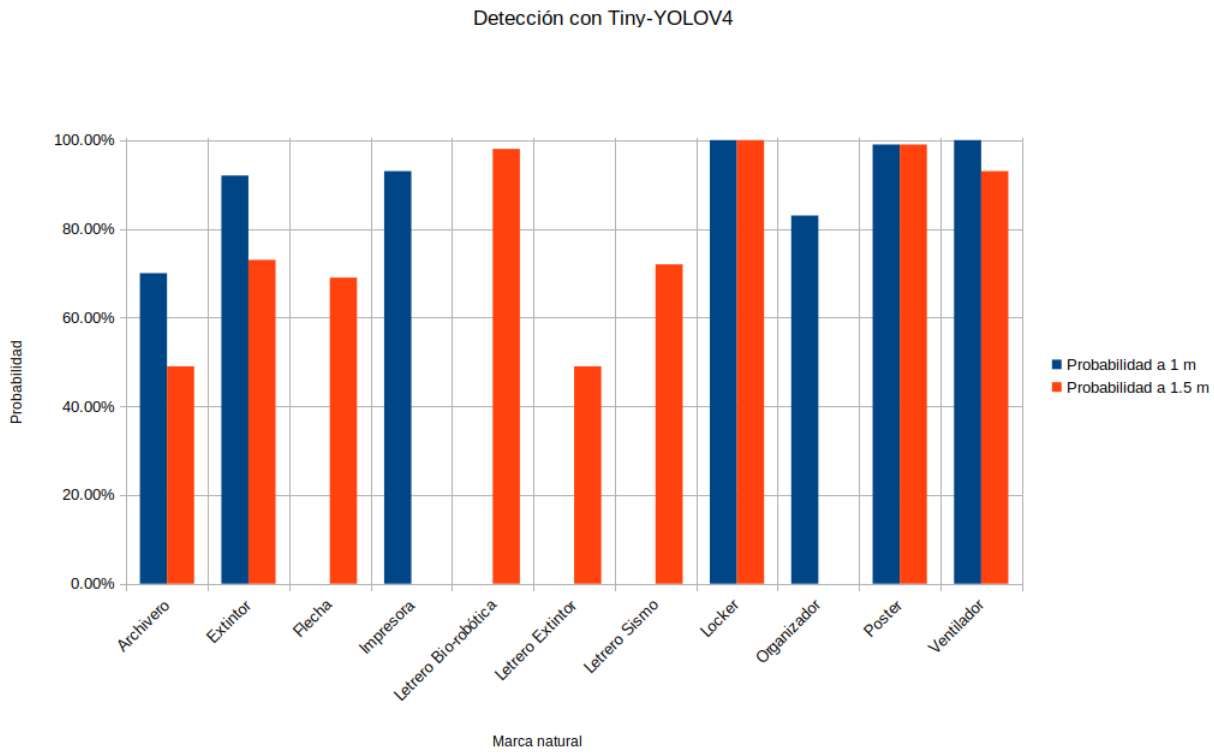

Figura 6.4: Probabilidades de detección con tiny-YOLOV4.

Dado lo anterior, se pudo determinar cuales marcas dentro del laboratorio no tienen problema en ser detectadas, ya sea que estén a un metro o a 1.5 metros. Esto se hizo para considerar las perspectivas de cada marca al momento en el que el robot las encuentre

considerando 1 metro y 1.5 metros que restringe el área del laboratorio. En la tabla 6.5 podemos ver las marcas que se pudieron considerar y las que no, para hacer la localización con el EKF.

ID de la clase	Nombre de la clase	Con YOLOV4	Con Tiny-YOLOV4
0	Archivero	si	si
1	Botiquín	no	no
2	Cámara	no	no
3	Estante	no	no
4	Extintor	si	si
5	Flecha	si	si
6	Impresora	si	si
7	Letrero biorobótica	si	si
8	Letrero extintor	si	si
9	Letrero sismo	si	si
10	Locker	si	si
11	Organizador	si	si
12	Poster	si	si
13	Refrigerador	no	no
14	Ventilador	si	si

Tabla 6.5: Conjunto de marcas naturales detectadas por cada modelo.

6.2. Resultados de la localización del robot

Una vez haciendo las pruebas de detección con los dos modelos mencionados previamente, se prosiguió a realizar las pruebas de localización pero con las marcas naturales. Estas pruebas consisten en hacer un mapeo previo del laboratorio con las marcas naturales, para tener como referencia las distancias fijas con respecto al origen del mapa (en este caso el origen se encuentra en la entrada del laboratorio), y posteriormente, el algoritmo EKF pueda hacer estimaciones de las distancias de las marcas detectadas con las medidas fijamente.

6.2.1. Creación del mapa de marcas naturales

Para crear la representación virtual del mapa en RViz se implementó un nodo de ROS el cual consiste en tomar las coordenadas de cada una de las marcas naturales del ambiente. Posteriormente se hizo la transformación respecto al origen del mapa (el cual se encuentra en la entrada del laboratorio) para tener las referencias con las cuales el EKF pueda hacer sus comparaciones.

Para la detección de las marcas se utilizó la arquitectura de YOLOV4, ya que es la arquitectura que tuvo mejores porcentajes de probabilidad al momento de reconocer las

marcas. Pese que los FPS eran lentos en está arquitectura, lo que nos importa es que sean reconocidos lo mejor posible para recrear el mapa.

En este caso, en la tabla tal se muestran las coordenadas $(x_{map}, y_{map}, z_{map})$ que resultaron al mapear las marcas naturales y ser transformadas respecto al origen del mapa. También en la figura 6.5 se muestran las ubicaciones de cada marca natural localizada en el laboratorio de bio-robótica. Estas marcas se encuentran almacenadas en un archivo donde se tiene un formato con las marcas de la siguiente manera: *ID de la marca* x_{map} y_{map} z_{map}.

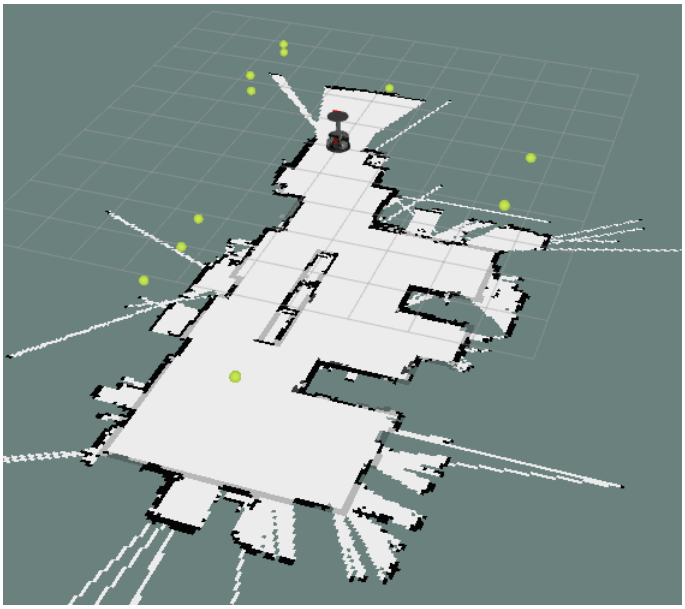

Figura 6.5: Recreación del mapa de marcas naturales en RViz.

Una vez teniendo virtualizado nuestro mapa del laboratorio por el cual navegó el robot, se pudo proceder a hacer las pruebas de localización poniendo al robot a navegar por ciertos puntos del laboratorio.

6.2.2. Localización con las marcas naturales

Para las pruebas de localización de nuestro sistema, se realizó una prueba de navegación dentro del laboratorio. Esta prueba consistió en hacer que el robot navegara hasta cierto puntos dentro del laboratorio para ver que tan bien el robot se localizaría con el EKF y con las marcas naturales. Para estás pruebas se utilizó el mapa creado previamente y un sistema de navegación.

Este sistema de navegación está basado en la tesis de doctorado del Dr. Marco Negrete (32), la cual consiste en un sistema compuesto, principalmente, de un bloque encargado de trazar rutas, un bloque para detectar obstáculos por medio de sensores del robot y un bloque para dar las instrucciones al robot de movimiento según el sistema

trazado. Cabe aclarar que este sistema de navegación es adaptable para un robot móvil autónomo, por lo cual es un sistema muy utilizado para la investigación dentro del laboratorio de bio-robótica.

Las pruebas consisten en llevar al robot desde la entrada hasta un punto dentro del laboratorio. Estos puntos tienen cerca una o varias marcas naturales para que sea más fácil el localizarse dentro del mapa. Para ello, se realizaron 3 trayectorias diferentes dentro del mapa como se muestra en la figura 6.6, en donde se observan las 3 áreas a donde el robot va a navegar estando cerca de las marcas naturales entrenadas. Además, se utilizaron los modelos de YOLOV4 y de Tiny-YOLOV4 para la detección de las marcas, y comparar rendimientos entre sí.

Figura 6.6: Áreas propuestas para la navegación del robot.

Para evaluar el funcionamiento del sistema de localización completo, fue desarrollado un nodo en ROS el cual gráfica las coordenadas de la odometría del robot (x_{odom}, y_{odom}, z_{odom}) contra las coordenadas del EKF calculados (x_{ekf}, y_{ekf}, z_{ekf}), para ver que tanto se parecen las posiciones reales contra las predichas.

6.2.3. Pruebas de localización

Para evaluar el sistema de localización desarrollado en este trabajo se usó el sistema de navegación previamente mencionado; éste sistema implemento las rutas por las cuales el robot navegó por las tres diferentes áreas del laboratorio mientras que el EKF localizaba al robot durante su trayecto. De igual manera y como se mencionó previamente, el nodo desarrollado para gráficar la predicción y la odometría tiene como función ver el comportamiento de las instrucciones dadas por el EKF en comparación de la odometría que tenga el robot.

Este nodo tuvo como fin dibujar las gráficas conforme a las coordenadas (x_{odom}, y_{odom}, z_{odom}) y las coordenadas (x_{ekf}, y_{ekf}, z_{ekf}). El nodo gráfica los valores conforme los tópicos de ambos nodos son recibidos, es decir, se genera una gráfica de número de valor recibido vs valor de la coordenada.

Para cada una de las 3 áreas de prueba vistas en la figura 6.6), el sistema trazo tres diferentes rutas para que el robot navegue por todo el laboratorio hasta llegar a su área destino. Durante las 3 pruebas no fue colocado ningún obstáculo que interfiera en la navegación del robot con el EKF. Para el área 1 la ruta que trazó se ve en la figura 6.7, para el área 2 la ruta trazada se muestra en la figura 6.8 y para el área 3 la figura 6.9.

En el trayecto de cada área se observó que el robot detecto diferentes marcas naturales, con las cuales se pudo localizar. Las marcas para cada área fueron las siguientes:

- Para el área 1 fueron el extintor, el letrero extintor, la flecha, el letrero sismo, el locker, la impresora y el organizador.
- Para el área 2 fueron el extintor, el letrero extintor, la flecha, el letrero sismo, el locker, y el ventilador.
- Para el área 3 fueron el extintor, el letrero extintor, la flecha y el letrero sismo.

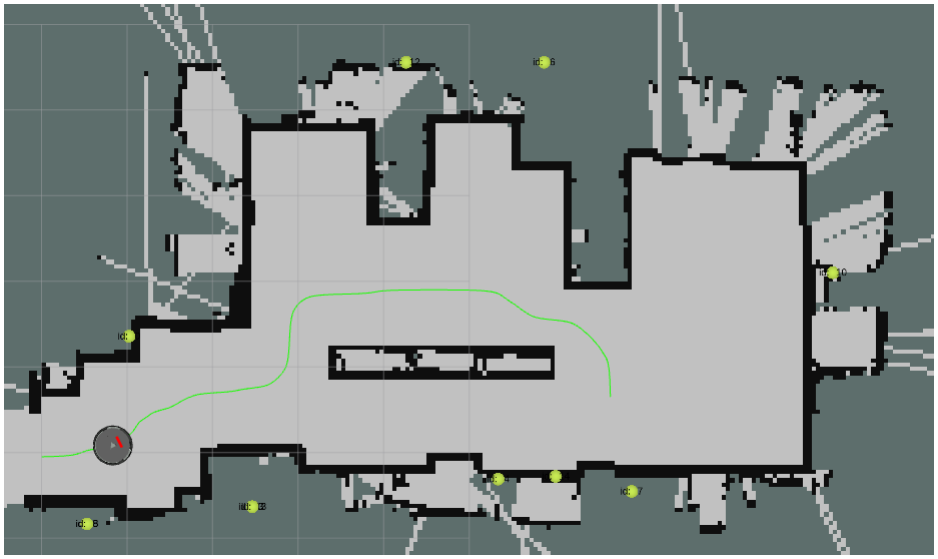

Figura 6.7: Navegación del robot hacia el área 1

Figura 6.8: Navegación del robot hacia el área 2

Figura 6.9: Navegación del robot hacia el área 3

También para de cada una de las pruebas, se utilizó los dos modelos entrenados YOLOV4 y Tiny-YOLOV4, comparar y analizar entre sí los modelos por medio de las gráficas resultantes de a cuerdo a la predicción del EKF del robot.

6.2.3.1. Pruebas en área 1

En las gráficas resultantes que se muestran en la figura 6.10, se observa que los valores $x_{ekf}, y_{ekf}, z_{ekf}$ realizan una predicción de las instrucciones con coordenadas (x, y, θ) que debe realizar el robot para localizarse y llegar al área 1. Mientras que los valores $x_{odom}, y_{odom}, z_{odom}$ presentados en la de odometría, se observan las instrucciones con coordenadas (x, y, θ) que el robot ejecutó para llegar a su punto destino en el área 1.

Para el modelo Tiny-YOLOV4 (figura 6.11), observamos el mismo caso que con el modelo YOLOV4, donde las gráficas pese a no ser las mismas entre modelos, sí tienen un comportamiento que las hace semejantes. La diferencia sobretodo se presenta al final, como terminan ambas gráficas, que visto desde la perspectiva del robot, en ambos casos entre los modelos, el robot no terminó exactamente en el mismo punto.

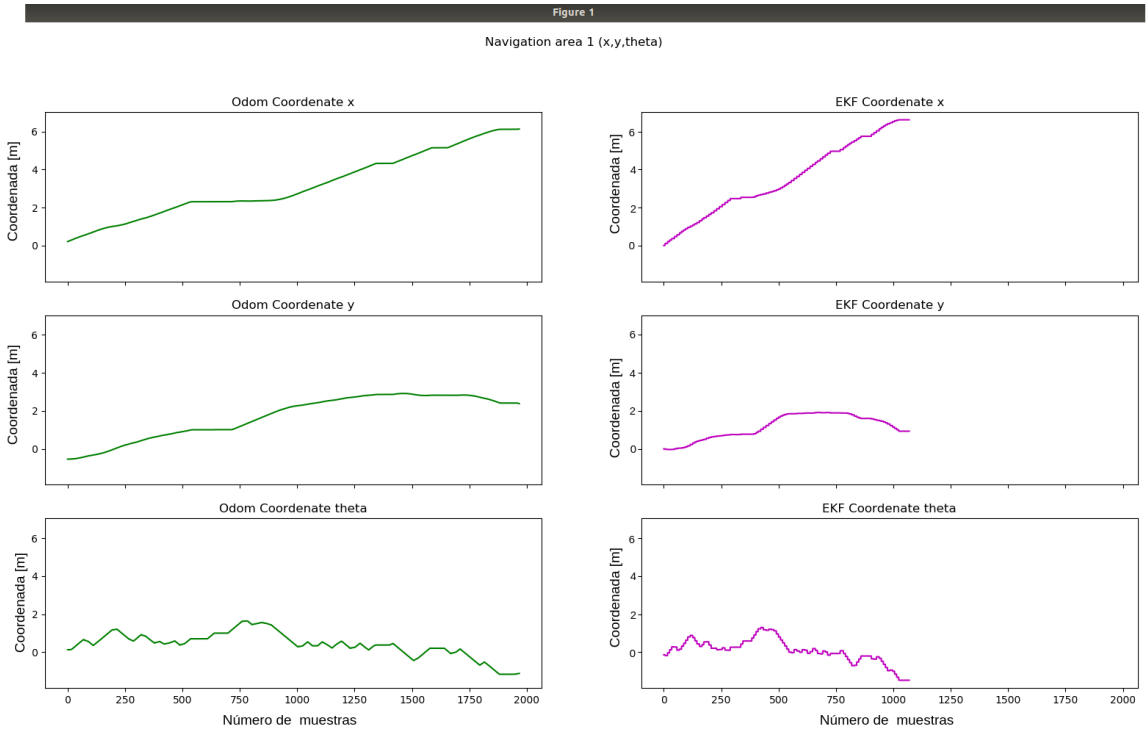

Figura 6.10: Gráficas de odometría y predicción EKF del área 1 con YOLOv4.

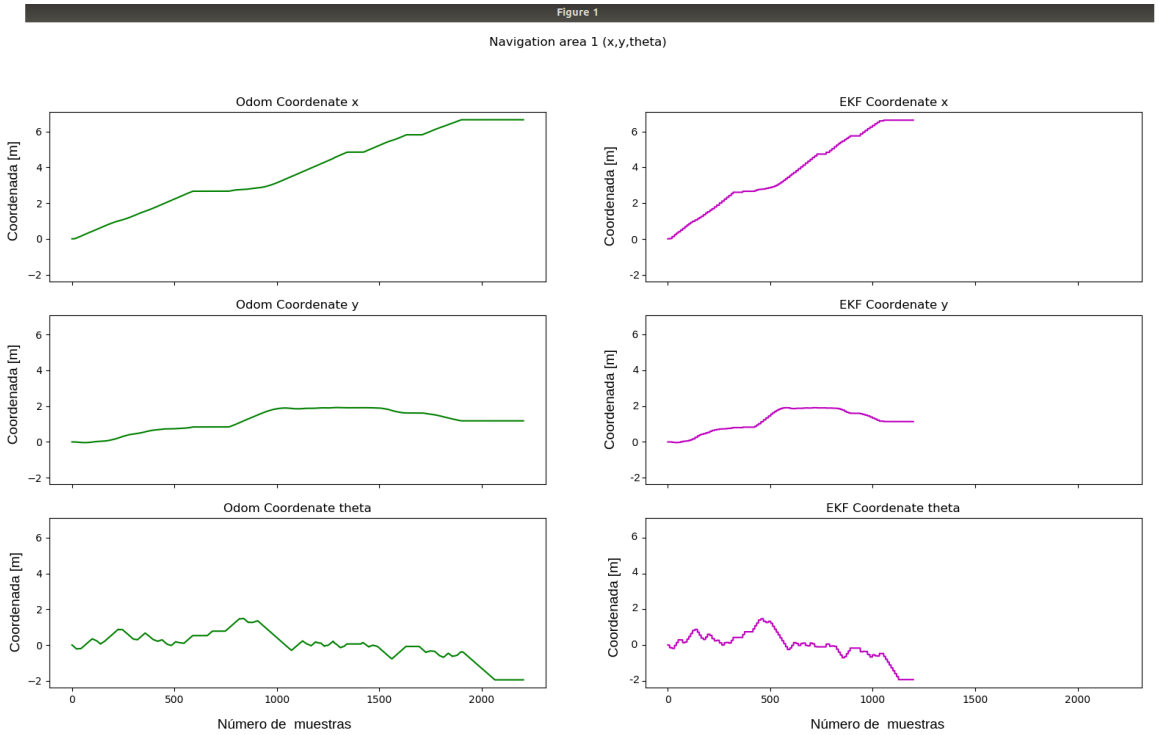

Figura 6.11: Gráficas de odometría y predicción EKF del área 1 con Tiny-YOLOv4.

6.2.3.2. Pruebas en área 2

Utilizando el modelo YOLOV4 (figura 6.12), podemos observar las gráficas de las coordenadas $x_{ekf}, y_{ekf}, z_{ekf}$ para la predicción del EKF y las coordenadas $x_{odom}, y_{odom}, z_{odom}$ de la odometría generadas de la navegación hacia el área 2. En ella podemos observar que ambas gráficas (de EKF y de odometría) se comportan de manera similar entre sí.

Para la figura 6.13 con el modelo Tiny-YOLOV4, tambien podemos observar las gráficas de las coordenadas $x_{ekf}, y_{ekf}, z_{ekf}$ para la predicción del EKF y las coordenadas $x_{odom}, y_{odom}, z_{odom}$ de la odometría generadas de la navegación hacia el área 2. De igual manera podemos ver un comportamiento que asemeja a las gráficas entre sí que vimos en el modelo anterior.

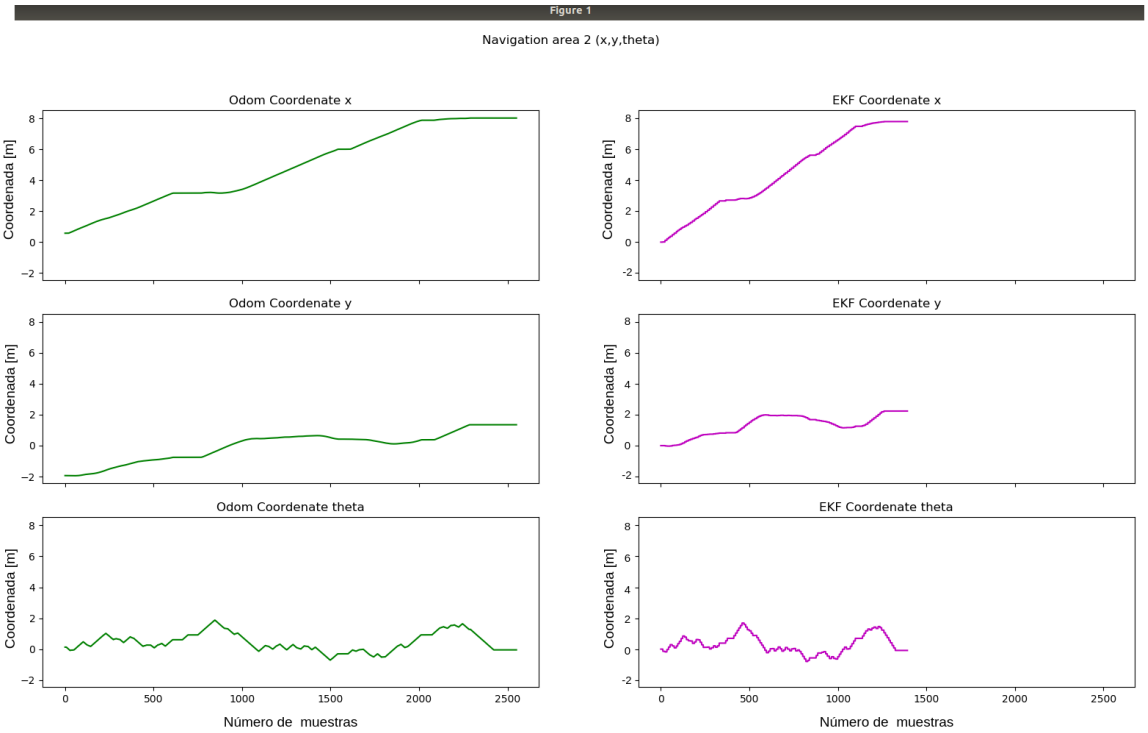

Figura 6.12: Gráficas de odometría y predicción EKF del área 2 con YOLOv4.

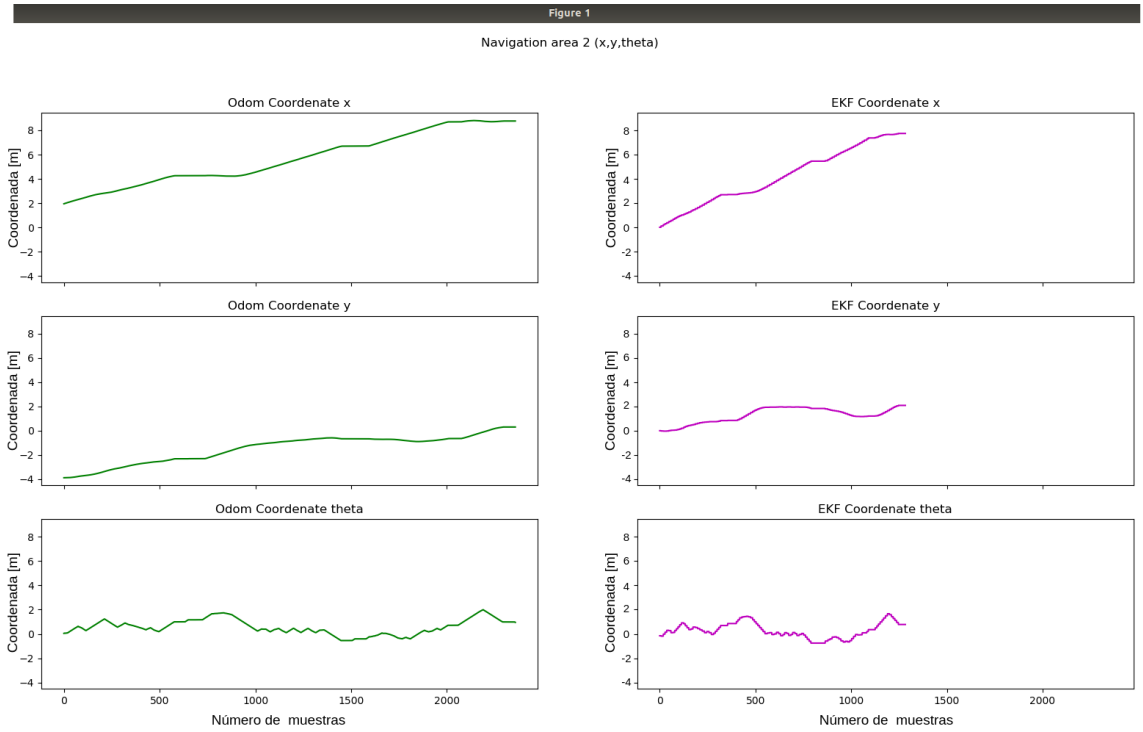

Figura 6.13: Gráficas de odometría y predicción EKF del área 2 con Tiny-YOLOv4.

6.2.3.3. Pruebas en área 3

Con el modelo YOLOV4 se pudo obtener las gráficas de la figura 6.14. En ellas se observa las coordenadas $x_{ekf}, y_{ekf}, z_{ekf}$ para la predicción del EKF y las coordenadas $x_{odom}, y_{odom}, z_{odom}$ de la odometría generadas de la navegación hacia el área 3. Observando las gráficas, podemos ver una similitud entre las gráficas de la predicción con EKF y la posición resultante de la navegación.

Implementando el modelo Tiny-YOLOV4 (la figura 6.15), también podemos observar las gráficas de las coordenadas $x_{ekf}, y_{ekf}, z_{ekf}$ para la predicción del EKF y las coordenadas $x_{odom}, y_{odom}, z_{odom}$ de la odometría generadas de la navegación hacia el área 3. En ambas gráficas se observa un comportamiento semejante entre ellas.

6.2.4. Análisis de las pruebas de localización

Analizando y comparando las gráficas generadas en las tres áreas con los modelos YOLOV4 y Tiny-YOLOV4, podemos ver las coordenadas $(x_{ekf}, y_{ekf}, z_{ekf})$ como las teóricas y las coordenadas $(x_{odom}, y_{odom}, z_{odom})$ las prácticas. El porque sucede que las gráficas son similares lo podemos ver en las lecturas tomadas:

- Para el área 1, tenemos que las lecturas predichas por el EKF terminan aproximadamente por la lectura 1000, las de la odometría terminan cerca de la lectura

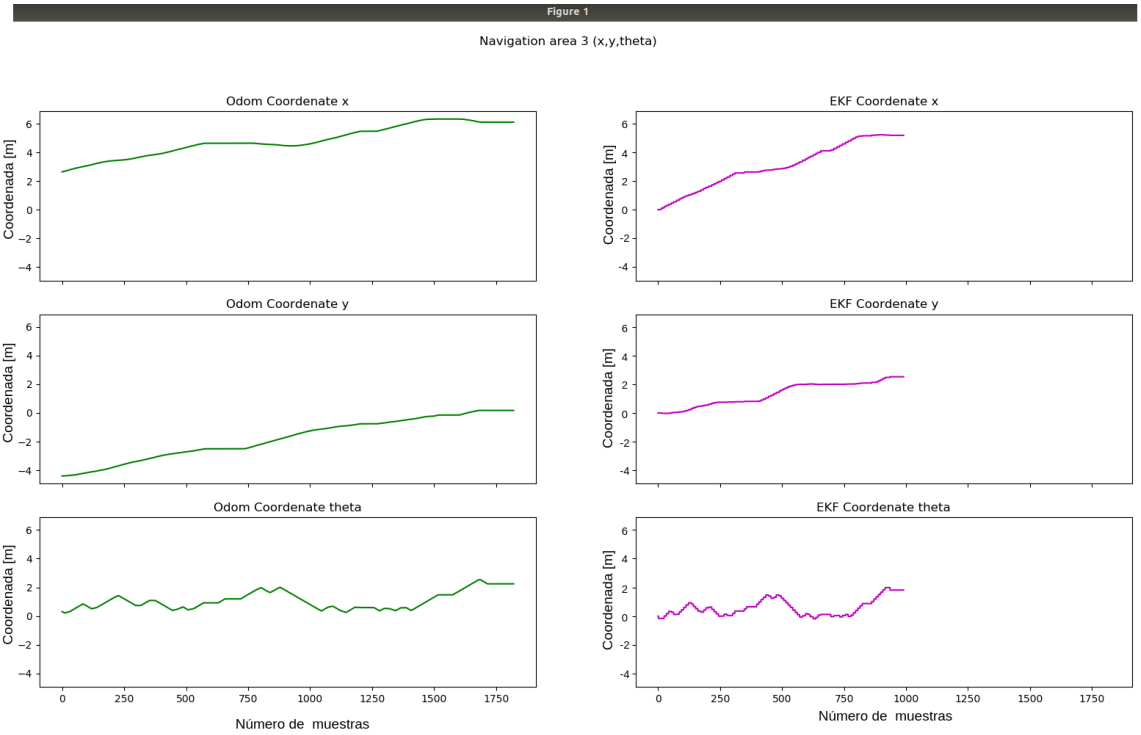

Figura 6.14: Gráficas de odometría y predicción EKF del área 3 con YOLOv4.

2000 con el modelo de YOLOV4. Para el modelo Tiny-YOLOV4 las lecturas de EKF, la gráfica termina aproximadamente por la iteración 1100 y las lecturas de la navegación por la iteración 2300.

- En el caso del área 2, vemos que las gráficas generadas con YOLOV4 terminan cerca de la iteración 1500 de las coordenadas $(x_{ekf}, y_{ekf}, z_{ekf})$ y en las gráficas de odometría $(x_{odom}, y_{odom}, z_{odom})$ se ejecuta la ultima instrucción aproximadamente en la iteración 2500. Con el modelo Tiny-YOLOV4 la gráfica de predicción termina en la iteración 1300 y la odometría de la navegación terminaba cerca de la iteración 2400.

- Con el área 3 las gráficas obtenidas con el modelo de YOLOV4 terminaban la iteración 1000 para el caso de la predicción con EKF y en la iteración 1800 para las gráficas de la odometría del robot al momento de navegar. En el caso del modelo Tiny-YOLOV4, la gráfica de predicción EKF terminaba aproximadamente de la iteración 900 y para la odometría terminaban en la iteración 1600.

Analizando los modelos YOLOV4 y Tiny-YOLOV4, vemos que en el caso de YOLOV4 tarda más iteraciones en llegar al área correspondiente en comparación con el modelo Tiny-YOLOV4. Esto se puede deber a que el modelo YOLOV4 es un modelo más completo, y por ende, requiere de utilizar más recursos de la Jetson TX2 Developer

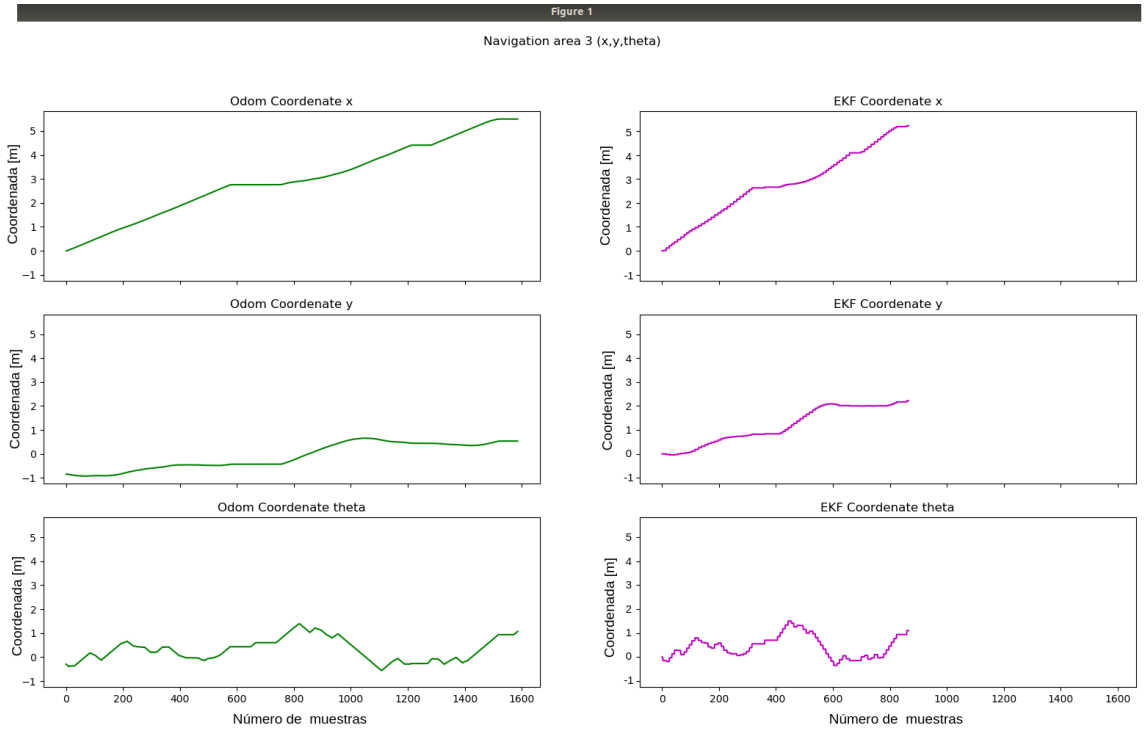

Figura 6.15: Gráficas de odometría y predicción EKF del área 3 con Tiny-YOLOv4.

kit para ejecutarse.

Otro aspecto también a considerar en las gráficas es la predicción del EKF. En ellas podemos ver que la gráfica de la odometría en tiempo real del robot es aproximadamente el doble de tamaño a la predicción con el EKF, pero manteniendo la misma forma de la gráfica en ambos casos. Esto se puede deber a que la gráfica generada con el EKF trazo la ruta a segur el robot, la odometría trata de seguir dicha ruta para llegar a su destino. Esto afecta en que el robot actualiza sus movimientos y tarda más en realizar su avance, reflejado en las gráficas de odometría de las pruebas.

Pese a esto, podemos ver que las marcas naturales ayudaron a la actualización de la pose del robot. La navegación del robot pudo realizarse gracias a que se fueron detectando las marcas naturales, haciendo que lleguen a su área correspondiente.

Capítulo 7

Conclusiones y trabajo futuro

Basado a las pruebas y el análisis redactados en el capítulo 6, se presentan las conclusiones a las que se llegaron, así como el trabajo futuro que se puede desarrollar para atender las áreas de oportunidad que mejoren al proyecto.

7.1. Conclusiones

Conforme a los puntos establecidos en la hipótesis del presente trabajo, se pudo reemplazar los códigos ArUco por marcas naturales al sistema de localización con el EKF. Utilizando el detector de marcas naturales desarrollado y el filtro de Kalman extendido que se tenía se pudo recrear y simular el sentido de localización dentro de un robot móvil. El uso de redes neuronales para crear el detector de marcas pudo detectar casi todas las marcas naturales propuestas. Esto se debe a que al entrenamiento le faltó considerar más características para el modelo entrenado. Sobretodo, el uso del GPU fue importante para entrenar y para ejecutar el detector dentro del robot móvil, ya que éste una vez reconocida la marca, le daba las coordenadas para la etapa de actualización al filtro de Kalman extendido.

Durante las pruebas, cuando fue recreada el área del laboratorio con las marcas naturales y con los pesos entrenados en tiny-YOLOV4, algunas de éstas no fueron detectadas correctamente, o definitivamente el sensor Kinect no pudo observarlos a la altura que se localizaba. En el caso de tratar de detectar el refrigerador no lo pudo observar ya que este se encuentra al nivel del suelo y con la altura de pocos centímetros. Para casos como el botiquín, la cámara de seguridad y el estante, aún cambiando de puntos de vista, no fue suficientemente capaz de detectarlos.

Otro aspecto que también influyó al momento de detectar las marcas y sobretodo en la recreación del mapa fue el tiempo de ejecución los dos modelos. En el modelo entrenado con YOLOV4 se pudo observar que los FPS que recibía del kinect oscilaban entre los 2.8 a los 3 FPS. Para el modelo Tiny-YOLOV4 los FPS del kinect oscilaban entre los 28 a 30 FPS, diez veces más rápido en consideración del otro modelo. Esto nos quiere decir que para detecciones en tiempo real la mejor opción para este trabajo es

implementar el modelo entrenado con Tiny-YOLOV4. Pese a que en algunos casos las marcas no pasan del 50 % de acierto en la detección, tenemos las información del kinect lo más fluido posible y la red neuronal puede realizar su clasificación mucho más rápido que usando el modelo de YOLOV4.

De las gráficas resultantes de las pruebas de localización realizadas, podemos concluir que el sistema del filtro de Kalman extendido tuvo mejorías usando la detección con el modelo tiny-YOLOV4 en comparación con el modelo YOLOV4. Esto se dedujo observando las gráficas de predicción por EKF, en donde en el modelo YOLOV4 hace mas iteraciones para predecir el siguiente movimiento en comparación con Tiny-YOLOV4. También se puede considerar el factor de ejecución en los FPS de los modelos, donde vimos previamente que Tiny-YOLOV4 tiene más FPS que YOLOV4 en su ejecución.

Por último, dados los resultados presentados, podemos decir que se pudo desarrollar un sistema de detección de marcas naturales implementado dentro de un robot móvil, conjuntarlo con el filtro de Kalman extendido para hacer las predicciones de la navegación, todo siendo ejecutado dentro de un GPU, que en este caso es nuestra Jetson TX2 del robot.

7.2. Trabajo futuro

Para trabajo futuro, se tienen algunas áreas de oportunidad que podrían, en un trabajo posterior, mejorar los resultados reportados en este trabajo. Estas consideraciones abarcan tanto la detección de marcas naturales, como ajustes en el EKF y son mencionadas a continuación:

- En la construcción del robot se consideró que el sensor Kinect estuviera fijo en una sola posición. Esto afectó a la detección de las marcas naturales, ya que pese a tener varias perspectivas de la marca en diferentes imágenes para su entrenamiento, no detectó marcas que estaban por debajo del Kinect, aún cambiando la posición del robot. Dicho esto, se propone como trabajo futuro agregar un sistema basado en motores los cuales permitan mover el el ángulo de visión del Kinect, dando más apertura a la detección.

- En el área de pruebas del robot se escogieron objetos como marcas naturales que fueran únicos, ya que en las pruebas se llegó a confundir con un extintor que se encontraba fuera del laboratorio. Para mejorar esto se puede recrear un área que contenga objetos que sean más fáciles de distinguirse entre sí e inclusive de diferentes colores para que sean una subclase de algún otro objeto. Por ejemplo, si existen varias sillas o muebles se pueden crear subclases partiendo de su color y/o tamaño.

- En la creación del conjunto de datos sólo fue considerado que las marcas naturales tuvieran varias tomas a partir de un vídeo. Al crear este vídeo existen varios factores que afectan a la toma, como lo puede ser la iluminación del lugar, la distancia

a la que se esta tomando el vídeo o la resolución de la cámara para realizar el vídeo. Por lo cual se propone crear un nuevo conjunto de datos considerando diferentes iluminaciones a diferentes distancias para cubrir más opciones y el sistema los pueda detectar pese a las perturbaciones del medio.

- Dentro del algoritmo del EKF, específicamente en la etapa de actualización, cuando un código ArUco es detectado se considera un 100 % de probabilidad de fiabilidad. Pero en la situación en la que es una marca natural, su fiablidad puede variar. Esto hace que las ecuaciones tengan que cambiarse, como se menciona en (31). Por lo cual, como parte del trabajo futuro se modificará estas ecuaciones para el reconocimiento de marcas naturales.

Apéndice A

Imágenes adicionales

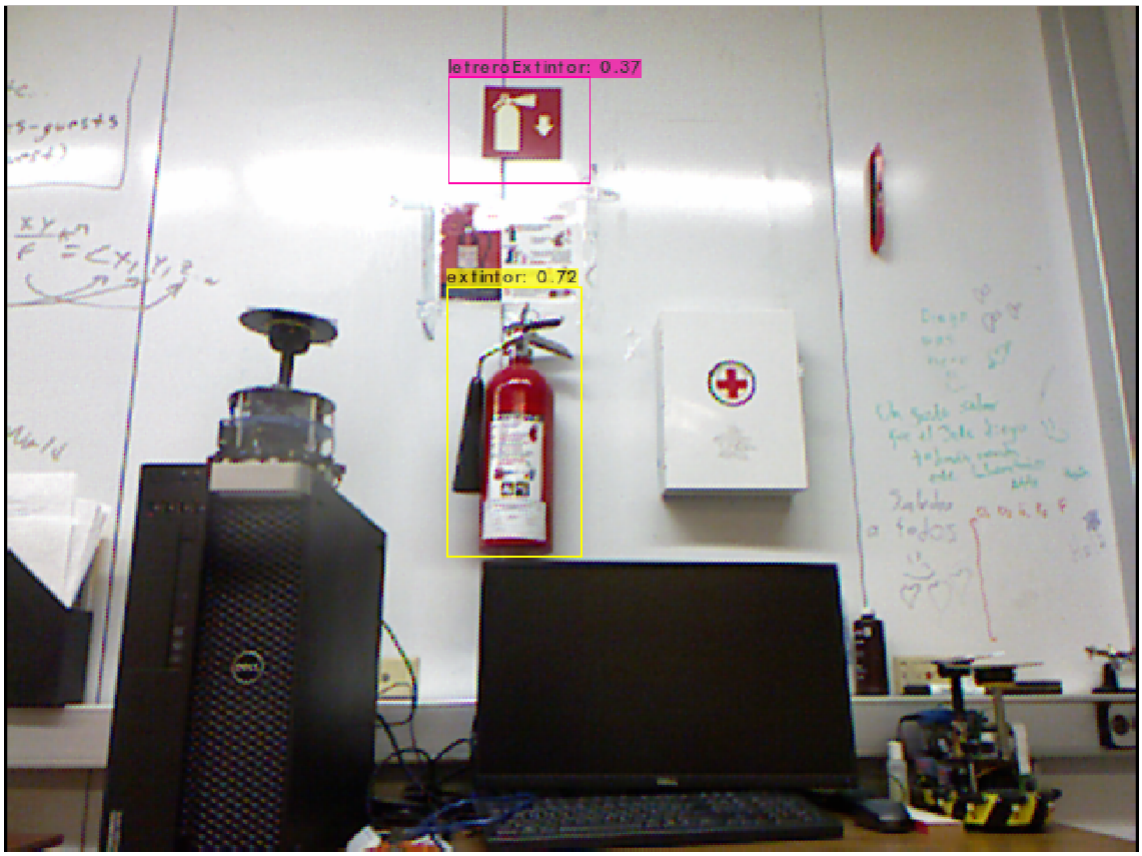

Figura A.1: Botiquín, Extintor y Letrero extintor

Figura A.2: Archivero

Figura A.3: Organizador e Impresora

Figura A.4: Locker y Refrigerador

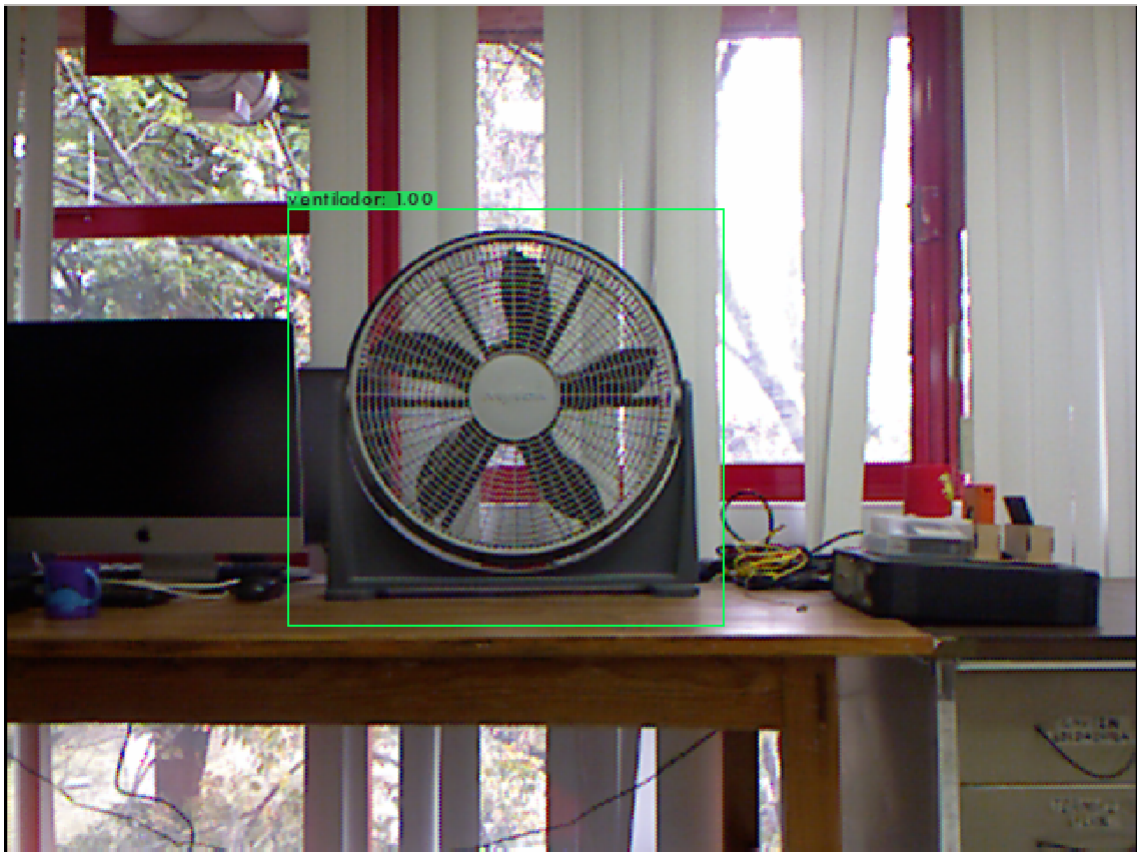

Figura A.5: Ventilador

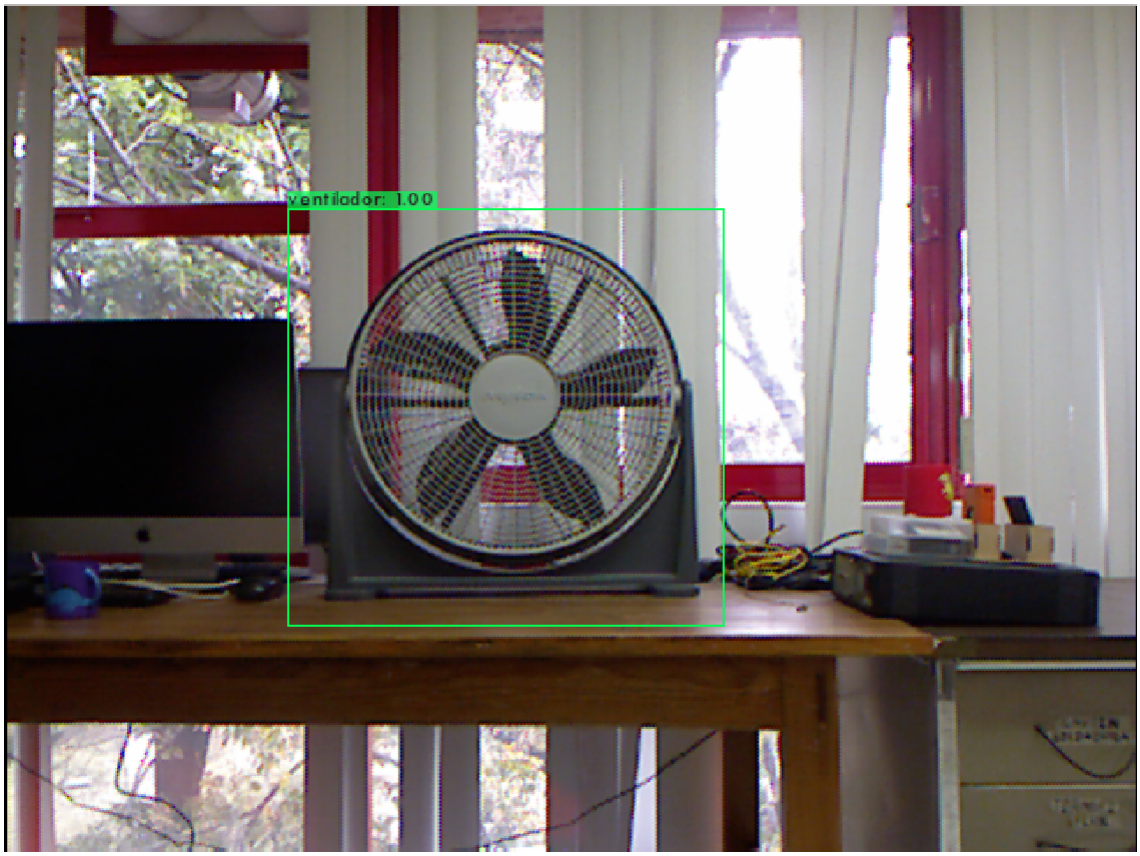

Figura A.6: Ventilador

Figura A.7: Flecha, Letrero sismo, Cámara y Estante

Figura A.8: Letrero bio-róbotica y Poster

Bibliografía

[1] Bochkovskiy, A., Wang, C.-Y., and Liao, H.-Y. M. (2020). Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*. 66, 67, 71

[2] Carmona, J. S. (2020). Filtro de kalman. Technical report, UNAM, https://biorobotics.fi-p.unam.mx/wp-content/uploads/Courses/robots_moviles/2020_2/lecciones/Leccion11_robots_moviles_2020_2.pdf. Presentation PDF.

[3] Castro, D. A. C. (2021). Localización de un robot móvil utilizando el filtro de kalman extendido y marcas visuales en el ambiente. Master's thesis, Posgrado en Ciencias e Ingeniería en Computación, UNAM. Ciudad de México.

[4] Cheng, J., Grossman, M., and McKercher, T. (2014). *Professional CUDA c programming*. John Wiley & Sons.

[5] Faragher, R. (2012). Understanding the basis of the kalman filter via a simple and intuitive derivation. *IEEE Signal processing magazine*, page 128–132.

[6] Fritz, D. (2019). depthjs (github repository). https://github.com/doug/depthjs.

[7] Goodfellow, I., Bengio, Y., and Courville, A. (2016). MIT Press. http://www.deeplearningbook.org.

[8] HP ("Septiembre 14, 2021"). Cómo escoger la mejor gpu para videojuegos. https://www.roadtovr.com/zed-mini-turns-rift-vive-high-end-ar-dev-kit/.

[9] Julier, S. J. and Uhlmann, J. K. (2004). Unscented filtering and nonlinear estimation". *Proceedings of the IEEE.*, page 401–422. (PDF).

[10] Kalman, R. E. (1960). A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, page "35–45".

[11] Kostadinov, S. (2019). Understanding backpropagation algorithm. https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd, (visitado 08-08-2022).

[12] Koyama, Y. M. and Carmona, J. S. (2004). *Determinación de la posición de un robot móvil usando transmisiones ultrasónicos*. PhD thesis, Universidad Nacional Autónoma de México.

[13] Levoy, M. and Whitted, T. (1985). The use of points as a display primitive.

[14] Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. (2014). Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer.

[15] Lippert, A. (2009). Nvidia gpu architecture for general purpose computing. https://www.cs.wm.edu/~kemper/cs654/slides/nvidia.pdf.

[16] MathWorks (2022a). Call and provide ros services. https://www.mathworks.com/help/ros/ug/call-and-provide-ros-services.html.

[17] MathWorks (2022b). Exchange data with ros publishers and subscribers. https://www.mathworks.com/help/ros/ug/exchange-data-with-ros-publishers-and-subscribers.html.

[18] MathWorks (2022c). Slam (simultaneous localization and mapping). https://www.mathworks.com/discovery/slam.html.

[19] Medium, E. A. (2019). Detección de objetos con yolo: implementaciones y como usarlas. https://medium.com/@enriqueav/detecci%C3%B3n-de-objetos-con-yolo-implementaciones-y-como-usarlas-c73ca2489246.

[20] Microsoft ("2010"). Microsoft xbox 360 kinect sensor user guide. https://www.manualslib.com/manual/1541058/Microsoft-Xbox-360-Kinect.html#product-Xbox%20360%20Kinect%20Sensor.

[21] Mrpt.org. (Retrieved March 16, 2011.). Kinect and mrpt | the mobile robot programming toolkit.

[22] Redmon, J., Divvala, S. K., Girshick, R. B., and Farhadi, A. (2015). You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640.

[23] Redmon, J. and Farhadi, A. (2017). Yolo9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271.

[24] roadtovr.com ("Diciembre 13, 2017"). Update: Zed mini turns rift and vive into an ar headset from the future. https://www.hp.com/pe-es/shop/tech-takes/como-escoger-la-mejor-gpu-para-videojuegos.

[25] Robotics, O. (2021). Ros documentation. https://docs.ros.org/.

[26] Rojas, R. (2013). *Neural networks: a systematic introduction*. Springer Science & Business Media.

[27] Rosenblatt, F. (1961). Principles of neurodynamics. perceptrons and the theory of brain mechanisms. Technical report, Cornell Aeronautical Lab Inc Buffalo NY.

[28] Row64 (2022). What is a gpu spreadsheet? a complete guide. https://row64.com/Blogs/Posts/What_Is_A_GPU_Spreadsheet_A_Complete_Guide.php.

[29] Sahoo, S. (2018). Residual blocks — building blocks of resnet. https://towardsdatascience.com/residual-blocks-building-blocks-of-resnet-fd90ca15d6ec, (visitado 08-08-2022).

[30] Savage, J., Rosenblueth, D. A., Matamoros, M., Negrete, M., Contreras, L., Cruz, J., Martell, R., Estrada, H., and Okada, H. (2019). Semantic reasoning in service robots using expert systems. *Robotics and Autonomous Systems*, 114:77–92.

[31] Sebastian Thrun, B. W. and Dieter, F. (2006). *Probabilistic Robotics*. The MIT Press Cambridge Massachusetts, London, England.

[32] Villanueva, M. A. N. (2019). *Sistema de navegacion para un Robot de Servicio*. PhD thesis, Universidad Nacional Autónoma de México.

[33] waveshare.com (2022). Jetson tx2 developer kit. https://www.waveshare.com/wiki/Jetson_TX2_Developer_Kit.

[34] Wortham, J. (November 21, 2010, Retrieved November 25, 2010.). "with kinect controller, hackers take liberties". *The New York Times*.

[35] Zhang, Z. (April 27, 2012). Microsoft kinect sensor and its effect. *IEEE MultiMedia*, page 4–10.