Departamento de Electrotecnia
Facultad de Ingeniería
Universidad Nacional del Comahue

Análisis y propuesta de estrategia para la implementación de técnicas colaborativas de mapeo y localización de robots en una plantación frutícola utilizando visión artificial

Naiara Eileen Sheffield - ING-6073

Dirigida por el Dr. Marcelo Leandro Moreyra
Codirigida por el Lic. Rafael Ignacio Zurita

Ingeniería Electrónica

Neuquén, 2 de Marzo de 2023

Resumen

El trabajo presentado a continuación enseña el estudio sobre una técnica colaborativa de *Visual Simultaneous Localization and Mapping* (Visual-SLAM), el *Centralized Collaborative Monocular Simultaneous Localization and Mapping* (CCM-SLAM), aplicada a un ambiente frutícola. Presenta las características más sobresalientes del algoritmo sobre otros de visión artificial, para luego ser implementado en un dataset propio. Este dataset fue generado en una chacra de la zona del Alto Valle y está acompañado de datos de un GPS asociado para la posterior comparación con los resultados provistos por CCM-SLAM. Luego se analiza el consumo de recursos del algoritmo para detectar las posibles dificultades en el rendimiento, y poder intuir qué placa de hardware embebido es capaz de ejecutar el programa. Tras este análisis, se procede a realizar ensayos sobre dos Raspberry Pi 3B y, en función de esto, posteriormente se plantea una posible estrategia de implementación del algoritmo en el campo.

Palabras clave: Visión artificial, SLAM, SLAM colaborativo, agricultura de precisión, consumo de recursos computacionales.

Abstract

The following work shows a study on a collaborative Visual Simultaneous Localization and Mapping (Visual-SLAM) technique, the Centralized Collaborative Monocular Simultaneous Localization and Mapping (CCM-SLAM), applied to a fruit-tree environment. It presents its most outstanding characteristics over other artificial vision algorithms, to later be implemented using a self-made dataset. This dataset was recorded in a farm in the Alto Valle region and associated GPS data was also provided for later comparison with the results given by CCM-SLAM. After that, the resources consumed by the algorithm are analized to detect possible difficulties in performance, and to estimate what kind of embedded hardware board is capable of executing the program. Then, tests on two Raspberry Pi 3B are performed and, based on this, a possible strategy of the algorithm's implementation on field is presented.

Keywords: Artificial vision, SLAM, collaborative SLAM, precision agriculture, computational resources consumption.

Agradecimientos

El camino recorrido para llegar a este punto ha sido un gran desafío, y es por ello que quisiera agradecer a todas las personas que me ofrecieron su ayuda y compañía en el trayecto. Quisiera comenzar agradeciendo a todos aquellos profesores de la universidad que me dedicaron horas de paciencia en mi enseñanza y soportando eternas horas de consultas. Yo misma practiqué la docencia y veo las dificultades que supone, así que estoy agradecida con todos aquellos que ponen lo mejor para que el alumnado se supere a sí mismo cada día en todos los aspectos.

Agradezco a Marcelo por su guía en este trabajo y a Rafael por su constante ayuda y paciencia; aprendí muchas cosas y me divertí trabajando a su lado. Agradezco también a las personas con las que trabajé durante mis años de estudio, que me ayudaban a mirar la vida desde otra perspectiva y me recordaban constantemente que no dejara de lado el estudio.

Agradezco a mis amigos que me acompañaron durante esta carrera. Cada risa de la cual formaban parte me ayudaba a superar los momentos difíciles, estresantes y agotadores del estudio. Particularmente a Matías, que su apoyo incondicional me acompañó siempre, y que siempre estuvo ahí en los peores y mejores momentos.

Finalmente, agradezco y dedico este trabajo a mi familia. A mi hermano, mi sostén económico y que sin él, es probable que no hubiese podido recibirme en tan poco tiempo y con las notas que lo hice. A mi madrina, la consejera eterna y recipiente de mis problemas. Y a mi madre, el soporte de todo y mi mayor motivación en el estudio; el crédito de llegar hasta acá es de ambas.

Índice

Resumen II
Abstract III
Agradecimientos IV

1. Introducción y objetivos **1**
1.1. Contexto y fundamentación . . . 1
1.2. Objetivos . . . 2
1.3. Organización del trabajo . . . 3
2. Uso de visión para mapeo y localización de robots de forma colaborativa **4**
2.1. Visual SLAM . . . 4
2.1.1. Introducción al V-SLAM . . . 4
2.1.2. Principios básicos del algoritmo . . . 5
2.1.3. Estado del arte . . . 9
2.1.4. ORB-SLAM . . . 10
2.2. CCM-SLAM . . . 13
2.2.1. Introducción al *framework* . . . 13
2.2.2. Descripción del funcionamiento . . . 14
3. Metodología experimental para la evaluación del CCM-SLAM **19**
3.1. Dataset a utilizar . . . 19
3.1.1. Chacra en Cipolletti . . . 20
3.1.2. Elección del dataset a utilizar . . . 21
3.1.3. Dificultades en el dataset . . . 22
3.1.4. Tratamiento del dataset . . . 24
3.2. Metodología de la experimentación con CCM-SLAM . . . 25
3.2.1. Preparación del ambiente para la experimentación . . . 25
3.2.2. Configuración de los experimentos para la medición de recursos . . . 28
3.2.3. Pruebas en placas de hardware embebido . . . 30
4. Resultados y discusiones **33**
4.1. Pruebas preliminares . . . 33
4.2. Pruebas con medición de recursos . . . 39
4.3. Pruebas con las placas de hardware embebido . . . 42
4.4. Propuesta de abordaje en campo . . . 46
4.4.1. Requisitos mínimos . . . 46
4.4.2. Estrategia a implementar . . . 48
5. Conclusiones **51**
5.1. Resumen del trabajo realizado . . . 51

5.2. Conclusiones 52
5.3. Cierre del trabajo y propuestas de mejora 53

Bibliografía **55**

Capítulo 1

Introducción y objetivos

El presente capítulo tiene por objetivo establecer el contexto y la línea de trabajo del proyecto integrador. La primera sección detalla los fundamentos que lideraron a la elaboración de este trabajo. Luego, la segunda sección plantea los objetivos del proyecto integrador, y la tercera describe la estructura que se seguirá a lo largo de todo el escrito para su desarrollo.

1.1. Contexto y fundamentación

La agricultura de precisión trabaja en la incorporación de nuevas tecnologías con el propósito final de mejorar la eficiencia y sostenibilidad de la agricultura en el tiempo. Dentro de las tecnologías que se utilizan, se pueden enunciar como mayores protagonistas el *Global Positioning System* (GPS), las imágenes satelitales y el SIG (Sistemas de Información Geográfica)[1]. Además, se destacan de manera emergente la visión artificial, el uso del LiDAR (del inglés, *Light Detection and Ranging*), el Internet de las Cosas (IOT, en inglés), el uso de *Machine Learning* [2] y los vehículos autónomos.

En el caso puntual de Argentina, la agricultura de precisión ha logrado hacerse un lugar en el país [3]; particularmente, resulta interesante el enfoque de esta ciencia en la fruticultura[4][5]. La fruticultura de precisión busca brindarle soluciones al fruticultor para mejorar el rendimiento de sus cosechas y, en simultáneo, disminuir el impacto que puedan tener sobre el medio ambiente.

Paralelamente, es de público conocimiento que la fruticultura en la región del Alto Valle de las provincias de Río Negro y Neuquén lleva ya varios años enfrentando problemas económicos que ponen en riesgo su perdurabilidad en el tiempo [6]. Los avances en la fruticultura de precisión fueron tomados para aplicarlos en la región [7] con el objetivo de ayudar a los

productores a mejorar la rentabilidad de sus cosechas. Sin embargo, el costo de algunas tecnologías complejiza el acceso de los productores a las mismas; por ejemplo, los LiDARes tienen precios que oscilan entre los 2000 y 6000 dólares según el caso. Esto, sumado a otros costos inherentes al uso de la fruticultura de precisión, resulta en una inversión que el chacarero de la zona no puede permitirse.

En ese sentido, el grupo de investigación en el cual se enmarca este proyecto integrador, el cual viene trabajando en conjunto con INTA Alto Valle desde 2015 en la investigación y desarrollo de la robótica, se ha propuesto poner foco en la visión artificial como principal fuente de sensado para los sistemas de navegación de vehículos inteligentes [8]. Esto es así dado que en los últimos años el costo de las cámaras se ha reducido y el hardware para procesamiento de imágenes se ha diversificado, resultando en que estas tecnologías sean más accesibles.

En consecuencia, el uso de la visión artificial da lugar a las técnicas de Visual-SLAM o V-SLAM (del inglés, *Visual Simultaneous Localization and Mapping*), las cuales buscan que el vehículo autónomo vaya guiándose mientras construye un mapa del lugar que recorre, a través de la información que adquiere a través de la cámara.

Si bien los algoritmos de visión artificial se encuentran en constante desarrollo, las operaciones que necesitan realizar pueden representar un costo de cómputo demasiado elevado. Como resultado de esto, el presente trabajo buscar poner a prueba una de estas técnicas de V-SLAM, particularmente una técnica colaborativa entre uno o más robots, para evaluar la posibilidad de reducir el costo de cómputo y en consecuencia el costo del hardware involucrado.

1.2. Objetivos

Los objetivos que se plantearon para este trabajo son los siguientes:

- Implementar y analizar el uso del algoritmo colaborativo de V-SLAM, CCM-SLAM (del inglés, *Centralized Collaborative Monocular Simultaneous Localization and Mapping*), para evaluar su posibilidad de uso en un sistema de bajo costo que use visión artificial como primer elemento de sensado en la estimación de trayectorias de un robot en una plantación frutícola.

- Medir el consumo de recursos computacionales del algoritmo y analizar posibles cuellos de botella.

- Testear el algoritmo en una o más placas de hardware de cómputo para evaluar sus

capacidades y limitaciones de ejecución del mismo.

- Proponer alguna estrategia factible de implementación del algoritmo en una plantación, que haga foco en los anchos de banda de comunicación y hardware mínimo que tanto cliente como servidor deban cumplir.

1.3. Organización del trabajo

La organización del proyecto integrador es la siguiente: en el Capítulo 2, se plantean las bases teóricas para introducir al lector al funcionamiento de los algoritmos de Visual SLAM, en función del estado del arte. Consecuentemente, esto dará pie para hablar sobre el CCM-SLAM, el algoritmo clave de este trabajo. Se resaltará mayormente sus distinciones respecto a otros algoritmos de Visual SLAM, haciendo clara alusión a la característica de trabajo colaborativo entre varios robots.

A continuación, en el Capítulo 3, se procederá a relatar todo lo relacionado con el desarrollo de los experimentos, basados en los objetivos planteados en la sección anterior. Esto cubrirá la instalación e implementación del CCM-SLAM usando un dataset del ambiente frutícola, y la forma en que se medirá el uso de recursos para el análisis de cuellos de botella.

Luego, en el Capítulo 4, se expondrán los resultados obtenidos según lo propuesto en el Capítulo 3, y se detallarán las bases de lo que sería la propuesta de abordaje en campo, en función de los mismos resultados.

Finalmente, en el Capítulo 5 se termina de resumir el trabajo realizado y qué conclusiones se obtuvieron, dando cierre al trabajo con las propuestas de trabajos a futuro.

Capítulo 2

Uso de visión para mapeo y localización de robots de forma colaborativa

El capítulo que se desarrolla a continuación establece un marco teórico para comprender los conceptos claves utilizados a lo largo del trabajo. La primera parte introduce definiciones básicas relacionadas al funcionamiento de la visión artificial y otros componentes claves que conforman el CCM-SLAM. Después, la segunda parte describe en detalle el algoritmo CCM-SLAM, cómo funciona y cuáles son las partes que lo integran, sumado a los parámetros que pueden configurarse.

2.1. Visual SLAM

2.1.1. Introducción al V-SLAM

El Mapeo y Localización Simultáneo (en inglés, *Simultaneous Localization and Mapping*, o SLAM) es un problema formulado por la comunidad robótica, que plantea la creación del mapa en el cual el robot se mueve, mientras que simultáneamente éste debe localizarse en el mapa conforme lo va creando. Dado que ambas tareas están íntimamente relacionadas, se realizan simultáneamente de manera tal que la información obtenida de una tarea sea empleada por la otra, y viceversa. Esto es así pues una buena localización depende de un mapa bien construido, y un mapa bien construido resulta de una localización adecuada.

En los últimos años, las herramientas de procesamiento de imágenes se han vuelto más rápidas y confiables [9], sumado a que la tecnología vinculada a las cámaras se ha vuelto más accesible. En consecuencia, los algoritmos de SLAM han evolucionado, en algunos casos, a

depender únicamente de la cámara como elemento de sensado; de ahí que se haya derivado el nombre Visual-SLAM o V-SLAM.

A grandes rasgos, se podría decir que las técnicas de V-SLAM engloban tres líneas de trabajo principales: la odometría visual [10] (en inglés, *Visual Odometry* o VO), la estructura a partir del movimiento (en inglés, *Structure from Motion* o SfM) [11] y el cierre de lazo (en inglés, *loop closure*). El primero se encarga de la estimación de la trayectoria del robot (localización) mediante la incorporación de información exclusivamente visual (imágenes captadas por la cámara); mientras que los dos últimos se enfocan en la construcción del ambiente en el cual se desplaza el robot (mapeo).

2.1.2. Principios básicos del algoritmo

Para comprender el funcionamiento básico de los algoritmos de V-SLAM, es necesario entender cómo trabajan los algoritmos de visión por computadora.

A grandes rasgos, a partir de una secuencia de imágenes, se buscan puntos o regiones de la imagen que son más informativas que otras, y se las sigue a lo largo de la secuencia. Observando su movimiento y suponiendo que el ambiente permanece estático, se puede estimar el movimiento de la cámara y del robot que la transporta, haciendo uso de modelos geométricos. Después, conociendo las relaciones geométricas entre las diferentes posiciones y orientaciones de la cámara a lo largo de la trayectoria, se estima la posición en el espacio de los puntos detectados, y esto permite dar el puntapié para comenzar a reconstruir el mundo 3D.

Modelo de la cámara

En primera instancia, es necesario entender cómo los elementos u objetos del mundo 3D son captados por la cámara y representados en una imagen en 2D. Esto se logra a través de un modelo geométrico de proyección para la cámara. Existen varios modelos, pero el más conocido es el modelo de *pinhole*. Básicamente, mapea un punto en el espacio tridimensional con coordenadas $\mathbf{x} = (x, y, z)^T$ al plano de la imagen $z = f$, donde f es la distancia focal de la cámara, uniendo el punto en el espacio al centro de proyección con una línea recta, como puede verse en la Figura 2.1. En consecuencia, el punto resultante tendrá en la imagen las coordenadas $(f_x\, x/z,\, f_y\, y/z)^T$. Se hace la distinción entre f_x y f_y pues no necesariamente la distancia focal en el eje x es la misma que en el eje y, según el tipo de cámara.

Este modelo, al ser relativamente básico, hace necesario añadirle otra serie de parámetros, como son el par de coordenadas de *offset* $(p_x, p_y)^T$ y el factor de sesgo o *skew*, s. El primero considera que el origen de coordenadas en el plano de la imagen puede estar desplazado,

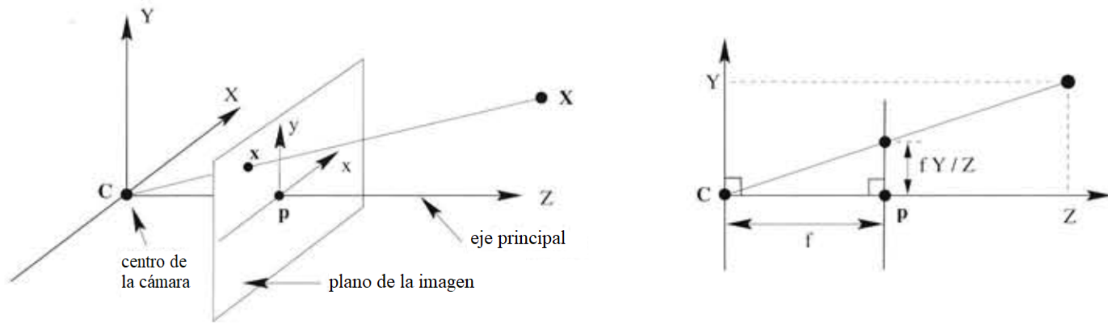

Figura 2.1: Geometría del modelo pinhole. Imagen extraída y modificada de [12].

mientras que el segundo surge cuando los ejes x e y de la cámara no son perpendiculares. Esto último es bastante inusual, así que ese parámetro suele ser nulo. Finalmente, este valor más los descritos anteriores conforman la matriz de parámetros intrínsecos de la cámara K [12]:

$$K = \begin{pmatrix} f_x & s & p_x \\ 0 & f_y & p_y \\ 0 & 0 & 1 \end{pmatrix} \tag{2.1}$$

Para determinar los parámetros intrínsecos de la cámara, existen diferentes métodos. Los más comunes consisten en utilizar un patrón cuyas dimensiones son conocidas, como por ejemplo un tablero cuadriculado en blanco y negro. Se pueden encontrar desde *toolboxes* de Matlab [13] hasta software de código abierto [14] que vienen preparados para simplemente ingresar las imágenes y, automáticamente, el programa calcula los valores de estos parámetros.

Detectores y descriptores

Una vez que los puntos de la imagen 3D fueron convertidos al sistema de coordenadas de la cámara, el algoritmo debe buscar características o *features* sobresalientes en la imagen actual que le sirvan de referencia para cuando vaya a comparar con la siguiente imagen que le llegue.

Este enfoque, comúnmente denominado basado en características o *feature-based*, consiste en dos pasos [15]: primero, se detectan los puntos de interés en las imágenes que puedan ser usados como características o *keypoints*; utilizando los detectores. Segundo, se calculan los descriptores, que consisten en vectores numéricos que representan de manera estadística

cómo es el entorno de cada *keypoint* en la imagen, y que permite comparar fácilmente los *keypoints* entre sí.

En lo que respecta a los detectores de *keypoints*, para que un punto resulte relevante o de buena calidad, debe cumplir las siguientes propiedades: debe ser notable (fácil de extraer), preciso (medido con buena precisión) e invariante a la rotación, traslación y a los cambios de iluminación y escala [16]; esto permitirá mejorar el rendimiento en los procesos de asociación o *matcheo* de imágenes y en la asociación de información.

La bibliografía del estado del arte respecto a los detectores es extensa debido a que es difícil conseguir detectar un punto que cumpla con todas las propiedades mencionadas, en diferentes ambientes o aplicaciones. En general, se suele buscar aquellos puntos que representen una esquina, o algún borde o región que resalte frente al resto de la imagen. Por mencionar algunos, los detectores más referenciados en la bibliografía son el Detector de Esquinas de Harris [17], SIFT (del inglés, *Scale-Invariant Feature Transform*) [18] y SURF (del inglés, *Speeded Up Robust Features*) [19].

Una vez que la característica o *keypoint* ha sido detectada, se procede a traducirla o representarla en un vector de componentes que más tarde pueda ser entendido por el algoritmo. Generalmente, los detectores vienen entonces acompañados de sus respectivos descriptores; es por eso que en la bibliografía en el estado del arte de descriptores se puede encontrar los análogos SIFT y SURF. También existen otros descriptores que buscan introducir mejoras respecto a estos últimos, tales como el BRIEF (del inglés, *Binary Robust Independent Elementary Features*) [20] y el ORB (del inglés, *Oriented FAST and Rotated BRIEF*) [21], basado en el primero mencionado.

Análisis de correspondencias y estimación de la posición

El análisis de correspondencias o *image matching*, compara las similitudes entre dos imágenes de una misma escena determinando cuáles de los *keypoints* se corresponden entre sí entre cada una de las vistas. Generalmente esta comparación entre imágenes se realiza con una métrica de distancia entre descriptores, y a partir de un valor o *threshold* determinado se decide si la correspondencia es cierta o no. La rigurosidad sobre este paso es alta, pues los falsos positivos pueden traer consecuencias graves en los resultados, no así como los falsos negativos.

A pesar de que se utilicen los mejores descriptores disponibles o diferentes tipos de medidas para evaluar las similitudes entre imágenes, las correspondencias falsas pueden guiar a errores relevantes para el sistema SLAM, tanto para la estimación de la pose de la cámara

(es decir, su posición y orientación en el espacio), como para la construcción del mapa. En consecuencia, estimadores robustos como RANSAC (del inglés, *Random Sample Consensus*) y PROSAC (del inglés, *Progressive Sample Consensus*), entre otros, son frecuentes en esta parte del proceso para el refinamiento de la información y correcta asociación de los datos [22].

Mediante las correspondencias, el algoritmo realiza un seguimiento o *tracking* de los *keypoints*, y por ende, sobre la cámara, para que simultáneamente vaya construyendo un mapa con todas las imágenes clave o *keyframes* que se van asociando entre sí. El término "*keyframe*" se adjudica para describir a aquella imagen que se destaca por sobre las otras por la cantidad de características nuevas y relevantes que puede aportar respecto a sus predecesoras/sucesoras.

Los *keypoints* capturados y asociados en cada imagen constituyen una nube de puntos 3D, que contribuyen a la estimación de la posición de la cámara mediante una serie de transformaciones. Para cada mismo punto en una secuencia de imágenes, este representa una traslación y rotación de la cámara en el mundo real. La transformación de ese punto desde las coordenadas globales al sistema de coordenadas de la cámara está representada por un vector t de traslación y una matriz R de rotación. Estos son conocidos como los parámetros extrínsecos de la cámara, pues definen su orientación y posición con respecto al mundo real. Si para el mismo punto pero identificado en distintos *keyframes* se recuperan los valores de rotación y traslación de la cámara, se puede ir reconstruyendo su movimiento por el mundo 3D mediante este punto y otros más que conforman el mapa.

En este punto, podría ocurrir que el algoritmo pierda el *tracking* porque no encuentra suficientes *keyframes* o no hay suficientes correspondencias entre *keyframes*. Las causas de esto suelen ser mayormente la mala calidad de las imágenes o por zonas muy parecidas entre sí, las pérdidas de información a causa de la falta de capacidad de cómputo o los objetos que presentan movimientos súbitos.

Reconocimiento del lugar y cierre de lazo

Finalmente, puede ocurrir que la cámara al desplazarse vuelva a pasar por un lugar del que ya tomó registro anteriormente. En ese caso, el algoritmo debe estar preparado para reconocer la situación y utilizar la nueva información para reforzar la existente, por ejemplo, corrigiendo la posición actual de la cámara.

Un problema típico asociado a los algoritmos de V-SLAM es que acumulan errores en el tiempo si no son corregidos con alguna otra fuente de información. Como solo se nutren de imágenes, estos errores acumulados resultan en una deriva de la estimación de la posición

respecto de la real. Al pasar por una misma posición más de una vez, esto se detecta y se produce un cierre de lazo. Los cierres de lazo o *loop closure* son particularmente útiles porque al ser detectados, el algoritmo puede corregir esta deriva y volver a alinearse con la trayectoria original.

En consecuencia, de manera paralela, otro hilo de procesamiento estará constantemente tomando la información nueva que llega y contrastándola con la existente para detectar si se está transitando por un lugar previamente recorrido o no. Existen varias técnicas para esta tarea, como por ejemplo, utilizar una bolsa de palabras visuales (más conocido del inglés, *Bag of Visual Words, BoVW*). Esta última, consiste en primero extraer descriptores de las características de la imagen; luego, se cuantizan estos descriptores a un "vocabulario" fijo de palabras visuales, y finalmente se usa un histograma de las palabras observadas como un descriptor de la imagen [23]. Los algoritmos de V-SLAM se valen de esta técnica para el contraste mencionado y así poder detectar las coincidencias entre dos lugares idénticos pero que se visitaron en distintos tiempos.

2.1.3. Estado del arte

Dentro del estado del arte de V-SLAM, pueden encontrarse varias alternativas, que fueron mutando y evolucionando con el tiempo a partir de otras versiones ya implementadas. Uno de los primeros registros comenzó con MonoSLAM, un sistema monocular en tiempo real que utilizaba un filtro de Kalman extendido para estimar la posición de la cámara [24]. Tenía varias limitaciones, tales como solo trabajar en espacios cerrados y con movimiento de la cámara constantes y no espontáneos.

Entre otras alternativas monoculares está el PTAM (del inglés, *Parallel tracking and Mapping*), el cual introdujo la característica principal de desarrollar dos hilos paralelos, uno para la localización (*tracking*) y otro para el mapeo (*mapping*); de ahí el nombre "*Parallel*"[25]. También se destaca el LSD-SLAM (del inglés, *Large-Scale Direct Monocular SLAM*), que a diferencia del anterior, no utiliza un método basado en *features* o características, sino que usa un método directo basándose en las intensidades de los píxeles, generando un mapa más denso y definido, pero a costa de mayor consumo computacional [26]. Por último, otro método muy popular es el ORB-SLAM [27], cuyo nombre hace referencia al uso de los descriptores ORB mencionados anteriormente. Debido a la importancia que tiene ORB-SLAM en este trabajo, se le dedicará una sección aparte.

En general, la desventaja de usar visión monocular frente a la estéreo, es que la primera no es capaz de calcular la profundidad de la imagen a partir de una sola observación, mientras

que la segunda sí. En consecuencia, los métodos monoculares tienen problemas para estimar la escala del mapa construido. Dentro de los métodos de V-SLAM basados en visión estéreo, se puede encontrar el análogo al PTAM, el S-PTAM (del inglés, *Stereo Parallel tracking and Mapping*) [28] y el ORB-SLAM 2 [29] y 3 [30], que incorporan, entre otras características, la posibilidad de trabajar con visión estéreo, a diferencia de su predecesor, ORB-SLAM.

Para ejemplificar lo descrito durante la sección 2.1.2, la siguiente sección se abocará a explicar de manera superficial el funcionamiento del ORB-SLAM, pues es la base para el algoritmo del CCM-SLAM, como se verá más adelante.

2.1.4. ORB-SLAM

ORB-SLAM (*Oriented FAST and Rotated BRIEF - SLAM*) es un sistema basado en la extracción de características o *feature-based*; es decir, se vale del uso de detectores y descriptores ORB para la búsqueda de características invariantes a la rotación y escala en las imágenes, con las cuales realiza el seguimiento del robot y el mapeo del lugar donde este transita. Para lograr esto, se basa de las mismas características extraídas por los detectores para realizar tres tareas simultáneas: seguimiento o *tracking*, mapeo o *mapping* y cierre de lazo o *loop closure*.

En la Figura 2.2 puede verse de forma detallada las tres tareas principales del ORB-SLAM: *tracking*, mapeo local y cierre de lazo, más los módulos del mapeo global y el reconocimiento del lugar o *place recognition*.

El *tracking* se encarga de localizar la cámara en cada imagen o *frame* y decidir si dicho *frame* debe convertirse en un nuevo *keyframe* o no. Para esto, realiza una estimación inicial de la pose de la cámara en función del *frame* anterior. Una vez obtenida la estimación inicial y la unión entre correspondencias de *frames*, comienza la construcción de un mapa local a través de un gráfico de covisibilidad o *covisibility graph*. Dicho gráfico establece uniones entre *keyframes* mediante líneas, que pueden tener mayor o menor peso (o grosor de línea) según la cantidad de *map points* (puntos característicos) locales que haya en común con los distintos *keyframes*. El algoritmo luego decide si el nuevo *frame* debe convertirse en *keyframe* o no, en función de la cantidad de *map points* hallados y de cuántos *frames* pasaron desde la última inserción de un *keyframe*.

La siguiente etapa, el mapeo local, procesa los *keyframes* y lleva a cabo un ajuste de paquetes o *bundle adjustment* local, para optimizar la reconstrucción de los alrededores de la cámara. Básicamente, esta parte corresponde a la etapa del análisis de correspondencias y construcción del mapa. Aquellas nuevas correspondencias del nuevo *keyframe* son buscadas

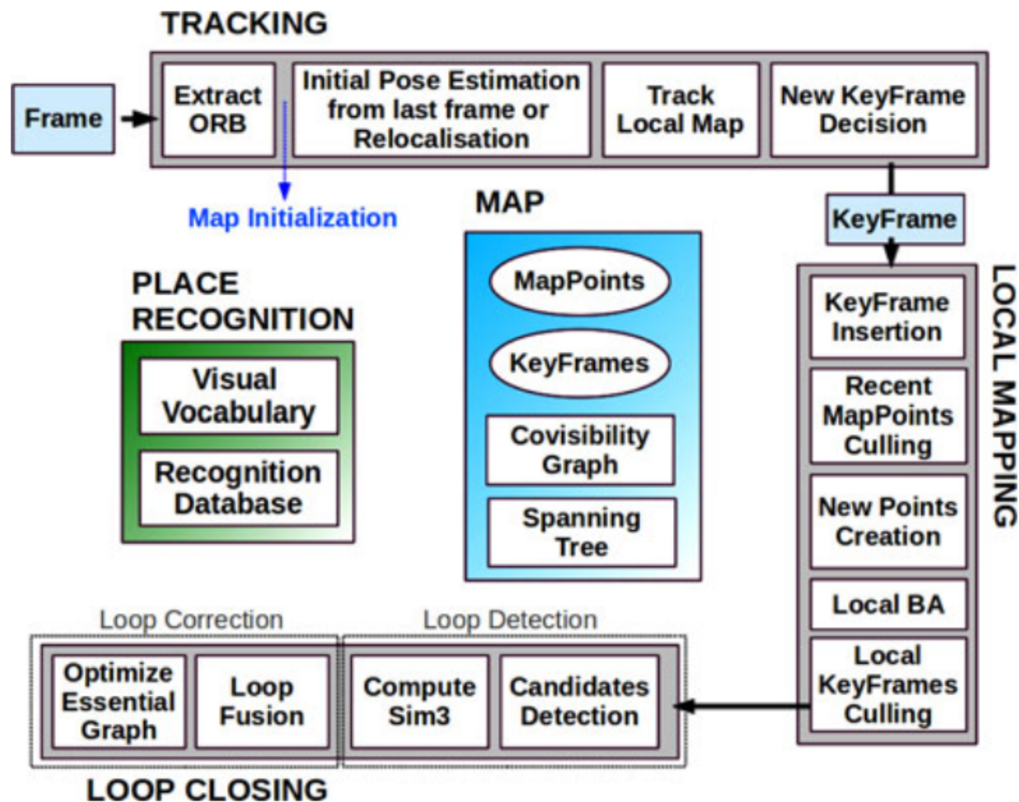

Figura 2.2: Diagrama de procesos del ORB-SLAM. Figura extraída de [27].

en los *keyframes* que ya están conectados en el gráfico de covisibilidad para triangular nuevos puntos. Luego, los puntos y *keyframes* que resulten redundantes o de baja calidad para el algoritmo, son filtrados en un proceso de “sacrificio” o *culling*. Estas actualizaciones se ven reflejadas en un árbol de expansión o *spanning tree*, que es un subgráfico del gráfico de covisibilidad, el cual conserva las uniones mínimas y necesarias para formar una especie de “esqueleto” del mapa, vinculando solo aquellos *keyframes* que tengan la mayor cantidad de puntos en común.

Luego, en la tercera parte, el cierre de lazo busca constantemente coincidencias en los nuevos *keyframes* para detectar si se está pasando por el mismo lugar. Cuando el lazo es detectado, se computa a través de una transformación cuánto es el error acumulado, para luego alinear ambos lados del lazo y fusionar los puntos duplicados. Finalmente, se realiza una optimización sobre el gráfico de estimación de las poses en función de unas restricciones de similaridad. Dicha optimización busca alcanzar una consistencia global entre todos los datos, y es realizada sobre un gráfico esencial o *essential graph*. Este gráfico engloba el de covisibilidad y el árbol de expansión, ya que retiene todos los *keyframes*, pero tiene menos uniones

que el de covisibilidad para que no sea tan denso y aún así entregue resultados precisos.

En cuanto al reconocimiento del lugar, el sistema utiliza el módulo de *Bags of Words* basado en DBoW2 [31] para detectar lazos y relocalizar al robot en caso de perderse. Estas "palabras visuales" son básicamente una discretización del espacio de descriptores ORB, en forma de vocabulario visual. De esta manera, se agiliza el procedimiento de corroborar las bases de datos para encontrar fácilmente los cierres de lazo.

Por último, el código trae la posibilidad de modificar ciertos parámetros que influyen en la manera en que se extraen las características de las imágenes a través del detector de esquinas FAST, y que posteriormente son computadas en los descriptores ORB. Estos valores se cambian en función de la calidad de las imágenes con las que se esté trabajando, pues pueden ser de distintas resoluciones, o puede ocurrir que el ambiente grabado tenga texturas poco distintivas o bajo contraste, y en consecuencia la detección de características se dificulte. Los parámetros de interés son:

- ORBextractor.nFeatures: Número de características o esquinas a buscar por imagen con FAST.
- ORBextractor.nLevels: Número de niveles en los que se escala la imagen para la búsqueda de esquinas con FAST.
- ORBextractor.scaleFactor: Factor de escala entre los niveles en los que se divide la imagen.
- ORBextractor.iniThFAST: Umbral inicial para el detector FAST. Toma este valor como inicial para definir si un píxel puede ser tomado como esquina o no.
- ORBextractor.minThFAST: Umbral mínimo para el detector FAST. Si con el umbral anterior no logró definir el píxel, toma este segundo umbral como alternativo.

Se verá que estos parámetros juegan un papel relevante en las siguientes secciones cuando se hagan las pruebas con el CCM-SLAM.

Si bien ORB-SLAM demuestra ser un algoritmo con buenas referencias en cuanto a sus resultados en el estado del arte, también es cierto que el costo computacional inherente es alto, y para poder ejecutarlo hace falta hardware muy específico y probablemente costoso. Cuando se analizan todas las tareas que la computadora debe realizar simultáneamente, como se mostraba en la Figura 2.2, resulta lógico que los requisitos mínimos de cualquier hardware embebido sean elevados, así como consecuentemente será elevado el precio de dicho hardware.

Entonces, surge la incógnita del trabajo colaborativo entre varios robots para el mapeo de un lugar, de manera tal que la memoria requerida para reconstruir un mapa extenso sea dividido entre muchos; no solo para distribuir esta carga, sino también para realizar un mapeo colaborativo más rápido y que no dependa exclusivamente de un robot. Existen extensos trabajos en el estado del arte que explotan esta posibilidad [32][33], además del que será utilizado en el presente trabajo, el CCM-SLAM [34], y cuyo funcionamiento será brevemente resumido en la siguiente sección.

2.2. CCM-SLAM

2.2.1. Introducción al *framework*

Como se anticipó en la sección anterior, la robótica colaborativa introduce un nuevo panorama en el arte del V-SLAM, debido a que la robustez de los algoritmos puede verse beneficiada al tener varios robots recolectando información y transmitiéndola entre ellos al mismo tiempo. En ese sentido, CCM-SLAM propone un *framework* de SLAM centralizado y colaborativo, donde cada robot, de ahora en más denominado agente, está equipado con una cámara monocular, una unidad de comunicación y una pequeña placa de procesamiento. La idea es que al poder aplicar cada agente odometría visual, se puede asegurar la autonomía de cada uno, mientras que paralelamente recolectan información de sus alrededores y la envían a un servidor central, quien se encarga de realizar las tareas de fusión y optimización de mapas.

De esta manera, aquellas tareas que resulten demasiado pesadas computacionalmente para los agentes, son derivadas a un servidor con mejores capacidades. Este se encargará de recolectar la información, actualizar los mapas, fusionarlos en caso de encontrar coincidencias, cerrar lazos, y transmitir eventualmente la información a los agentes en caso que sea necesario. Es decir, si uno de los agentes pasa por un lugar que ya había sido recorrido por otro agente, el servidor le descarga la información existente al primero para sus tareas de localización. Para lograr todas estas tareas, la comunicación entre agentes y servidor resulta fundamental; aunque de todas formas, si dicha comunicación fuese afectada, los agentes estarían preparados para seguir funcionando de manera autónoma.

Lo descrito en los últimos párrafos puede resumirse a continuación en las principales contribuciones que aporta CCM-SLAM al sistema de V-SLAM colaborativo:

- Arquitectura cliente-servidor eficiente: Los agentes preservan su autonomía al realizar tareas de navegación críticas sobre sus placas, mientras que las tareas más complejas vinculadas al manejo de los datos son derivadas al servidor.

- Robustez: El sistema cuenta con una estrategia de comunicación para lidiar con los anchos de banda limitados, las pérdidas de mensajes y los retrasos en la red.
- Intercambio de la información: Las experiencias recolectadas por un agente pueden ser compartidas con otros, si visitan el mismo área que el primero.
- Escalabilidad: El servidor puede encargarse de manera eficiente de los datos mapeados para remover la data redundante sin comprometer la robustez de la estimación.

2.2.2. Descripción del funcionamiento

En la Figura 2.3 se puede ver de forma resumida el funcionamiento del CCM-SLAM. Se distinguen dos módulos principales: los agentes y el servidor.

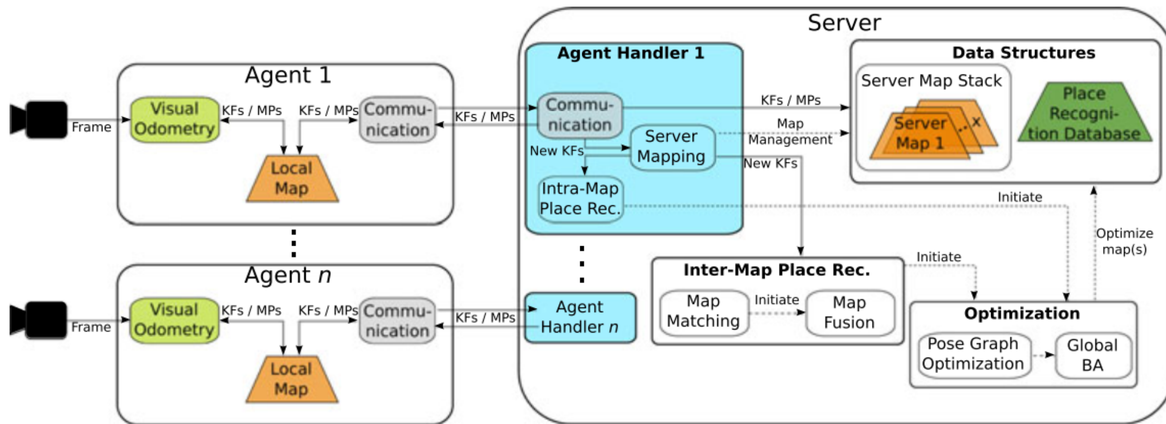

Figura 2.3: Vista general del funcionamiento de CCM-SLAM. Figura extraída de [34].

Los agentes

Cada agente ejecuta un sistema de odometría visual para estimar su pose y un mapa 3D del ambiente por el cual se desplaza. Esta odometría visual se basa en ORB-SLAM, donde básicamente los autores del CCM-SLAM adoptaron el *front-end* del mismo para utilizarlo como base de odometría visual en su código. Por tanto, el principio es similar al explicado en la sección 2.1.4; a partir de las imágenes que le llegan por cámara, la odometría estima *frame* por *frame* el movimiento de la misma con los *map points* locales, que son guardados en el mapa local. Luego, dentro del hilo de *tracking*, define si el *frame* que recibió se debe convertir en *keyframe* o no. Mientras tanto, el hilo de mapeo va procesando los nuevos *keyframes* para conectarlos con los *keyframes* ya procesados.

El mapa local representa básicamente la vecindad del agente, y hereda las mismas características que se detallaron cuando se explicó el mapeo durante la sección 2.1.4. La diferencia radica en que estos mapas tienen un límite de N *keyframes* que pueden almacenar. Conforme van almacenando nuevos *keyframes*, los más viejos se envían al servidor para que este los almacene en un *stack* de mapas. Cuando el agente recibe la confirmación de parte del servidor de que el *keyframe* fue correctamente recibido, procede a desecharlo. Naturalmente, puede ocurrir que por problemas de comunicación, no pueda recibir las confirmaciones del servidor y su memoria comience a llenarse. Para ello, cuenta con un *buffer* adicional que contiene hasta B *keyframes* más, mientras espera recuperar la comunicación. Si tanto la memoria como el *buffer* se llenan, y el agente aún no pudo enviar sus datos, comienza borrando los *keyframes* más viejos. En general, la elección del valor de N y B depende de la potencia computacional de cada agente, pero mientras más grandes sean estos valores, más robusto se volverá el sistema frente a las pérdidas de comunicación, aunque tendrá que procesar y comunicar más información a la vez.

El servidor

Por su parte, el servidor se encarga de ejecutar las tareas de: reconocimiento del lugar, optimización global de paquetes (en inglés, *bundle adjustment*) y detección de redundancia. Todas estas tareas las realiza sobre los *keyframes* que le llegan de los diferentes agentes, y que almacena en un *stack* de mapas. Por cada agente, el servidor controla un gestor de agentes (en inglés, *agent handler*), que se encarga de recopilar el mapa local de cada robot y lo apila con los demás.

Cada gestor de agentes se ocupa de la información que cada agente envía, una vez inicializado el módulo de comunicación. Para esto, inicializa un submódulo de V-SLAM, es decir, un mapa y un submódulo de reconocimiento interno del lugar para ese mapa. Cada uno de estos bloques trabaja en un hilo paralelo a los demás. En caso de producirse la fusión de mapas entre agentes, la información se junta en un solo mapa y se comparte con los gestores de agentes correspondientes.

En cuanto al mapeo del lugar, el servidor trabaja con tres módulos distintos: los submódulos de reconocimiento interno de cada gestor, la base de datos de *keyframes* y el módulo de *map matching*. Si bien estos tres últimos pueden parecer similares, cada uno tiene su propia función. El submódulo de reconocimiento interno (que en la Figura 2.3, está referenciado como *Intra-Map Place Recognition*) detecta dentro de un mapa del servidor aquellos lugares ya previamente visitados, lo que permite optimizar las poses para mejorar la precisión del mapa en general. La base de datos de *keyframes* es utlilizada por el sistema de *place recognition* del

servidor, que busca solapamientos entre los distintos mapas almacenados en el *map stack*. Y, por último, el módulo de *map matching* busca solapamientos entre dos mapas del servidor, y aplica restricciones a la búsqueda sobre esos dos mapas particulares. En caso de detectar el solapamiento, la información se envía al módulo de fusión de mapas.

En cualquiera de los tres casos mencionados arriba, el objetivo es muy similar al descrito en el apartado de ORB-SLAM. Recibe los *keyframes* nuevos y de allí los deriva a los tres módulos para establecer conexiones entre los *keyframes* y los *map points* nuevos con los existentes en cada mapa. Sin embargo, también debe aplicar una política de redundancia o rechazo, ya que de otra manera el tamaño del mapa crecería sin control. Dicha política se basa en un esquema probabilístico donde se selecciona aleatoriamente un *keyframe* y se lo compara con sus *keyframes* vecinos en el mapa de covisibilidad, chequeando los *map points* observados. Si el valor umbral de *map points* son observados en otros tres *keyframes* vecinos, entonces el *keyframe* elegido aleatoriamente se considera redundante y es eliminado.

En caso de que el módulo de *place recognition* detecte una coincidencia entre dos mapas del servidor, o en caso de que dentro de un mapa ocurra un cierre de lazo, se lleva a cabo un ajuste de haces global o *bundle adjustment* global. Este paso optimiza el mapa respectivo al minimizar el error de proyección para todos los *keyframes* o *map points*, mejorando la precisión del gráfico y reduciendo el error de deriva por la escala. Es importante recordar que, al estar trabajando con visión monocular y no estéreo, la escala del mapa no puede estimarse porque no se tiene medida sobre la profundidad a partir de una foto. En consecuencia, el cierre de lazo es crucial para reducir el error asociado a la deriva de la escala. Antes de realizar el *bundle adjustment*, la optimización previa del gráfico de poses es realizada sobre un subgráfico del gráfico de covisibilidad, conocido en ORB-SLAM como gráfico esencial o *essential graph*, el cual solo usa los bordes más importantes o con mayor peso de correspondencias, para mejorar los tiempos de ejecución. Además, durante esta tarea, el servidor deja de procesar los *keyframes* entrantes, por lo cual son almacenados en *buffers* para ser procesados luego de finalizada la tarea de *bundle adjustment*.

La comunicación

Para la comunicación entre agentes y servidor, CCM-SLAM hace uso del sistema de comunicación ROS (*Robot Operating System*), el cual provee una capa de comunicaciones estructurada [35] a través de una red inalámbrica. El servidor crea un nodo maestro, que a su vez tiene canales vinculados a los nodos clientes (los agentes, en este caso). El *framework* no asegura una comunicación en tiempo real, aunque esto no es necesario, pues tanto servidor como agente no esperan recibir información del otro a un tiempo específico.

Desde el agente, el módulo de comunicación almacena los *keyframes* y *map points* llevando control sobre los cambios constantes del mapa local. Según el caso, puede comunicarle al servidor información nueva o puede enviarle actualizaciones a una frecuencia configurable. En el primer caso, los nuevos *keyframes* y *map points* son enviados junto con los *keypoints* asociados extraídos de las imágenes, así como los descriptores de sus características. En el segundo caso, cuando se envían actualizaciones, para los mismos *keyframes* registrados pero ahora trasladados, sus respectivos *keypoints* y descriptores ya habían sido calculados, así que solo es necesario enviar la actualización de las nuevas poses de los *keyframes* en el mapa. Este envío de actualizaciones busca evitar enviar información ya conocida para no sobrecargar el tráfico de la red. Para asegurar que el mensaje llegó bien al servidor, este le devuelve con cierta un frecuencia cuáles *keyframes* fueron procesados por el mismo. En caso de detectar que el *keyframe* anterior al procesado no tuvo mensaje de recibo, se vuelve a enviar la información, asumiendo que no llegó bien en primer lugar.

Por parte del servidor, en cada mensaje que le envía a los agentes, este contiene los k *keyframes* con mayor peso en el gráfico de covisibilidad relativos a un *keyframe* de referencia, y también envía sus *map points* asociados, para que los agentes aumenten el tamaño de su mapa local. La frecuencia de publicación de estos mensajes está limitada, de manera tal que el ancho de banda pueda ser limitado a los requisitos mínimos, así como para el esfuerzo computacional de cada agente. Ajustando el valor de k y de la frecuencia de publicación, el tráfico de información puede ajustarse al ancho de banda máximo disponible en la red. A diferencia de la comunicación agente-servidor, en este caso, el envío correcto de los mensajes del servidor al agente no deben ser corroborados necesariamente.

En caso de pérdida de comunicación, el agente dejará de recibir información del servidor para mejorar su mapa local y también dejará de corroborar que su información llegó adecuadamente al servidor. En el primer caso, el agente se puede valer de su propia odometría visual para continuar trasladándose de manera autónoma por el lugar; mientras que en el segundo caso, la nueva información que cree la almacenará en un *buffer* con un tamaño máximo, y en caso de ser desbordado, procederá a borrar los *keyframes* más antiguos.

En resumen, además de los parámetros asociados a la odometría visual heredados de ORB-SLAM, descritos en la Sección 2.1.4, ahora también se cuenta con los parámetros propios de CCM-SLAM vinculados a la frecuencia de comunicación agente-servidor, a saber:

- Comm.Client.PubFreq: Frecuencia [hz] a la cual el agente le publica nueva información del mapa local al servidor.

- Comm.Server.PubFreq: Frecuencia [hz] a la cual el servidor envía datos a sus agentes

para que aumenten o actualicen su mapa local.

Además, existe otro parámetro que será de relevancia en los siguientes capítulos, propio de CCM-SLAM:

- Mapping.LocalMapSize: Son los *n keyframes* que componen el tamaño del mapa del agente.

En las siguientes secciones se detallará la metodología experimental que se llevará a cabo, no solo para la implementación de CCM-SLAM sino también para la medición de los recursos consumidos, y allí se pondrá en juego la relevancia de los parámetros descritos en esta sección.

Capítulo 3

Metodología experimental para la evaluación del CCM-SLAM

El presente capítulo detallará la metodología a utilizar para la evaluación del CCM-SLAM: desde el dataset a utilizar, pasando por la instalación del código con sus dependencias correspondientes, hasta los valores utilizados en los parámetros configurables descritos en el capítulo anterior. También se describirá la estrategia a seguir para la medición de recursos consumidos por el algoritmo, que posteriormente ofrecerá información sobre los principales cuellos de botellas que puede presentar. Finalmente, atravesadas las pruebas preliminares y análisis correspondientes, se brindará un avance de cómo se trabajará con las placas de hardware embebido para simular las pruebas en campo.

3.1. Dataset a utilizar

La presente sección tiene por objetivo introducir el conjunto de datos experimentales, es decir, el dataset, con el que se realizaron los ensayos a lo largo del trabajo. Este dataset consiste en un conjunto de imágenes que el algoritmo utilizará, más los datos provistos por el GPS para tomarlos como referencia. Se detallará cómo se hicieron las capturas de imágenes y qué cámaras fueron utilizadas. Posteriormente se explicará con cuál de las versiones adquiridas se eligió trabajar y porqué, qué post-procesamientos se realizaron, y qué dificultades surgieron durante su adquisición, que luego tuvieron influencia en los resultados.

Figura 3.1: Vista aérea de la chacra en Cipolletti. En rojo se resalta el área donde se grabó el dataset.

3.1.1. Chacra en Cipolletti

Este dataset fue generado en una chacra ubicada sobre Calle Rural A15, en la ciudad de Cipolletti, misma chacra donde reside el codirector del presente trabajo. En la Figura 3.1 se puede apreciar una vista aérea de la zona donde se generó el dataset, remarcada en rojo, dedicada al cultivo de manzanas.

Se adquirieron dos versiones para el mismo recorrido: uno utilizando una cámara monocular Logitech C525 de 8 MP y otro utilizando una cámara estéreo Zed Stereolabs. En ambos casos, se tomaron en paralelo mediciones de un GPS para utilizarlas posteriormente como *ground truth* de la posición de la cámara.

Debido a que CCM-SLAM solo trabaja con visión monocular, en el caso de la grabación con la Zed Stereolabs, se tomó solo una de las imágenes (izquierda, en este caso) para simular la visión monocular. En cuanto a los *frames per second* (fps), en este caso resultaron irrelevantes pues, como se verá más adelante, es posible a partir de las imágenes reconstruir el dataset a diferentes fps.

Para el caso de la Logitech, se tomaron 14111 imágenes, con una resolución de 640x480. Para los parámetros intrínsecos de la cámara, se utilizó la siguiente matriz, cuyos valores fueron extraídos de un paper [36]:

$$K = \begin{pmatrix} 820{,}2028 & 0 & 255{,}4357 \\ 0 & 819{,}97 & 222{,}3254 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.1}$$

Además de un vector k para las distorsiones radiales:

$$[k_1, k_2] = \begin{pmatrix} 0{,}0378 & -0{,}3324 \end{pmatrix} \tag{3.2}$$

En cuanto a la Zed, las 9201 imágenes grabadas tuvieron una resolución de 672x376, luego de quedarse con la imagen izquierda. Para los parámetros intrínsecos, se consideraron los valores empleados en un proyecto integrador profesional anterior [37] que también formó parte del grupo de investigación en el cual se enmarca el actual trabajo. Dado que en dicho trabajo se utilizó la misma cámara, los parámetros intrínsecos resultaban ser los mismos. Sin embargo, los valores tuvieron que adaptarse a la resolución de la imagen del presente dataset, como se comentará más adelante, resultando:

$$K = \begin{pmatrix} 363{,}7842 & 0 & 342{,}9634 \\ 0 & 363{,}7842 & 191{,}8737 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.3}$$

También se extrajo, del mismo proyecto integrador profesional mencionado, el siguiente vector de distorsiones radiales k:

$$[k_1, k_2] = \begin{pmatrix} -0{,}172019 & 0{,}023772 \end{pmatrix} \tag{3.4}$$

Y un vector de componentes de distorsión tangencial, p:

$$[p_1, p_2] = \begin{pmatrix} 0{,}00019359 & 1{,}76187114x10^{-5} \end{pmatrix} \tag{3.5}$$

3.1.2. Elección del dataset a utilizar

Para la realización de todas las pruebas que siguen se utilizó el dataset grabado con la cámara Zed; es decir que a partir de ahora en adelante, salvo que esté especificado, se entenderá que cada vez que se refiera al “dataset”, se hablará del que fue adquirido con esta

Figura 3.2: Imágenes del dataset de la chacra en Cipolletti, tomadas con la cámara Zed Stereolabs (imagen izquierda).

cámara. De acuerdo a los registros obtenidos con el GPS, la longitud total de la trayectoria fue de 369 metros aproximadamente, realizando el recorrido remarcado en rojo de la Figura 3.1, luego de cerrar la vuelta alrededor de las hileras. En la Figura 3.2 se pueden ver algunas de las imágenes captadas durante la grabación del mismo.

El dataset obtenido con la Logitech, si bien fue grabado en las mismas condiciones que con la Zed, presentó algunos problemas que imposibilitó su utilización para los propósitos de este trabajo y por ende no formó parte de los resultados que se enseñan en el Capítulo 4. En el siguiente apartado se darán más detalles de por qué esto fue así. Finalmente, en la Figura 3.3 pueden verse algunas de las imágenes que se tomaron durante la grabación con la Logitech.

3.1.3. Dificultades en el dataset

Es importante resaltar el hecho que ambos datasets fueron tomados con las respectivas cámaras colocadas sobre un tractor; en consecuencia, todas las vibraciones propias del tractor fueron transmitidas a la cámara, haciendo que esta vibrara y transmitiera ese movimiento a las imágenes. La intensidad de las vibraciones en el tractor fueron mayores de lo previstas y dificultaron el trabajo, pues los algoritmos de V-SLAM tienen problemas para lidiar con movimientos súbitos.

Otra problemática que surgió fue debido a que no hubo suficiente solapamiento entre el

Figura 3.3: Imágenes del dataset de la chacra en Cipolletti, tomadas con la cámara Logitech.

inicio y final de la grabación. El dataset consistía en dar la vuelta a dos hileras del cultivo en la chacra, de manera tal que inicio y final del recorrido coincidan, y el algoritmo detecte esto para cerrar el lazo apropiadamente. Sin embargo, al cortarse la grabación apenas habiendo llegado al punto de inicio, resultó que eran muy pocos los *frames* en común en el inicio y final. Esto hace que el algoritmo deba trabajar con muy poco margen de error, ya que de otra forma no logra cerrar el lazo.

Es importante recordar que los algoritmos de V-SLAM presentan dificultades bajo tres condiciones:

1) En caso de movimientos súbitos o que producen imágenes borrosas por falta de pulso o estabilidad de la cámara. Esto lleva a que una característica detectada no pueda ser fácilmente reconocida en las siguientes imágemes.

2) En ambientes con objetos muy dinámicos, como objetos rígidos o personas en movimiento constante. Esto puede desencadenar en falsos positivos que acarrean errores a todo el sistema.

3) En lugares con patrones repetitivos, es decir, con texturas muy similares. Esto dificulta la búsqueda de esquinas o características en la imagen.

En función de estas tres condiciones descritas, se podría decir que los datasets cumplen con al menos dos de las tres características que dificultan el rendimiento de los algoritmos de V-SLAM. Las vibraciones transmitidas del tractor a la cámara generaron imágenes borrosas o con movimientos repentinos, y, por la característica propia de una hilera de cultivos frutícolas, es lógico que las texturas sean similares, pues se está trabajando con hileras de árboles solamente.

Ahora bien, estas condiciones son inherentes a los dos datasets con las dos cámaras. El problema que surgió con la Logitech es que por un lado presentó problemas de *rolling shutter*. El *rolling shutter* es una característica propia de los métodos de captura de imagen realizados con sensores CMOS. Cuando la velocidad del obturador para capturar la imagen no es lo suficientemente rápida comparado con el movimiento presente en la escena a capturar, hace un barrido que captura una escena en dos o más tiempos distintos de una sola vez. Esto genera una distorsión en el resultado, ya que no todas las escenas presentes corresponden al mismo momento.

Por otro lado, durante el primer giro de la trayectoria con la Logitech, sucedió un efecto de “encandilamiento” donde el sol dio directo en la cámara, “enceguecíéndola”, por un montículo de tierra que hizo que el tractor se elevara. Esta condición, sumada a la del *rolling shutter*, no se presentaban en el caso de la Zed, y considerando las problemáticas mencionadas al inicio de este apartado, resultó imposible lograr que el algoritmo no se perdiera durante el primer giro de la trayectoria, por más que se intentaran reiteradas pruebas variando los parámetros relacionados a la detección de características de la imagen. Consecuentemente, el dataset tuvo que ser descartado y se trabajó únicamente con el correspondiente a la Zed.

3.1.4. Tratamiento del dataset

Para realizar las pruebas, es necesario darle un tratamiento al dataset luego de su adquisición para que sea procesado por CCM-SLAM. Debido a que CCM-SLAM trabaja con ROS, es necesario agrupar las imágenes bajo el formato .bag. Este formato, propio de ROS, consiste en englobar las imágenes en una “bolsa de imágenes”, comúnmente denominada *rosbag*, donde se especifica el formato de las imágenes (.png, .jpg, etc.) y los fps que tendrá la grabación. Esto da la libertad de especificar en el archivo *rosbag* los fps con los que se querrá reproducirlo más tarde, ya que en función de este parámetro, se ajusta la duración total del dataset. Para la generación del *rosbag*, se utilizó un algoritmo de código abierto en Github [38] recomendado por los autores del CCM-SLAM.

3.2. Metodología de la experimentación con CCM-SLAM

Para la experimentación con CCM-SLAM, primero se preparó el entorno de trabajo y se aseguró que éste funcionase correctamente. Para ello, se utilizó el sistema operativo Linux, ya que dentro del estado del arte de los algoritmos de V-SLAM, Linux es el sistema mayormente utilizado, y en consecuencia es más probable encontrar soporte para los problemas que surjan. Además, para la instalación de CCM-SLAM, fue necesario tener en consideración todas las bibliotecas/programas dependientes para su uso, tales como son OpenCV y ROS. Para este trabajo, se utilizó la versión 4.2.0 de OpenCV y la versión Noetic de ROS.

Las siguientes subsecciones detallan, en primer lugar, cómo se prepararon y configuraron las herramientas que participaron en los experimentos para una ejecución exitosa del dataset; luego, se presenta el formato de los experimentos realizados en laptops, y, a continuación, en placas de hardware embebido.

3.2.1. Preparación del ambiente para la experimentación

Pruebas preliminares

En primer lugar, se realizaron pruebas previas a la medición de recursos para entender el funcionamiento del CCM-SLAM, es decir, cómo funciona la arquitectura agente-servidor, cómo reacciona el algoritmo ante el mismo dataset pero variando la forma en que se crea el *rosbag*, y cómo se representan las poses estimadas al final del procesamiento, ya que esa es la información que se utilizó para comparar los resultados obtenidos frente a un *ground truth*.

Otro objetivo en estos ensayos preliminares era determinar cuáles eran los parámetros que aseguraban un cierre de lazo en el algoritmo, y consecuentemente una buena estimación de la trayectoria. Estos parámetros son aquellos relacionados a la detección de características en la imagen, es decir, aquellos que fueron introducidos junto con ORB-SLAM en la Sección 2.1.4.

Como se mencionó en la Sección 3.1.3, la falta de solapamiento entre inicio y final del dataset resulta en que el mapa debe inicializarse rápidamente para poder cerrar el lazo. De esa manera, se pierden la menor cantidad de *keyframes* iniciales posibles, y así se asegura que haya superposición suficiente entre *keyframes* iniciales y finales, que se detecte el lazo y el servidor lo cierre adecuadamente.

El algoritmo, para comenzar a mapear, primero debe realizar una localización y estimación inicial de la pose de la cámara. Este proceso de inicialización puede tomar más o menos tiempo, en función de los parámetros seleccionados de ORB-SLAM para la extracción de

características. Hasta que esta inicialización no ocurra, el mapa no guardará ningún *keyframe* inicial, que son los que precisamente necesita para luego cerrar el lazo cuando los *keyframes* finales se superpongan con los iniciales. El objetivo de las pruebas preliminares es en parte también buscar qué valores aseguran que esto suceda lo antes posible.

Es importante tener presente además que CCM-SLAM solo brinda información sobre las poses estimadas una vez que se ejecuta el *bundle adjustment* global; de otra manera, no aplica la transformación correspondiente sobre los puntos estimados y no genera ningún archivo a la salida. Como se mencionó en la Sección 2.2, para que esto suceda, es necesario que haya fusión de dos mapas o un cierre de lazo. Como en las primeras pruebas no es de interés probar la fusión entre dos mapas (es decir, entre dos agentes), asegurar el cierre de lazo es crucial, no solo para corregir el error asociado a la escala, sino también para obtener las poses estimadas y consecuentemente poder graficarlas.

Una vez conseguido el cierre de lazo, se debe contar con un *ground truth* para comparar los resultados obtenidos. Como ya se había comentado, se utilizarán los datos de GPS para dicho fin. Para la comparación entre ambos conjuntos de puntos, los autores de CCM-SLAM recomiendan en su Github el uso de una herramienta denominada *evo* [39], la cual consiste en una biblioteca de Python para la evaluación y comparación de trayectorias obtenidas a partir de algoritmos de odometría y SLAM.

Análisis preliminar para la medición de recursos

En esta etapa del trabajo se definió qué recursos se requiere medir y evaluar. A grandes rasgos, dentro de una computadora se pueden divisar cinco clásicos componentes: la entrada, la salida, la memoria, el camino de datos y la unidad de control, siendo estos dos últimos en conjunto lo que se conoce como procesador [40]. En general, las características más restrictivas en una computadora, tanto por su costo como por su escalabilidad, suelen ser el procesador (CPU) y la memoria. Por tanto, esas son las principales variables de interés a medir.

Por otra parte, el CCM-SLAM explora otra variable que en los algoritmos tradicionales de V-SLAM no es tomada en cuenta: el ancho de banda. Al trabajar de manera colaborativa comunicándose a través del pasaje de mensajes, resulta lógico que será necesario trazar una estrategia para la medición del consumo de red.

Linux, al ser un clon del sistema operativo UNIX, tiene como objetivo el cumplimento de las especificaciones definidas por POSIX y la Single UNIX Specification [41]. Por tal motivo, cuenta con una serie de utilidades (programas del sistema) propias de todo sistema operativo de tipo UNIX, que permiten realizar el seguimiento del uso de los recursos. También

permiten almacenar la información registrada, para un procesamiento y análisis posterior. Dentro de ese análisis posterior, se debe identificar a la medición registrada que corresponda al proceso específico de interés, pues el programa toma registro de todos los procesos que se estén ejecutando en la computadora. Por poner un ejemplo, al medir el consumo de ancho de banda, se debe tener en cuenta que hay un tráfico constante de información debido a otros procesos ajenos al CCM-SLAM, pero en general, los programas que monitorean la red no se enfocan tanto en qué aplicación se lleva qué porcentaje del consumo, ya que sería un análisis más minucioso.

Cada proceso dentro del sistema tiene asignado un ID, comúnmente llamado PID (*Process ID* o ID del Proceso), pero a su vez ese mismo proceso podría generar varios procesos hijos dependientes de ese primero, cada uno con sus propios consumos y propios ID. En ese caso, habría que discernir si el proceso de interés a medir es el proceso padre o alguno de los hijos, algo que no es tan simple a primera vista. Tomando otro ejemplo, para ejecutar todo el sistema CCM-SLAM es necesario disponer de varias terminales Linux (intérpretes de comandos) al mismo tiempo, para ejecutar en cada una diferentes componentes de todo el sistema. Una terminal es para iniciar el nodo maestro de ROS, otra para iniciar al servidor, otra para iniciar a un agente, y otra para reproducir el *rosbag* (el dataset) en el agente asociado. Cuando estas terminales se inician con el comando correspondiente, en paralelo se inicializan varios procesos hijos vinculados a esa ventana, y resulta que el proceso que más consume memoria es uno que se deriva de insertar el comando en la terminal, no del comando en sí.

En síntesis, el propósito de lo anterior es demostrar que no es una tarea trivial medir los recursos utilizados por un proceso específico. En efecto se sabe qué se quiere medir, pero a qué proceso se quiere medir no es tan cierto. Sin embargo, si se pudiese localizar el número del ID asociado al proceso de interés, esto se podría usar para localizarlo entre todos los demás procesos y así efectuar las mediciones. Por ejemplo, Linux cuenta con los comandos *top* o *htop* que detallan cada proceso padre con sus procesos hijos y muestran el consumo de memoria RAM de cada uno de ellos. También se puede acceder al árbol de procesos mediante el comando *tree*. De esta manera, haciendo una prueba del CCM-SLAM, se puede analizar mientras se ejecuta, cuáles son los procesos más relevantes y cómo se identifican, para luego pasar su nombre respectivo a la función que haga la medición.

Hasta este punto, queda definido qué se quiere medir y a qué proceso; resta determinar el cómo. Para el caso de la memoria RAM y el CPU, se hace uso de los programas *ps* y *top*, respectivamente. La primera, *ps*, hace alusión a las siglas PS por "*Process Status*", o estado del proceso, y es un programa del sistema Linux que ofrece información relacionada a los procesos de un sistema; en este caso, el uso de memoria RAM. La segunda, *top*, muy similar

a la primera, es un comando de Linux que muestra los procesos o hilos que están siendo controlados por el kernel de Linux. También ofrece un resumen de la información en tiempo real; la diferencia entre ambas está en cómo se mide el consumo de CPU.

Para *ps*, la medición de CPU se basa en el uso acumulado desde que el proceso se ejecutó por primera vez, y promedia el total sobre el tiempo para obtener un porcentaje como medida. Por otro lado, *top* reporta el promedio de uso del CPU desde la última vez que fue muestreado, es más bien una foto instantánea del estado del CPU en cada muestreo. Por este motivo, resulta más apropiado el comando *top* para medir el consumo de CPU, ya que un acumulado no brinda información de los picos y promedio de consumo a lo largo de la prueba.

En cuanto al ancho de banda, la búsqueda de una herramienta que mida los datos enviados y recibidos por un proceso en particular fue más específica, ya que generalmente las herramientas de monitoreo analizan el tráfico por protocolo o por subcapa. Sin embargo, en este caso resulta de interés medir todo el tráfico relacionado a un proceso único, sin desmenuzar los paquetes por protocolo. Para ello se hace uso de la herramienta *nethogs* la cual se define como una "herramienta *top* para redes" [42], donde la medición de ancho de banda se hace por ID de proceso. El código fuente de esta herramienta se modificó en colaboración y bajo la guía del cotutor de este trabajo para que, además de reportar el ancho de banda utilizado por el proceso de interés, coloque una marca del tiempo, necesaria para su procesamiento posterior.

3.2.2. Configuración de los experimentos para la medición de recursos

Según lo definido en el apartado anterior para realizar las mediciones durante las pruebas, se crearon una serie de *scripts* que ejecutan las líneas de comando necesarias para ejecutar los comandos *ps, top* y *nethogs*. Estos *scripts* se prepararon con el fin de poder especificar a qué procesos se le efectuarán las mediciones, y en qué formato se grabarán las mismas. Los datos recolectados son guardados en archivos de texto plano, para ser posteriormente analizados.

Para reconocer cuáles son los procesos de mayor relevancia, primero se midieron todos aquellos procesos relacionados al funcionamiento del CCM-SLAM, y a partir de allí se filtraron los que representaran el mayor consumo computacional. Esto es principalmente porque el algoritmo ejecuta muchos procesos hijos, y con el propósito de luego graficar la información, resultaba necesario aplicar algún filtrado sobre los datos obtenidos. En consecuencia, se detectaron seis procesos en ejecución relativos a todo el sistema CCM-SLAM. De los mismos, se seleccionaron únicamente dos, un proceso de parte del servidor y otro de parte del agente. Los otros cuatro procesos que se descartaron, fueron procesos que presentaron un uso constante y poco relevante de recursos durante todo el tiempo de vida del proceso. Por ejemplo,

tres de esos cuatro procesos tenían un promedio de uso de CPU/RAM que representaba entre el 1 y el 2 % del promedio máximo de CPU/RAM medido correspondientemente. En cuanto al cuarto proceso, su consumo de memoria promedio representaba el 10 % del máximo promedio medido, mientras que el CPU promedio representaba el 6 % del máximo. Este proceso se descartó de todas formas por estar relacionado a la gráfica de la trayectoria durante el experimento, algo que no es inherente al funcionamiento del CCM-SLAM, sino que tiene como propósito hacer seguimiento al desarrollo del mismo.

Para las pruebas, se hizo uso de dos computadoras laptop, una simulando ser el servidor y otra el agente. En el caso de la computadora servidor, esta contaba con un procesador AMD Ryzen 9 5900HS @3.30 GHz y alrededor de 10 GB de RAM (dado que se trataba de una máquina virtual donde se ejecutaba Linux); mientras que la computadora que simulaba ser agente contaba con un procesador Intel® Core™ i3-4005U @1.70GHz y 8 GB de RAM. Siendo que ambas computadoras están conectadas a la misma red Wi-Fi, se inicializa la comunicación entre ambas mediante nodos ROS, especificando a las computadoras cuál es su dirección IP propia y cuál es la dirección IP del nodo maestro (servidor).

En cuanto a las configuraciones de los parámetros del algoritmo, primero fue necesario encontrar aquellos valores que aseguraran una buena ejecución del algoritmo sobre el dataset, es decir, que proporcionaran un cierre de lazo exitoso y la estimación de una trayectoria similar a la efectivamente realizada en el campo. Para ello, se emplearon los siguientes valores:

- ORBextractor.nFeatures: 2000
- ORBextractor.nLevels: 8
- ORBextractor.scaleFactor: 1.2
- ORBextractor.iniThFAST: 10
- ORBextractor.minThFAST: 7
- Mapping.LocalMapSize: 50

Estos valores ofrecen las bases del análisis de los cuellos de botella, ya que con varias mediciones, se puede analizar el consumo de RAM y CPU por parte de agente y servidor, y consecuentemente intuir qué placas de hardware embebido serían capaces de soportar las exigencias del CCM-SLAM. Si bien también podrían variarse y ver qué ocurre, por ejemplo, con 2000 o 5000 *features* por imagen, en estos casos se tuvo que limitar bastante los extremos de análisis por el problema que tuvo el dataset al ser grabado, como se detalló durante la Sección 3.1.1.

Entonces, con el propósito de querer sacarle provecho a los experimentos, se decidió variar los parámetros de frecuencia de comunicación entre agente y servidor, no solo para comparar con los resultados del paper de los autores del CCM-SLAM, sino también para evaluar las implicancias que tenían los valores elegidos sobre el consumo de ancho de banda en servidor y cliente. Esto permitiría estimar las características que una red inalámbrica debería cumplir en el campo para asegurar una comunicación apropiada entre ambas computadoras. Con ese fin, se proyectaron tres pruebas, cada una con los parámetros indicados por el Cuadro 3.1.

Prueba	Comm.Client.PubFreq [Hz]	Comm.Server.PubFreq [Hz]
1	0.5	0.1
2 (predeterminado)	5	1
3	500	100

Cuadro 3.1: Valores de frecuencia de comunicación a usar durante las mediciones

De los valores elegidos, la Prueba 2 del Cuadro 3.1 tiene aquellos que vienen de forma predeterminada por CCM-SLAM. Es decir que a partir de los valores por default, se los disminuyó 10 veces para la Prueba 1 y se los aumentó 100 veces para la Prueba 3. En el caso de la Prueba 1, solo se disminuyó 10 veces respecto a los valores originales ya que con una disminución de 100 veces la comunicación traía problemas al rendimiento del programa.

En lo que respecta a la duración del dataset, para la forma en que se construyó el archivo *rosbag*, la misma es de 10 minutos aproximadamente. Esto hizo que cada experimento tuviese que planearse con cuidado, ya que una vez iniciado, si se perdía el tracking o si los valores de los parámetros habían sido mal ingresados, había que repetir todo de nuevo. Para construir el archivo .bag se utilizaron los 9201 *frames* y se grabó con 15 fps usando imágenes en formato .png, resultando un archivo con un peso final de alrededor 7 GB. Todos estos detalles tuvieron que ser considerados a la hora de probar el CCM-SLAM usando placas de hardware embebido como agente, tal como se verá en la siguiente sección.

3.2.3. Pruebas en placas de hardware embebido

Una vez corroborado el correcto funcionamiento de CCM-SLAM en las computadoras y realizadas las mediciones, se procedió a la siguiente etapa de pruebas. Estas consistieron en evaluar qué tan bien podían reproducirse los resultados de la sección anterior en las placas de hardware embebido, considerando los análisis basados en las mediciones de recursos. Para este caso, se contó con dos Raspberry Pi Modelo 3B, con 1 GB de RAM cada una.

Las Raspberry Pi disponían de un sistema operativo Debian 10 Buster instalado, única versión de Debian compatible con la versión Noetic de ROS. La instalación de CCM-SLAM presentó dificultades debido a la poca RAM que disponían las placas; 1 GB de RAM no era suficiente para compilar todo el programa con sus dependencias. En consecuencia, se hicieron algunas modificaciones, como añadir 2 GB de memoria *swap*. Sin embargo, aún con los GB de memoria añadidos, la compilación del CCM-SLAM junto con sus dependencias tomó entre 8 y 9 horas.

También se tuvieron que reducir la cantidad de paquetes de dependencias a instalar, ya que el espacio disponible en disco comenzó a representar una restricción también. Se contaba con menos de 8 GB para la instalación de estos paquetes, dado que parte de esos 8 GB estaban ocupados por el sistema operativo. Por tanto, solo se instalaron aquellos paquetes indispensables para el funcionamiento desde la terminal de Linux, y, en el caso del dataset de 7 GB, se añadió espacio con otro USB montado a la Raspberry Pi.

El procedimiento para probar los datasets fue muy similar al descrito anteriormente entre dos computadoras, solo que ahora una laptop hizo de servidor y la placa de agente. Una de las ventajas que tiene ROS para trabajar con los datasets en formato .bag, es que a la hora de reproducirlos, se puede especificar con que velocidad se quiere enviar la información. Por ejemplo, si al reproducir el dataset se le añade el parámetro “-r 0.5”, entonces el dataset transcurrirá a la mitad de la velocidad, por lo que si originalmente duraba 5 minutos, ahora durará 10.

Debido a las limitaciones de las Raspberry Pi y al problema de falta de solapamiento del dataset ya descrito, la velocidad con que se reprodujo el dataset fue reducida 4 veces respecto a la velocidad original con la que se probó en las PCs, extendiendo así la duración normal de 10 a 40 minutos, para que el mapa se pueda construir exitosamente. Más adelante se detallará una estrategia aplicada para aumentar la velocidad a la mitad de la original y aún así obtener resultados positivos.

En cuanto a los parámetros usados para las pruebas con las Raspberry, de manera análoga a lo enseñado antes, se utilizaron para los ensayos los siguientes valores:

- ORBextractor.nFeatures: 3000
- ORBextractor.nLevels: 12
- ORBextractor.scaleFactor: 1.2
- ORBextractor.iniThFAST: 7
- ORBextractor.minThFAST: 2

- Mapping.LocalMapSize: 100

Comparado a las pruebas en computadora, se aumentaron el número de características por imagen y los niveles de división para la extracción, con el fin de incrementar la cantidad de información y evitar que el *tracking* se pierda. También se bajaron los umbrales de detección del FAST, para alivianar las exigencias sobre la placa a la hora de detectar las esquinas. Los valores de frecuencia de comunicación entre agente y servidor se dejaron en default, que son los correspondientes a la Prueba 2 del Cuadro 3.1.

Capítulo 4

Resultados y discusiones

Este capítulo expondrá los resultados obtenidos en función de las metodologías planteadas durante el capítulo anterior. Primero se mostrará un resultado general obtenido durante las pruebas preliminares, que corrobora el buen funcionamiento del algoritmo. Luego se enseñarán los resultados pertenecientes a las pruebas con medición de recursos consumidos, que son los experimentos base para analizar la viabilidad de que se pueda replicar lo mismo en las Raspberry Pi. En paralelo, el análisis sobre los consumos de CPU, RAM y ancho de banda brindarán una idea de cuáles son los requisitos mínimos de funcionamiento, además de qué condiciones debe cumplir la red inalámbrica que vincule el servidor con el o los agentes. Finalmente, en la última sección, se presentarán los resultados de ejecutar el algoritmo tomando las Raspberry Pi como agentes, incluyendo el caso de tener dos agentes, cada uno representado por una placa, y cómo se da la fusión de mapas en el servidor.

4.1. Pruebas preliminares

Hubieron muchos factores que pusieron en juego la funcionalidad del algoritmo, dado que al principio éste se perdía con facilidad, no pudiendo cerrar el lazo después, ya que no lograba detectar suficientes características entre imágenes para poder hacer el seguimiento de la posición de la cámara. Por ejemplo, la influencia de los parámetros intrínsecos de la cámara fue clave a la hora de hacer funcionar correctamente CCM-SLAM. En trabajos anteriores del grupo de investigación, al trabajar con la misma cámara Zed, se utilizaban una serie de parámetros que pertenecían a una resolución de imagen en particular. En este caso, al ser la resolución de la imagen distinta, este conjunto de valores ya no era más válido. Por tanto, se los adaptó a la nueva resolución mediante la multiplicación de un factor de 0.5, permitiendo

así que el algoritmo estimara correctamente las poses de la cámara en cada *frame*. Este factor proviene básicamente de hacer la relación entre las resoluciones nuevas y las anteriores asociadas a los parámetros intrínsecos originales.

Luego, una práctica común en estos algoritmos es que cuando las características en las imágenes del dataset resultan difíciles de detectar (ya sea por la falta de texturas o por el movimiento de la cámara) se suele incrementar el número de características ORB a extraer (parámetro conocido como *ORBextractor.nFeatures*). Para esta situación, esto no resulta conveniente, pues no solo ralentiza el proceso de *tracking*, sino que también hace que el tiempo de inicialización del mapa se extienda. Esto se traduce en una pérdida de la información inicial, que luego impide lograr el cierre de lazo adecuadamente por falta de solapamiento. Como se explicó en la sección de la configuración de los experimentos, un valor de *nFeatures* entre 2000 y 3000 sirve para una buena relación de compromiso entre el tiempo de inicialización y un *tracking* adecuado de la cámara.

Finalmente, gracias a la ventaja que tiene trabajar con *rosbags*, se configuraron a los mismos de manera tal que la reproducción del dataset siempre fuese a 15 fps. Si bien se hicieron pruebas con el dataset a 5, 7 y 20 fps, no se identificaron mejorías en el rendimiento del algoritmo, sobre todo con fps menores a 15.

Lo detallado en los párrafos anteriores pone en evidencia la gran cantidad de pruebas previas que debieron realizarse para entender cómo adaptar el algoritmo al dataset objetivo.

Para ejemplificar el formato de los resultados, en la Figura 4.1 se presenta con color azul sólido la trayectoria luego del cierre de lazo obtenida por CCM-SLAM utilizando el dataset. En línea punteada se enseña la referencia, que corresponde al trazado de la trayectoria a partir de los datos del GPS recolectados. Los parámetros usados en el algoritmo son los mismos que aquellos enunciados durante la Sección 3.2.2, para la configuración de agente-servidor utilizando laptops para ambos, y son los mismos que se utilizarán en la siguiente sección.

Como se mencionó antes, la desventaja de la visión monocular frente a la estéreo es que

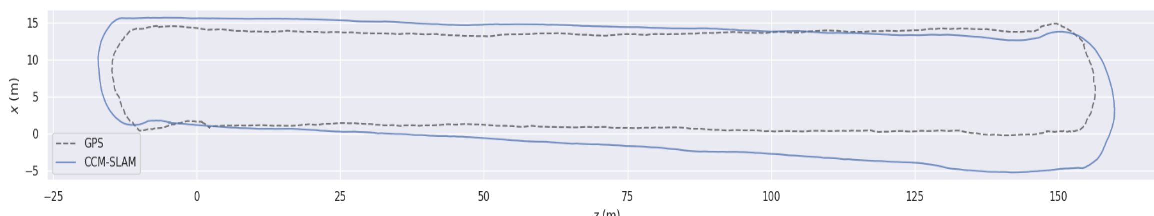

Figura 4.1: Trayectoria obtenida con CCM-SLAM, comparada con el *ground truth* en línea punteada.

no puede estimar la escala del mapa. Esto hace que no sea posible comparar directamente la trayectoria obtenida por el algoritmo con la del GPS. El resultado que se muestra es posterior a un ajuste de escala y alineación mediante el método de alineación de Umeyama [43], que normalmente se utiliza en el ámbito del V-SLAM. La herramienta *evo*, mencionada en el capítulo anterior, aplica a los datos ambas operaciones tomando de referencia el *ground truth*, y luego grafica ambos. Como puede apreciarse, si bien hay un desvío entre la trayectoria real y la estimada, la forma del recorrido fue muy bien respetada.

Los mayores desvíos se identifican al final de los tramos rectos, previo a los giros. La suposición es que, durante estos tramos rectos, la deriva asociada a estos algoritmos incrementa el error, siendo un tramo que es casi constante y sin grandes cambios de contraste (recordar que es un ambiente donde los árboles son prácticamente iguales, y el entorno no varía mucho). Durante los giros, el algoritmo tiene una mayor variación en los puntos característicos que extrae, lo cual se traduce en más información y en una mejor estimación de la trayectoria. Luego, con el cierre de lazo, se corrigen los errores al unir el punto inicial con el final, ya que la computadora itera sobre lo obtenido y encuentra la trayectoria más lógica que una ambas puntas considerando la información disponible. En ese sentido, corregir los giros puede resultar más sencillo, pero durante los tramos rectos, es difícil definir qué tanto se pudo desviar hacia izquierda o derecha el robot, pues el objetivo final (coincidir inicio con llegada) no se ve tan afectado.

En simultáneo, y para poner a prueba las funciones de *top, ps* y *nethogs*, se tomaron las mediciones de CPU, RAM y ancho de banda. En las Figuras 4.2, 4.3, 4.4, 4.5 se pueden visualizar el porcentaje de CPU utilizado, la memoria RAM en MegaBytes [MB] consumida, la tasa de datos enviada y la tasa de datos recibida, ambas en KiloBytes por segundo [KB/s], respectivamente. En todo los casos se hace distinción entre agente y servidor. El porcentaje de CPU hace referencia a qué porción del procesador fue utilizada en cada instante de muestreo, pudiendo hacerse uso de uno o más núcleos, según la computadora.

Por otro lado, previamente se había mencionado que el dataset duraba alrededor de 10 minutos, aunque en los gráficos se observa un tiempo total menor. Esto se debe a que durante el primer minuto la cámara estaba grabando quieta en el lugar. En consecuencia, ese primer minuto no contaría en las mediciones al no estar el *tracking* inicializado del todo. Si bien el primero minuto del dataset se podría haber recortado, se priorizó dejar todos los *frames* para analizar si esto ayudaría en la falta de solapamiento que se había comentado antes.

En cuanto al uso del CPU, en la Figura 4.2 se puede ver que el agente, desde el inicio hasta los 48 segundos aproximadamente, se mantiene usando la capacidad de un núcleo, y luego salta a dos, cuando comienza con el proceso de *mapping* de su mapa local. A partir de

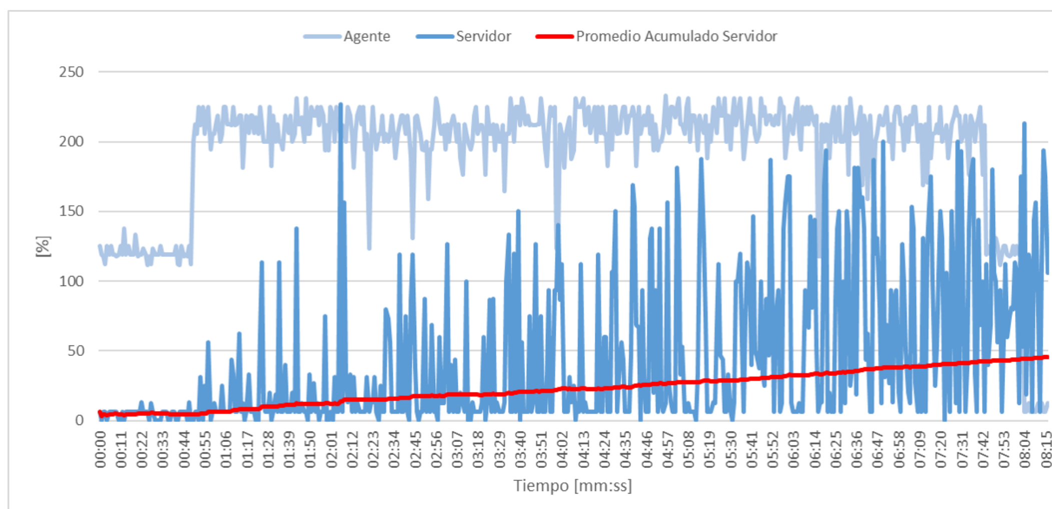

Figura 4.2: Consumo de CPU a lo largo del experimento, para agente y servidor, en [%]. Un consumo mayor al 100 % (o 200 %) implica la utilización de más de un (o dos) núcleo(s).

ahí mantiene un uso del procesador relativamente constante entre hacer el seguimiento de la cámara e ir intercambiando información con el servidor sobre los *keyframes* que va generando conforme se desplaza. Respecto al servidor, este exhibe su mayor consumo de procesamiento sobre el final, según muestra la recta del promedio acumulado, algo lógico pues conforme el mapa se va agrandando tiene mayor carga para efectuar operaciones, sumado a que en los últimos segundos está el cierre de lazo y se ejecuta el *bundle adjustment* global.

A partir de la Figura 4.3, ya es posible intuir, en primera instancia, la cantidad de memoria RAM que una computadora debería tener como mínimo para repetir estos mismos experimentos. La diferencia entre agente y servidor es muy clara: mientras un agente necesitaría alrededor de 1 GB de RAM, o un poco más quizás, el servidor, al tener que cargar con el mayor peso computacional, requiere más de 4 GB de RAM. De todas formas, esto es memoria RAM destinada exclusivamente al funcionamiento del algoritmo, así que en una situación real hay que considerar el uso de RAM asociado a los procesos internos de la computadora. Por tanto, un agente debería cumplir con al menos 2 GB de RAM, mientras que el servidor debería ofrecer 6 GB como base.

La Figura 4.4 enseña la tasa de transferencia enviada, en KB/s, para agente y servidor. Naturalmente, la gráfica más relevante corresponde al agente, ya que el servidor solo envía ocasionalmente información sobre la vecindad del agente, en caso de disponerla. Al promediar los valores de la gráfica, resulta que el agente tiene una tasa de envío de datos de 330 KB/s promedio; este valor presenta congruencia con los valores que los autores del CCM-SLAM reportaban en sus propios experimentos. A su vez, dicho valor ofrece una idea de los

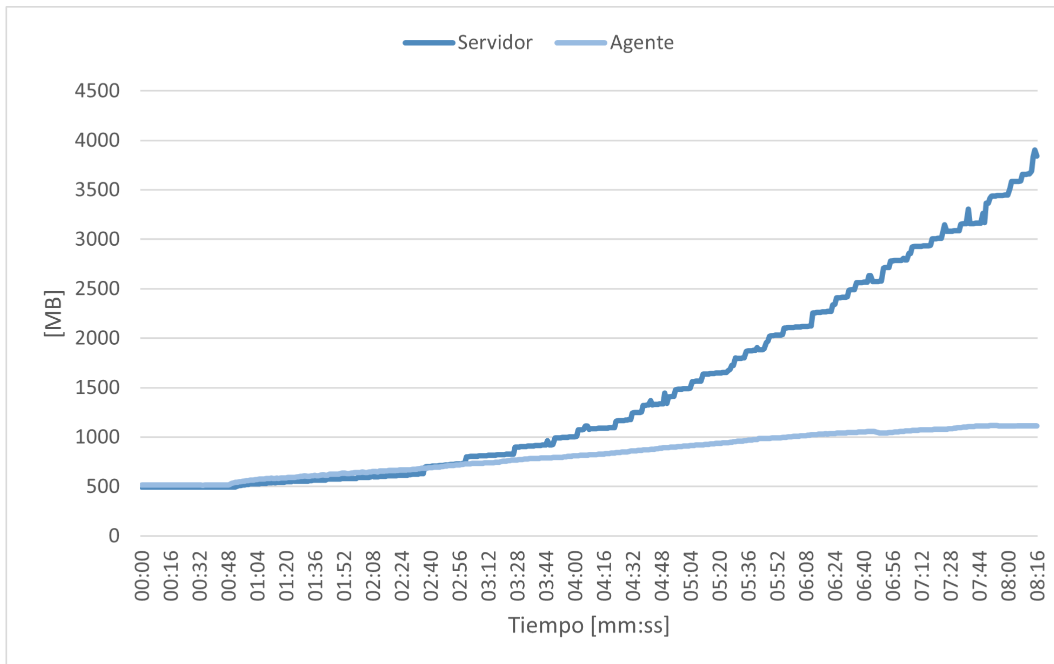

Figura 4.3: Consumo de memoria RAM a lo largo del experimento, para agente y servidor, en [MB].

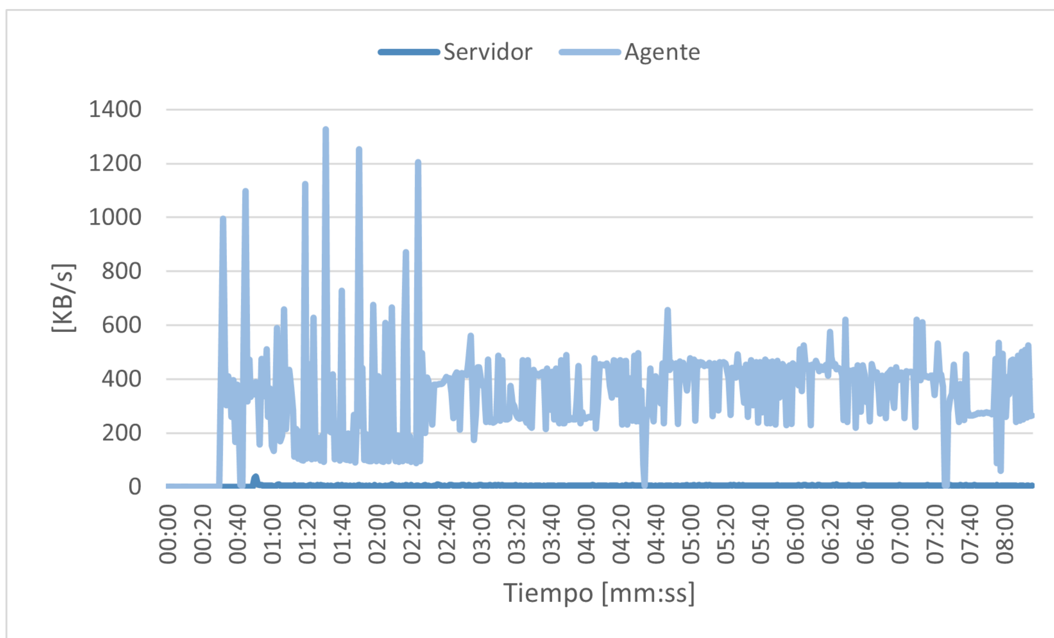

Figura 4.4: Velocidad de datos enviados a lo largo del experimento, para agente y servidor, en [KB/s].

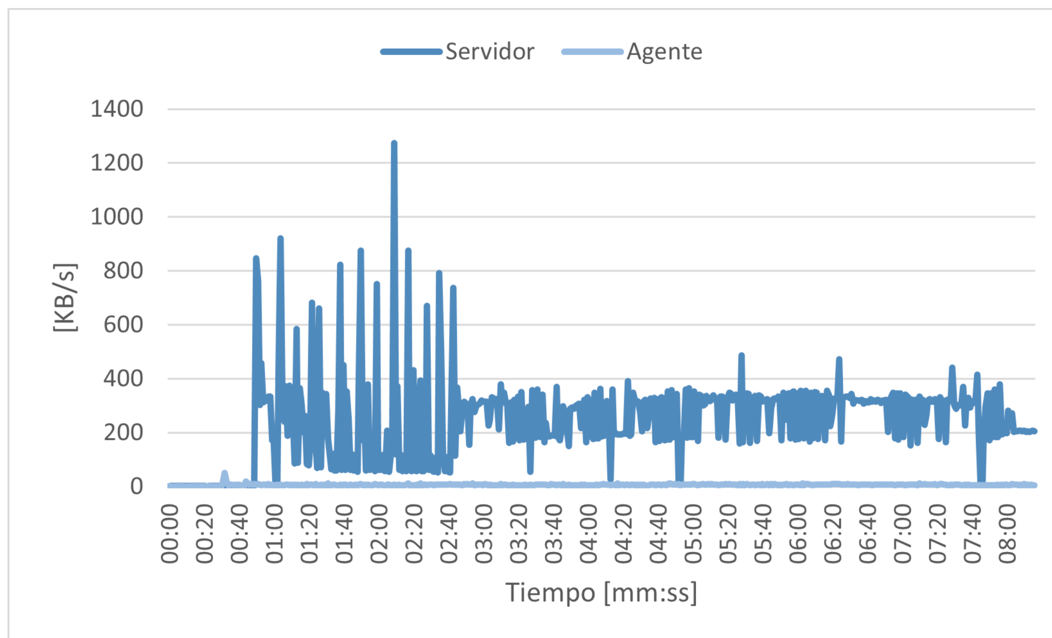

Figura 4.5: Velocidad de datos recibidos a lo largo del experimento, para agente y servidor, en [KB/s].

requerimientos que la red inalámbrica deba poseer como mínimo, así como se analizó con la memoria RAM.

Finalmente, la Figura 4.5 muestra, de manera similar a la Figura 4.4, la tasa de transferencia recibida en KB/s, para agente y servidor. En este caso se presenta la inversa a la figura anterior y es la gráfica del servidor la que cobra relevancia, aunque de todas formas se deje al agente, quien ocasionalmente recibe las confirmaciones de los *keyframes* recibidos por el servidor. Nuevamente, tomando el promedio de las mediciones, se tiene que la tasa de transferencia que recibe el servidor es alrededor de 240 KB/s.

Si bien las gráficas de las Figura 4.4 y 4.5 parecen ser simétricas, no se espera que el intercambio de datos sea exactamente igual en ambas direcciones, como lo demuestran los promedios de las tasas de transferencia. Se recuerda que las frecuencias de comunicación entre ambas partes son diferentes en función de si se trata de servidor o agente. En general, la frecuencia de comunicación del servidor al agente suele ser menor que la frecuencia de comunicación agente a servidor, dado que el servidor no tiene tanta urgencia por comunicar la información nueva que disponga al agente, que incluso en este caso no es necesario pues hay un solo agente trabajando.

En cuanto a la forma de la gráfica, puede verse al inicio picos mayores de transferencia en comparación con el final. Nuevamente, esto es algo relativo, no solo a las frecuencias de

comunicación, sino también a cómo se vaya transmitiendo la información conforme sea necesario. Al inicio resulta lógico ver un incremento en la tasa pues el agente está conformando su mapa inicial y no tiene ningún *keyframe*. Luego, una vez que el mapa local está conformado, la variación de *keyframes* es menor ya que paulatinamente va borrando *keyframes* mientras que otros nuevos ingresan.

4.2. Pruebas con medición de recursos

El mismo análisis realizado en la sección anterior se hizo para todas las pruebas 1, 2 y 3 planteadas en el Cuadro 3.1. Para estos tres experimentos se reutilizaron los mismos parámetros enunciados durante la Sección 3.2.1, variando según corresponda las frecuencias en la comunicación. En principio, lo que se esperaba era un aumento o disminución en la tasa de transferencia según el caso, comparado con los valores predeterminados correspondientes a la Prueba 2.

En la Figura 4.6 pueden observarse las trayectorias obtenidas luego de los cierres de lazo respectivos, comparadas con la trayectoria de referencia graficada con los datos de GPS. El número de trayectoria hace referencia al número de prueba.

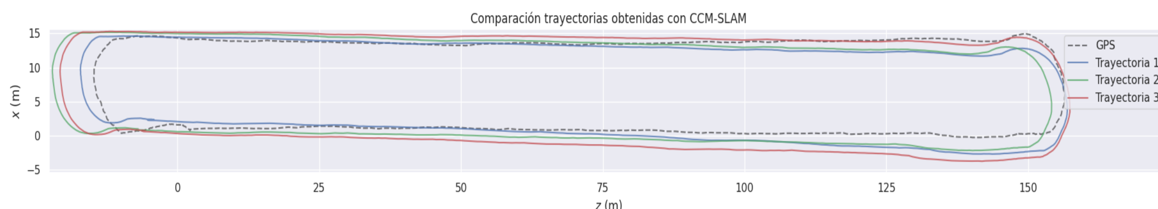

Figura 4.6: Trayectorias estimadas obtenidas para los tres experimentos, comparadas con el *ground truth*, en línea punteada.

En cuanto a la calidad de las trayectorias, puede verse en la Figura 4.6 que a simple vista no hay una gran diferencia cualitativa entre las mismas. Esta es una observación interesante, ya que como todas respetan la misma forma, similar a la vista en la Figura 4.1, utilizar una u otra frecuencia no afecta al resultado final de manera relevante. En consecuencia, provee cierta libertad para trabajar con el ancho de banda, pues este podría ajustarse a la necesidad de la red y aún así seguir brindando resultados satisfactorios.

Respecto al consumo de recursos, en las siguientes gráficas se comparan algunas medidas entre los diferentes ensayos. Para la primera, la Figura 4.7, se eligió mostrar el promedio de uso porcentual de CPU en cada una de las pruebas, tanto para el agente como para el

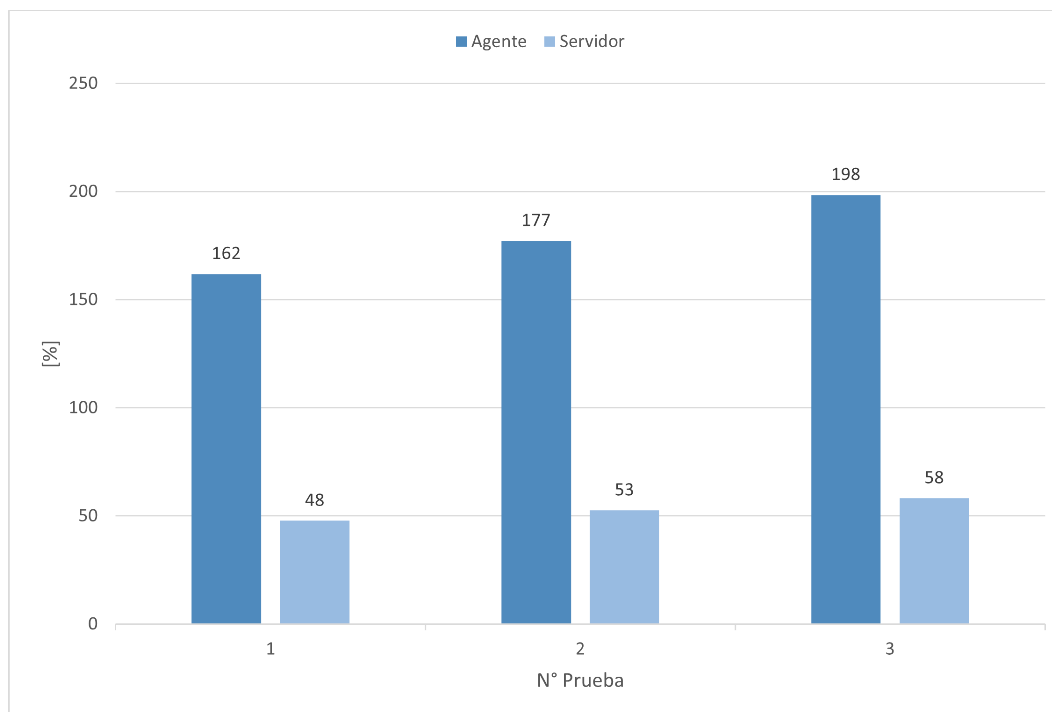

Figura 4.7: Uso promedio de CPU en las tres pruebas, en [%], para agente y servidor.

servidor. En general, el servidor no se vio muy afectado por la variación en la frecuencia de comunicación, aunque si sucedió que tanto para este como para el agente, el promedio de CPU utilizado se vio incrementado conforme la comunicación entre ambos era más regular, lo cual resulta lógico, al procesarse más mensajes por segundo.

En el caso del uso de RAM, resulta más interesante comparar la memoria máxima usada por agente y servidor en cada uno de los experimentos que el promedio, como enseña la Figura 4.8. Para esta situación, las conclusiones obtenidas llamaron más la atención que en el análisis del CPU; si bien el servidor mostró un aumento en el consumo de RAM con el incremento de la frecuencia de intercambio de datos, el cliente mantuvo el consumo prácticamente constante. Una posible hipótesis que respalde esto es la siguiente: con un envío de datos más frecuente por parte del agente al servidor, este último debe hacer actualizaciones en el mapa global y buscar posibles cierres de lazo con más regularidad; esto se traduce en el aumento de RAM por parte del servidor.

Por otro lado, el servidor le envía con mayor frecuencia información al agente, pero esta información está relacionada a la vecindad del mismo, es decir, le envía *keyframes* que tenga en su base de datos cercanos a donde está desplazándose el agente para así aumentar y mejorar la información de su mapa local. Al solo haber un agente presente en estos experimentos, solamente está recibiendo información de sí mismo, dado que no hay quién más colabore. En

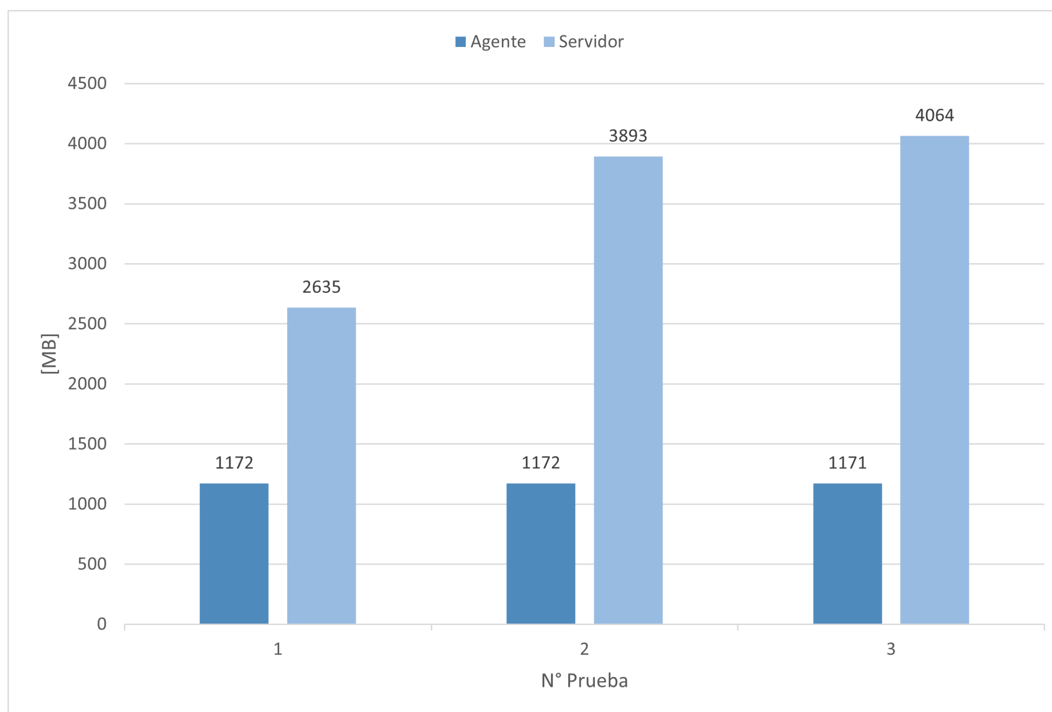

Figura 4.8: Memoria RAM máxima usada en las tres pruebas, en [MB], para agente y servidor.

consecuencia, su trabajo de mejorar el mapa local no le insume más RAM, ya que no hay información nueva presente.

Finalmente, en lo que respecta al uso del ancho de banda, la Figura 4.9 enseña el promedio de transferencia de datos, respectivamente, para el agente y el servidor durante las pruebas. Este gráfico distingue además la transferencia enviada y recibida solamente por cuestiones de prolijidad, ya que resulta lógico que los valores relevantes para el servidor sean los asociados a la información recibida, y para el agente sean de interés los correspondientes a los datos enviados.

De esta última imagen se tiene que, como resultaría lógico de esperar, una mayor frecuencia de comunicación a lo largo de las pruebas se tradujo en un mayor consumo de ancho de banda. Sin embargo, considerando la forma de las trayectorias resultantes de la Figura 4.6, se ve que existe cierta libertad para variar los anchos de banda según se requiera en el campo, ya que al final se pueden seguir obteniendo resultados satisfactorios.

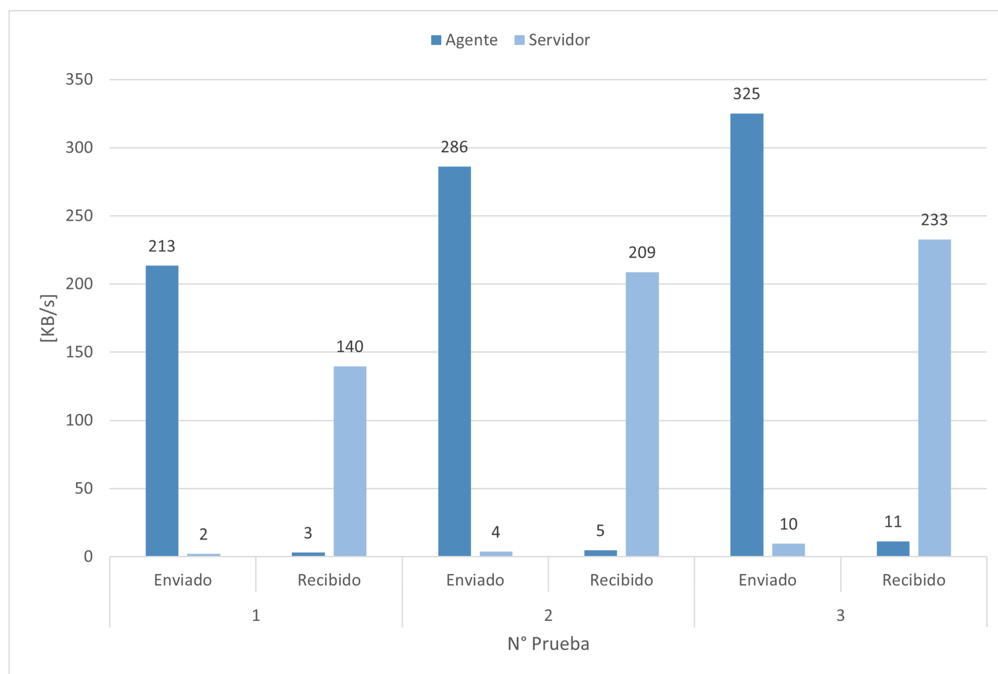

Figura 4.9: Ancho de banda promedio utilizado en las tres pruebas, en [KB/s], para agente y servidor.

4.3. Pruebas con las placas de hardware embebido

Una vez concluidas las pruebas en computadora, se procedió a probar las Raspberry Pi 3B disponibles. Si bien ya se conocían las limitaciones de antemano, debido a las mediciones de RAM principalmente, se eligió experimentar con las Raspberry Pi para ver de qué manera se podía conseguir que el algoritmo funcione. Dichas placas solamente tienen 1 GB de RAM, por lo tanto, considerando el consumo exclusivo que se midió de 1.1 GB de RAM, más el utilizado por la placa, se observaba claramente cual sería el primer cuello de botella. Como se mencionó en el capítulo anterior, el problema se solucionó añadiendo memoria *swap* y reduciendo la velocidad de reproducción del dataset a 4 veces la original. Esto último sería equivalente a simular que el tractor se desplaza más lento, de manera tal que acompaña la velocidad del procesamiento de la placa en función de su capacidad.

De la misma forma que antes, las trayectorias resultantes en cada ensayo se sometieron al proceso de ajuste de escala y alineado con el método de Umeyama, utilizando la herramienta *evo*, para poder compararlas apropiadamente con los datos del GPS como *ground truth*.

Primero se comenzó con un esquema de un servidor y un agente, para probar cada Raspberry Pi por separado. Luego de obtener el cierre de lazo, el resultado de la trayectoria de la primera placa quedó como se muestra en la Figura 4.10, remarcado en línea sólida azul.

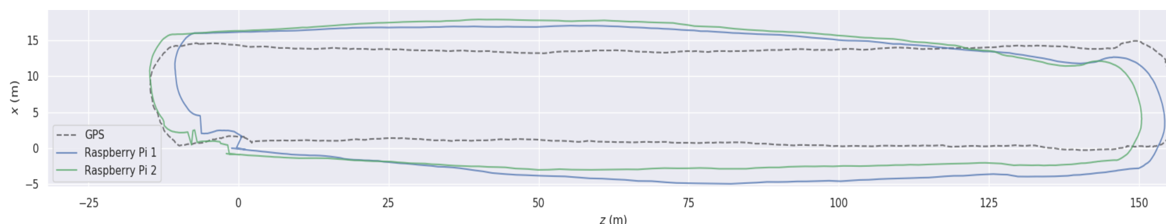

Figura 4.10: Trayectorias estimadas con las dos Raspberry Pi, comparadas con el *ground truth* en línea punteada.

A primera vista, resalta que la calidad de la trayectoria es menor que la obtenida durante las pruebas en computadora, si se compara por ejemplo, con la Figura 4.1. Sin embargo, es necesario aclarar que en términos generales, la calidad con la que la trayectoria resulta depende fundamentalmente de la cantidad de *keyframes* que el algoritmo extrajo durante la reproducción del dataset, ya que con ellos se reconstruye el mapa final. Mientras mas *keyframes* hayan, más información habrá, y por ende el mapa resultará más preciso. Durante las pruebas anteriores, cada trayectoria estimada estaba compuesta por alrededor de 800 *keyframes*, mientras que todas las enseñadas en esta sección consisten aproximadamente de 420 *keyframes* cada una. Esto justifica la forma menos precisa usando la Raspberry Pi respecto a las conseguidas anteriormente. La causa de la disminución de *keyframes* totales se relaciona principalmente a la falta de capacidad de la placa, ya que la información esencialmente es la misma (si bien variaron un poco los parámetros, esto debería haberse reflejado en incluso más *keyframes* que antes, algo que no sucedió). Al no poder procesar la información con tanta rapidez, extrae menos características y esto se refleja en la disminución de *keyframes*. A pesar de eso, resulta interesante ver que la forma del recorrido se sigue manteniendo reconocible, y más importante, que fue posible lograr el cierre de lazo.

Luego, se repitió el mismo experimento pero con la segunda Raspberry Pi. El resultado obtenido se muestra también en la Figura 4.10, trazado en una línea verde sólido. Comparando las dos trayectorias de Raspberry Pi, se ve que ambas son similares, lo cual resulta esperable al poseer ambas placas las mismas especificaciones.

Después de que ambas placas fueron ensayadas con el mismo dataset, se procedió a probar un sistema con dos agentes y un servidor, tomando ambas Raspberry como agentes. Para seguir trabajando con la misma información, se dividió el dataset a la mitad, dejando suficientes *frames* de solapamiento entre sub-datasets, de manera tal que el servidor pueda detectar los dos tramos y fusionarlos. De esta manera, se simula como si el mismo recorrido lo hubiesen hecho entre los dos agentes, una mitad cada uno. Cuando el servidor recibe la misma información de ambos agentes, lo detecta y lleva a cabo la fusión de mapas.

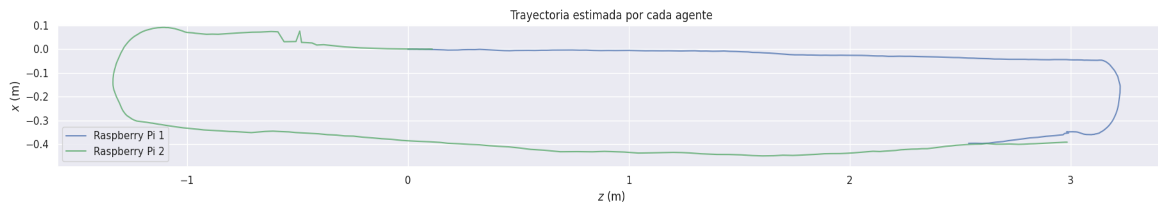

Figura 4.11: Trayectorias estimadas por cada agente, sin ajuste de escala ni alineación aplicadas.

Para mostrar en forma más detallada el funcionamiento, la Figura 4.11 muestra que tramo "recorrió" cada agente entre las hileras de frutales. Cabe aclarar que la imagen muestra los datos crudos de cada agente, sin el ajuste de escala y el alineamiento de Umeyama que siempre se mostraba en las otras imágenes.

La Figura 4.11 enseña la porción del dataset que cubrió cada agente; puede verse también algunos tramos donde ambos agentes se superponen, que corresponden a los *frames* en común que se añadieron a cada sub-dataset para que luego el servidor fusionara los mapas. Por otro lado, si bien la gráfica se puede ver invertida, puede apreciarse que tendrá una forma muy similar a la de las trayectorias presentadas en la Figura 4.10, luego de ser alineada y escalada apropiadamente.

Finalmente, en la Figura 4.12 se muestra el resultado final de la fusión de los mapas de cada agente en el servidor, que conforman la trayectoria producto del trabajo colaborativo entre dos robots. El redondeado en rojo indica los puntos donde se fusionaron los inicios y finales de cada agente que coincidieron en el servidor.

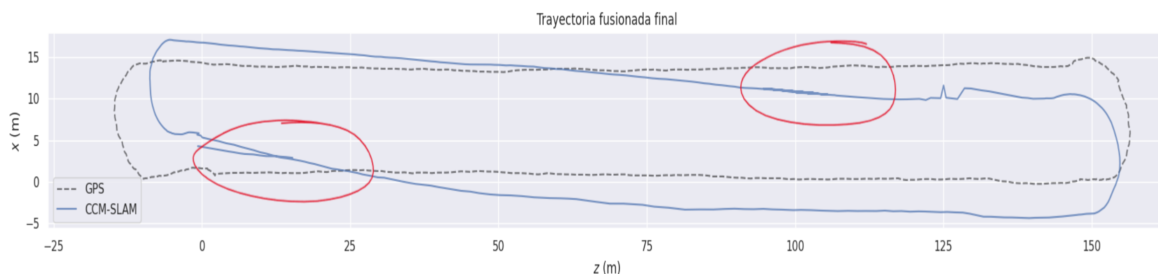

Figura 4.12: Trayectoria final fusionada por el servidor, comparada con el *ground truth* en línea punteada.

Este último resultado es muy importante, pues muestra en funcionamiento el punto más fuerte del CCM-SLAM frente a los demás algoritmos de V-SLAM: el trabajo colaborativo de robots. A partir de dos robots, representados por las Raspberry Pi, se logró reconstruir

resultados muy similares a los encontrados con un solo agente, en computadora. Si bien esta misma metodología de dos agentes y un servidor se podría haber hecho durante la etapa de medición de recursos, resultaba más interesante poner en juego la práctica de más de un agente con las placas de hardware embebido, que son lo más cercano a la realidad.

Aunque las trayectorias enseñadas en esta sección no son mejores que las de la sección anterior, se entiende que esto está relacionado con la capacidad limitada de las Raspberry Pi. Por tanto, una placa que presente mejores especificaciones (por ejemplo, al menos 2 GB de RAM) que estas últimas con seguridad otorgará los mismos resultados o mejores. Por otro lado, una desventaja de estos ensayos fue que al encontrarse la RAM como principal cuello de botella, impidió poder analizar si el CPU era también un cuello de botella o no.

Los resultados anteriores enseñados con las Raspberry Pi fueron producto de reproducir el dataset 4 veces más lento que lo normal. El motivo de esto está principalmente vinculado a las limitaciones de las placas, y a la falta de solapamiento que se explicó durante la Sección 3.1.3. Esto último se traduce en que, al tener la placa muy pocos *frames* iniciales coincidentes con el final, debe inicializar antes de que se terminen esos *frames* coincidentes, ya que sino no habrán *keypoints* en común para luego hacer el cierre de lazo.

Para resolver esto, se "estiró" el dataset, copiando y pegando *frames* del inicio del mismo pero al final con otro nombre, repitiéndolos. Técnicamente, esto simula que al llegar al final el dataset continuaba, pero en realidad solo estaba repitiendo el principio. La idea final de esto no era engañar los resultados, sino probar aumentar la velocidad de reproducción del dataset. Efectivamente, esto se logró, y se reprodujo de nuevo la trayectoria; solo que esta vez, a la mitad de la velocidad original, tardando 20 minutos menos de lo que se tardó en las otras pruebas. El resultado fue el que muestra la Figura 4.13.

La forma extraña que se observa en la trayectoria estimada de la Figura 4.13 corresponde al punto donde se marcaba el inicio y el final, que es donde se "engañó" al algoritmo haciéndole

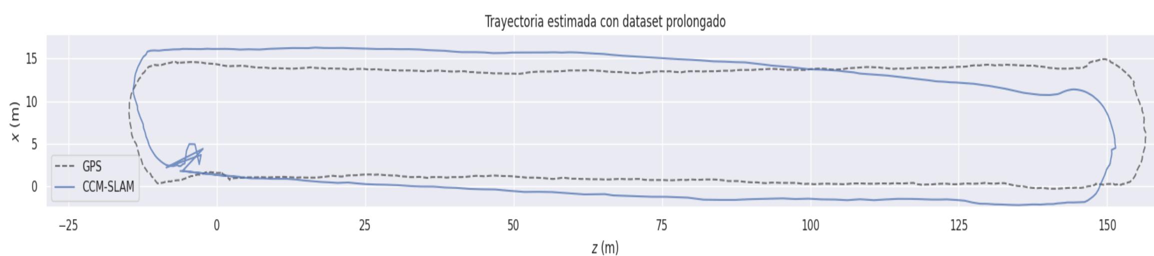

Figura 4.13: Trayectoria obtenida con un agente, usando el dataset con frames añadidos, comparada con el *ground truth* en línea punteada.

creer que el video continuaba. Sin embargo, este resultado demuestra que la Raspberry Pi es capaz de ofrecer un mejor rendimiento, si el dataset provisto ofrece una mayor flexibilidad en cuanto al solapamiento de imágenes.

A pesar de que se intentó reproducir esto mismo a una velocidad normal, no hubo forma de repetir los resultados, pues el agente perdía el *tracking* fácilmente al no poder procesar la información con suficiente rapidez ante las limitaciones del hardware.

4.4. Propuesta de abordaje en campo

Tras haber presentado los resultados de los ensayos en la sección anterior, se está en condiciones de plantear una propuesta de cómo se podría abordar la implementación del CCM-SLAM en campo. La presente sección detallará cuáles serían los requisitos mínimos a cumplir para que esto sea factible, y cómo debería ser la disposición del agente, servidor y conexiones, considerando un cuadro típico de chacra de una hectárea, más o menos similar al utilizado durante la grabación del dataset.

4.4.1. Requisitos mínimos

En lo que respecta al servidor, se comentó que durante todo el trabajo la computadora que funcionaba como servidor contaba con un procesador AMD Ryzen 9 5900HS @3.30 GHz y alrededor de 10 GB de RAM, debido al uso de máquina virtual. Se trabajó con uno o dos agentes todo el tiempo y, fuera de los resultados mostrados, se intentó ejecutar el algoritmo con tres agentes pero resultó ser una carga más difícil de gestionar en el servidor. Cuando se hacían pruebas con esa cantidad de agentes, el servidor siempre se detenía y lanzaba error a causa de la exigencia que le representaban tantos hilos de trabajo en paralelo.

En consecuencia, es importante tener en cuenta el número de agentes a usar para lo que respecta a las especificaciones del servidor. Si se toma como referencia el trabajo realizado por los autores del CCM-SLAM, estos utilizaron un procesador menos potente que el de este trabajo, pero su servidor disponía 20 GB de RAM, y pudieron hacer trabajar en paralelo a tres agentes. En función de esto y de las mediciones realizadas, se puede estimar que un servidor con 8 GB de RAM, un valor comercial, basta para un solo agente. Sin embargo, es importante considerar el aumento de la memoria RAM en caso de querer trabajar con más robots en simultáneo.

Desde el punto de vista del agente, se comprobó que una placa como la Raspberry Pi 3B no es suficiente. Si bien bajo ciertas condiciones puede otorgar resultados tolerables, es

necesario elevar el requisito de la memoria RAM, en lo posible, a los 4 GB como mínimo.

Como también se había mencionado, debido al problema de la RAM, no se pudo evaluar si el procesador de la placa, un Broadcom BCM2837 Cortex-A53 @1.2GHz, podría ser un posible cuello de botella o no. De todas formas, considerando las mediciones realizadas para este procesador que tiene cuatro núcleos, el promedio del uso de CPU nunca superó el 200 % para el agente; por tanto, podría afirmarse que la mayor restricción está dada por la RAM.

Otro punto importante a tener en cuenta es la capacidad de almacenamiento del agente. Una característica del CCM-SLAM, que aumenta su robustez frente a problemas de comunicación es que almacena los *keyframes* obtenidos de su mapa local en un *buffer* y los retiene hasta que el servidor le confirma haber recibido la información, para luego descartar el *keyframe* en cuestión y seguir almacenando nuevos *keyframes* conforme se va moviendo. Como tal, este *buffer* tiene una restricción de espacio, y si dicho espacio no es suficiente, sumado a que la comunicación no sea buena, se correrá riesgo de perder información. Más adelante, en la estrategia a implementar, se verá la relevancia que tiene esta característica debido a la forma de comunicación elegida.

En cuanto a la comunicación, durante el inicio de este trabajo se había planteado el uso de tecnologías como LoRa (*Long Range*), ya que otorgan grandes áreas de cobertura y bajo consumo de alimentación, algo que para un entorno frutícola resulta ideal. Sin embargo, los anchos de banda asociados a esta tecnología son demasiado bajos, alrededor de los 30 Kbps. Considerando las mediciones tomadas durante las pruebas, que otorgaban un promedio de 300 KB/s de uso del tráfico, resulta inviable el uso de LoRa.

Como ya se vio, también se intentó disminuir ese ancho de banda, ya que CCM-SLAM tiene parámetros configurables vinculados a la transmisión de la información. Sin embargo, la brecha entre el ancho de banda ofrecido por LoRa y los datos que el agente debe enviar es demasiado amplia, así que no hubo manera de reducir lo suficiente la tasa de transferencia para adecuarse a la especificaciones del LoRa.

Por otro lado, de las mediciones acumuladas con *nethogs* se calculó que el agente envía a lo largo del dataset un total de 180 MB aproximadamente, la cual es una cantidad considerable. El envío de *keyframes* lógicamente no es tan pesado como lo sería enviar todos los *frames* que registra el agente, pero no deja de ser una transferencia de imágenes. Se sopesó en consecuencia la posibilidad de utilizar varios módulos LoRa para un agente y dividir el envío de la información, o utilizar más agentes donde cada uno trabaje con una menor porción de información, traduciéndose en una menor tasa de transferencia requerida. Sin embargo, ya sea en un caso u otro, la complejidad que implica dichas acciones llevó a descartar estos módulos IoT por completo.

La siguiente opción quizás más factible y sencilla de implementar resultaba ser una antena 4G. Desgraciadamente, la mayoría de las chacras en el Alto Valle no cuentan con cobertura 4G al día de hoy, así que tampoco era viable esta opción, pensando en una solución útil para cualquier productor. Eso dejaba el Wi-Fi como última opción, el cual cumpliría con las prestaciones de ancho de banda, pero que reduciría en gran medida el alcance.

Un *Access Point*, que sería el dispositivo a utilizar para crear la red Wi-Fi, según el modelo, tiene un alcance alrededor de 30 metros para una buena velocidad de transferencia, al menos aceptable para los valores obtenidos en los resultados. Si se considera como modelo de análisis el cuadro de una hectárea chacra mencionado anteriormente, para tener todos los puntos comunicados en todo momento sería necesario instalar varios *Access Point* a lo largo del cuadro y luego interconectarlos entre sí. Evidentemente, esta tampoco resultaba ser una solución factible, ya que es poco práctica y costosa.

4.4.2. Estrategia a implementar

Finalmente, tomando ventaja del *buffer* que el agente tiene para guardar los *keyframes* hasta entablar conexión con el servidor, se planteó dejar un solo *Access Point*, de manera tal que cada vez que el agente esté lo suficientemente cerca de éste, pueda descargar rápidamente la información acumulada. Como referencia, la Figura 4.14 muestra un esquema de cómo sería la estrategia a implementar en campo. Idealmente, se enseña un solo agente, pero podrían ser más de uno.

Mientras las hileras son recorridas, en los puntos donde la conexión lo permita, el agente seguirá descargando los *keyframes* y *map points* que vaya generando y el servidor le comunicará que fueron bien recibidos, desde algún otro punto cualquiera. A pesar de que el agente tenga espacio para almacenar la información mientras intenta reestablecer la conexión, este *buffer* no es infinito. En función de la longitud de la hilera, podría suceder que se acumule demasiada información y, si el agente rebalsa su *buffer*, comenzará a descartar los *keyframes* más viejos para dejarle lugar a los nuevos.

Además de la longitud de la hilera, la cantidad de información que el agente le debe transmitir al servidor depende de la cantidad de *keyframes*. Mientras más imágenes clave deba enviar, más irá almacenando en la memoria. A su vez, la cantidad de *keyframes* depende de varios subfactores, como son los fps, la capacidad de procesamiento del agente, los parámetros del sistema de odometría visual, entre otros. Como no se puede saber el valor exacto de *keyframes* que se extraerán, a lo sumo se puede tomar de referencia los experimentos ya realizados o hacer nuevas pruebas para tener una aproximación de cuánta información se transmitirá.

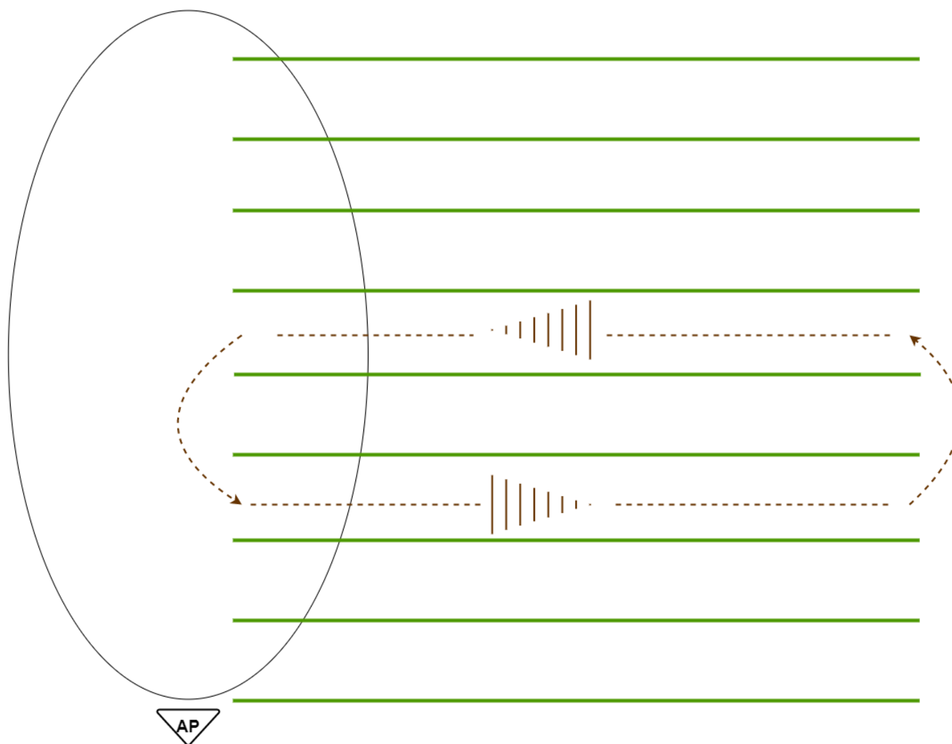

Figura 4.14: Posible disposición del AP y el agente en la distribución de hileras de frutales para el funcionamiento del CCM-SLAM.

Para tomar una referencia, se mencionó antes que durante los resultados de la Sección 4.2, se había encontrado que el agente generaba alrededor de 800 *keyframes* y enviaba un total de 180 MB de datos en todo el dataset, aproximadamente. Revisando las mediciones de datos enviados por el agente, y considerando solo uno de los giros, se calcula que durante ese tramo se envían 40 MB. Si durante el giro, que sería cuando el agente tiene visibilidad al *Access Point*, envía 40 MB, quiere decir que estaría sin conexión durante los restantes 140 MB acumulados.

En principio, este sería el peor caso, dado que no es tan directo que el agente gira y al seguir el tramo recto de la hilera pierde automáticamente la comunicación; existe un cierto rango donde todavía se pueden enviar datos, lo cual depende principalmente del alcance del *Access Point*. En el caso hipotético de que la cantidad de memoria en el *buffer* no sea posible de configurar en el CCM-SLAM o simplemente la placa no tenga dicho espacio, se puede plantear colocar otro *Access Point* más en el otro extremo de la hilera, como muestra el esquema de la Figura 4.15.

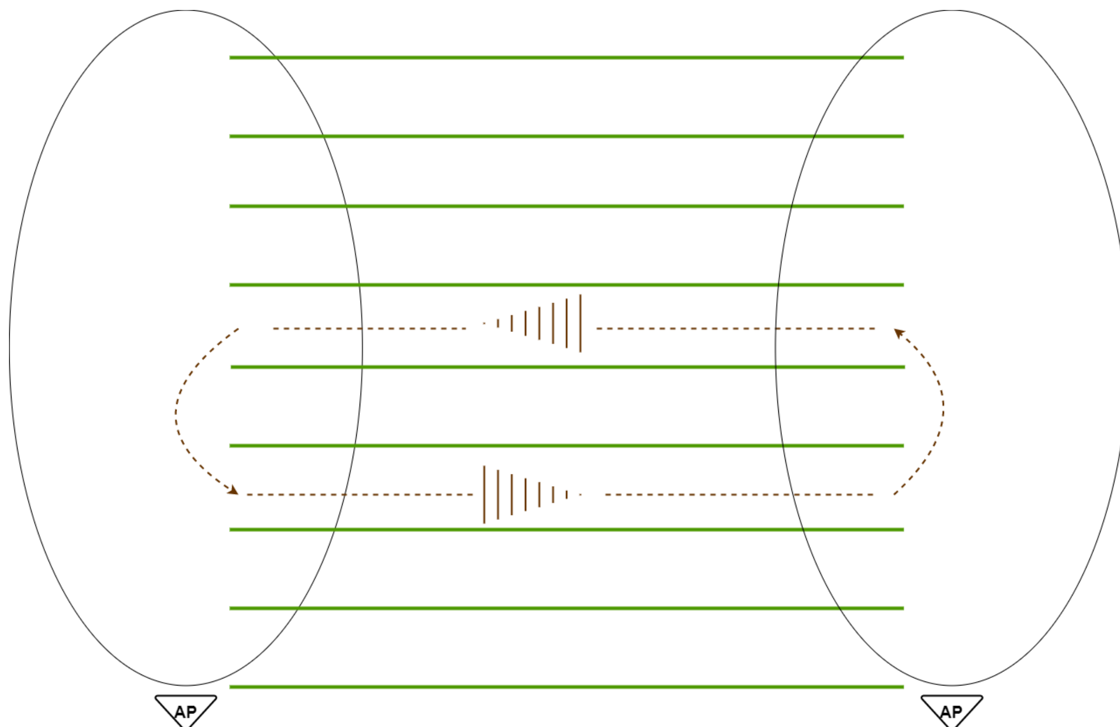

Figura 4.15: Nueva disposición en el cuadro de chacra para el funcionamiento del CCM-SLAM, incorporando un segundo *Access Point*.

De esta manera, si se divide equitativamente como aproximación, serían 90 MB por cada hilera, tomando que 40 MB podrían ser descargados automáticamente en cada giro, por la proximidad a cada *Access Point*. Eso dejaría una carga de 50 MB solamente para el agente durante el trayecto recto, en el peor de los casos.

De todas formas, estas cantidades eran las que se encontraron durante las pruebas con computadora; como se mencionó antes, con las Raspberry Pi se consiguieron la mitad de los *keyframes* por las limitaciones de las placas. Eso quiere decir que incluso serían menores las cantidades a transmitirse. Sin embargo, también es cierto que una placa con mejor prestaciones sería lo ideal para que la reconstrucción de la trayectoria tenga una forma más cercana a la real.

Capítulo 5

Conclusiones

El presente capítulo tiene como fin, en primera instancia, resumir el desarrollo presentado a lo largo de todo el trabajo. Luego de esto, la siguiente sección presentará las conclusiones obtenidas en función de los objetivos planteados en la Sección 1.2 y de los resultados redactados en el Capítulo 4. Para finalizar, se dará cierre al trabajo y se propondrán posibles acciones o mejoras a seguir.

5.1. Resumen del trabajo realizado

Este trabajo comenzó en el marco de las becas EVC-CIN, buscando aportar nuevo conocimiento al ámbito de algoritmos de visión artificial enfocados a la fruticultura. Consecuentemente, se profundizó el estudio sobre técnicas de visión artificial colaborativas a través del CCM-SLAM. Durante este trabajo, fue necesario investigar sobre el funcionamiento del mencionado algoritmo para saber cómo aplicarlo en el entorno deseado, y eso requirió la comprensión de los conceptos teóricos que se introdujeron durante el Capítulo 2.

Para poner a prueba el CCM-SLAM, fue necesario elegir un dataset objetivo con el cual trabajar, y esto es algo que se detalló durante el Capítulo 3. Luego de eso, se presentó la metodología a seguir por el resto del trabajo, la cual se dividió en dos partes. La primera parte, donde se hicieron pruebas para entender cómo funcionaba CCM-SLAM y qué había que tener en cuenta para obtener resultados positivos, y cómo se llevarían a cabo las mediciones de RAM, CPU y ancho de banda, para su posterior análisis. La segunda parte consistió en los ensayos sobre placas de hardware embebido, que fueron las dos Raspberry Pi 3B. Tanto para las mediciones como para las pruebas sobre las placas se especificaron diferentes parámetros vinculados al funcionamiento de la odometría visual, basada en ORB-SLAM, para la

detección y extracción de características en las imágenes.

Después, en el Capítulo 4, se expuso lo obtenido según la metodología planteada anteriormente. Primero se enseñaron los resultados de los ensayos exclusivos en las computadoras, y de cómo fue posible reconstruir la trayectoria grabada alrededor de dos hileras de perales, tras conseguir un cierre de lazo. Conseguir ese cierre no fue tarea sencilla, pero una vez logrado, dio lugar a continuar con los análisis de consumo de recursos. Se llevaron a cabo los tres ensayos caracterizados por sus frecuencias de comunicación agente-servidor y se vio que a pesar de las variaciones que hubo en el ancho de banda, RAM y CPU, los resultados obtenidos fueron muy similares entre sí, y que al mismo tiempo eran trayectorias muy cercanas a la real, la cual estaba representada por los valores tomados con el GPS.

En la segunda parte de los resultados, se pudieron observar los rendimientos de ambas placas al reproducir los dataset. Si bien las especificaciones de ambas eran conocidas, y con la información obtenida de las mediciones de recursos anterior, se podía estimar la viabilidad de que las placas funcionasen, aún así era de utilidad ver cómo eran las trayectorias finales estimadas. A pesar de los problemas que presentaron las Raspberry Pi por sus limitaciones, los mapas obtenidos se acercaron mejor a la realidad de lo que se habría esperado, y ya haber conseguido el cierre de lazo teniendo en cuenta la falta de solapamiento del dataset fue un gran logro en sí.

5.2. Conclusiones

Luego de lo presentado a lo largo del Capítulo 4, se puede concluir que se logró exitosamente la implementación y análisis del uso del CCM-SLAM, como una alternativa a algoritmos de V-SLAM que incorporen colaboración entre robots. El estudio sobre este software fue extensivo, evaluando todo el abanico de posibles configuraciones que contiene y qué ventajas y desventajas implica cada una de ellas. La opción de utilizar solamente visión artificial como elemento de sensado demostró ser efectiva, pudiendo reconstruirse con éxito una trayectoria de aproximadamente 369 metros alrededor de dos hileras de perales. Es importante destacar que esto se logró en un ambiente frutícola, donde los algoritmos de visión artificial suelen presentar dificultades para hacer el seguimiento de la cámara, pues el ambiente presenta muy pocos puntos de contraste, al ser uniforme. Si bien es cierto que mientras más dispositivos de sensados se incorporen, más cercanos serán los resultados a la realidad, es interesante indagar en la viabilidad de sistemas que presenten menor costo al incorporar menos tecnologías y aún así puedan seguir entregando un buen rendimiento.

Por otro lado, una vez conseguido el cierre de lazo sobre el dataset elegido, el estudio del

consumo de recursos como RAM, CPU y ancho de banda permitió comprender las exigencias y requisitos mínimos que CCM-SLAM presenta, y hasta qué punto se pueden reducir los costos para seguir obteniendo resultados satisfactorios. En general, se puede afirmar que el mayor cuello de botella está en la memoria RAM, en especial del lado del agente. El ancho de banda también representó una limitación, pero esto depende de qué tipo de tecnología decida usarse; desde el punto de vista de tecnología LoRa resulta inviable lidiar con las tasas de transferencia medidas, pero desde el lado del Wi-Fi esto es fácilmente cubierto. Sin embargo, considerando la estrategia que se planteó en la Sección 4.4 para la implementación en campo, es necesario tener cuidado con la memoria del agente, ya que será necesario utilizar tamaños de *buffer* considerables, como bien se estimó.

En cuanto al hardware mínimo que deben satisfacer cliente y servidor, según lo analizado durante la medición de recursos, el servidor debe ser un equipo potente, con al menos 8 GB de RAM, un procesador de 8 núcleos y una frecuencia de reloj superior a los 3 GHz. El servidor no necesariamente debe estar en la misma localización geográfica que el agente; basta con que esté conectado a la misma red.

En cuanto al agente, se concluyó que una Raspberry Pi, al menos el modelo 3B, no basta para entregar resultados satisfactorios; en todo caso, se podría evaluar el uso de una placa con más memoria RAM, por ejemplo una Odroi Xu4, la cual tiene un procesador de 8 núcleos a 2 GHz y 2 GB de RAM. De todas formas, el abanico de parámetros que tiene CCM-SLAM para variar es muy amplio. Como se vio a lo largo del trabajo, modificar un conjunto de parámetros resulta en obtener más o menos *keyframes*, lo que puede traducirse en mayor o menor consumo de recursos; lógicamente esto tiene un efecto sobre la calidad de las trayectorias resultantes, pero como todo juego de compromiso entre variables, es simple cuestión de encontrar el punto más óptimo.

5.3. Cierre del trabajo y propuestas de mejora

En conclusión, se logró un aporte importante en el área de algoritmos de visión artificial orientados a la robótica aplicada a la fruticultura. La posibilidad de trabajar colaborativamente sin tener que dejar toda la carga en un solo robot es una propuesta interesante, no necesariamente la única a utilizar, pero este trabajo deja precedente sobre una posibilidad que puede reconsiderarse en el futuro.

Sin embargo, aún quedan otras opciones a probar en lo que respecta al CCM-SLAM. En primer lugar, durante las mediciones de RAM, CPU y ancho de banda solo se contempló el uso de un solo agente; sería interesante ver qué tanto influye la incorporación de más agentes en el

sistema, en todos los aspectos. También sería de interés hacer mediciones desde las placas y no desde la computadora que simulaba ser agente, para que los valores entregados se acerquen más a la realidad. En este caso, como la Raspberry Pi estaba muy limitada en memoria, no se pudo hacer, pero se podría intentar con una placa con mejores prestaciones.

En segundo lugar, sería útil hacer ensayos con otras placas, superiores a la Raspberry Pi 3B, para ver qué tanto evolucionan los resultados con respecto a los presentados en este trabajo. Esto no implica necesariamente usar las mismas placas para ambos agentes o saltar a una placa con prestaciones que exceden a las utilizadas aquí; se puede ir probando combinaciones y placas cuyas prestaciones no se alejen mucho de las recomendadas en la sección anterior.

En tercer lugar, se podría incluir en el futuro análisis cuantitativos. Durante el análisis de resultados, las trayectorias siempre se compararon cualitativamente; sería más objetivo introducir alguna métrica de comparación, como puede ser el error absoluto y/o relativo.

En cuanto a lo que respecta la implementación en campo, durante la sección correspondiente a esto, se mencionó el *buffer* y su configuración, pero no se realizaron experimentos con el mismo, ya que quedaba fuera del alcance del trabajo. En un futuro, sería práctico incluirlo en pruebas para determinar cuánta información puede retener el agente hasta comenzar a sufrir pérdidas permanentes.

Finalmente, otro posible trabajo futuro sería volver a grabar el dataset pero utilizando la cámara Logitech. Durante la Sección 3.1.3 se habían mencionado varios problemas que ocurrieron, pero si el dataset se grabara de nuevo teniendo en cuenta dichos conflictos, CCM-SLAM no debería tener problema en recrear las trayectorias.

Bibliografía

[1] Darío Fernández, Liliana Cichón, Silvina Garrido, and Hugo Álvarez. Agricultura de precisión. *Fruticultura&Diversificación. INTA*, 53:26–34, 2007. ISSN 1669-7057.

[2] Anna Chlingaryan, Salah Sukkarieh, and Brett Whelan. Machine learning approaches for crop yield prediction and nitrogen status estimation in precision agriculture: A review. *Computers and Electronics in Agriculture*, 151:61–69, 2018. ISSN 0168-1699. doi: https://doi.org/10.1016/j.compag.2018.05.012. URL `https://www.sciencedirect.com/science/article/pii/S0168169917314710`.

[3] Alejandro Rollán. Agricultura de precisión: una revolución que cumple 25 años y que vale oro. *La Voz*, 2020. URL `https://www.lavoz.com.ar/agro/actualidad/agricultura-de-precision-una-revolucion-que-cumple-25-anos-y-que-vale-oro/`.

[4] Edgardo Benítez Piccini, Carlos Magdalena, and Marcelo Muñoz. Hacia una fruticultura de precisión en la aplicación de agroquímicos. *Fruticultura&Diversificación. INTA*, 62: 8–11, 2010. ISSN 1669-7057.

[5] E. Benitez Piccini, M. Romitelli, Á. Muñoz, M. Muñoz, D. Fernandez, M. Curetti, P. Fonovich, F. Cabezas, and C. Magdalena. Yield maps in tree fruit production: an automatic system for implementing precision agriculture. *Acta Hortic*, pages 427–434, 2021. doi: https://doi.org/10.17660/ActaHortic.2021.1303.59. URL `https://www.actahort.org/books/1303/1303_59.htm`.

[6] República Argentina. Poder Legislativo Nacional. Ley nacional 27354 - declarase emergencia económica, financiera y social., 2017.

[7] Dario Eduardo Fernandez. Avances en fruticultura de precisión en la norpatagonia. *Resúmenes del Curso Internacional de Fruticultura de Precisión - Innovación en mecanización. INTA*, pages 95–109, 2017.

[8] Francisco Raverta Capua, Sebastian Sansoni, and Marcelo Moreyra. Comparative analysis of visual-slam algorithms applied to fruit environments. pages 1–6, 11 2018. doi: 10.23919/AADECA.2018.8577360.

[9] Joan Solà. Multi-camera vslam : from former information losses to self-calibration. 2007.

[10] D. Nister, O. Naroditsky, and J. Bergen. Visual odometry. In *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.*, volume 1, pages I–I, 2004. doi: 10.1109/CVPR.2004.1315094.

[11] Marc Pollefeys, Luc Gool, Maarten Vergauwen, Frank Verbiest, Kurt Cornelis, Jan Tops, and Reinhard Koch. Visual modeling with a hand-held camera. *International Journal of Computer Vision*, 59:207–232, 09 2004.

[12] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in computer vision*. Cambridge University Press, 2nd edition, 2003. ISBN 0521 54051 8.

[13] The MathWorks. Computer vision toolbox - camera calibration, 2013. URL `https://www.mathworks.com/products/computer-vision.html#camera-calibration`.

[14] Joern Rehder, Thomas Schneider Janosch Nikolic, Timo Hinzmann, and Roland Siegwart. Extending kalibr: Calibrating the extrinsics of multiple imus and of individual axes. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 4304–4311, Stockholm, Sweden, 2016.

[15] Arturo Gil, Oscar Mozos, Monica Ballesta, and Óscar Reinoso. A comparative evaluation of interest point detectors and local descriptors for visual slam. *Machine Vision and Applications*, 21:905–920, 10 2010. doi: 10.1007/s00138-009-0195-x.

[16] Thomas Lemaire, Cyrille Berger, Il-Kyun Jung, and Simon Lacroix. Vision-based slam: Stereo and monocular approaches. *International Journal of Computer Vision*, 74:343–364, 2007.

[17] Christopher G. Harris and M. J. Stephens. A combined corner and edge detector. In *Alvey Vision Conference*, Manchester, UK, 1988.

[18] David Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60:91–, 11 2004. doi: 10.1023/B:VISI.0000029664.99615.94.

[19] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. volume 3951, pages 404–417, 07 2006. ISBN 978-3-540-33832-1. doi: 10.1007/11744023_32.

[20] Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua. Brief: Binary robust independent elementary features. volume 6314, pages 778–792, 09 2010. ISBN 978-3-642-15560-4. doi: 10.1007/978-3-642-15561-1_56.

[21] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *2011 International Conference on Computer Vision*, pages 2564–2571, 2011. doi: 10.1109/ICCV.2011.6126544.

[22] Jorge Fuentes-Pacheco, Jose Ascencio, and J. Rendon-Mancha. Visual simultaneous localization and mapping: A survey. *Artificial Intelligence Review*, 43, 11 2015. doi: 10.1007/s10462-012-9365-8.

[23] Ethan Eade and Tom Drummond. Unified loop closing and recovery for real time monocular slam. 01 2008. doi: 10.5244/C.22.6.

[24] Andrew Davison, Ian Reid, Nicholas Molton, and Olivier Stasse. Monoslam: real-time single camera slam. *IEEE transactions on pattern analysis and machine intelligence*, 29:1052–67, 07 2007. doi: 10.1109/TPAMI.2007.1049.

[25] Georg Klein and David Murray. Parallel tracking and mapping for small ar workspaces. In *2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality*, pages 225–234, 2007. doi: 10.1109/ISMAR.2007.4538852.

[26] Jakob J. Engel, Thomas Schöps, and Daniel Cremers. Lsd-slam: Large-scale direct monocular slam. In *ECCV*, Zurich, Switzerland, 2014.

[27] Raúl Mur-Artal, J. M. M. Montiel, and Juan D. Tardós. ORB-SLAM: a versatile and accurate monocular SLAM system. *IEEE Transactions on Robotics*, 31(5):1147–1163, 2015. doi: 10.1109/TRO.2015.2463671.

[28] Taihú Pire, Thomas Fischer, Gastón Castro, Pablo De Cristóforis, Javier Civera, and Julio Berlles. S-ptam: Stereo parallel tracking and mapping. *Robotics and Autonomous Systems*, 93, 04 2017. doi: 10.1016/j.robot.2017.03.019.

[29] Raúl Mur-Artal and Juan D. Tardós. ORB-SLAM2: an open-source SLAM system for monocular, stereo and RGB-D cameras. *IEEE Transactions on Robotics*, 33(5):1255–1262, 2017. doi: 10.1109/TRO.2017.2705103.

[30] Carlos Campos, Richard Elvira, Juan J. Gómez, José M. M. Montiel, and Juan D. Tardós. ORB-SLAM3: An accurate open-source library for visual, visual-inertial and multi-map SLAM. *IEEE Transactions on Robotics*, 37(6):1874–1890, 2021.

[31] Dorian Gálvez-López and Juan D. Tardós. Bags of binary words for fast place recognition in image sequences. *IEEE Transactions on Robotics*, 28:1188–1197, 2012.

[32] Ming Ouyang, Xuesong Shi, Yujie Wang, Yuxin Tian, Yingzhe Shen, Dawei Wang, Peng Wang, and Zhiqiang Cao. A collaborative visual slam framework for service robots, 2021. URL `https://arxiv.org/abs/2102.03228`.

[33] Patrik Schmuck, Thomas Ziegler, Marco Karrer, Jonathan Perraudin, and Margarita Chli. Covins: Visual-inertial slam for centralized collaboration. *arXiv preprint arXiv:2108.05756*, 2021.

[34] Patrik Schmuck and Margarita Chli. CCM-SLAM: Robust and efficient centralized collaborative monocular simultaneous localization and mapping for robotic teams. In *Journal of Field Robotics (JFR)*, 2018.

[35] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. volume 3, 01 2009.

[36] Jian-Huang Lai, Cheng-Lin Liu, Xilin Chen, Jie Zhou, Tieniu Tan, Nanning Zheng, and Hongbin Zha. *Pattern Recognition and Computer Vision First Chinese Conference, PRCV 2018, Guangzhou, China, November 23-26, 2018, Proceedings, Part IV*. 11 2018. ISBN 978-3-030-03341-5. doi: 10.1007/978-3-030-03341-5.

[37] Francisco Raverta Capua. Evaluación de técnicas de visual-slam para estimar la trayectoria de un vehículo terrestre en un ambiente frutícola. Universidad Nacional del Comahue, Neuquén, Argentina, 07 2019.

[38] Raúl Mur-Artal. Bagfromimages. `https://github.com/raulmur/BagFromImages`, 2014.

[39] Michael Grupp. evo: Python package for the evaluation of odometry and slam. `https://github.com/MichaelGrupp/evo`, 2017.

[40] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Revised Fourth Edition, Fourth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2011. ISBN 0123747503.

[41] The Open Group. Single unix®specification, 2018. URL https://publications.opengroup.org/t101?_ga=2.101964993.1339867181.1676815112-1656385643.1676815112.

[42] Ravi Saive. Nethogs – monitor linux network traffic usage per process. 2021. URL https://www.tecmint.com/nethogs-monitor-per-process-network-bandwidth-usage-in-real-time/.

[43] S. Umeyama. Least-squares estimation of transformation parameters between two point patterns. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(4):376–380, 1991. doi: 10.1109/34.88573.