Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

SLAM denso, globalmente consistente y acelerado por GPU

Tesis de Licenciatura en Ciencias de la Computación

Emiliano Höss

Director: Pablo De Cristóforis

Buenos Aires, noviembre de 2024

SLAM DENSO, GLOBALMENTE CONSISTENTE Y ACELERADO POR GPU

Un sistema de SLAM (Simultaneous Localization and Mapping) denso es esencial para los robots móviles, ya que no sólo proporciona la localización del agente sino que también permite la navegación, la planificación de trayectorias, la evasión de obstáculos y la toma de decisiones en entornos no estructurados. A medida que aumentan las demandas computacionales, crece el uso de GPU en sistemas de SLAM denso. En este trabajo, presentamos coVoxSLAM, un novedoso sistema de SLAM volumétrico acelerado por GPU que aprovecha al máximo la potencia del procesamiento paralelo para construir mapas globalmente consistentes incluso en entornos de gran escala. El sistema se evaluó en diferentes plataformas (GPU discreta e integrada) y se comparó con el estado del arte. Los resultados obtenidos utilizando conjuntos de datos públicos muestran que coVoxSLAM ofrece una mejora significativa del rendimiento considerando los tiempos de ejecución manteniendo al mismo tiempo una localización precisa. Como contribución a la comunidad, el sistema desarrollado está disponible como código abierto en GitHub[1].

Palabras claves: SLAM, Aceleración por GPU, Mapeo Volumétrico.

[1] https://github.com/lrse-uba/coVoxSLAM

GPU ACCELERATED AND GLOBALLY CONSISTENT DENSE SLAM

A dense SLAM system is essential for mobile robots, as it provides no only localization, but also allows navigation, path planning, obstacle avoidance, and decision-making in unstructured environments. Due to increasing computational demands the use of GPUs in dense SLAM is expanding. However, porting algorithms designed initially for CPUs to GPUs are not straightforward because of significant differences in hardware architectures and resources. In this work, we present coVoxSLAM, a novel GPU-accelerated volumetric SLAM system that takes full advantage of the parallel processing power of the GPU to build globally consistent maps even in large-scale environments. It was deployed an tested on different platforms (discrete and embedded GPU) and compared with the state of the art. The results obtained using public datasets show that coVoxSLAM delivers a significant performance improvement considering execution times while maintaining accurate localization. The presented system is available as open-source on GitHub[2].

Keywords: SLAM, GPU-accelerated, Volumetric Mapping.

[2] https://github.com/lrse-uba/coVoxSLAM

Índice general

1. INTRODUCCIÓN

1.1. El problema de SLAM

La Localización y Mapeo Simultáneo (o SLAM por sus siglas en inglés, *Simultaneous Localization and Mapping*) es una técnica utilizada para construir un modelo o mapa de un entorno desconocido al mismo tiempo que se localiza a sí mismo dentro de ese mapa en tiempo real. El SLAM ha sido un foco de investigación en robótica móvil en las últimas décadas. Se han ido proponiendo soluciones cada vez más eficientes, utilizando diferentes enfoques para distintos tipos de sensores [1].

Para alcanzar el objetivo de navegar e interactuar de manera autónoma en entornos desconocidos, los sistemas de SLAM necesitan construir un modelo interno del mundo observado, es decir, una representación abstracta que actue como mapa aproximandose al ambiente real que esta siendo explorado. Las estimaciones que producen los robots de su propio movimiento se realiza de manera incremental a medida que se mueven por el entorno, esto conduce a errores de aproximación en la pose y a un mapeo inconsistente. Para mitigar los efectos inevitables que produce la incertidumbe sobre estos cálculos y mantener un mapa globalmente consistente se utilizan técnicas de cierre de ciclo (*loop closure* en inglés) que consiste en detectar cuando el robot ha regresado a una posición ya visitada para actualizar el mapa y reducir errores acumulados en la localización.

1.1.1. Mapeo esparzo

Una representación común en sistemas de SLAM es modelar el mapa como un conjunto disperso de puntos de referencia, que representan características identificadas arbitrariamente en el ambiente explorado, que pueden corresponderse a puntos, líneas, esquinas, u otras características. El mapeo basado en características convierte los datos crudos del sensor en un conjunto de puntos de referencia (o claves) que se utilizan para la construcción del mapa. Muchos sistemas SLAM bien conocidos siguen este enfoque y han demostrado crear mapas basados en características globalmente consistentes en tiempo real [2–4]. Sin embargo, estos mapas basados en características son de uso limitado para tareas que no exceden más allá de la localización, debido a las dificultades de extraer la forma y la conectividad de las superficies a partir de un conjunto escaso de muestras.

1.1.2. Mapeo denso

Otra manera de representar el ambiente explorado es a partir de mapas densos. Al contrario de los mapas esparzos, las representaciones densas buscan una estimación de la geometría del ambiente explorado, tan detallada como sea necesario. Estos mapas son adecuados no solo para la estimación de la pose, sino también para la reconstrucción de escenas, la planificación de trayectorias, la detección de objetos, la evasión de obstáculos y el control del movimiento. El primer sistema SLAM denso que utiliza la consistencia fotométrica de cada píxel para estimar la trayectoria de una cámara en mano fue [5]. Posteriormente, se desarrollaron varios sistemas SLAM densos basados en la técnica de ajuste de paquetes fotométricos (*Bundle Adjustment*) [6, 7] y actualmente se estan desarrollando técnicas que utilizan aprendizaje profundo [8–10]. Sin embargo, estos sistemas aún requieren muchos recursos computacionales y no logran ejecutar en tiempo real en escenarios de gran escala. Tener la capacidad de crear mapas densos y globalmente consistentes en tiempo real sigue siendo un desafío para resolver el problema de la navegación autónoma a largo plazo en entornos complejos. A su vez, el costo computacional de los métodos de SLAM denso sigue representando un desafío para aplicaciones de tiempo real que deben procesarse en unidades de cómputo limitadas como las que se encuentran a bordo de los robots móviles [6].

1.1.3. Particionamiento espacial del entorno

El mapa que se genera en los sistemas de SLAM a partir de la trayectoria de exploración del robot en el entorno solo incluye las zonas alcanzadas por sus sensores. Por lo tanto, se debe elegir una representación que permita mapear las áreas exploradas y saltear las áreas que aún no han sido exploradas. Pero también debe ser una representación flexible que permita actualizar y agregar zonas nuevas a medida que robot explora el ambiente, sin necesidad de reconstruir el mapa por completo. Por lo tanto, se busca generar una representación de más alto nivel que refleje alguna gerarquía o disposición en los datos registrados desde los sensores. Los sistemas SLAM utilizan comunmente dos opciones para este problema: los árboles octales también conocidos como octomaps [11] y las tablas de hash [12–15]. Los árboles octales ofrecen una mayor flexibilidad al momento de adaptarse a los distintos niveles de fragmentación que se producen en los espacios no explorados al momento de recorrer el mapa, pero no resultan tan eficientes para la indexación de un elemento puntual. Por su parte, las tablas de hash son más simples de implementar y los algoritmos de indexación resultan más eficientes. Su mayor dificultad viene dada por seleccionar una función de hash adecuada que minimice la cantidad de colisiones dentro de la tabla. La implementación en GPU de este tipo de estructuras conlleva cierta

complejidad, uno de los trabajos más recientes que aborda este problema es ASH [15] que propone un método para resolver el hashing espacial de manera paralela en GPU.

1.1.4. Funciones de distancia con signo

Las Funciones de Distancia con Signo (SDF por su nombre en inglés *Signed Distance Functions*) calculan la distancia entre un punto dado y una superficie o un objeto, indicando además de qué lado del objeto se encuentra el punto. Cuando el punto está fuera del objeto o superficie la distancia es positiva, cuando el punto está dentro del objeto es negativa y cuando el punto está exactamente sobre la superficie la distancia es cero. Las funciones de distancia con signo son una herramienta poderosa para representar superficies de manera implícita. Introducidas por primera vez en [16], han demostrado ser una representación efectiva para el mapeo denso [17]. La eficacia de estas representaciones en la fusión de datos de profundidad ruidosos de alta frecuencia provenientes de cámaras de profundidad de bajo costo, se mostró primero en [18]. Las ventajas de esta representación han llevado a su adopción en diversas plataformas robóticas [19], donde el contar con información de distancia ha mostrado una utilidad adicional para la planificación de movimiento [20].

Un mapa representado por una Función de Distancia con Signo Truncada (TSDF) es una representación volumétrica utilizada comúnmente en Visión por Computadora para realizar modelos o reconstrucciones 3D, especialmente con sensores RGB-D del tipo del dispositivo Kinect [17]. Mide la distancia desde cada punto en el espacio hasta el origen del sensor: los valores positivos indican puntos fuera de la superficie, los valores negativos representan puntos dentro y cero cuando se encuentra en la propia superficie. La característica de estas funciones "truncadas" limita las distancias más allá de un umbral específico, lo que ayuda a priorizar los datos más relevantes y minimizar el ruido de mediciones distantes. Los mapas TSDF se organizan en una cuadrícula de vóxeles, facilitando la integración eficiente de datos de profundidad a lo largo del tiempo, lo que ayuda a suavizar inconsistencias y ruido.

1.1.5. Consistencia global

Alejarse de la representación del entorno mediante mapas esparsos de características para adoptar una representación densa hace que la construcción de un mapa consistente a nivel global sea un gran desafío, ya que la optimización global del mapa rápidamente se vuelve intratable a medida que aumenta el tamaño del mapa. Por lo tanto, muchos sistemas existentes, se limitan a la operación en entornos de pequeña escala donde la deriva es limitada, o almacenan datos de entrada del sensor sin procesar, de modo que

se pueda calcular un mapa globalmente preciso desde cero a medida que se dispone de más datos, pero ésto vuelve a limitar el entorno de trabajo. La falta de un método sistemático para corregir el mapa a partir de relacionar la nueva información capturada con las poses pasadas, como típicamente son los cierres de ciclo en la trayectoria, significa que los mapas basados en SDF son susceptibles a la inconsistencia global.

Por otro lado, en ambientes donde no podemos contar con información de geolocalización absoluta como GPS o ésta puede estar limitada, resulta necesario construir mapas que sean globalmente consistentes. Como los robots van estimando su movimiento de manera incremental a medida que navegan por el ambiente, esto lleva a errores en la construcción del mapa y en el cálculo de la localización en el mismo, que se van acumulando de manera no acotada con el tiempo. Por lo tanto, contar con la posibilidad de construir mapas densos y globalmente consistentes del ambiente en tiempo real representa una bala de plata para resolver el problema de la navegación autónoma a largo plazo en ambientes complejos [6].

Una alternativa para abordar este problema es representar el entorno reconstruido como una colección de submapas. La ventaja de este enfoque es que la pose del sensor, en el momento en que se realiza una nueva registración de un nuevo submapa, solo necesita ser registrada con respecto al submapa actual. Si la trayectoria completa no se considera durante la optimización del mapa denso, los submapas pueden ser restringidos entre sí a través de la alineación geométrica [21]. En [22] esta idea se extiende proponiendo usar una alineación libre de correspondencia basada en la Función de Distancia con Signo Euclidiana (ESDF) que representa una cuadrícula de vóxeles donde cada punto contiene su distancia euclidiana al obstáculo más cercano.

1.1.6. Sistemas SLAM acelerados por GPU

El uso unidades de procesamiento de gráficos (GPU por sus siglas en inglés) en robots móviles, tales como vehículos aéreos y terrestres no tripulados, está aumentando gracias a la disponibilidad en el mercado de placas equipadas con GPU de bajo costo, pequeñas dimensiones y bajo consumo. Esto permite el procesamiento de grandes volúmemes de datos provenientes de los sensores en tiempo real. Portar sistemas diseñados originalmente para CPU en placas GPU no es sencillo debido a que se requiere de modificaciones en la arquitectura y en la forma en la que se procesan los datos.

Durante la última década, se han realizado esfuerzos para implementar algoritmos SLAM en múltiples unidades de procesamiento en sistemas heterogéneos CPU-GPU dado que esto representa un enfoque más realista para su uso en la vida real, como el caso de [23] donde se presenta una versión optimizada para GPU del sistema ORB-SLAM2 [2]. Este trabajo representa el flujo de datos como un grafo, lo que permite subdividirlo en componentes

que facilitan su ejecución paralela en GPU. Para aprovechar al máximo la naturaleza heterogénea de las placas que se comercializan hoy en dia (es decir, CPUs multinúcleo combinadas con GPU de muchos núcleos), el trabajo utiliza dos niveles de paralelismo. El primero se da por la implementación paralela para GPU de un conjunto de sub-bloques de seguimiento. El segundo se da por la implementación de una diseño en cascada de 8 etapas de dichos sub-bloques.

Para lograr eso, primero se presenta el modelo del bloque de extracción de características de ORB-SLAM como un grafo acíclico dirigido (DAG) adoptando el estándar OpenVX. La transformación de la implementación original, que fue concebida originalmente solo para CPUs, en una ejecución paralela CPU/GPU requiere un control sobre la comunicación entre el código que se ejecuta en las CPUs y en la GPU. En particular, la fase de mapeo es crítica, requiriendo una sincronización temporal entre los bloques del algoritmo para tener éxito. La implementación propuesta en este trabajo resulto ser 5 veces más rápida que la original sobre placas Nvidia sin perder precisión en el resultado.

Aún así, la portabilidad de algoritmos originalmente implementados en CPUs para su re-implementación en GPUs no es sencilla debido a las diferencias en las arquitecturas de hardware y recursos disponibles. Tal es el caso de [23–25], donde se evalúan versiones aceleradas por GPU de sistemas SLAM basados en características. La GPU también ha sido utilizada para algunos componentes de los sistemas de SLAM RGB-D denso [17, 26]. En el contexto de SLAM utilizando LiDAR, el uso de GPU se limitó principalmente a acelerar la detección de patrones en las imágenes provenientes de los sensores en el Front-end [27, 28]. Sin embargo, en la mayoría de los trabajos anteriores, el Back-end del sistema, y específicamente, la optimización del grafo de poses que subyase a muchos de los sistemas de SLAM sigue siendo realizada en una CPU.

1.2. Contribuciones de esta tesis

El sistema de SLAM presentado en este trabajo, al que acuñamos co-VoxSLAM, formula el problema como una optimización de grafos de poses, incluyendo restricciones de odometría y de registración del sensado. Siguiendo el enfoque de Voxgraph, representamos el entorno como una colección de submapas modelados como SFDs, que se alinean mediante la optimización del grafo de poses. La coherencia local se mantiene mediante restricciones de registro entre pares de submapas superpuestos. El costo computacional de estas alineaciones entre submapas se reduce significativamente aprovechando la representación subyacente basada en SFDs de los submapas. Esto permite optimizar el mapa global cada vez que se agrega un nuevo submapa, manteniendo así la consistencia global sin perjudicar el desempeño del sistema.

Las principales contribuciones de este trabajo pueden resumir de la siguiente manera:

- Una biblioteca para optimizar la gestión y disposición (*layout*) de la memoria en sistemas de SLAM de hardware embebido utilizando el modelo de organización de datos concocido como Estructura-de-Arrays (SoA por sus siglas en inglés)
- Una tabla de hash con índices y valores que utiliza la biblioteca SoA antes descripta para implementar el marco de trabajo ASH que permite el intercambio óptimo de datos entre diferentes módulos.
- Un nuevo método para la generación de mapas volumétricos basado en el casteo de rayos (*ray casting*) que mejora los tiempos de ejecución respecto de los métodos basados en mapeo de proyecciones.
- La implementación en GPU del algoritmo de Levenberg-Marquardt para la resolución iterativa por cuadrados mínimos de sistemas no-lineales, inspirado en la biblioteca Ceres Solver y optimizado para el problema de SLAM.
- La implementación de un conjunto de funcionalidades de la biblioteca Eigen para el álgebra lineal, de la biblioteca Kindr para las operaciones sobre cuaterniones y de la biblioteca JET de números duales para diferenciación automática utilizando la biblioteca SoA antes descripta.
- La publicación completa del código fuente (Front-end y Back-end) del sistema coVoxSLAM en GitHub[1].

1.3. Disposición de los capítulos

El resto de la tesis se organiza de la siguiente manera: el Capítulo 2 introduce los conceptos preliminares para entender la formulación probabilística del problema de SLAM, el uso de grafos de factores como una representación para abordarlo como un problema de optimización no lineal, los métodos iterativos basados en cuadrados mínimos que se utilizan para resolverlo, las estructuras de datos que se utilizan para el particionamiento espacial del entorno y las métricas de error que se definen para evaluar los sistemas de SLAM. En el Capítulo 3 se presenta el estado del arte, haciendo foco en los trabajos previos en los cuales se basa el desarrollo del sistema que se presenta en esta tesis, analizando sus ventajas y limitaciones. En el Capítulo 4 se describe detalladamente el nuevo sistema coVoxSLAM, incluyendo las cuestiones de implementación en GPU que resultan fundamentales para alcanzar un desempeño óptimo. El Capítulo 5 muestra los resultados

[1] https://github.com/lrse-uba/coVoxSLAM

obtenidos en los experimentos realizados. Finalmente, el Capítulo 6 presenta las conclusiones de todo el trabajo realizado.

2. CONCEPTOS PRELIMINARES

2.1. Notación matemática

- Variables en minúscula y negrita ($\mathbf{x}$) denotan vectores o, elementos que podrían representarse como vectores. Mientras que variables en mayúscula y negrita ($\mathbf{A}$) denotan matrices, y variables en minúsculas que no están en negrita indican escalares o elementos (x). Por último, variables en mayúsculas que no están en negrita indican conjuntos (X).
- Un superíndice delantero en minúsculas indica el marco de referencia para un elemento: ${}^{W}\mathbf{x}$
- Un subíndice y un superíndice delanteros indica un vector de traslación entre orígenes de los marcos de referencia: ${}^{w}_{r}\mathbf{t}$
- Se notará una matriz de transformación como ${}^{w}_{r}\mathbf{T}$, de manera que transforma vectores de un marco de referencia R a un marco W : ${}^{w}\mathbf{x} = ({}^{w}_{r}\mathbf{T})^{r}\mathbf{x}$.
- $P(\mathbf{x})$ denota la función de densidad de probabilidad (PDF) de una variable aleatoria (v.a.) $\mathbf{x}$.
- $P(x|y)$ denota la PDF condicional de una v.a. $\mathbf{x}$ dada otra v.a. $\mathbf{y}$.
- $\mathbf{x} \sim \mathcal{N}(\mu, \sigma^2)$ denota una v.a. $\mathbf{x}$ que sigue una distribución de probabilidad normal con media μ y desvío estándar σ.
- Análogamente $\mathbf{x} \sim \mathcal{N}(\mu, \Sigma)$ denota una variable que sigue una distribución normal multivariada con media $\mu \in \mathbb{R}^n$ y matriz de covarianza $\Sigma \in \mathbb{R}^{n \times n}$, donde Σ es siempre semidefinida positiva.
- Dadas dos v.a. $\mathbf{a}$, $\mathbf{b} \in \mathbb{R}^n$ extraídas de la misma distribución normal con matriz de covarianza $\mathbf{C} \in \mathbb{R}^{n \times n}$, $||a-b||_C$ denota la distancia Mahalanobis entre a y b, la cual se calcula como $\sqrt{(a-b)^{\intercal} > \mathbf{C}^{-1}(a-b)}$.

2.2. Distribución Normal Multivariada

Los modelos probabilísticos utilizados en esta tesis asumen una distribución normal (o gaussiana) multivariada para las v.a. La parametrización más común de una distribución normal es utilizando sus momentos:

$$\mathbf{x} \sim \mathcal{N}(\mu, \Sigma), \qquad P(\mathbf{x}) := \frac{1}{(2\pi)^{n/2}\sqrt{|\Sigma|}} \exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^{\intercal}\Sigma^{-1}(\mathbf{x}-\mu)\right)$$

donde $\mathbf{x}$ denota una v.a. normal multivariada con media $\mu \in \mathbb{R}^n$ y matriz de covarianza $\Sigma \in \mathbb{R}^{n \times n}$. Dependiendo de la formulación del problema, otra parametrización alternativa es dada en forma de información:

$$\mathbf{x} \sim \mathcal{N}^{-1}(\nu, \Omega), \qquad P(\mathbf{x}) := \frac{\exp\left(-\frac{1}{2}\nu^\intercal \Omega^{-1} \nu\right)}{(2\pi)^{n/2}\sqrt{|\Omega^{-1}|}} \exp\left(-\frac{1}{2}\mathbf{x}^\intercal \Omega^{\mathbf{x}} + \mathbf{x}^\intercal \nu\right)$$

donde $\nu \in \mathbb{R}^n$ es el vector de información y $\Omega \in \mathbb{R}^{n \times n}$ es la matriz de información de Fisher. Ambas matrices Σ y Ω son simétricas semidefinidas positivas. Ambas parametrizaciones son equivalentes y se desprenden las siguientes igualdades:

$$\nu = \Sigma^{-1}\mu, \qquad \Omega = \Sigma^{-1}$$

Por lo general, para optimizar funciones de costo para la estimación de las media de la distribución, se reducen estas expresiones tomando el logaritmo natural. Considerando la función de densidad normal para una variable aleatoria $\mathbf{x} \sim \mathcal{N}(\mu, \Sigma)$:

$$P(\mathbf{x}) = \eta \exp\left(-\frac{1}{2}||\mathbf{x} - \mu||^2_{\Sigma^{-1}}\right)$$

donde η es un factor de normalización constante que no depende de μ. El logaritmo de la función de probabilidad es entonces:

$$ln(P(\mathbf{x})) \propto -\frac{1}{2}||\mathbf{x} - \mu||^2_{\Sigma^{-1}} \tag{2.1}$$

ya que el termino $ln(\eta)$ no depende de μ y puede ser ignorado, en el caso de una optimización.

2.3. Propagación de la incertidumbre

Las variables estimadas que modelan elementos físicos del mundo real están predestinadas a acarrear un cierto grado de incertidumbre debido a las limitaciones de los sensores al obtener mediciones. Suponiendo variables que siguen una determinada distribución estadística, estos errores se pueden modelar y cuantificar en términos de la desviación estándar. Además, si se aplica alguna función sobre estas variables aleatorias, resulta beneficioso propagar la incertidumbre existente al resultado de la función.

Sea $f : \mathbb{R}^n \to \mathbb{R}^m, f = \mathbf{A}\mathbf{x} + \mathbf{b}$ una función lineal sobre una variable aleatoria gaussiana multivariada $\mathbf{x}$ con matriz de covarianza $\mathbf{\Sigma}_\mathbf{x} \in \mathbb{R}^{n \times n}$. Entonces, la matriz de covarianza $\mathbf{\Sigma}_f$ está dada por

$$\mathbf{\Sigma}_f = \mathbf{A}\mathbf{\Sigma}_{\boldsymbol{x}}\mathbf{A}^T.$$

Si f es una combinación no lineal de $\mathbf{x}$, generalmente, no existe una formula exacta que pueda ser derivada para propagar la incertidumbre. Por lo tanto es usual emplear la expansión de Taylor de primer orden de f como aproximación:

$$f \approx f_0 + \mathbf{J}\mathbf{x},$$

donde $\mathbf{J}$ son las derivadas (parciales) de primer orden de f con respecto a $\mathbf{x}$, comúnmente referida como la matriz Jacobiana de f. Dado que esta ecuación es lineal y f_0 es constante, y no contribuye al error, la propagación puede ser aproximada como

$$\mathbf{\Sigma}_f \approx \mathbf{J}\mathbf{\Sigma}_\mathbf{x}\mathbf{J}^T.$$

Es importante señalar que la propagación a través de funciones no lineales realizada de esta manera produce estimadores sesgados para la covarianza. El alcance de este sesgo depende del grado de no linealidad de la función f. Para más detalle sobre la propagación de la incertidumbre en el contexto de la robótica móvil, se recomienda leer el informe técnico de Kai Oliver Arras [29] donde se hace un estudio exhaustivo sobre el tema, derivando explícitamente las ecuaciones de propagación y caracterizando la extensión y precisión de las aproximaciones aplicadas.

2.4. Modelo probabilístico de SLAM

El enfoque de la robótica probabilística persigue el objetivo de obtener estimaciones del estado del robot, caracterizando de manera precisa las incertidumbres inherentes a su actuación y percepción. La percepción del entorno por parte de los robots se lleva a cabo mediante sensores que suministran mediciones con distintos niveles de incertidumbre y error. Estos errores pueden surgir por diversas razones, como entornos altamente impredecibles, la presencia de ruido en los sensores, representaciones discretas, entre otros. Los algoritmos probabilísticos destacan sobre las técnicas alternativas en numerosas aplicaciones debido a su capacidad para realizar un cálculo de la calidad de las estimaciones, modelando la propagación de las incertidumbres a través de los diferentes procesos. Además, los enfoques probabilísticos permiten planificar mejor las acciones a tomar a partir de evaluar las incertidumbres de las variables estimadas.

A continuación, se presenta una derivación probabilística para el problema del SLAM. La mayoría de los conceptos y modelos han sido desarrollados a lo largo de décadas de investigación, destacando la importancia del modelo probabilístico gráfico conocido como Grafo de Factores (*Factor Graph* en inglés), el cual ha adquirido un considerable reconocimiento en la comunidad de SLAM. Para obtener una comprensión más detallada sobre el tema, se recomienda consultar la siguiente bibliografía: Probabilistic Robotics [30], Factor Graphs for Robot Perception [31], State Estimation for Robotics [32].

2.4.1. Formulación del problema

El SLAM puede verse como un problema de inferencia a gran escala donde el objetivo principal es estimar los estados del robot y del entorno, dadas las mediciones obtenidas de los sensores a lo largo de la trayectoria. A medida que el robot se mueve y explora el ambiente, el problema de estimación del SLAM aumenta su número de variables. Definimos las variables de estado como $X = \{X_p, X_l\}$ donde $X_p = \{\mathbf{x}_{p_1} ... \mathbf{x}_{p_n}\}$ representa n poses de robot y $X_l = \{\mathbf{x}_{l_1} ... \mathbf{x}_{l_m}\}$ representa m landmarks. El conjunto de mediciones se indicará como $Z = \{\mathbf{z}_{ij}\} \cup \{\mathbf{u}_{ij}\}$ donde $\mathbf{z}_{ij}$ es una observación del landmark j desde la pose i y $\mathbf{u}_{ij}$ es un incremento de pose relativo medido entre las poses i y j, estas mediciones pueden surgir de la integración de movimiento entre estados contiguos del robot por medio de la utilización de sensores propioceptivos de movimiento como encoders, una IMU o a partir de un método de detección y cierre de ciclos.

La distribución de probabilidad objetivo, para la cual desearemos estimar sus parámetros, se puede establecer y factorizar usando la regla de Bayes como:

$$P(X|Z) = \frac{P(Z|X)P(X)}{P(Z)}, \tag{2.2}$$

donde $P(Z|X)$ es la verosimilitud de las mediciones, $P(X)$ es el término prior que determina la probabilidad marginal del estado y $P(Z)$ es considerado como un factor de normalización. De esta manera, se puede obtener una solución para las variables de estado maximizando la distribución de probabilidad 2.2, este enfoque se conoce como estimación de probabilidad máxima a posteriori (maximum a posteriori, MAP):

$$\begin{aligned} \widehat{X}^{MAP} &= \arg\max_X P(X|Z) \\ &= \arg\max_X \frac{P(Z|X)P(X)}{P(Z)} \\ &= \arg\max_X P(Z|X)P(X). \end{aligned}$$

El factor de normalización $P(Z)$ se puede eliminar ya que no afecta la maximización, y asumiendo mediciones condicionalmente independientes, el término de probabilidad se puede factorizar aún más como:

$$\widehat{X}^{MAP} = \arg\max_X \prod_{\mathbf{z}_{ij} \in Z} P(\mathbf{z}_{ij}|X) \prod_{\mathbf{u}_{ij} \in Z} P(\mathbf{u}_{ij}|X)\, P(X), \tag{2.3}$$

donde se pueden distinguir tres grupos diferentes de factores, cada uno de ellos representa una fuente de información diferente: 1) las mediciones de observación $P(\mathbf{z}_{ij}|X)$, 2) mediciones de movimiento $P(\mathbf{u}_{ij}|X)$ y 3) información

prior del estado ($P(X)$). Asumiendo que estos tres términos están distribuidos normalmente, se presentan a continuación los modelos que describen el comportamiento esperado para cada uno.

El modelo de observación describe las mediciones esperadas del sensor, cuando el j-ésimo *landmark* es observado desde la i-ésima pose, $\mathbf{z}_{ij} = h_{ij}(\mathbf{x}_{p_i}, \mathbf{x}_{l_j}) + \mathbf{v}_{ij}$ donde $h_{ij} : \mathbb{R}^{dim(\mathbf{x}_{p_i}, \mathbf{x}_{l_j})} \rightarrow \mathbb{R}^{dim(\mathbf{z}_{ij})}, \mathbf{v}_{ij} \sim \mathcal{N}(0, \mathbf{R}_{ij})$ y $\mathbf{R}_{ij}$ es la matriz de covarianza de observación. De esta manera, se obtiene la distribución condicional $\mathbf{z}_{ij}|X \sim \mathcal{N}(h_{ij}|_X, \mathbf{R}_{ij})$.

El modelo de movimiento describe las mediciones de incremento de movimiento entre dos poses, $\mathbf{u}_{ij} = \mathrm{g}_{ij}(\mathbf{x}_{p_i}, \mathbf{x}_{p_j}) + \mathbf{w}_{ij}$ donde $\mathrm{g}_{ij} : \mathbb{R}^{dim(\mathbf{x}_{p_i}, \mathbf{x}_{p_j})} \rightarrow \mathbb{R}^{dim(\mathbf{u}_{ij})}$ es un vector de ruido aditivo que sigue una distribución normal $\mathbf{w}_{ij} \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}_{ij})$ y $\mathbf{Q}_{ij}$ representa la incertidumbre del movimiento. Este modelo define la distribución condicional, $\mathbf{u}_{ij}|X \sim \mathcal{N}(\mathrm{g}_{ij}|_X, \mathbf{Q}_{ij})$. El incremento de movimiento entre poses se calcula con g, y de esta manera, es posible modelar mediciones de movimiento de manera análoga al modelo de observación de landmarks, permitiendo tratar de manera unificada las mediciones provenientes de diferentes fuentes.

El estado prior refleja un conocimiento previo resumido sobre las variables de estado, ya sea por información de inicialización de los estados o a través de un proceso de marginalización. Este término prior es también asumido como distribuido normalmente e impone una expectativa sobre las variables que se están estimando, es decir, $\mathbf{x} \sim \mathcal{N}(\mathbf{x}_\Pi, \Sigma_\Pi)$.

Habiendo definido distribuciones para todas las variables involucradas, es posible simplificar aún más la maximización de la Ecuación (2.3) omitiendo el término de normalización de las probabilidades normales multivariadas, de manera de introducir la siguiente función proporcional:

$$
\begin{aligned}
P(\mathbf{z}_{ij}|X) &\propto \exp\left(-\frac{1}{2}|\mathbf{z}_{ij} - h_{ij}(\mathbf{x}_{p_i}, \mathbf{x}_{l_j})|_{\mathbf{R}_{ij}}{}^2\right) \\
P(\mathbf{u}_{ij}|X) &\propto \exp\left(-\frac{1}{2}|\mathbf{u}_{ij} - \mathrm{g}_{ij}(\mathbf{x}_{p_i}, \mathbf{x}_{pj})|_{\mathbf{Q}_{ij}}{}^2\right) \\
P(X) &\propto \exp\left(-\frac{1}{2}\|\mathbf{x}_\Pi - \mathbf{x}\|^2_{\Sigma_\Pi}\right),
\end{aligned}
$$

donde $\|\mathbf{x}_\Pi - \mathbf{x}\|^2_{\Sigma_\Pi} = (\mathbf{x}_\Pi - \mathbf{x})^\intercal \Sigma^{-1} (\mathbf{x}_\Pi - \mathbf{x})$ denota la distancia de Mahalanobis cuadrada. La maximización en la Ecuación (2.3) puede ser entonces escrita como:

$$\widehat{X}^{MAP} = \arg\max_{X} \prod_{\mathbf{z}_{ij} \in Z} P(\mathbf{z}_{ij}|X) \prod_{\mathbf{u}_{ij} \in Z} P(\mathbf{u}_{ij}|X)\, P(X) \tag{2.4}$$

$$= \arg\max_{X} \Big(\prod_{\mathbf{z}_{ij} \in Z} \exp(-\frac{1}{2} \|\mathbf{z}_{ij} - h_{ij}(\mathbf{x}_{p_i}, \mathbf{x}_{l_j})\|^2_{\mathbf{R}_{ij}}) \tag{2.5}$$

$$\prod_{\mathbf{u}_{ij} \in Z} \exp(-\frac{1}{2} \|\mathbf{u}_{ij} - \mathrm{g}_{ij}(\mathbf{x}_{p_i}, \mathbf{x}_{p_j})\|^2_{\mathbf{Q}_{ij}})$$

$$\exp(-\frac{1}{2} \|\mathbf{x}_\Pi - \mathbf{x}\|^2_{\Sigma_\Pi}) \Big).$$

Es posible además aplicar la función logarítmica en la Ecuación (2.5) para convertir un producto de factores en una suma y, al ser una función monótona, los máximos se conservan. Además, es una práctica común aplicar el logaritmo negativo para traducir la maximización, en una minimización sobre el logaritmo negativo, con lo cual el problema de SLAM queda formulado como sigue a continuación:

$$\widehat{X}^{MAP} = \arg\min_{X} \Bigg(\sum_{\mathbf{z}_{ij} \in Z} \|\mathbf{z}_{ij} - h_{ij}(\mathbf{x}_{p_i}, \mathbf{x}_{l_j})\|^2_{\mathbf{R}_{ij}} \tag{2.6}$$

$$+ \sum_{\mathbf{u}_{ij} \in Z} \|\mathbf{u}_{ij} - \mathrm{g}_{ij}(\mathbf{x}_{p_i}, \mathbf{x}_{pj})\|^2_{\mathbf{Q}_{ij}} \tag{2.7}$$

$$+ \|\mathbf{x}_\Pi - \mathbf{x}\|^2_{\Sigma_\Pi} \Bigg). \tag{2.8}$$

2.5. Grafos de factores

Formalmente, un grafo de factores es un grafo bipartito que representa la factorización de una función. Habiendo planteado el problema SLAM desde la perspectiva de la teoría de la probabilidad, se utilizarán grafos de factores para representar la factorización de la función de distribución de probabilidades objetivo.

Sea $\mathcal{G} = (\mathcal{F}, \mathcal{X}, \mathcal{E})$ un grafo de factores bipartito, con dos tipos de nodos: factores $\mathrm{f}_k \in \mathcal{F}$ y variables $\mathbf{x}_i \in \mathcal{X}$; las aristas $e_{ki} \in \mathcal{E}$ relacionan nodos de factores con nodos de variables. En este caso, los nodos de variables $\mathbf{x}$ corresponden a las variables de estado previamente definidas como X (Sec. 2.4.1), y cada factor f_k es una función de las variables $\mathbf{x}$ en su adyacencia. La Figura 2.1 muestra una situación de ejemplo en la que un robot navega mapeando un espacio, donde se agregan dos poses y varios landmarks son sensados a lo largo de la trayectoria.

El conjunto de nodos de variables adyacentes a un factor f_k se indicará como $\mathbf{x}_k$, y el grafo definirá la factorización de una función global $F(\mathbf{x})$:

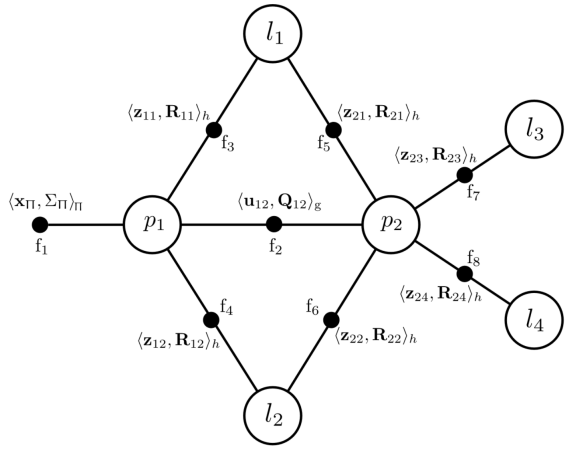

Fig. 2.1: Ejemplo de grafo de factores compuesto por dos poses de robot y cuatro landmarks. Los puntos negros representan factores denominados f_k y los pares $\langle \bullet, \bullet \rangle_\bullet$ representan parámetros embebidos, como las mediciones de los sensores, su incertidumbre estimada y la función del modelo.

$$F(\mathbf{x}) = \prod_{k \in \mathcal{G}} \mathrm{f}_k(\mathbf{x}_k). \tag{2.9}$$

Las mediciones $\mathbf{z}_k \in Z$ obtenidas y sus covarianzas acompañantes serán parámetros embebidos de cada factor y, de esta manera, se define

$$\mathrm{f}_i(\mathbf{x}_k) \propto \Big\{ \exp(-\frac{1}{2} \|\mathbf{z}_k - m_k(\mathbf{x}_k)\|^2_{\Sigma_k}) \Big\}, \tag{2.10}$$

donde m_k representa cualquier función de modelo, como se explicó anteriormente, de observación, movimiento o prior. Notar que se flexibiliza la notación utilizada, dado que $\mathbf{z}_k$ también puede tomar la forma de un prior sobre las variables relacionadas por un factor prior. La estimación de probabilidad máxima a posteriori en la ecuación (2.5) se reduce a maximizar el producto de todos los factores del grafo:

$$\begin{aligned} \widehat{\mathbf{x}}^{MAP} &= \arg \max_X F(\mathbf{x}) \\ &= \arg \max_X \prod_{k \in \mathcal{G}} \mathrm{f}_k(\mathbf{x}_k). \end{aligned}$$

Aplicando el logaritmo negativo, análogamente a como se hizo en la Ecuación 2.8, y eliminando el término $-\frac{1}{2}$ es posible resolverlo de manera equivalente minimizando

$$\widehat{\mathbf{x}}^{MAP} = \arg\min_{\mathbf{x}} \sum_{k \in \mathcal{G}} \|\mathbf{z}_k - m_k(\mathbf{x}_k)\|^2_{\Sigma_k}. \tag{2.11}$$

Esta última función objetivo es conocida como un problema no lineal de optimización de cuadrados mínimos, el cual a partir de valores iniciales para $\mathbf{x}_k$ se puede resolver aplicando métodos iterativos como el de Gauss-Newton [33], el algoritmo de Levenberg-Marquardt [34, 35] o el método de región de confianza Dogleg [36]. Incluso cuando las funciones de modelo m_k son no lineales, estos métodos aplican sucesivas aproximaciones lineales en la Ecuación 2.11 y son capaces de converger al mínimo. Una observación no evidente es que, a excepción de los factor priors, las mediciones $\mathbf{z}_k$ y las funciones de modelo m_k son típicamente de menor dimensión que las incógnitas $\mathbf{x}_k$. En esos casos, un único factor determina la misma probabilidad para un subconjunto infinito del dominio de $\mathbf{x}_k$. Por ejemplo, una medición en la imagen 2D de una cámara se corresponde con un rayo completo de puntos 3D que se proyectan en la misma posición de la imagen. En este caso, solo cuando se combinan múltiples mediciones podemos esperar recuperar una solución única para las variables.

Hay varias implementaciones disponibles altamente eficientes para realizar la optimización de grafos de factores, e incluso con interfaces diseñadas especialmente para aplicaciones robóticas. A modo de ejemplo, cabe mencionar algunas de las más populares como g2o [37], Ceres Solver [38] y GTSAM [39]. Todas estas bibliotecas utilizan algoritmos iterativos de optimización no-lineal basados en cuadrados mínimos. En la siguiente sección se presentan los más utilizados.

2.6. Algoritmos iterativos de optimización no-lineal

2.6.1. Introducción

Existen diversos algoritmos iterativos para resolver problemas de optimización no lineal. En esta sección vamos a describir los que consideramos para este trabajo y que son los más utilizandos en las bibliotecas disponibles. Estos algoritmos comparten una estructura básica:

- Se requiere una estimación inicial de las variables.
- Se aplica una estrategia determinada de linealización, aproximando la función de costo objetivo no lineal entorno a la estimación inicial.
- En cada iteración se calcula y aplica un paso de actualización para obtener una estimación más adecuada.
- El procedimiento se repite hasta cumplir un determinado criterio de convergencia o de parada, basado en las estimaciones más recientes.

Dependiendo de cuan rápido el algoritmo se acerca a la solución, pueden dividirse en asintóticamente cuadráticos (también llamados de segundo orden), lo que significa que la magnitud del error decrece aproximadamente al cuadrado en cada iteración cerca de la solución; y asintóticamente o de convergencia lineal (también llamados de primer orden), lo que significa que el error decrece de manera proporcional en cada iteración. Si bien los métodos de convergencia lineal requieren de más iteraciónes para llegar a la solución, y por ende a simple vista pueden parecer menos eficientes a los métodos de segundo orden, en terminos de computo, la cantidad de operaciones requerida en cada iteración, vuelven a estos métodos lineales más convenientes para resolver ciertos tipos de problemas. Además, los algoritmos de convergencia lineal resultan más robustos para problemas de grandes dimensiones o mal condicionados, incluso con inicializaciones pobres de las variables no cercanas a la solución.

2.6.2. Algoritmo de Guass-Newton

Definimos al método iterativo en terminos de la transición desde la actual iteración $\mathbf{x}_c$ hacia la siguiente $\mathbf{x}_+$. Para el caso de sistemas de ecuaciones no lineales $\mathbf{x}_+$ es la raiz del modelo local lineal de F al rededor de $\mathbf{x}_c$

$$\mathbf{x}_+ = \mathbf{x}_c - F'(\mathbf{x}_c)^{-1} + F(\mathbf{x}_c) \tag{2.12}$$

Tambien puede verse a $\mathbf{x}_+$ como la raiz del segundo termino de la expansión de Taylor al rededor de $\mathbf{x}_c$:

$$M_c(\mathbf{x}) = F(\mathbf{x}_c) + F'(\mathbf{x}_c)(\mathbf{x} - \mathbf{x}_c) \tag{2.13}$$

Este modelo se deriva de la expansión de Taylor de F e incluye la primera y segunda derivada, en n dimensiones, su gradiente y la matriz Hessiana.

$$m_c(\mathbf{x}) = f(\mathbf{x}_c) + \nabla f(\mathbf{x}_c)^T(x - \mathbf{x}_c) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_c)^T\nabla^2 f(\mathbf{x}_c)(\mathbf{x} - \mathbf{x}_c) \tag{2.14}$$

Cuando $\nabla^2 f(\mathbf{x}_c)$ es definida positiva, entonces el minimizador de $\mathbf{x}_+$ de m_c tiene solución única $\nabla m_c(x) = 0$, por lo tanto:

$$0 = \nabla m_c(\mathbf{x}_+) = \nabla f(\mathbf{x}_c) + \nabla^2 f(\mathbf{x}_c)(\mathbf{x}_+ - \mathbf{x}_c) \tag{2.15}$$

Por lo tanto la iteración puede expresarse como:

$$\mathbf{x}_+ = \mathbf{x}_c - (\nabla^2 f(\mathbf{x}_c))^{-1}\nabla f(\mathbf{x}_c) \tag{2.16}$$

Los problemas de cuadrados mínimos no lineales ya mencionados en la sección anterior, pueden expresarse como:

$$f(\mathbf{x}) = \frac{1}{2}\sum_{i=1}^{M}\|r_i(\mathbf{x})\|_2^2 = \frac{1}{2}\mathbf{r}(\mathbf{x})^T\mathbf{r}(\mathbf{x}) \tag{2.17}$$

El vector $\mathbf{r}(x) = (r_1, ..., r_M)$ es llamado el Residual. Cuando definimos N como el número de parámetros del problema tenemos que: Si $M = N$ el problema es no lineal, si $M > N$ decimos que el problema esta sobre determinado y si $M < N$ el problema está subdeterminado. Definimos la matriz del Jacobiano $\mathbf{R'}$ de R que es una matriz de $M \times N$ como:

$$\mathbf{R'}(\mathbf{x})_i j = \partial r_i / \partial x_j, 1 \leq i \leq M, 1 \leq j \leq N \tag{2.18}$$

Con esta notación queda que:

$$\nabla f(\mathbf{x}) = \mathbf{R'}(\mathbf{x})^T\mathbf{r}(\mathbf{x}) \in R^N \tag{2.19}$$

El algoritmo de Gauss-Newton descarta el termino de segundo orden $\nabla^2 f$ y computa el paso como:

$$s = -(\mathbf{R'}(\mathbf{x}_c)^T\mathbf{R'}(\mathbf{x}_c))^{-1}\nabla f(\mathbf{x}_c) = -(\mathbf{R'}(\mathbf{x}_c)^T\mathbf{R'}(\mathbf{x}_c))^{-1}\mathbf{R'}(\mathbf{x}_c)^T\mathbf{r}(\mathbf{x}_c) \tag{2.20}$$

Donde la iteración del método se define como $\mathbf{x}_+ = \mathbf{x}_c + s$. La motivación para este cambio es que $\mathbf{R''}(x)\mathbf{r}^T(\mathbf{x})$ se vuelve insignificante para residuales pequeños.

2.6.3. Método de Descenso Pronunciado

Se define la dirección de descenso más pronunciada para $\mathbf{x}$ de la forma $d = -\nabla f(\mathbf{x})$. Este método utiliza esta dirección y actualiza el paso de la iteración $\mathbf{x}_c$ siguiendo:

$$\mathbf{x}_+ = \mathbf{x}_c + \lambda\mathbf{d} \tag{2.21}$$

donde

$$\mathbf{d} = -\nabla f(\mathbf{x}) = -\mathbf{R'}(\mathbf{x})^T\mathbf{r}(\mathbf{x}) \tag{2.22}$$

Los métodos de convergencia lineal utilizan esta dirección para realizar la busqueda de la solución. Se suele utilizar la regla de Armijo para definir el valor de λ y de esta forma controlar el tamaño del paso de cada iteración.

También se pueden utilizar direcciones descendientes basadas en modelos cuadráticos de f de la forma:

$$m(\mathbf{x}) = f(\mathbf{x}_c) + \nabla f(\mathbf{x}_c)^T(\mathbf{x} - \mathbf{x}_c) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_c)^T\mathbf{H}_c(x - \mathbf{x}_c) \tag{2.23}$$

donde $\mathbf{H}_c$ la matriz Hessiana es simetrica definida positiva. Tomando $d = x - \mathbf{x}_c$ minimice $m(x)$ queda que:

$$\nabla m(\mathbf{x}) = \nabla f(\mathbf{x}_c) + \mathbf{H}_c(\mathbf{x} - \mathbf{x}_c) = 0 \tag{2.24}$$

Y por lo tanto la dirección es:

$$\mathbf{d} = -\mathbf{H}_c^{-1}\nabla f(\mathbf{x}_c) \tag{2.25}$$

2.6.4. Iteración de Gauss-Newton Amortiguada

La dirección de descenso del método de Gauss-Newton se define como:

$$\mathbf{d} = -(\mathbf{R'}(\mathbf{x}_c)^T\mathbf{R'}(\mathbf{x}_c))^{-1}\mathbf{R'}(\mathbf{x}_c)^T\mathbf{r}(\mathbf{x}_c) \tag{2.26}$$

Esta dirección no es definida si la matriz **R'** no presenta independencia lineal en todas sus columnas. Caso contrario, queda una dirección descendiente dado que:

$$\mathbf{d}\nabla f(\mathbf{x}) = -(\mathbf{R'}(\mathbf{x})^T\mathbf{r}(\mathbf{x}))^T(\mathbf{R'}(\mathbf{x})^T\mathbf{R'}(\mathbf{x}))^{-1}\mathbf{R'}(\mathbf{x})^T\mathbf{r}(\mathbf{x}) < 0 \tag{2.27}$$

La combinación de la regla de Armijo junto con la dirección del método de Gauss-Newton se define la Iteración amortiguada de Gauss-Newton.

2.6.5. Intervalo de confianza

El método de intervalo de confianza permiten una transición suave de la dirección de descenso pronunciado y la dirección del método de Newton, permitiendo por un lado la convergencia global del método de descenso pronunciado combinada con la rápida convergencia local del método de Newton.

La idea del método del intervalo de confianza es definir una región geométrica alrededor de $\mathbf{x}_c$ en el cual el modelo cuadrático de la Ecuación 2.23 aproxima a la función f definida en la Ecuación 2.17. El radio del intervalo de confianza al rededor del punto $\mathbf{x}_c$ se denota como $\triangle_c$ (tambien se lo suele denotar como $\triangle$)

$$m_c(\mathbf{x}) = f(\mathbf{x}_c) + \nabla f(\mathbf{x}_c)^T(\mathbf{x} - \mathbf{x}_c) + (\mathbf{x} - \mathbf{x}_c)^T\mathbf{H}_c(\mathbf{x} - \mathbf{x}_c)/2 \tag{2.28}$$

Se define el intervalo de confianza de radio $\triangle_c$ al rededor del punto $\mathbf{x}_c$ como:

$$\mathrm{T}(\triangle) = \{\mathbf{x} | \|\mathbf{x} - \mathbf{x}_c\| \leq \triangle\} \tag{2.29}$$

El intervalo de confianza restringe el tamaño del paso en el proceso de optimización para asegurar que el modelo se mantenga valido. Esto ayuda

a prevenir pasos grandes que puedan provocar la divergencia del algoritmo iterativo. El radio $\triangle_c$ puede ajustarse según que tan buenas son las aproximaciones paso a paso. Si el modelo aproxima bien se incrementa, en caso contrario se decrementa.

Se computa el paso $\mathbf{x}_+$ minimizando $\mathbf{m}_c$ en el intervalo de confianza $\mathrm{T}(\triangle_c)$.

Cálculo del paso para el intervalo de confianza

Tanto el nuevo punto $\mathbf{x}_+$ como el intervalo de confianza se evaluan simultaneamente para medir qué tan bien el modelo cuadrático aproxima la función dentro de la región de confianza. Medimos esto comparando la reducción real en f:

$$ared = f(\mathbf{x}_c) - f(\mathbf{x}_t) \tag{2.30}$$

Y la reducción predecida en el modelo cuadrático:

$$pred = m_c(\mathbf{x}_c) - m_c(\mathbf{x}_t) = -\nabla f(\mathbf{x}_c)^T s_t - {s^T}_t \mathbf{H}_c s_t/2 \tag{2.31}$$

Se introducen tres parametros de control $\mu_0 \leq \mu_{low} < \mu_{high}$ que se utilizan para determinar si el intervalo debe ser abortado ($ared/pred < \mu_0$), incrementado ($ared/pred > \mu_{high}$) o decrementado ($ared/pred < \mu_{low}$). Los valores usuales para estos parametros son de $\mu_{low} = 0{,}25$ y $\mu_{high} = 0{,}75$. Para μ_0 suele usarse tanto 10^{-4} o μ_{low}

El intervalo se incrementa o decrementa multiplicandolo por:

$$0 < \omega_{down} < 1 < \omega_{up} \tag{2.32}$$

Finalmente se limita la cantidad de veces que puede crecer el intervalo de confianza limitando $\triangle_c$

$$\triangle_c \leq C_T \|\nabla f(\mathbf{x}_c)\| \tag{2.33}$$

2.6.6. Algoritmo de Levenberg-Marquardt

Este método presenta una convergencia asintóticamente cuadrática garantizada para funciones convexas. Se puede interpretar como una extención del método clásico de Guass-Newton, pero en vez de realizar la busqueda del mínimo de la función de error no lineal, solamente se estima el optimo global de la aproximación local de Taylor en un estado inicial θ. Este mínimo en $\theta + \delta\theta$ se utiliza como semilla para la próxima iteración. Este paso del método de Guass-Newton se estima como $\delta\theta = -\mathbf{H}^{-1}g$, donde g esta dado por la ecuación 2.27

Algoritmo de Levenberg-Marquardt utilizando la regla de Armijo

Un problema que existe con el método amortiguado de Gauss-Newton es que la matriz $\mathbf{R}'(\mathbf{x}_k)^T\mathbf{r}(\mathbf{x}_k)$ no solo debe tener rango completo de columnas, sino que también debe estar uniformemente acotada y bien condicionada, lo cual no siempre se cumple.

El algoritmo de Levenberg-Marquardt mejora este condicionamiento agregando un parámetro $v_c > 0$ a $\mathbf{R}'(\mathbf{x}_c)^T\mathbf{r}(\mathbf{x}_c)$ para obtener:

$$\mathbf{x}_+ = \mathbf{x}_c - (v_c\mathbf{I} + \mathbf{R'}(\mathbf{x}_c)^T\mathbf{R'}(\mathbf{x}_c))^{-1}\mathbf{R'}(\mathbf{x}_c)^T\mathbf{r}(\mathbf{x}_c) \tag{2.34}$$

Donde $\mathbf{I}$ es la matrix identidad de $N \times N$. La matrix $v_c\mathbf{I}+\mathbf{R}'(\mathbf{x}_c)^T\mathbf{R}'(\mathbf{x}_c)$ es definida positiva. Este parametro v_c es llamado el parámetro de Levenberg-Marquardt

No es necesario computar $\mathbf{R}'(\mathbf{x}_c)^T\mathbf{R}'(\mathbf{x}_c)$ para calcular el paso del algoritmo, se puede resolver el problema de cuadrados mínimos lineal definido como:

$$\min_{\mathbf{x}_c} \frac{1}{2}\left|\left|\begin{bmatrix}\begin{pmatrix}\mathbf{R}'(\mathbf{x}_c)\\ \sqrt{v_c}I\end{pmatrix}\end{bmatrix}s+\begin{bmatrix}\begin{pmatrix}\mathbf{r}(\mathbf{x}_c)\\ 0\end{pmatrix}\end{bmatrix}\right|\right|^2 \tag{2.35}$$

Para funciones con continuidad de Lipschitz, el algoritmo iterativo de Levenbeg-Marquardt-Armijo converge cuadraticamente.

Algoritmo de Levenberg-Marquardt utilizando intervalo de confianza

Puede probarse que, dado $g \in R^N$ y dado A una matriz simetrica de $N \times N$. Siendo:

$$m(\mathbf{s}) = \mathbf{g}^T\mathbf{s} + \mathbf{s}^T\mathbf{A}\mathbf{s}/2 \tag{2.36}$$

Un vector s es solución para:

$$\min_{\|q\|\leq\triangle} m(\mathbf{s}) \tag{2.37}$$

Si y solo si existe un $0 \leq v$ tal que:

$$(\mathbf{A} + v\mathbf{I})\mathbf{s} = -\mathbf{g} \tag{2.38}$$

Y puede que ocurrir que $v = 0$ o $\|s\| = \triangle$

De este resultado se puede definir:

$$\mathbf{s}_t = -(v\mathbf{I} + \mathbf{H}_c)^{-1}\mathbf{g} \tag{2.39}$$

Se desprende que el intervalo de confianza se puede ajustar por medio de v en vez de utilizar $\triangle$, utilizando la relación $ared/pred$. A este parametro se

lo denomina el parametro de Levenberg-Marquardt y resulta en un algoritmo más simple para manejar el tamaño del paso de cada iteración y aún asi seguir manteniendo convergencia global.

El modelo cuadrático de Levenberg-Marquardt para la ecuación 2.17 con el parametro v_c en el punto $\mathbf{x}_c$ esta dado por:

$$\begin{aligned} m_c(\mathbf{x}) &= f(\mathbf{x}_c) + (x - \mathbf{x}_c)^T \mathbf{R'}(\mathbf{x}_c)^T \mathbf{r}(\mathbf{x}_c) \\ &\quad + \frac{1}{2}(x - \mathbf{x}_c)^T (\mathbf{R'}(\mathbf{x}_c)^T \mathbf{R'}(\mathbf{x}_c) + \mathbf{v}_c \mathbf{I})(\mathbf{x} - \mathbf{x}_c) \end{aligned} \tag{2.40}$$

El minimizador del modelo cuadrático es:

$$\mathbf{x}_t = \mathbf{x}_c - (\mathbf{R'}(\mathbf{x}_c)^T \mathbf{R'}(\mathbf{x}_c) + v_c I)^{-1} \mathbf{R'}(\mathbf{x}_c)^T \mathbf{r}(\mathbf{x}_c) \tag{2.41}$$

Y el paso de la iteración $\mathbf{s} = \mathbf{x}_t - \mathbf{x}_c$ y la predicción de la reducción:

$$\begin{aligned} pred &= m(\mathbf{x}_c) - m(\mathbf{x}_t) \\ &= -s^T \mathbf{R'}(\mathbf{x}_c)^T \mathbf{r}(\mathbf{x}_c) - \frac{1}{2} s^T (\mathbf{R'}(\mathbf{x}_c)^T \mathbf{R'}(\mathbf{x}_c) + v_c I) s \\ &= -s^T \mathbf{R'}(\mathbf{x}_c)^T \mathbf{r}(\mathbf{x}_c) + \frac{1}{2} s^T \mathbf{R'}(\mathbf{x}_c)^T \mathbf{r}(\mathbf{x}_c) \\ &= -\frac{1}{2} s^T \nabla f(\mathbf{x}_c) \end{aligned} \tag{2.42}$$

Para decidir aceptar el punto $\mathbf{x}_c$ y realizar ajustes en el parámetro Levenberg-Marquardt se examina la relación:

$$\begin{aligned} \frac{ared}{pred} &= \frac{f(\mathbf{x}_c) - f(\mathbf{x}_t)}{m(\mathbf{x}_c) - m(\mathbf{x}_t)} \\ &= -2 \frac{f(\mathbf{x}_c) - f(\mathbf{x}_t)}{pmbs^T \nabla f(\mathbf{x}_c)} \end{aligned} \tag{2.43}$$

Además de los parametros para el intervalo de confianza $0 < \omega_{down} < 1 < \omega_{up}$ y $\mu_0 \leq \mu_{low} < \mu_{high}$ se agrega el parametro de Levenberg-Marquardt v_0

2.6.7. Complemento de Schur

Una matriz cuadrada $\mathbf{M} \in \mathbb{R}^{q+p,q+p}$ puede descomponerse como se describe en la ecuación 2.44, donde $\mathbf{A} \in \mathbb{R}^{p \times p}$ es inversible y $\mathbf{D} \in \mathbb{R}^{q \times q}$

$$\begin{aligned} \mathbf{M} &= \begin{pmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{pmatrix} \\ &= \begin{pmatrix} \mathbf{I} & 0 \\ \mathbf{C}\mathbf{A}^{-1} & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{A} & 0 \\ 0 & \mathbf{S} \end{pmatrix} \begin{pmatrix} \mathbf{A} & 0 \\ 0 & \mathbf{S} \end{pmatrix} \begin{pmatrix} \mathbf{I} & \mathbf{A}^{-1}\mathbf{B} \\ 0 & \mathbf{I} \end{pmatrix} \end{aligned} \tag{2.44}$$

La matriz $\mathbf{S} = \mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B}$ se conoce como el complemento de Schur de $\mathbf{A}$.

Algorithm 1 Algoritmo de Levenberg-Marquardt.

Require: Matriz de Jacobianos J, parámetro de amortiguación ϱ, parametros f

function LMA
  $k \leftarrow 0$; $v \leftarrow 2$; $x \leftarrow x_0$
  $A \leftarrow J(x)^T J(x)$; $g \leftarrow J(x)^T f(x)$
  $found \leftarrow \|x\|_\infty \leq \epsilon_1; \mu \leftarrow \tau * max(a_{ii})$
  while not found and $k \leq k_max$ **do**
    $k \leftarrow k + 1$;
    $solve(A + \mu I); h_{lm} \leftarrow -g$;
    if $\|h_{lm}\| \leq \epsilon_2(\|x\| + \epsilon_2)$ **then**
      $found \leftarrow true$;
    else
      $x_{new} \leftarrow x + h_{lm}$;
      $\varrho \leftarrow (F(x) - F(x_{new}))(L(0) - L(h_{lm}))$
      if $\varrho \leq 0$ **then**
        $x \leftarrow x_{new}$
        $A \leftarrow J(x)^T J(x)$;
        $g \leftarrow J(x)^T f(x)$;
        $found \leftarrow \|g\|_\infty \leq \epsilon_1$
        $\mu \leftarrow \mu * max(\frac{1}{3}, 1 - (2\varrho - 1)^3$;
        $v \leftarrow 2$
      else
        $\mu \leftarrow \mu * v$
        $v \leftarrow 2 * v$
      end if
    end if
  end while
end function

Dado un sistema de ecuaciones $\mathbf{Mx} = \mathbf{b}$, donde $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2)^T$ y $b = (\mathbf{b}_1, \mathbf{b}_2)^T$ con $\mathbf{x}_1, \mathbf{b}_1 \in \mathbb{R}^p$, $\mathbf{x}_2, \mathbf{b}_2 \in \mathbf{R}^q$, $\mathbf{x}_2$ puede ser calculada desde $\mathbf{Sx}_2 = \mathbf{b}_2 - \mathbf{CA}^{-1}\mathbf{b}_1$. Despejando, $\mathbf{x}_1$ puede ser calculada como $\mathbf{Ax}_1 = \mathbf{b}_1 - \mathbf{Bx}_2$. Este tipo de sistemas son mucho mas fáciles de resolver cuando $p \ll q$

Esta misma optimización se puede aplicar en SLAM substituyendo ($\mathbf{M} = \mathbf{H}, \mathbf{x} = \delta\theta, \mathbf{b} = -\mathbf{g}$) y usando $\mathbf{A}$ como el bloque de datos del sensor y $\mathbf{D}$ el bloque de datos de odometría.

Las matrices en la ecuación del complemento Schur tienen forma diagonal y pueden ser resueltas por bloques. Cada bloque es definido positivo, simétrico y denso en los casos donde la matriz fue aumentada con λ. Para resolver este sistema usualmente se utiliza la descomposición Cholesky $\mathbf{M} = \mathbf{LL}^T$ (L es una matriz triangular inferior). El sistema $\mathbf{M}x = b$ con $\mathbf{M} = \mathbf{LL}^T$ se resuelve trivialmente por sustitución.

La matriz del sistema siempre es simétrica y definida positiva, pero podria no ser densa. Para estos casos se utiliza la variante esparza del algoritmo de descomposición de Cholesky.

2.6.8. Linealización de factores y método Gauss-Newton

La ecuación (2.11) se conoce como un problema de cuadrados mínimos ponderados, ya que es posible expresarla como una suma de errores de predicción ponderados $\mathbf{e}_k(\mathbf{x}_k) = \mathbf{m}_k(\mathbf{x}_k) - \mathbf{z}_k$ como

$$\mathbf{F}(\mathbf{x}) = \sum_{k \in \mathcal{G}} \|\mathbf{z}_k - \mathbf{m}_k(\mathbf{x}_k)\|^2_{\Sigma_k} \tag{2.45}$$

$$= \sum_{k \in \mathcal{G}} \|\mathbf{e}_k(\mathbf{x}_k)\|^2_{\Sigma_k} \tag{2.46}$$

$$= \sum_{k \in \mathcal{G}} \mathbf{e}_k(\mathbf{x}_k)^\top \mathbf{\Omega}_k \mathbf{e}_k(\mathbf{x}_k), \tag{2.47}$$

donde $\mathbf{\Omega}_k$ se conoce como la matriz de información de la medición que corresponde a la inversa de las matrices de covarianza modeladas $\mathbf{\Omega}_k = \Sigma_k^{-1}$ definidas en la Sección 2.4.1 y sirve como una forma de ponderar la influencia de cada error de predicción. Notar que la distancia de Mahalanobis es conmutativa cuando Σ_k es simétrica (que, por definición de la covarianza, es el caso), $\|\mathbf{z}_k - m_k(\mathbf{x}_k)\|^2_{\Sigma_k} = \|m_k(\mathbf{x}_k) - \mathbf{z}_k\|^2_{\Sigma_k} = \|\mathbf{e}_k(\mathbf{x}_k)\|^2_{\Sigma_k}$.

Para que $\mathbf{F}(X)$ sea lineal, cada $\mathbf{e}_k(\mathbf{x}_k)$ debe aproximarse linealmente, dado que está compuesto por la función de modelo no lineal m_k. Dado un valor inicial $\breve{\mathbf{x}}_k$ para las variables involucradas, es posible aproximar $\mathbf{e}_k(\mathbf{x}_k)$ aplicando una expansión de primer orden de Taylor centrada en $\breve{\mathbf{x}}_k$,

$$\mathbf{e}_k(\mathbf{x}_k) = \mathbf{e}_k(\breve{\mathbf{x}}_k + \Delta\mathbf{x}_k) \approx \mathbf{e}_k(\breve{\mathbf{x}}_k) + \mathbf{J}_k \Delta\mathbf{x}_k, \tag{2.48}$$

donde $\Delta\mathbf{x}_k = \mathbf{x}_k - \breve{\mathbf{x}}_k$ es el vector de actualización del estado y el Jacobiano del modelo $\mathbf{J}$ se define como la derivada parcial (multivariada) de $\mathbf{e}_k(\bullet)$ en el punto de linealización dado $\breve{\mathbf{x}}_k$.

Asumiendo que utilizar el Jacobiano resulta en una aproximación suficiente del modelo y que $\Delta\mathbf{x}_k$ es típicamente pequeño, podemos ignorar los términos de órdenes superiores de la expansión de Taylor. Aún más, si se considera que las variables en $\mathbf{x}_k$ viven en un espacio vectorial euclidiano o, de manera equivalente, se pueden representar como un vector, cada Jacobiano se define como

$$\mathbf{J}_k = \frac{\partial \mathbf{e}_k(\mathbf{x}_k)}{\partial \mathbf{x}_k}\Big|_{\mathbf{x}=\breve{\mathbf{x}}} = \frac{\partial m_k(\mathbf{x}_k)}{\partial \mathbf{x}_k}\Big|_{\mathbf{x}=\breve{\mathbf{x}}}. \tag{2.49}$$

Algo a notar es que en varias aplicaciones SLAM, las variables en $\mathbf{x}_k$ en realidad no son euclidianas, por ejemplo, algunas variables en $\mathbf{x}_k$ incluyen componentes angulares (como poses de robot) que pertenecen a un grupo rotacional 2D o 3D no euclidiano SO(2), SO(3).

La función de costo objetivo $\mathbf{F}(\mathbf{x}) = \sum_k \mathbf{F}_k(\mathbf{x})$ en la ecuación 2.47 puede entonces ser aproximada aplicando la expansión de Taylor de primer orden en cada término. La notación se simplifica usando $\mathbf{x}$ como un vector columna de todas las variables del estado y $\mathbf{e}_k = \mathbf{e}_k(\breve{\mathbf{x}})$ como el error predicho k valuado en los valores de variables iniciales $\breve{\mathbf{x}}$,

$$\begin{aligned}
\mathbf{F}_k(\breve{\mathbf{x}} + \Delta\mathbf{x}) &= \mathbf{e}_k(\breve{\mathbf{x}} + \Delta\mathbf{x})^\top \mathbf{\Omega}_k \mathbf{e}_k(\breve{\mathbf{x}} + \Delta\mathbf{x}) && (2.50)\\
&\approx (\mathbf{e}_k + \mathbf{J}_k\Delta\mathbf{x})^\top \mathbf{\Omega}_k (\mathbf{e}_k + \mathbf{J}_k\Delta\mathbf{x}) && (2.51)\\
&= \underbrace{\mathbf{e}_k^\top \mathbf{\Omega}_k \mathbf{e}_k}_{\mathbf{c}_k} + 2\underbrace{\mathbf{e}_k^\top \mathbf{\Omega}_k \mathbf{J}_k}_{\mathbf{b}_k^\top}\Delta\mathbf{x} + \Delta\mathbf{x}^\top \underbrace{\mathbf{J}_k^\top \mathbf{\Omega}_k \mathbf{J}_k}_{\mathbf{H}_k}\Delta\mathbf{x} && (2.52)\\
&= \mathbf{c}_k + 2\mathbf{b}_k^\top\Delta\mathbf{x} + \Delta\mathbf{x}^\top\mathbf{H}_k\Delta\mathbf{x}. && (2.53)
\end{aligned}$$

De esta manera, $\mathbf{F}(\mathbf{x})$ puede escribirse como

$$\begin{aligned}
\mathbf{F}(\breve{\mathbf{x}} + \Delta\mathbf{x}) &= \sum_{k\in\mathcal{G}} \mathbf{F}_k(\breve{\mathbf{x}} + \Delta\mathbf{x}) && (2.54)\\
&\approx \sum_{k\in\mathcal{G}} (\mathbf{c}_k + 2\mathbf{b}_k^\top\Delta\mathbf{x} + \Delta\mathbf{x}^\top\mathbf{H}_k\Delta\mathbf{x}) && (2.55)\\
&= \underbrace{\sum_{k\in\mathcal{G}} \mathbf{c}_k}_{\mathbf{c}} + 2\underbrace{\sum_{k\in\mathcal{G}} \mathbf{b}_k^\top}_{\mathbf{b}^\top}\Delta\mathbf{x} + \Delta\mathbf{x}^\top \underbrace{\sum_{k\in\mathcal{G}} \mathbf{H}_k}_{\mathbf{H}}\Delta\mathbf{x} && (2.56)\\
&= \mathbf{c} + 2\mathbf{b}^\top\Delta\mathbf{x} + \Delta\mathbf{x}^\top\mathbf{H}\Delta\mathbf{x} && (2.57)\\
&=: \mathbf{G}(\Delta\mathbf{x}). && (2.58)
\end{aligned}$$

La expresión resultante $\mathbf{G}(\Delta\mathbf{x})$ es una función cuadrática de $\Delta\mathbf{x}$ que aproxima linealmente a $\mathbf{F}(\breve{\mathbf{x}}+\Delta\mathbf{x})$, lo que nos permite proponer la siguiente minimización,

$$\Delta\mathbf{x}^* = \arg\min \Delta\mathbf{x}\mathbf{G}(\Delta\mathbf{x}) \tag{2.59}$$

El mínimo $\Delta\mathbf{x}^*$ puede ser determinado inspeccionando las primeras y segundas derivadas de $\mathbf{G}(\Delta\mathbf{x})$:

$$\mathbf{G}'(\Delta\mathbf{x}) = 2\mathbf{H}\Delta\mathbf{x} + 2\mathbf{b} \tag{2.60}$$
$$\mathbf{G}''(\Delta\mathbf{x}) = 2\mathbf{H}. \tag{2.61}$$

Dado que $\mathbf{G}(\bullet)$ es un paraboloide, es posible encontrar su mínimo $\Delta\mathbf{x}^*$ de forma cerrada simplemente estableciendo la primera derivada a cero, $\mathbf{G}'(\Delta\mathbf{x}^*) = 0$ lo cual conduce al sistema lineal

$$\mathbf{H}\Delta\mathbf{x}^* = -\mathbf{b}. \tag{2.62}$$

La matriz $\mathbf{H} = \mathbf{J}^\top\mathbf{\Omega}\mathbf{J}$ se conoce como la matriz Hessiana o Hessiano del problema, lo cual es abuso de terminología ampliamente aceptado, dado que es en realidad una aproximación de la matriz Hessiana analítica de derivadas segundas de $\mathbf{F}(\mathbf{x})$. $\mathbf{H}$ es simétrica semidefinida positiva por construcción (ver Sec. 2.6.9), y como la segunda derivada $\mathbf{G}''(\Delta\mathbf{x})$ no depende de $\Delta\mathbf{x}$, está garantizado que la solución del sistema lineal es un mínimo.

Se puede obtener una nueva estimación del estado $\mathbf{x}^*$ agregando el incremento $\Delta\mathbf{x}^*$ al valor inicial,

$$\mathbf{x}^* = \breve{\mathbf{x}} + \Delta\mathbf{x}^*. \tag{2.63}$$

Debido a las aproximaciones aplicadas a través del procedimiento de linealización, la estimación resultante $\mathbf{x}^*$ generalmente no alcanza el mínimo local en un único paso de actualización. El procedimiento se aplica de manera iterativa varias veces hasta que el incremento $\Delta\mathbf{x}^*$ sea lo suficientemente pequeño y/o se cumpla cierto criterio. En cada iteración, la solución anterior de $\mathbf{x}^*$ se utiliza como el nuevo punto de linealización.

El propósito de realizar la optimización no lineal es obtener el estimador de probabilidad máxima a posteriori $\hat{\mathbf{x}}^{MAP}$ de la media para la distribución $P(X|Z)$ introducida en la Sección 2.4.1, y dado que $\mathbf{x}^*$ se resuelve aproximando el problema a través de linealizaciones, resulta intuitivo que $\hat{\mathbf{x}}^{MAP} \approx \mathbf{x}^*$, y $\mathbf{x}^*$ es, entonces, nuestro estimador de la media para la distribución.

El método descrito en esta sección corresponde al método de Gauss-Newton y, aunque en general no hay garantías de convergencia a un mínimo

local, es el componente básico inspirador de muchos otros algoritmos para abordar cuadrados mínimos no lineales. Por ejemplo, el algoritmo de Levenberg-Marquardt [34,35] garantiza convergencia introduciendo un paso de actualización diferente (ver Ecuación 2.63) con un factor de amortiguación que controla en qué región uno está dispuesto a confiar en la aproximación cuadrática. Adicionalmente, estos métodos de optimización se pueden combinar con varias estrategias numéricas para resolver la Ecuación (2.62), las más populares y exitosas se basan en la factorización de Cholesky o la descomposición **QR** [31].

2.6.9. Estructura dispersa del sistema lineal

Como se describió anteriormente, el problema SLAM que nos propusimos resolver se reduce a la minimización de una función objetivo no lineal (Ec. 2.47) la cual es una contribución aditiva de muchas funciones de costo al cuadrado,

$$\mathbf{F}(\mathbf{x}) = \sum_{k \in \mathcal{G}} \mathbf{e}_k(\mathbf{x}_k)^\top \mathbf{\Omega}_k \mathbf{e}_k(\mathbf{x}_k). \tag{2.64}$$

Esta formulación de cuadrados mínimos puede ser reformulada mediante la recopilación de todos los errores y las matrices de información, en un vector de error único y una matriz diagonal de información por bloques única,

$$\mathbf{F}(\mathbf{x}) = \mathbf{e}(\mathbf{x})^\top \mathbf{\Omega} \mathbf{e}(\mathbf{x}) \tag{2.65}$$

con,

$$\mathbf{e}(\mathbf{x}) = \begin{bmatrix} \mathbf{e}_1(\mathbf{x}_1) \\ \vdots \\ \mathbf{e}_K(\mathbf{x}_K) \end{bmatrix} \quad , \quad \mathbf{\Omega} = \begin{bmatrix} \mathbf{\Omega}_1 & & \\ & \ddots & \\ & & \mathbf{\Omega}_K \end{bmatrix}, \tag{2.66}$$

donde cada par de $\mathbf{e}_k \in \mathbb{R}^{dim(\mathbf{z}_k)\times 1}$ y $\mathbf{\Omega}_k \in \mathbb{R}^{dim(\mathbf{z}_k)\times dim(\mathbf{z}_k)}$ proviene del factor k del grafo de factores (Sec. 2.6.8), el cual modela una observación, una medición de movimiento o una restricción prior.

El proceso de linealización aplicando la expansión de Taylor de primer orden (Sec. 2.6.8) introduce los Jacobianos de la función del error $\mathbf{J}_k$ que son dispersos por construcción, principalmente porque el grafo es disperso. Cada Jacobiano $\mathbf{J}_k$ presenta bloques distintos de cero solo en las variables afectadas por su factor correspondiente. Como ejemplo, el Jacobiano correspondiente a un factor entre las variables $\mathbf{x}_i$ y $\mathbf{x}_j$ es:

$$\mathbf{J}_k = \Big[\mathbf{0} \cdots \mathbf{0} \underbrace{\frac{\partial \mathbf{e}_k(\mathbf{x}_i, \mathbf{x}_j)}{\partial \mathbf{x}_i}}_{\mathbf{J}_{ki}} \mathbf{0} \cdots \mathbf{0} \underbrace{\frac{\partial \mathbf{e}_k(\mathbf{x}_i, \mathbf{x}_j)}{\partial \mathbf{x}_j}}_{\mathbf{J}_{kj}} \mathbf{0} \cdots \mathbf{0}\Big]. \tag{2.67}$$

Notar que las derivadas del error $\mathbf{e}_k$ dependen de las derivadas del modelo m_k (Ec. 2.49), dando como resultado $\mathbf{J}_k \in \mathbb{R}^{dim(\mathbf{z}_k)\times dim(\mathbf{x})}$ siendo $\mathbf{x}$ el vector del estado completo de las variables. Estas matrices Jacobianas del factor se pueden apilar componiendo el sistema Jacobiano

$$\mathbf{J} = \begin{bmatrix} \mathbf{J}_1 \\ \vdots \\ \mathbf{J}_8 \end{bmatrix}. \tag{2.68}$$

Tomando como ejemplo el grafo presentando en la Figura 2.1, y ordenando las variables como $\mathbf{x} = [\mathbf{x}_{p_1}, \mathbf{x}_{p_2}, \mathbf{x}_{l_1}, \mathbf{x}_{l_2}, \mathbf{x}_{l_3}, \mathbf{x}_{l_4}]$, se obtiene el siguiente Jacobiano,

$$\mathbf{J} = \begin{bmatrix} \mathbf{J}_1 \\ \vdots \\ \mathbf{J}_8 \end{bmatrix} = \begin{array}{c} \begin{array}{cccccc} \mathbf{x}_{p_1} & \mathbf{x}_{p_2} & \mathbf{x}_{l_1} & \mathbf{x}_{l_2} & \mathbf{x}_{l_3} & \mathbf{x}_{l_4} \end{array} \\ \begin{bmatrix} \mathbf{J}_{11} & & & & & \\ \mathbf{J}_{21} & \mathbf{J}_{22} & & & & \\ \mathbf{J}_{31} & & \mathbf{J}_{33} & & & \\ \mathbf{J}_{41} & & & \mathbf{J}_{44} & & \\ & \mathbf{J}_{52} & \mathbf{J}_{53} & & & \\ & \mathbf{J}_{62} & & \mathbf{J}_{64} & & \\ & \mathbf{J}_{72} & & & \mathbf{J}_{75} & \\ & \mathbf{J}_{82} & & & & \mathbf{J}_{86} \end{bmatrix} \end{array} \begin{array}{c} \\ \mathrm{f}_1 \\ \mathrm{f}_2 \\ \mathrm{f}_3 \\ \mathrm{f}_4 \\ \mathrm{f}_5 \\ \mathrm{f}_6 \\ \mathrm{f}_7 \\ \mathrm{f}_8 \end{array}, \tag{2.69}$$

donde se han omitido los ceros para una mejor legibilidad y los índices de la matriz están etiquetados. El sistema lineal finalmente obtenido en la ecuación 2.62 requiere la matriz Hessiana y un vector del lado derecho,

$$\mathbf{H} = \mathbf{J}^\top \mathbf{\Omega} \mathbf{J} \quad , \quad \mathbf{b}^\top = \mathbf{e}(\mathbf{x})^\top \mathbf{\Omega} \mathbf{J}. \tag{2.70}$$

La estructura de bloques de la matriz Hessiana $\mathbf{H}$ corresponde a la matriz de adyacencia del grafo de factores, con una cantidad de bloques distintos de cero relacionado al numero de aristas del grafo, el cual en aplicaciones SLAM resulta en una matriz dispersa. Más aún, $\mathbf{H}$ resulta simétrica semidefinida positiva por construcción.

2.7. Particionamiento espacial del entorno

Una cuestión clave a la hora de diseñar un sistema de SLAM es definir las estrcuturas de datos que vamos a utilizar para el particionamiento espacial. Estas estructuras deben organizar y gestionar datos espaciales, permitiendo operaciones rápidas de consulta, inserción y eliminación. Como mencionamos en la Sección 1.1.3, entre los distintos tipos que podemos encontrar en la literatura, se destacan los árboles y las tablas hash, que son particularmente populares en campos como la Computación Gráfica, la Visión por Computadora y tambié en SLAM.

2.7.1. Árboles

Las estructuras basadas en árboles, como los Octomaps [11], los árboles BVH [40] y los KD-trees [41], son ampliamente utilizadas por su capacidad para particionar el espacio de manera jerárquica, lo que las hace ideales para tareas como la detección de colisiones, la búsqueda del vecino más cercano y el mapeo del entorno. Estos arboles, son estructuras de datos construidas mediante la partición recursiva del espacio. El KD-tree es particularmente útil para aplicaciones que involucran claves de búsqueda multidimensionales, como búsquedas de rango y búsquedas del vecino más cercano. Cada nodo en un KD-tree representa un punto k-dimensional. Los nodos internos (no hojas) actúan como hiperplanos que dividen el espacio en dos semiespacios.

Los árboles de jerarquía de volúmenes delimitadores (BVH) es otra representación jerárquica que organiza primitivas en 2D o 3D. Cada nodo en el BVH representa un volumen delimitador que encierra un subconjunto de los objetos. Es fundamental en el área de la Computación Gráfica, especialmente para tareas de detección de colisiones y trazado de rayos. Sin embargo, la mayoría de las versiones aceleradas por GPU de los KD-trees [42, 43] y los BVH [40, 44, 45] no permiten un tamaño dinámico para los datos, ya que la asignación de memoria no escala en la GPU, por lo que agregar o eliminar elementos requiere de reinicializar por completo los nodos del arbol.

Otra estructura de datos, el Octree [46], basa su particionamiento en la división recursiva del espacio 3D en ocho subvolúmenes equivalentes, condicionada a la ocupación de un objeto. Esta estrcutura ha sido utilizada en mapeo 3D adaptativo [11, 47] y para la navegación autónoma de robots móviles [15]. Tal y como se menciona en Voxgraph [22] la utilización de este tipo de estructuras es compleja para ambientes dinamicos y esparsos. Los Octrees pueden terminar degradando el rendimiento en escenarios donde se presentan una gran cantidad de datos, con consultas frecuentes y que necesiten ejecutar en tiempo real. Esto se debe principalmente a la necesidad de atravesar todo el árbol cada vez que se realiza una consulta. Por otra parte, las implementaciones en GPU de Octrees, comparte las mismas dificultades que los otros tipos de árboles mencionados anteriormente.

2.7.2. Tablas de hash

Por otro lado, las tablas de hash son estructuras que utilizan un conjunto de pares *(clave, valor)*, proveen una forma eficiente de indexar y obtener el valor de data espacial a través de distintas técnicas de hashing debido a su compacto modelo de datos y su complejidad temporal cuasi constante para insertar y buscar. Se utiliza una función de hash $h : \mathcal{K} \to \mathcal{I}_n$ donde $\mathcal{I}_n = \{0, 1, \ldots, n-1\} \in \mathbb{N}$ para minimizar el número de colisiones y mejorar la distribución de los valores en memoria de acuerdo a un criterio arbitrario.

A pesar de la función de hash, aún se necesita un esquema de resolución

de colisiones. La mayoría de las implementaciones en GPU utilizan direccionamiento abierto [14, 48, 49] debido a las limitaciones de la GPU para trabajar con estructuras de tamaño dinámico y la posibilidad de implementar operaciones concurrentes. Mientras que las implementaciones de tablas de hash están ampliamente disponibles para CPU, sus contrapartes en GPU solo han surgido en la última década [14, 15].

El hashing espacial es otra variación de la gestión espacial con un tiempo de acceso $O(1)$ dependiendo de las tablas de hash. Combinado con pequeños arreglos densos en 3D, ha sido ampliamente utilizado en la reconstrucción volumétrica de escenas en tiempo real dentro de una región de interés arbitrariamente extensas.

Algunas implementaciones como stdgpu [50] no están optimizadas para grandes volúmenes de datos. Recientemente, WarpCore [14] propone soportar valores no enteros e inserciones dinámicas, aunque el dominio de claves sigue limitado, como máximo, a enteros de 64 bits. ASH [15] proporciona una tabla hash dinámica, genérica y libre de colisiones. Una implementación independiente de la arquitectura del hardware donde se ejecuta, con un diseño de datos SoA mejorado y con varios caso de uso donde se logran rendimientos superiores, incluyendo reconstrucciones basadas en mapas TSDF.

Varias aplicaciones de mapeo 3D requieren asociar coordenadas a diversas propiedades. Por ejemplo, una coordenada 3D puede estar asociada a una normal, un color o una etiqueta en una nube de puntos. Si bien los valores asociados pueden empaquetarse en una estructura de datos dentro de un arreglo (es decir, con disposición en memoria AoS), la complejidad del código podría aumentar. Por lo tanto, se deberían implementar funciones a nivel de estructura. ASH admite tablas de hash de múltiples valores que se almacenan organizados en disposición de memoria SoA. Generalizando la funcionalidad de la tabla de hash al extender el búffer de un solo valor a un arreglo de búfferes de valores [15]. Este simple cambio permite almacenar tipos de valores complejos en SoA, facilitando consultas e indexaciones vectorizadas.

2.8. Métricas de error en SLAM

La evaluación de rendimiento de sistemas SLAM es, en sí mismo, un tema de investigación muy activo debido a que existe un ecosistema muy diverso de métodos y soluciones. Existen, también, muchas configuraciones de robots posibles con diferentes capacidades de sensado que generan diferentes tipos de representaciones del entorno, es decir, mapas. Por esta razón, un enfoque es medir la precisión de un sistema SLAM en términos de la comparación entre poses de robot estimadas con respecto a un conjunto de poses reales medidas externamente y consideradas como valores de referencia o verdad (*ground-truth*) [37,51,52] Aunque puede ser falaz en situaciones en las que el error de estimación cometido al comienzo de la trayectoria se traduce en un

grandes errores al final de la misma. Un ejemplo de esto podría ser un error considerable en la estimación de la orientación de la pose, el cual podría corromper estimaciones posteriores "curvando" la trayectoria estimada. Dependiendo de la aplicación, este tipo de error propagado puede no ser un problema siempre que las estimaciones sean consistentes localmente, lo que puede ser cierto incluso en presencia de grandes errores métricos absolutos en la pose del robot. Para abordar este problema, se suelen emplear dos tipos diferentes de métricas, una basada en el error absoluto de la pose para analizar la precisión global de un sistema SLAM y otra basada en errores relativos de la pose para analizar la precisión local de un sistema SLAM.

El error absoluto de la pose (también conocido como APE, por las siglas en inglés de *Absolute Pose Error*) para una secuencia estimada de poses $\mathbf{x}_1 \dots \mathbf{x}_n$ y su correspondiente conjunto de poses de los valores de referencia (*groundtruth*) $\mathbf{x}_1^* \dots \mathbf{x}_n^*$, es definido como

$$E_k = (\mathbf{x}_k \ominus \mathbf{x}_k^*) \quad , \quad APE_k = \|\delta(E_k)\|_2 \,, \tag{2.71}$$

donde $\ominus$ es el inverso del operador de composición de movimiento $\oplus$ explicado en el Apéndice 7.1, y $\delta(\bullet)$ denota una función proyectiva que puede tomar tanto la parte traslacional como la parte rotacional del error E_k, el cual es representado como una matriz de transformación. Más aún, se define una métrica unificadora para toda la trayectoria como el error cuadrático medio absoluto (también conocido como absolute RMSE por las siglas en inglés de *Root Mean Square Error*):

$$RMSE_{abs} = \sqrt{\frac{1}{n}\sum_{k=1}^{n} APE_k^2}. \tag{2.72}$$

Es importante tener en cuenta que las poses groundtruth deben encontrarse sincronizadas con las estimaciones del SLAM, lo que generalmente no es el caso, ya que los datos de referencia (*groundtruth*) provienen de sistemas externos que capturan el estado de la pose del robot en momentos diferentes, a los que lo hace el algoritmo SLAM. Sin embargo, las poses de referencia pueden ser interpoladas para que coincidan con las marcas de tiempo requeridas.

Las métricas de error absoluto son subóptimas [53] para la comparación de sistemas de SLAM y es posible introducir una métrica alternativa, comparando incrementos de pose relativos entre la trayectoria estimada y la de referencia. Sea $E_{i,j} = \boldsymbol{x}_j \ominus \boldsymbol{x}_i$ la diferencia (o movimiento) entre dos poses de la misma secuencia, y $E_{i,j}^* = \boldsymbol{x}_j^* \ominus \boldsymbol{x}_i^*$ el incremento correspondiente al mismo movimiento en la trayectoria de referencia. El error relativo de pose (también conocido como RPE, por las siglas en inglés de *Relative Pose Error*) entre frames i y j se puede definir como:

$$RPE_{i,j} = \left\|\delta(E_{i,j} \ominus E_{i,j}^*)\right\|_2 \,, \tag{2.73}$$

donde $\delta(\bullet)$ denota una función proyectiva que puede tomar tanto la parte traslacional como la parte rotacional del error entre los incrementos relativos $E_{i,j}$ y $E^*_{i,j}$, los cuales son representados como matrices de transformación. De manera similar al caso absoluto, se puede definir una métrica unificadora para toda la trayectoria bajo esta formulación de error relativo, esta es conocida como error cuadrático medio relativo (RMSE de las siglas en inglés de *Relative Root Mean Square Error*), siguiendo la ecuación:

$$RMSE_{rel} = \sqrt{\frac{1}{n}\sum_{\forall i,j} RPE^2_{i,j}}. \tag{2.74}$$

Las métricas de error presentadas en esta sección son las que utilizamos para realizar los experimentos de esta tesis (ver Cap. 5).

3. ESTADO DEL ARTE

3.1. Trabajos Previos

Los sistemas de SLAM densos han sido foco de desarrollo en la última decada, empujados por la facilitación en el acceso a hardware con capacidad de cómputo para generar mapas densos en tiempo real. Distintos tipos de mapas se desarrollaron, como las representaciones de superficies (b-reps) que definen objetos 3D en términos de los puntos del sensor que intersectan con la superficie de un objeto observado. Algunos trabajos como los de Castle y Kaess [54, 55] propone modelos basados en planos, por su eficiencia y facilidad para representar y almacenar información de alto nivel extraida de la nube de puntos del sensor. También existen representaciones más generales basadas en curvas de NURBS o B-splines, modelos de malla superficial (conjuntos conectados de polígonos) y representaciones de superficies implícitas. Estas últimas especifican la superficie de un sólido como los valores donde una función definida en $\mathbb{R}^3$ vale cero; ejemplos de estas representaciones incluyen funciones de base radial [56], funciones de distancia con signo (SDF) [16] y funciones de distancia con signo truncada (TSDF) [57]. Por su parte, las representaciones que utilizan particionamiento espacial definen objetos 3D como una colección de primitivas contiguas no intersectantes. La representación de particionamiento espacial más popular es la llamada enumeración de ocupación espacial, que descompone el espacio 3D en cubos idénticos llamados vóxeles dispuestos en una cuadrícula 3D regular. Esquemas de particionamiento más eficientes incluyen octree, octree de mapa poligonal y árbol de particionamiento binario [58]. Las representaciones de octree se han utilizado exitosamente para la creación de mapas 3D [59].

Algunos trabajos recientes de SLAM denso como LSD-SLAM [60] y DVO-SLAM [6] construyen un mapa que consiste en mapas de profundidad de fotogramas clave. Kintinuous [61] y ElasticFusion [7] generan reconstrucciones en forma de una malla deformable y una colección de surfels, respectivamente. Más recientemente, Kimera [62] crea una malla semántica del entorno. Estos enfoques generan reconstrucciones visualmente atractivas, pero son difíciles de usar para la planificación de caminos, ya que carecen de información sobre el espacio libre observado y se centran solo en las superficies.

Siguiendo este enfoque, se ha presentado el sistema de código abierto llamado Voxgraph[1], que presenta un método para construir mapas volumétricos globalmente consistentes en una CPU [22]. El sistema estima las poses

[1] Disponible on-line en: https://github.com/ethz-asl/voxgraph

de los submapas en función del propio mapa denso, utilizando la representación SDF subyacente para realizar la alineación geométrica. Se utilizan restricciones geométricas que son lo suficientemente eficientes para realizar la optimización global del mapa con alta frecuencia. Además, este trabajo formula el problema como una optimización no lineal basada en un grafo de factores e incluye restricciones de odometría, cierre de ciclos y registración. El presente trabajo de tesis se basó fuertemente en el estudio de este sistema para desarrollar una nueva versión basada en GPU. En la sección siguiente presentamos un estudio detallado de este sistema que incluye un análisis de los perfiles de los tiempos de ejecución. Posteriormente presentaremos el sistema nvBlox [63], que puede considerarse una evolución del sistema Voxgraph para el mapeo volumétrico acelerado por GPU.

Particionamiento espacial del entorno

Es necesario que la representación del mapa sea eficiente para permitir altas resoluciones incluso en hardware embebido, y también debe ser flexible para facilitar la incorporación de diversas técnicas de ponderación y fusión para sus partes. En el mapeo volumétrico, la representación de los puntos del entorno en una estructura de datos resulta un desafío. La distribución de las superficies es incierta y dispersa, y puede seguir varios patrones diferentes. Por lo tanto, es necesaria una representación adecuada. Los trabajos actuales ofrecen opciones discretas [64] o neuronales [65–67] para categorizar los objetos 3D en la escena.

Los mapas de ocupación utilizan una cuadrícula de celdas (vóxeles en 3D) donde cada celda contiene un valor de probabilidad que indica si está ocupada, libre o desconocida [68]. Los mapas TSDF almacenan la distancia al punto de superficie más cercano dentro de una cierta distancia de truncamiento. La principal ventaja de TSDF sobre los mapas de ocupación es que suaviza el ruido después de observaciones sucesivas [16].

Los trabajos más recientes reemplazan la representación densa de la función de distancia con signo volumétrica (SDF) utilizada en sistemas tradicionales de reconstrucción de superficies por un conjunto de SDF continuas aprendidas localmente definidas por una red neuronal [69]. Algunos utilizan representaciones de alto nivel para los objetos previamente identificados en la escena para reducir la disimilitud visual y geométrica entre las formas generadas y una colección de figuras [70]. DeepSDF representa un SDF a nivel de objeto con una red neuronal y un único código latente. DeepLS almacena una cuadrícula de códigos latentes independientes, cada uno responsable de almacenar información sobre superficies en un pequeño vecindario local.

Como ya se mencionó previamente, para los sistemas de SLAM esparzos suelen utilizarse dos estructuras de datos principalmente: los árboles octales y las tablas de hash. Los árboles octales suelen ser mejores para representar niveles virtualmente infinitos de precisión, posibilitando la flexibilidad

de integrar mapas de cualquier tamaño. Por otro lado, las tablas de hash brindan acceso más eficiente tanto para lectura y escritura de los datos [1].

Utilizar una tabla hash para discretizar todo el entorno puede no escalar adecuadamente tanto en términos del tamaño del mapa como de su resolución, y también podría decrementar la tasa de aciertos en caché, ya que los vóxeles vecinos quedarán dispersos en la tabla hash. El estado actual del arte utiliza dos niveles de jerarquía [71, 72] para la discretización volumétrica de mapas TSDF. Siendo el primer nivel un conjunto de bloques y el segundo nivel una grilla tridimensional, usualmente cúbica, de vóxeles. Trabajos recientes [22, 73] extienden esta idea al representar el mapa global como una combinación de submapas superpuestos junto con su posición y orientación para lograr consistencia global. Luego, cada submapa utiliza una colección dispersa de bloques, con el objetivo de representar solo las partes del entorno donde se han recopilado datos del sensor.

3.2. Voxgraph

El sistema Voxgraph prioriza la modularización y extensibilidad por sobre el rendimiento, con una arquitectura que fue diseñada para ser ejecutada en una CPU. El sistema es principalmente de un solo hilo y la concurrencia se utiliza solo para acelerar el procesamiento por lotes de datos dentro de los módulos de integración de nube de puntos del sensor, tal como se describe en [71]. Este sistema Voxblox se usa para construir incrementalmente las ESDFs directamente a partir de Funciones de Distancia con Signo Truncada (TSDFs) y aprovechar la información de distancia ya contenida dentro del radio de truncamiento. Los mapas TSDFs resultan más eficientes que los octomaps para ser construídos y suavizar el ruido del sensor a medida que se repiten las observaciones sobre una misma área del entorno explorado.

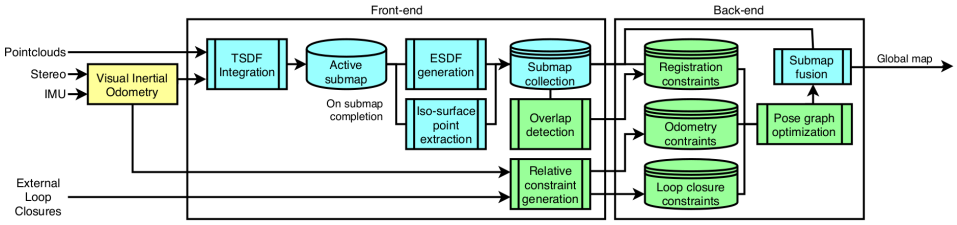

Fig. 3.1: Descripción general del sistema Voxgraph, extraído de [22].

3.2.1. Definición del problema

Dada una secuencia de nubes de puntos producidas por un sensor (en el marco de referencia C) que se mueve por el mundo (marco W, alineado por gravedad), el objetivo es generar una reconstrucción consistente basada en SDF. En el instante de tiempo i tenemos una estimación de la posición

del marco del sensor a partir de la odometría, con respecto a un marco local O, parametrizado como una transformación rígida $T_{OC^i} \in \mathrm{SE}(3)$. Si se pudiera calcular T_{OC^i} con precisión en el momento de la captura de la nube de puntos, existen técnicas conocidas [12, 18] que podrían usarse para construir un mapa TSDF en O. Sin embargo, la estimación por odometría de T_{OC^i} acumula error a lo largo del tiempo. Por lo tanto, una alternativa es contruir una serie de submapas $\{\mathcal{S}_i\}_{i=1}^N$ con sus marcos $\{S_i\}_{i=1}^N$ a partir de los datos del sensor. Los submapas se definen por su pose en el mundo $T_{WS^i} \in \mathbb{R}(3)\times\mathrm{SO}(2)$ y por una SDF. La SDF de un submapa $\mathcal{S}_i$ se compone de las funciones de distancia $\Phi_{\mathcal{S}_i} : \mathbb{R}^3 \to \mathbb{R}$ y peso $\omega_{\mathcal{S}_i} : \mathbb{R}^3 \to \mathbb{R}$, que respectivamente mapean puntos en $\mathbb{R}^3$ a d, una distancia con signo a la superficie observable más cercana, y w, una medida de confianza. Bajo esta configuración, el problema de reconstrucción se reduce a determinar las poses óptimas de cada submapa $\{T_{WS^i}\}_{i=1}^N$.

Como la mayoría de los sistemas de SLAM, Voxgraph se divide en un Front-end y un Back-end. El Front-end convierte las mediciones del sensor en submapas y genera las restricciones que se pasarán al Back-end. El Back-end mantiene este conjunto de restricciones y estima la alineación de la colección de submapas más probable mediante un proceso de optimización no-lineal. En la Figura 3.1 pueden observarse los módulos principales del sistema.

3.2.2. Front-end

El Front-end de Voxgraph puede dividirse en dos módulos: uno responsable de convertir las mediciones de sensores en submapas y otro para crear las restricciones que se pasarán al Back-end para corregir los errores durante la creación de los submapas. Por su parte, el primer módulo también se divide en dos etapas. Primero, los datos crudos del sensor se integran en un volumen TSDF para incluir o actualizar los vóxeles que construyen el mapa TSDF. En una segunda etapa, se propagan los vóxeles actualizados desde el TSDF a la representación de volúmenes ESDF. Los vóxeles creados durante ambos pasos se agrupan en bloques de tamaño fijo que a su vez se indexan utilizando una tabla hash de bloques que representa internamente el mapa esparso global del entorno explorado. Este mapa global podría verse como una tabla hash de bloques dispersa en memoria (ver Fig. 3.2). El tamaño de los bloques puede ser modificado por un parámetro de configuración del sistema, pero permanece igual durante toda la ejecución. Cuando una nueva nube de puntos llega desde el sensor, se lanzan múltiples hilos de ejecución que procesan un punto dentro de la nube y de manera serializada incorpora todos los vóxeles que intersectan con el rayo que se proyecta desde ese punto. Una vez que termina de incorporar todos los vóxeles, continúa con otro punto dentro de la nube de puntos. Este procesamiento se repite hasta alcanzar todos los puntos de la nube de puntos. Un aspecto negativo de este enfoque es que requiere de mecanismos de sincronización para no caer en

condiciones de carrera al escribir sobre la tabla de hash. En la Fig. 3.2 se marcan con una señal de detenerse los tres momentos en que el sistema debe sincronizar. Este tipo de flujo de ejecución no es posible de llevar a cabo en GPU ya que esta fuertemente desaconsejado realizar bloqueos de escritura o lectura sobre colecciones de datos dado que impacta drásticamente en el rendimiento.

Creación de submapas

Las secuencias contiguas de nubes de puntos de entrada se combinan en submapas mediante la proyección de rayos en una grilla de vóxeles espacialmente codificada mediante una tabla de *hash* (ver [71, 73] para más detalles). El origen de la grilla de vóxeles se ubica y alinea con su marco correspondiente S, que está parametrizado por su pose con respecto al marco del mundo T_{WS}. Los mapas se crean a una frecuencia fija (N pasos de tiempo) siguiendo la premisa que los errores de odometría se acumulan de manera lenta y suave, y, por lo tanto, los submapas que se crean en intervalos de tiempo lo suficientemente cortos siguen siendo consistentes internamente. Durante la construcción, también se almacena la trayetoria del sensor a través de cada submapa como una colección de poses relativas al mismo,

$$\mathcal{T}_{\mathcal{S}i} = \{T_{S^iC^j}, \ldots, T_{S^iC^{j+N}}\}. \tag{3.1}$$

Luego de generar el submapa, calculamos la información utilizada por el back-end para la registración del submapa. La ESDF (*Euclidian* SDF) $\Phi_{\mathcal{S}}$ se calcula a partir de su TSDF siguiendo [71]. Este proceso propaga las distancias euclidianas fuera del rango de truncamiento utilizado por el TSDF, lo que permite buscar distancias más lejanas desde las superficies.

Generación de restricciones

El Front-end le pasa al Back-end las restricciones al problema de optimización subyacente. En Voxgraph se utiliza tres tipos de restricciones: odometría, cierre de ciclos y registración. A continuación se detalla cada una de estas restricciones.

a) Restricciones de odometría

Para penalizar la desviación de la pose de cada nuevo submapa con la estimación de su pose relativa calculada por odometria, se estima la transformación como la concatenación acumulada de las poses de los submapas anteriores:

Cada vez que se crea un nuevo mapa $\mathcal{S}_{i+1}$, y a partir de $\mathcal{S}_i$ creado a partir de las mediciones del sensor $\{C_l\}_{l=k}^{k+N}$, la transformada acumulada queda definida como:

$$\hat{T}_{S^iS^{i+1}} = T_{C^kC^{k+1}} T_{C^{k+1}C^{k+2}} \ldots T_{C^{k+N-1}C^{k+N}}. \tag{3.2}$$

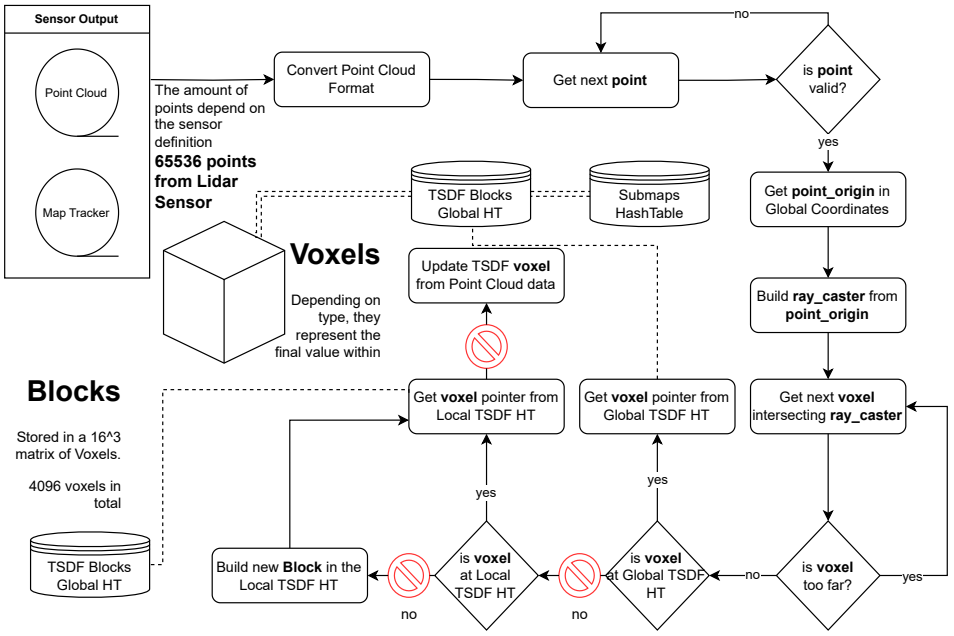

Fig. 3.2: Flujo de ejecución del Front-end de Voxgraph.

El error se mide como la sumatoria de las diferencias o residuales entre la posición esperada del robot al inicio de cada submapa, basada en un calculo de odometria, respecto a la posición observada al inicio de la integración de una nube de puntos.

El termino del costo esta definido como:

$$e^{i,j}_{\text{odom}}(T_{WS^i}, T_{WS^j}) = \log(\hat{T}^{-1}_{S^iS^j} T^{-1}_{WS^i} T_{WS^j}). \tag{3.3}$$

b) Restricciones de cierre de ciclos

Voxgraph acepta cierres de ciclos de fuentes externas. La detección de ciclos toma la forma de una transformación estimada $\hat{T}_{C^lC^k}$ que vincula el marco del sensor C en los instantes de tiempo l y k que no están contenidos dentro del mismo submapa. Los cierres de ciclo relacionan la posición del sensor en instancias de tiempo arbitrarias y, por ende, no relacionan marcos de submapa directamente. Para resolver esto, se determinan los dos submapas, $\mathcal{S}_i$ y $\mathcal{S}_j$ que contenían estos instantes de tiempo, y se buscan las poses relativas a cada submapa $T_{S^iC^l}$ y $T_{S^jC^k}$ en la Ecuación 3.1.

c) Restricciones de registro de submapa

Como se detallará más adelante, Voxgraph utiliza el algoritmo *Iterative Closest Point* (ICP) para medir la distancia entre cada submapa superpuesto en la colección, lo que implica minimizar la distancia euclidiana entre

los puntos de cada submapa. Cada vez que se crea un nuevo submapa, este podría influir potencialmente en la posición y orientación de cualquier otro submapa e incluso activar la fusión entre dos submapas preexistentes. El cálculo de submapas superpuestos se aproxima utilizando Cajas Delimitadoras Alineadas por Ejes (AABBs Axis Aligned Bounding Boxes).

Una cuestión a tomar en cuenta es que esta aproximación podría producir falsos positivos, pero siempre incluirá todos los pares de submapas superpuestos. Este compromiso vale la pena al considerar que calcular la superposición exacta de cualquier par de submapas seria menos eficiente para el caso general.

Durante el primer paso, los puntos relevantes del algoritmo ICP se toman arbitrariamente de cada submapa según sus pesos. Este subconjunto de datos necesita recalcularse cada vez que se incluye un nuevo submapa en la colección, ya que las optimizaciones anteriores podrían haber modificado la posición y orientación de cualquier submapa.

Un enfoque para generar restricciones por pares sería determinar correspondencias punto a punto entre los puntos de la iso-superficie de pares de submapas y minimizar la distancia entre estos pares. Voxgraph presenta una alternativa, en la que, para un punto en la iso-superficie de S_i, podemos determinar la distancia a la iso-superficie de S_j leyendo el valor en el mapa ESDF para ese punto. La restricción de registro entre dos submapas expresa entonces la diferencia total de distancia al cuadrado evaluada para todos los puntos arbitrariamente seleccionados $p^m_{\mathcal{S}_i}$ del submapa $\mathcal{S}_i$. La función de costo del requerimiento de registración se define a continuación como:

$$e^{i,j}_{\text{reg}}(T_{WS^i}, T_{WS^j}) = \sum_{m=0}^{N_{\mathcal{S}_i}} r_{\mathcal{S}_i\mathcal{S}_j}(\vec{p}^{\,m}_{\mathcal{S}_i}, T_{S^jS^i})^2,$$

donde $N_{\mathcal{S}_i}$ es el total de puntos de la iso-superficie $\mathcal{U}_{\mathcal{S}_i}$ del submapa $\mathcal{S}_i$. El residuo $r_{\mathcal{S}_i\mathcal{S}_j}$ viene dado por:

$$\begin{aligned} r_{\mathcal{S}_i\mathcal{S}_j}(p^m_{\mathcal{S}_i}, T_{S^jS^i}) &= \Phi_{\mathcal{S}_i}(p^m_{\mathcal{S}_i}) - \Phi_{\mathcal{S}_j}(T_{S^jS^i}\ p^i_{\mathcal{S}_i}) \\ &= -\Phi_{\mathcal{S}_j}(T_{S^jS^i}\ p^m_{\mathcal{S}_i}) \end{aligned}$$

donde $T_{S^jS^i}$ es una función de variables optimizadas en $\mathcal{X}$ a través de

$$T_{S^jS^i} = T^{-1}_{WS^j} T_{WS^i}. \tag{3.4}$$

Además de la función de perdida, el backend tambien utiliza su primera derivada. Este Jacobiano se obtiene al diferenciar el residual sobre las coordenadas $(x, y, zyyaw)$ de la pose de los submapas S_i y S_j. Tomando q como la representación de combinación de estas coordenadas y declaramos la forma general de la solución como:

$$\frac{\partial r_{\mathcal{S}_i\mathcal{S}_j}(\vec{p}^{\,m}_{\mathcal{S}_i})}{\partial q} = -\frac{\partial \left[\Phi_{\mathcal{S}_j}(T_{S^jS^i}\ \vec{p}^{\,m}_{\mathcal{S}_i})\ \right]}{\partial q}$$
$$= -\frac{\partial \Phi_{\mathcal{S}_j}(\vec{s})}{\partial \vec{s}}\bigg|_{\vec{s}=T_{S^jS^i}\ \vec{p}^{\,m}_{\mathcal{S}_i}} \frac{\partial \left(T_{S^jS^i}\ \vec{p}^{\,m}_{\mathcal{S}_i}\right)}{\partial q}.$$

El primer termino corresponde a la derivada de la función de interpolación trilinear de la ESDF [74], dada por:

$$\Phi_{\mathcal{S}_j}(\vec{s}) = \vec{g}^T B^T h(\vec{s})$$

donde el vector $\vec{g}$ mantiene las distancias de 8 vóxeles al rededor del punto $\vec{p}$, la matriz B representa una matriz binaria constante de 8 x 8 matriz y el vector h esta dado por:

$$\vec{h} = [1\ \ \Delta x\ \ \Delta y\ \ \Delta z\ \ \Delta x\Delta y\ \ \Delta y\Delta z\ \ \Delta z\Delta x\ \ \Delta x\Delta y\Delta z]^T$$

donde Δx, Δy y Δz son las distancias relativas desde $\vec{s}$ a su vecino izquierdo inferior $\vec{v}$.

d) Subsampling

Dado el número típicamente grande de puntos en las iso-superficies dentro de cada submapa, la minimización conjunta del error de registro para todos los puntos de superficie de todos los pares de submapas superpuestos es computacionalmente costosa, incluso para mapas de tamaño moderado. Por otro lado, no todos los puntos tienen la misma confianza sobre la geometria que representan. Voxgraph presenta un método de submuestreo para aumentar la eficiencia computacional de las restricciones de registro del backend. Utilizando solo un subconjunto aleatorio de sus residuos dentro de cada iteración del solucionador. La optimización no lineal resultante comparte algunas similitudes con el Descenso de Gradiente Estocástico (SGD), que ha tenido aplicaciones exitosas en problemas de SLAM en el pasado. Para cada punto de iso-superficie $p_m^{S_i}$ en U_{S_i}, determinamos un peso $\omega_{S_i}(p_m^{S_i})$ interpolando los pesos de los vóxeles que lo rodean. Los pesos de los vóxeles proporcionan una medida de confianza sobre la distancia estimada por cada vóxel. Obtenemos el subconjunto V_k extrayendo N_{V_k} muestras de U_{S_i} con reemplazo, donde la probabilidad de extraer cada punto $v_m^{S_i}$ es proporcional a su peso $w = \omega_{S_i}(v_m^{S_i})$.

e) Particionamiento espacial del entorno

Voxgraph utiliza una tabla hash de tamaño dinámico que hace uso del enfoque presentado por [12]. Hay tres niveles de particiones al indexar posiciones en la representación del mapa global. El primer nivel utiliza una

colección de submapas superpuestos, junto con sus restricciones y poses relativas para estimar su alineación más probable. Esta colección se implementa utilizando una tabla hash de la Biblioteca Estándar de C++ para indexar cada submapa. Dentro de cada submapa hay un segundo nivel de partición representado por bloques, los cuales contienen una matriz de un número fijo de vóxeles, usualmente 8×8 o 16×16, estos tamaños son configurables y suelen elegirse en base a la arquitectura de hardware. Los bloques se almacenan en una tabla de hash de la Biblioteca Estándar de C++ utilizando dos funciones de hash. La primera función transforma las coordenadas continuas representadas como una tupla de puntos flotantes de 64 bits, a coordenadas globales discretas representadas por una tupla de enteros sin signo de 64 bits. Esto crea un segundo nivel de partición llamado Discrete Global Grid (DGG), después de eso se aplica la segunda función hash denominada DECO para mejorar la eficiencia al recorrer los datos, siguiendo el método de [13].

El problema con este enfoque es que cuando se crea un bloque, se crea un nuevo objeto y la tabla de hash guarda el puntero a su posición en memoria. Creando así un patrón de memoria dispersa dentro de cada submapa. El último nivel de partición se realiza dentro de los bloques, utilizando una matriz cuadrada densa de vóxeles del mismo tamaño (ver Fig. 3.3). Por lo tanto, las tablas de hash utilizadas por Voxgraph sufren de un rendimiento deficiente inducido por patrones de acceso a memoria irregulares.

Los avances recientes [14, 15] han demostrado que la adopción de la disposición de memoria SoA para las tablas de hash y sus claves, mejora sustancialmente el rendimiento de las operaciones de inserción y búsqueda. La disposición AoS proporciona una localidad de caché relativamente alta si se accede tanto a la clave como al valor simultáneamente. Sin embargo, si sólo se accede a la clave, la efectividad de la caché se reduce. Esto es especialmente crítico si el tipo de valor es grande en comparación con el tipo de clave, como es en el caso de Voxgraph. En este trabajo, adoptamos estas ideas junto con la biblioteca SoA para implementar la tabla de hash.

3.2.3. Back-end

El Back-end de Voxgraph es le responsable de mantener el conjunto de restricciones generadas por el Front-end (odometría, cierre de ciclos y registro de submapas) y estima la alineación de la colección de submapas más probable minimizando el error total de todas las restricciones del grafo de poses. Para hacer eso, tiene que resolver la aproximacion por mínimos cuadrados no lineales calculando las matrices de residuos y jacobianos para cada vóxeles y vértice de los submapas superpuestos. El cálculo no ponderado de los valores de los elementos de estas matrices es completamente independiente entre sí, el único punto en común es el valor de la sumatoria del peso de los vóxeles o vértices relevantes, necesarios para calcular la media aritméti-

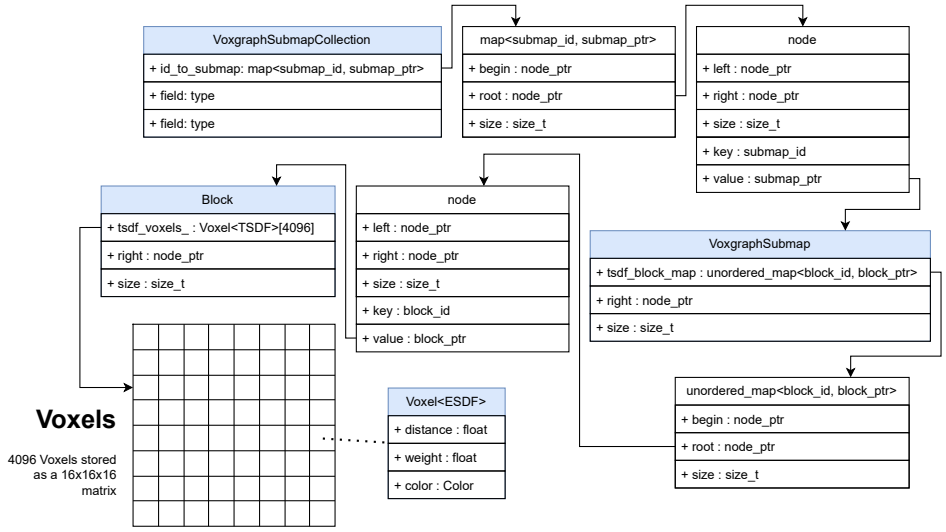

Fig. 3.3: Diagrama del particionamiento espacial utilizado en Voxgraph, cada flecha implica una indirección por un puntero.

ca ponderada de los valores finales de las matrices de residuos y jacobianas. Sin embargo, como este valor se precalcula para cada submapa cuando este es finalizado, no añade ninguna dependencia al cálculo, y por lo tanto, su implementación en GPU es perfectamente paralelizable [75].

El back-end alinea la colección de submapas minimizando el error total de todas restricciones generadas por el Front-end. Esta minimización no lineal se resuelve por mínimos cuadrados:

$$\begin{aligned}\arg\min_{\mathcal{X}} \sum_{(i,j)\in\mathcal{R}} ||e^{i,j}_{\text{reg}}(T_{WS^i}, T_{WS^j})||^2_{\sigma_\mathcal{R}} + \\ \sum_{(i,j)\in\mathcal{O}} ||\boldsymbol{e}^{i,j}_{\text{odom}}(T_{WS^i}, T_{WS^j})||^2_{\Sigma_\mathcal{O}} + \\ \sum_{(i,j)\in\mathcal{L}} ||\boldsymbol{e}^{i,j}_{\text{loop}}(T_{WS^i}, T_{WS^j})||^2_{\Sigma_\mathcal{L}}\end{aligned} \tag{3.5}$$

donde

$$\mathcal{X} = \{T_{WS^1}, T_{WS^2}, \ldots, T_{WS^N}\} \tag{3.6}$$

son las poses de los submapas, $T_{WS^i} \in \mathbb{R} \times SO(2)$, y $\mathcal{O}$, $\mathcal{L}$ y $\mathcal{R}$ son conjuntos que contienen pares de índices de submapas unidos por restricciones de odometría, cierre de ciclos y registración respectivamente. La distancia de Mahalanobis al cuadrado para las restricciones de odometría se escribe como $||\boldsymbol{e}||^2_{\Sigma_\mathcal{O}}$ y $||\boldsymbol{e}||^2_{\Sigma_\mathcal{L}}$, donde $\Sigma_\mathcal{O}$ y $\Sigma_\mathcal{O}$ representan sus matrices de covarianza. Las restricciones de registración $||e||^2_{\sigma_\mathcal{R}}$ corresponden al cuadrado de la distancia total ponderada, con peso $\sigma_\mathcal{R}$ escalar. Notar que las variables

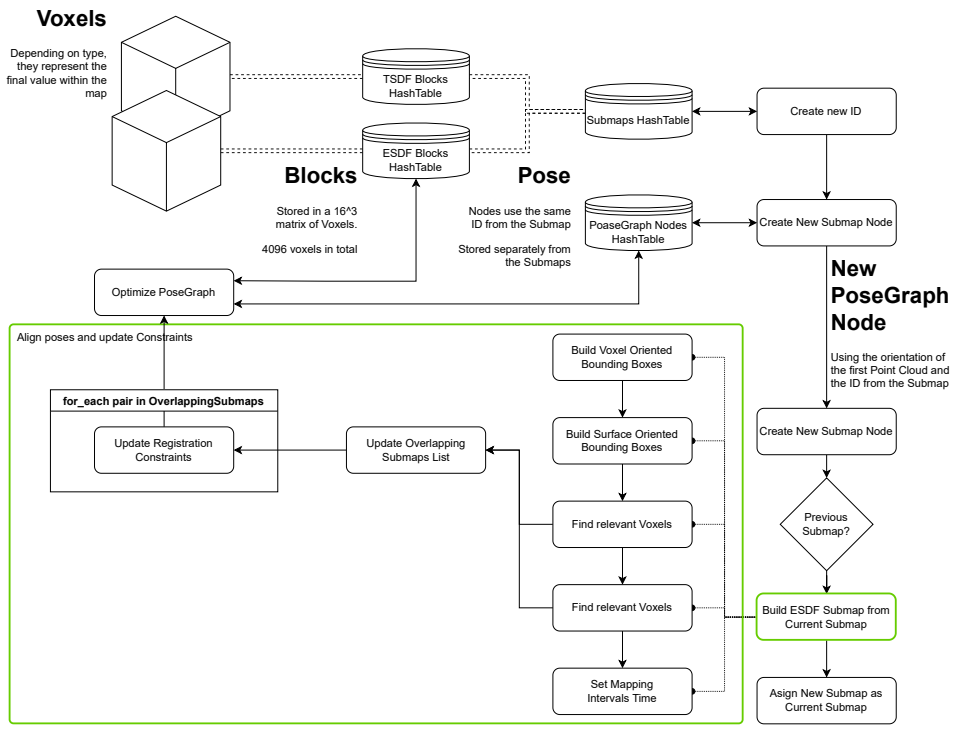

Fig. 3.4: Flujo de ejecución del Back-end de Voxgraph.

de optimización T_{WS^i} viven en una variedad (*manifold*) no lineal, por lo que deben manejarse en la correspondiente Álgebra de Lie.

Los residuos de odometría se implementan como términos de costo en la pose relativa entre marcos base de los submapas, de modo que para el caso general de los submapas $\mathcal{S}_i$ y $\mathcal{S}_j$ tenemos que

$$\boldsymbol{e}^{i,j}_{\text{odom}}(T_{WS^i}, T_{WS^j}) = \log(\hat{T}^{-1}_{S^iS^j} T^{-1}_{WS^i} T_{WS^j}) \tag{3.7}$$

En el caso particular de la odometría, $j = i + 1$ y $\hat{T}_{S^iS^{i+1}}$ es la estimación de la pose relativa calculada en el front-end.

Los residuos de cierre de ciclos restringen la posición relativa de los puntos dentro de dos submapas que no están ubicados con los propios marcos de submapa. Para los submapas $\mathcal{S}_i$ y $\mathcal{S}_j$ y los marcos del sensor C^l y C^k se tiene:

$$\boldsymbol{e}^{i,j}_{\text{loop}}(T_{WS^i}, T_{WS^j}) = \log(\hat{T}_{C^lC^k}(T_{WS^j} T_{S^jC^k})^{-1} T_{WS^i} T_{S^iC^l}) \tag{3.8}$$

donde $T_{S^iC^l}$ y $T_{S^jC^k}$ son las poses del sensor en el instante de tiempo l y k con respecto a sus marcos de submapa. En este trabajo no implementamos un algoritmo de detección de cierre de ciclos, sin embargo se ofrece la posibilidad de utilizar el error relativo entre submapas junto con una funcionalidad

para incluir esta restricción entre cualquier par de submapas del mapa global. Por otro lado, la consistencia global la logramos utilizando la restricción de registracción como se verá más adelante. Esta restricción evalua todo posible par de submapa que se superponga, evaluando asi cualquier camino que pueda generar un cierre de ciclo que deba ser optimizado. Finalmente, respecto de la generación de restricciones por la registración por pares para los submapas producidos por el Front-end, se sigue el razonamiento detrás del algoritmo *Iterative Closest Point* (ICP), es decir, dos submapas se pueden alinear minimizando la distancia euclidiana entre puntos en sus superficies y se lo adapta para utilizar la representación SDF subyacente. Los submapas generados tienen un conjunto de puntos en una isosuperficie de nivel cero y una ESDF $\Phi_{\mathcal{S}}$. En Voxgraph se utiliza una alineación libre de correspondencias. Para un punto en la isosuperficie de $\mathcal{S}_i$ se puede determinar la distancia a la isosuperficie de $\mathcal{S}_j$ leyendo el valor de la ESDF en ese punto. La restricción de registración entre dos submapas expresa la diferencia de distancia total al cuadrado evaluada para todos los puntos $\boldsymbol{p}^m_{\mathcal{S}_i}$ de la isosuperficie de nivel cero del submapa $\mathcal{S}_i$,

$$e^{i,j}_{\text{reg}}(T_{WS^i}, T_{WS^j}) = \sum_{m=0}^{N_{\mathcal{S}_i}} r_{\mathcal{S}_i\mathcal{S}_j}(\boldsymbol{p}^m_{\mathcal{S}_i}, T_{S^jS^i})^2,$$

donde $N_{\mathcal{S}_i}$ es el número total de puntos de la isosuperficie del submapa $\mathcal{S}_i$. El residuo $r_{\mathcal{S}_i\mathcal{S}_j}$ está dado por:

$$\begin{aligned} r_{\mathcal{S}_i\mathcal{S}_j}(\boldsymbol{p}^m_{\mathcal{S}_i}, T_{S^jS^i}) &= \Phi_{\mathcal{S}_i}(\boldsymbol{p}^m_{\mathcal{S}_i}) - \Phi_{\mathcal{S}_j}(T_{S^jS^i}\boldsymbol{p}^i_{\mathcal{S}_i}) \\ &= -\Phi_{\mathcal{S}_j}(T_{S^jS^i}\boldsymbol{p}^m_{\mathcal{S}_i}) \end{aligned}$$

donde $T_{S^jS^i}$ es una función de optimización de las variables en $\mathcal{X}$ mediante:

$$T_{S^jS^i} = T^{-1}_{WS^j} T_{WS^i}. \tag{3.9}$$

Notar que $\Phi_{\mathcal{S}_i}(\boldsymbol{p}^m_{\mathcal{S}_i}) = 0$ para todos los puntos $\boldsymbol{p}^m_{\mathcal{S}_i}$, dado que esos puntos se hayan en la isosuperficie de nivel cero de $\mathcal{S}_i$ por construcción.

Dada la gran cantidad de puntos en isosuperficies dentro de cada submapa, la minimización conjunta del error de registración para todos los puntos de superficie de todos los pares de submapas superpuestos es muy costosa. Por lo tanto, Voxgraph aproxima las restricciones de registración utilizando solo un subconjunto aleatorio (sub-sampleo) de sus residuos dentro de cada iteración del *solver* que se utiliza para resolver la optimización, siguiendo el algoritmo descripto en [22].

3.2.4. Perfiles de tiempos de ejecución

Se realizó un analisis exhaustivo de los tiempos de ejecución de cada módulo de Voxgraph. Esto proporciona información valiosa sobre el consumo de tiempo y también sobre la distribución del uso de la CPU, detectando

cuellos de botella, y guiando los esfuerzos para mejorar el sistema en términos de rendimiento para la versión acelerada en GPU.

Voxgraph requiere poder ser ejecutado en tiempo real para garantizar la estabilidad de la cantidad de datos de cada submapa. Las nubes de punto de entrada, destinadas a ser parte del submapa actual, se omitirán si el sistema no puede procesarlas en tiempo real. Esta variabilidad afecta la cantidad final de datos que se utilizan para cada submapa y, por lo tanto, el tiempo que demanda. Por esta razón, para realizar el análisi de tiempos de ejecución, se modificó el código oringal para evitar la omisión de nube de puntos, de manera similar a lo que se conoce como *process-every-frame* [76]. El conjunto de datos utilizado para el análisis de tiempos fue el mismo que Voxgraph (para más detalles, consulte la Sección 5.3).

Cada determinado tiempo se crea un nuevo submapa en Voxgraph, en ese momento el sistema necesita realizar la integración de las nubes de puntos recolectadas en esa ventana de tiempo, la actualización del mapa ESDF y la optimización del grafo de poses. La Fig. 3.5 muestra la distribución de tiempo en el momento en el que se crea un nuevo submapa. Considerando que, entre una medición y otra, hay alrededor de 20 ejecuciones de la integración de la nube de puntos, los tres módulos que demandan más tiempo de ejeción son: la integración de la nube de puntos, la actalización del mapa ESDF y la optimización del grafo de poses, como ya fuera mencionado tambien por los autores de Voxgraph [22]. Como puede observarse, el tiempo de ejecución del Front-end (actualización del mapa ESDF y la integración de nube de puntos), se mantiene constante debido a que no hay cambios significativos en la cantidad de nube de puntos que se integran. Sin embargo, el tiempo de ejecución del Back-end (específicamente de la optimización de grafo de poses) aumenta a medida que crece la cantidad de submapas totales en el mapa global.

3.3. Nvblox

Un trabajo muy reciente que puede considerarse una evolución de Voxgraph y que implementa mapeo volumétrico acelerado por GPU es nvBlox [63]. La arquitectura del sistema está configurada para el mapeo TSDF desde un sensor de tipo cámara de profundidad RGB-D. El mapa reconstruido (llamado LayerCake) está compuesto por grillas de vóxeles superpuestas, co-localizadas y alineadas, basada en el trabajo de [71]. Los mapas de profundidad y las imágenes de color se integran en las capas de vóxeles TSDF y Color, de las cuales se derivan las grillas de vóxeles que contienen el ESDF y una reconstrucción de malla.

El sistema nvBlox sostiene su capacidad para realizar mapas globalmente consistentes en la aproximación de la odometría que propone Fast-LIO [77]. Por otro lado, logra mejoras significativas de velocidad tanto para la construcción del mapa TSDF ($\times$177) como para la actualización del mapa ESDF

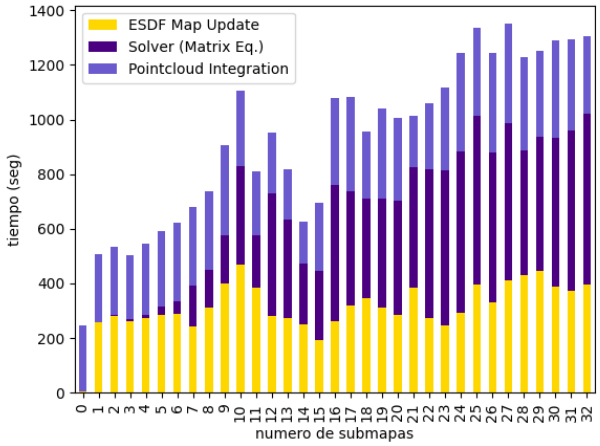

Fig. 3.5: Mediciones de tiempos de ejecución de cada módulo de Voxgraph.

(×31). Estos resultados convienten a nvBlox en el estado del arte en lo que se refiere a mapeo volumétrico acelerado por GPU. Por este motivo, en esta tesis realizamos experimentos para comparar el rendimiento de nuestro sistema coVoxSLAM contra nvBlox.

El mapa que construye nvBlox es disperso, de modo que los vóxeles solo se asignan en las regiones del espacio 3D que se observan durante el mapeo. Esta dispersión se logra utilizando una jerarquía de dos niveles, siguiendo el trabajo de [12].

El primer nivel es una tabla hash que mapea los índices de la grilla 3D a VoxelBlocks. En el sistema nvBlox, esta tabla hash se puede consultar en los núcleos de GPU utilizando una interfaz basada en la biblioteca stdgpu [50]. En el segundo nivel, cada VoxelBlock contiene un grupo de $8 \times 8 \times 8$ vóxeles que se almacenan de manera contigua en la memoria de la GPU, lo que lleva a cargas fusionadas (coalescentes) en los núcleos de GPU.

Los datos que llegan del sensor se añaden a la reconstrucción almacenada en una de las capas del mapa. Esto ocurre en varios pasos. Primero, se trazan rayos a través de la grilla de VoxelBlock en la GPU para determinar qué VoxelBlocks están a la vista utilizando [78] y se asignan aquellos que aún no están en el mapa. Finalmente, para actualizar los vóxeles, se llama a una función de actualización por vóxel en paralelo en la GPU sobre todos los vóxeles en vista. Para calcular el mapa ESDF el trabajo se basa en el algoritmo de bandas paralelas [79]

La librería nvBlox es una de las primeras en su tipo, logrando implementar en GPU casi de manera completa el Front-end de los sistemas SLAM densos. Incluye un nuevo método incremental acelerado por GPU para calcular el mapa ESDF y una arquitectura de dos niveles que le permite construir mapas esparsos utilizando una tabla de hash en GPU que contiene única-

mente la información del ambiente explorado.

La tabla de hash que se utiliza en el sistema es la stdgpu [50], que corresponde a una implementación que ejecuta en GPU. Su diseño esta orientado en ofrecer soporte para tipos de datos de la libreria estandar de C++ tales como *unordered_map* y *unordered_set*. Brinda una opción rápida para reemplazar una tabla de hash tradicional sin demasiado esfuerzo. Sin embargo, no esta optimizada para soportar particionamiento espacial, inserciones y busqueda libres de colisiones como es el caso de ASH y al abarcar un rango general de estructuras, no implementa una optimización para el manejo de claves y valores de tipo de 64 bits como es el caso de warpcore.

El algoritmo de integración de nube de puntos funciona casi enteramente en GPU, brindando la posibilidad de representar mapas de mayores dimensiones y con mayor resolución sin perder la ejecución en tiempo real. Sin embargo, al utilizar mapeo de proyecciones no se puede analizar el peso de todos los rayos que caen sobre un mismo voxel de la grilla, por lo que resulta necesario utilizar alguna técnica de subsampleo. Nvblox utiliza una aproximación lineal (interpolación) del peso de un voxel utilizando un selección de nueve puntos al rededor del voxel original. Esto impacta en el nivel de confianza resultante de la ponderación de los pesos de los vóxeles, lo cual disminuye la correctitud del método, ya que puede inducir problemas de *aliasing* o perdida de detalles del entorno explorado en la representación del mapa.

Por otro lado, nvBlox no implementa un grafo de poses para mantener la consistencia global del mapa. En contraste con el método utilizado en Voxgraph, nvBlox fusiona los datos del sensor en la representación del mapa de manera directa al momento de la integración. Esto dificulta o imposibilita la toma de decisiones sobre errores acumulados a lo largo de la navegación dado que se pierde la noción de como los datos fueron recolectados a lo largo de la exploración del robot. Por ende, tampoco existe un grafo de poses para realizar la corrección de errores de cierre de ciclos, por lo que podemos afirmar que nvBlox no se trata de un sistema de SLAM completo.

4. SISTEMA COVOXSLAM

Los sistemas SLAM que se implementan en arquitecturas heterogéneas deben soportar diversas configuraciones de hardware interactuando entre sí en tiempo real. Esto puede incluir sistemas embebidos dedicados al manejo de sensores y actuadores, unidades de control que facilitan la toma de decisiones, así como conexiones remotas para monitoreo, gestión de datos e internet de las cosas (IoT). Recientemente, ha habido un notable aumento en el uso de GPUs con notables mejoras de rendimiento para estos sistemas [80].

En los sistemas SLAM en tiempo real, al igual que en otros sistemas de esta naturaleza, el rendimiento y la precisión dependen en gran medida de la cantidad de datos que se pueden procesar en un intervalo de tiempo determinado. Estos datos, que generalmente provienen de los sensores del robot, deben ser procesados inmediatamente una vez que se obtienen, para maximizar la utilización de información que, de otro modo, se descartaría. Este procesamiento implica la creación de un modelo que represente el entorno conocido, el desarrollo de un método para minimizar el error acumulado a lo largo de la trayectoria y el empleo de estructuras de datos que facilite la toma de decisiones durante la navegación.

4.1. Descripción general

La descripción general del funcionamiento del sistema coVoxSLAM puede verse en la Figura 4.1. El Front-End comienza con la llegada de las mediciones de los sensores que se encolan para ser procesadas de manera asincrónica. El Front-end se encarga en primer lugar de desencolarlas y durante una cantidad arbitraria de tiempo y/o distancia recorrida, genera el submapa TSDF correspondiente. También actualiza la lista de pares de mapas solapados y el subsampleo de puntos relevantes para cada uno. Estos datos son necesarios para los cálculos de errores de odometría, cierre de ciclos y registración para el algorítmo de aproximación por cuadrados mínimos ejecutado en el Back-end. Al finalizar este proceso, el submapa generado se agrega a la cola de submapas TSDF para ser procesados también de manera asincrónica tanto por el Back-end como por el generador de mapas ESDF. El generador de ESDF es el último paso realizado por el Front-end, se encarga de generar el mapa que luego se utilizará para realizar el cálculo de trayectorias para la navegación autonoma, para evitar colisiones con objetos presentes en el ambiente explorado o para la visualización entre otras cosas. Finalmente, el Back-end utiliza los datos previamente cálculados en el Front-end para realizar la optimización del grafo de poses, garantizando la consistencia global del mapa.

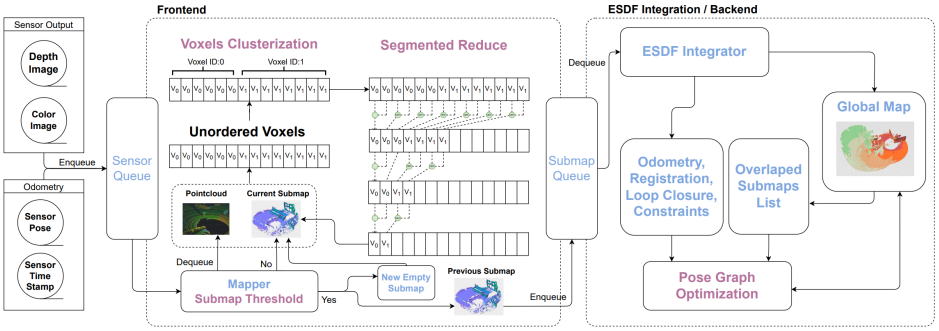

Fig. 4.1: Flujo de datos del sistema coVoxSLAM. Puede observarse la utilización de dos colas independientes para el Front-end y el Back-end. El módulo mapeador lee de su cola cuando hay una nueva nube de puntos disponible. Cada vez que pasa esto, si el umbral del submapa se satisface, se devuelve un nuevo submapa vacío al Integrador TSDF y se encola el submapa actual para ser procesado por el Integrador ESDF y el Back-end. El tamaño de ambas colas puede modificarse como parámetros globales del sistema.

4.2. Manejo de memoria

La organización de los datos en memoria es el principal factor limitante en la reducción de la latencia de las operaciones del sistema, así como en el aumento de su rendimiento. Dependiendo de las características del sistema SLAM, esto se reflejará en el tiempo requerido para la construcción del mapa y en la obtención de características específicas durante la toma de decisiones para la planificación de trayectorias o en la minimización del error acumulado de la misma.

Los arreglos son la estructura de datos más comúnmente utilizada para el almacenamiento de información, debido a su representación explícita de la memoria física en cualquier arquitectura de computador de tipo von Neumann. Además, permiten definir estrategias que optimizan el uso del hardware en el que se ejecuta el sistema. Esta representación, aunque limitada, facilita la optimización del rendimiento del código a través de tecnicas como desenrollado de bucles, pre-carga de datos, alineación de caché y vectorización por parte de arquitecturas de tipo SIMD (*Single Instruction Multiple Data*) [81].

No obstante, esta rigidez en la representación puede generar dificultades al intentar modelar datos del mundo real dentro del sistema. Por ejemplo, cuando los datos carecen de una continuidad preestablecida, la representación mediante arreglos se vuelve poco práctica. En estos casos, el uso de arreglos dispersos se vuelve más eficiente dado que minimizan la necesidad de representar datos nulos. En sistemas SLAM suele encontrarse frecuentemente este problema por ejemplo cuando se proyecta explorar mapas indeterminadamente grandes. Para solucionarlo suelen utilizarse dos tipos de

estructuras para representar los arreglos esparzos, estas son las tablas de hash y los arboles octales.

Otro aspecto a tomar en cuenta para las estructuras de datos tiene que ver con soportar distinto tipos de disposiciones (*layout*) en memoria. Este no es un problema que este ampliamente abordado en la actualidad y sus desarrollos son más bien variados y específicos para ciertos tipos de problemas [15,81,82]. Esto se debe a que es necesario comprometer ciertas variables del sistema para favorecer otras, como por ejemplo mejorar la latencia o la performance en pos de la cantidad de memoria total utilizada. Por otro lado, el manejo de datos multivariado aumenta relativamente más la complejidad y resulta en valores de performance que dependen de la coordinación entre el funcionamiento del método/algoritmo y la estructura de datos elegida. Por eso, cuando se selecciona un tipo de estructura para almacenar los datos en pos de una característica del sistema, otra podría verse afectada. En sistemas SLAM, que presentan módulos heterogeneos con caracteristicas diversas esto se vuelve más evidente. Por ejemplo, la visualización del mapa que se explora requiere de un subconjunto de datos distinto al que se utiliza para minimizar el error acumulado.

En las arquitecturas de hardware modernas, debemos considerar además las jerarquías de caché, las latencias, los anchos de banda, las tecnologías de memoria volátil (por ejemplo, DDR, GDDR, HBM), las arquitecturas de memoria no volátil direccionables por bytes (por ejemplo, SCM), las memorias programables de scratchpad (por ejemplo, memoria compartida en CUDA), los tamaños de memoria (por ejemplo, tamaño de la caché L1) y el tipo de acceso (lectura, escritura, atómico, sin caché, secuencial, no temporal). También es necesario considerar los patrones de acceso espaciales y temporales del algoritmo. Todas estas distintas opciones podrian estar presentes en un sistema SLAM.

4.2.1. Disposición de datos SoA

Para abordar la mencionada heterogeneidad del hardware utilizado en diversos sistemas de SLAM, como parte de esta tesis se desarrolló una biblioteca de gestión de memoria que permite, en esencia, elegir libremente entre la manipulación de grandes colecciones de datos utilizando las dispocisiones en memoria SoA (*Struct of Arrays*), AoS (*Arrays of Structs*) o la combinación de ambas como por ejemplo SoAoS (*Struct of Arrays of Structs*). También permite utilizar arreglos dinámicos, para los casos donde no se sabe o no se puede calcular una cota para el tamaño del arreglo. [83–86]

Definir la disposición en memoria de grandes colecciones de datos multivaluados es indispensable para el desarrollo de modelos acelerados por GPU. Usualmente existen dos principales modelos AoS y SoA. La elección indistinta no siempre es posible, por lo que debe evaluarse cada caso particularmente, no obstante la elección de SoA sobre AoS por lo general resulta en una

mejora debido a una serie de factores que se mencionan en la bibliografía [82]

- Maximiza la adaptación del código para el uso de instrucciones vectoriales SIMD.
- Mejora la gestión de caché maximizando la tasa de aciertos (hit rate)
- No requiere memoria extra inutilizada para alinear internamente los miembros de una estructura.
- Por lo anterior, la cantidad de memoria necesaria para alinear un arreglo y sus elementos no depende de la longitud del arreglo.
- Mejora el rendimiento de la transferencia de datos a través de la fusión de lecturas y escrituras (lo que se conoce como coalescencia de memoria)
- Facilita la serialización y deserealización de grandes colecciones de datos cuando son enviados entre módulos.
- Aumenta la probabilidad de optimizaciones por parte del compilador como desenrollado de ciclos (loop unrolling)

Para la implementación de este trabajo, se desarrolló una biblioteca para gestionar la disposición de memoria, que controla el flujo de datos y las diversas operaciones realizadas desde que el sensor genera la colección de valores de la muestra, hasta su almacenamiento final en el sistema. Este flujo de datos abarca tanto las funcionalidades dentro de cada módulo del sistema, como la minimización de errores y la visualización del mapa, como también la comunicación entre ellos, incluyendo la transferencia de datos de la GPU a la CPU para su serialización y posterior almacenamiento en un disco externo remoto.

Dada la creciente disparidad entre la velocidad de procesamiento de las CPU y la velocidad de suministro de datos de la memoria, este tipo de disposición puede mejorar notablemente el rendimiento en aplicaciones limitadas por la memoria, especialmente aquellas que manejan grandes volúmenes de datos y requieren procesamiento paralelo, como en el caso del SLAM que utilizan LiDAR o cámaras RGB-D como sensores principales.

4.2.2. Disposición de memoria utilizando estructuras

En C++, una estructura (o `struct`) es un tipo de dato compuesto definido por el usuario que agrupa variables de diferentes tipos bajo un solo nombre. Estas variables, conocidas como miembros, deben tener nombres distintos y se declaran dentro de las llaves de la estructura [87].

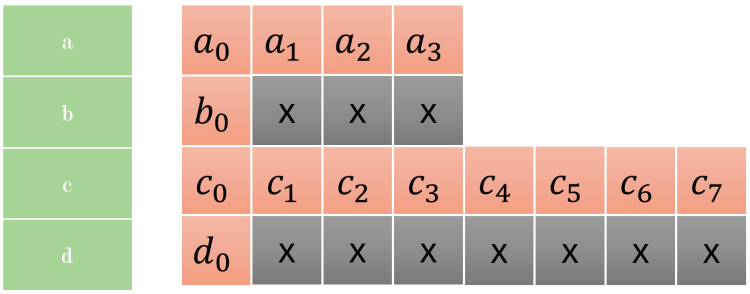

Fig. 4.2: Relleno (*padding*) de la estructura 4.1 con dispocisión en memoria AoS. Cada celda representa un byte, las celdas grises representan el relleno.

```
struct ejemplo_t {
    float a;
    bool b;
    double c;
    bool d;
}
```

Listing 4.1: Ejemplo declaración de estructura

Dentro de la disposición de memoria utilizada para representar la declaración de la Estructura 4.1 *ejemplo_t*, ordena los miembros en el mismo orden en el que figuran en la definición, para este caso (a,b,c,d). Además alinea utilizando el máximo entre la alineación minima especificada de manera explicita en el código (en caso de existir) y el tamaño del tipo del miembro dentro de la estructura.

En la Fig 4.2 se observa el relleno necesario para mantener la alineación de cada uno de los miembros de la estructura. En este caso la alineación de cada miembros es de 4 bytes para a, 1 bytes para b y d, 8 bytes para c. Por su parte, la estructura está también alineada al máximo entre cada uno de sus miembros y el especificado explicitamente, para este caso es de 8 bytes, esto último es condición suficiente para garantizar la alineación de cada miembro. Finalmente, para garantizar la alineación de cada miembro se agrega relleno (*padding*) donde sea necesario. En este ejemplo es necesario agregar relleno entre b y c es de 3 bytes, para garantizar la alineación de c dentro de la estructura y 7 bytes al final para alinear la estructura completa dentro de un arreglo. Como puede observarse, de los 24 bytes finales que ocupa la estructura, 10 se desperdician en relleno para el alineamiento.

Algunos lenguajes de programación elijen la manera óptima para ordenar los campos declarados de una estructura de datos para minimizar el relleno, sin embargo esto no siempre garantiza su ausencia. Por otro lado, esta técnica agrega complejidad al momento de transferir datos de una arquitectura a otra dado que podría variar el orden de los miembros dentro de la estructura.

4.2.3. Disposición de memoria utilizando arreglos

Mientras que cualquier estructura (`struct`) es un tipo heterogéneo, cualquier arreglo (`array`) es un tipo homogéneo. Dado una variable entera k y un tipo T, se obtiene el tipo de secuencia de tamaño constante $array < T, k >$. Según el estándar de C++, un arreglo es una secuencia de variables de tipo T almacenados de manera contigua en memoria. Esto significa que los elementos de un arreglo se almacenan uno al lado del otro en memoria, sin espacios intermedios. El tamaño total del arreglo en memoria es igual al tamaño de un elemento multiplicado por el número de elementos.

Las reglas implícitas para el relleno en un arreglo se heredan y se determinan según los requisitos de alineación del tipo de elemento T y sus miembros. El estándar de C++ especifica que cada elemento de un arreglo debe estar alineado en memoria de acuerdo con los requisitos de alineación de su tipo. Para el caso del ejemplo 4.1 la disposición en memoria de un arreglo de tipo $ejemplo\_t$ se representaría de como se muestra en la Figura 4.3:

a_0^0	a_1^0	a_2^0	a_3^0	b_0^0	x	x	x	c_0^0	c_1^0	c_2^0	c_3^0	c_4^0	c_5^0	c_6^0	c_7^0	d_0^0	x	x	x	x	x	x	x
a_0^1	a_1^1	a_2^1	a_3^1	b_0^1	x	x	x	c_0^1	c_1^1	c_2^1	c_3^1	c_4^1	c_5^1	c_6^1	c_7^1	d_0^1	x	x	x	x	x	x	x
a_0^2	a_1^2	a_2^2	a_3^2	b_0^2	x	x	x	c_0^2	c_1^2	c_2^2	c_3^2	c_4^2	c_5^2	c_6^2	c_7^2	d_0^2	x	x	x	x	x	x	x
a_0^3	a_1^3	a_2^3	a_3^3	b_0^3	x	x	x	c_0^3	c_1^3	c_2^3	c_3^3	c_4^3	c_5^3	c_6^3	c_7^3	d_0^3	x	x	x	x	x	x	x

Fig. 4.3: Arreglo de 4 elementos del tipo de la estructura 4.1. Cada celda representa un byte, las celdas grises representan el relleno. Las celdas $\{a, b, c, d\}_i^j$ reprensetan el i-ésimo byte del miembro $\{a, b, c, d\}$ del j-ésimo elemento del arreglo.

4.2.4. Bibliotecas actuales de SoA

Las implementaciones actuales de contenedores SoA incluyen varias bibliotecas disponibles [81,82,86,88–90]. A pesar de sus ventajas en rendimiento, la integración de estas bibliotecas en un código existente puede resultar complicada debido a la necesidad de redefinir los tipos de datos y reescribir largas secciones de código.

En particular, LLAMA [82] permite la configuración de disposiciones de datos como SoA o AoS. Sin embargo, esta configuración se limita únicamente al primer nivel de la definición de la estructura y no abarca estructuras anidadas. Además, estas implementaciones no cuentan con la capacidad de copiar datos de una definición SoA a otra mediante una descripción de mapeo entre sus miembros. En su lugar, solo ofrecen un mecanismo de copia por campo, lo que puede ser menos eficiente y más difícil de implementar para estructuras de datos complejas. Aunque copiar entre dos objetos SoA con la misma definición y tamaño es sencillo, cuando las definiciones o tamaños son diferentes esta operación resulta más compleja. Estas limitaciones subrayan

la necesidad de contenedores SoA más avanzados y flexibles que soporten completamente estructuras anidadas y mecanismos eficientes de copia de datos.

En contraste con las implementaciones actuales de contenedores SoA, en este trabajo se presenta una biblioteca de manejo de disposición en memoria que ofrece una capa de abstracción sin costo, lo que significa que no genera una indirección extra en tiempo de ejecución para realizar lecturas o escrituras y se integra fácilmente con bases de código preexistentes. Támbien hace uso de la vectorización automática para maximizar las capacidades SIMD, lo que se traduce en un procesamiento de datos más rápido. Según nuestros experimentos, el rendimiento de SoA mejora los tiempos de ejecución de consultas de búsqueda, proporcionalmente al tamaño del tipo consultado en relación con el tamaño total de la estructura principal. Por ejemplo, para una consultan únicamente la mitad de los campos de la estructura original, la mejora es del doble.

Es importante destacar también que la biblioteca desarrollada requiere un código mínimo por parte del usuario, facilitando su adopción. Esta flexibilidad es especialmente útil para aplicaciones que implican iterar sobre grandes conjuntos de datos o que utilizan acceso selectivo para diferentes funcionalidades. Al soportar tanto formatos SoA como AoS, el contenedor ofrece la flexibilidad necesaria para optimizar el uso de memoria y el rendimiento en diversos componentes de hardware, convirtiéndolo en una herramienta valiosa para aplicaciones que requieren un funcionamiento eficiente en sistemas heterogéneos.

4.2.5. Declaración de nuestra biblioteca SoA

La biblioteca de manejo de memoria desarrollada como parte de este trabajo utiliza un *tag* en la declaración del tipo para definir la disposición en memoria de los miembros del mismo. Para el caso de no existir dicho *tag*, se toma por defecto la disposición AoS que es la estándar dentro de C++

Siguiendo con el ejemplo 4.1, en caso de querer utilizar SoA únicamente es necesario definirlo como:

```
struct ejemplo_soa_t {
    using memory_layout = soa;
    float a;
    bool b;
    double c;
    bool d;
}
```

Listing 4.2: Ejemplo declaración de estructura

La disposición en memoria de un arreglo de tipo 4.2 difiere de la representación original 4.1 tal y como se muestra en la Figura 4.4. El relleno ya

no es necesario entre los miembros de la estructura y los valores de cada uno se encuentran continuos en memoria:

a_0^0	a_1^0	a_2^0	a_3^0	a_0^1	a_1^1	a_2^1	a_3^1	a_0^2	a_1^2	a_2^2	a_3^2	a_0^3	a_1^3	a_2^3	a_3^3
b_0^0	b_0^1	b_0^2	b_0^3	x	x	x	x	c_0^0	c_1^0	c_2^0	c_3^0	c_4^0	c_5^0	c_6^0	c_7^0
c_0^1	c_1^1	c_2^1	c_3^1	c_4^1	c_5^1	c_6^1	c_7^1	c_0^2	c_1^2	c_2^2	c_3^2	c_4^2	c_5^2	c_6^2	c_7^2
c_0^3	c_1^3	c_2^3	c_3^3	c_4^3	c_5^3	c_6^3	c_7^3	d_0^0	d_0^1	d_0^2	d_0^3	x	x	x	x

Fig. 4.4: Arreglo de 4 elementos de tipo 4.2. Cada celda representa un byte, las celdas grises representan el relleno. Las celdas $\{a, b, c, d\}_i^j$ reprensetan el i-ésimo byte del miembro $\{a, b, c, d\}$ del j-ésimo elemento del arreglo.

La biblioteca permite crear tipos con disposición en memoria mixta, es decir, un tipo de datos con disposición SoA que incluya miembros con disposición AoS y viceversa. Cualquier combinación tiene un unico valor de representación en memoria.

Siguiendo con el ejemplo 4.1 y 4.2, una estructura mixta podria definirse de la siguiente manera:

```
struct ejemplo_mixto_t {
    ejemplo_soa_t soa;
    ejemplo_t aos;
}
```

Listing 4.3: Ejemplo declaración de estructura

Para este caso, los miembros con disposición SoA se organizan individualmente en arreglos separados mientras que los miembros de la disposición AoS se acomodan dentro de un mismo bloque. Este tipo de layout mixto es muy útil cuando trabajamos con librerias que tienen requerimientos de disposición en memoria para devolver una colección de valor de resultado, como ocurre si utilizamos la biblioteca Ceres Solver, que modifica las transformaciones de cada nodo del grafo de poses al finalizar la aproximación utilizando un puntero a un arreglo. En el caso de tener una estructura con disposición SoA que incluye uno de sus elementos con disposición AoS, se dice que la estructura es Estructura-de-Arrays-de-Estructuras (SoAoS).

El relleno necesario para alinear los datos dentro de una disposición SoA es necesaria únicamente entre los arreglos utilizados por cada miembro de la estructura original. Este relleno depende de la longitud del arreglo y del requerimiento de alineación de cada miembro. Puede evitarse todo tipo de relleno si la longitud del arreglo es múltiplo del máximo entre los requerimientos de alineación de los miembros de la estructura. Para el caso de la Fig.4.4 si el arreglo fuese de 8 elementos no serían necesarios bytes de relleno.

4.2.6. Implementación de nuestra biblioteca de SoA

Para generar una colección de datos de un tipo que declare explicitamente la disposición de memoria, tal como se definió en la sección anterior, se utiliza el tipo:

```
draft<Type, N> collection;
```

Este tipo es escencialmente una tupla de arreglos que pueden utilizar tanto disposiciones SoA como AoS dependiendo de la declaración en `Type`.

Un `draft` de los datos difiere con un arreglo tradicional en cuanto al orden en que almacena cada uno de los valores de los miembros del tipo original. A diferencia de un arreglo tradicional, un `draft` no puede indexarse utilizando el operador `[]`, para observar su contenido es necesario crear una `view` del mismo.

Una `view` es una manera de indexar dentro de un `draft`, representa el tipo observado para cada elemento y reemplaza el operador `[]`, utilizado en un arreglo tradicional.

La manera de manipular datos tradicionalmente en lenguajes como C o C++ es a partir del operador[] `[]` que indexa unicamente utilizando el índice y el tamaño del tipo del elemento del arreglo. El resultado de $n\times$ sizeof(`Type`) devuelve el offset al n-ésimo elemento dentro del arreglo.

Para el caso de un `draft`, acceder al n-ésimo elemento requiere de información adicional, en particular el offset de cada colección de datos para cada miembro de la estructura original. Este offset pre calcula de forma de no agregar indirecciones extras sobre el operador `[]` de indexación.

Finalmente, en lo referido a la tranferencia de datos, podemos mencionar que un `draft` puede tener multiples `view` dependiendo de lo que sea necesario para cada uso. Además de limitar el acceso para cada modulo del sistema, también permite facilitar y optimizar la transferencia de datos que se realiza únicamente sobre los miembros observados de la estructura destino. Además, al utilizar disposición en memoria SoA, se optimiza la serialización y deserealización de los datos.

4.3. Front-End

4.3.1. Representación de datos

En coVoxSLAM se adopta el enfoque de la tabla de hash ASH [15] a través de nuestra biblioteca de manejo de memoria con disposición SoA. Específicamente, se utilizan búferes con disposición de memoria SoA para mejorar la eficiencia del procesamiento de datos. Además, para las operaciones de inserción y búsqueda, ASH introduce una nueva funcionalidad que activa las claves de entrada al insertarlas en la tabla de hash y obtener los índices de búffer asociados. Hacemos uso de esta característica de ASH para

acelerar la asignación de bloques en nuestro proceso de integración de mapas TSDF en el Front-end.

Sin embargo, a diferencia de ASH, que llena los búferes resultantes de una busqueda en disposición SoA, nuestro sistema mantiene los datos con disposición SoA internamente en los búferes de la tabla de hash. Esta elección de diseño simplifica el proceso para consultas de rango, eliminando la necesidad de pasos adicionales para construir el conjunto de los valores que se desean devolver. Mantener el formato SoA a lo largo del proceso asegura un acceso y manipulación de datos más eficientes, mejorando el rendimiento y reduciendo la complejidad. Además, este enfoque maximiza el rendimiento de la memoria al garantizar un uso eficiente de las líneas de caché y minimizar la latencia de acceso a la memoria [15].

```
struct punto {
    float x;
    float y;
    float z;
}
struct color {
    uint8_t x;
    uint8_t y;
    uint8_t z;
}

struct ESDF_voxel {
    float distance;
    float weight;
    punto position;
    color rgb_color;
}
```

Listing 4.4: Ejemplo declaración de estructura

En la Figura. 4.5 se representa la disposición en memoria de un voxel del mapa ESDF dentro de la tabla de hash, junto con los índices de la tabla, declarado como el ejemplo de código 4.4. Los campos de la estructura se almacenan consecutivos y al realizarse la busqueda por índice, esta ofrece una mayor flexibilidad al momento de devolver cualquier subconjunto de datos definido por el usuario. Por ejemplo: para la localización o exploración es posible que solo sean necesarios los datos de `distance` y `position`, mientras que para actualizar el mapa también va a ser necesario utilizar el dato del peso `weight` y para la visualización por su parte es posible que la consulta solo requiera de `rgb_color` y `position`.

Para el caso de uso que se le da a la tabla de hash en este trabajo, la utilidad de la biblioteca de disposición en memoria SoA ofrece un beneficio considerable. Los elementos que se almacenan son tanto bloques como submapas, ambos son estructuras de datos con una numerosa cantidad de datos que, no obstante, son utilizados de manera muy fraccionada por los distintos módulos del sistema. La elección de una arquitectura SoA junto

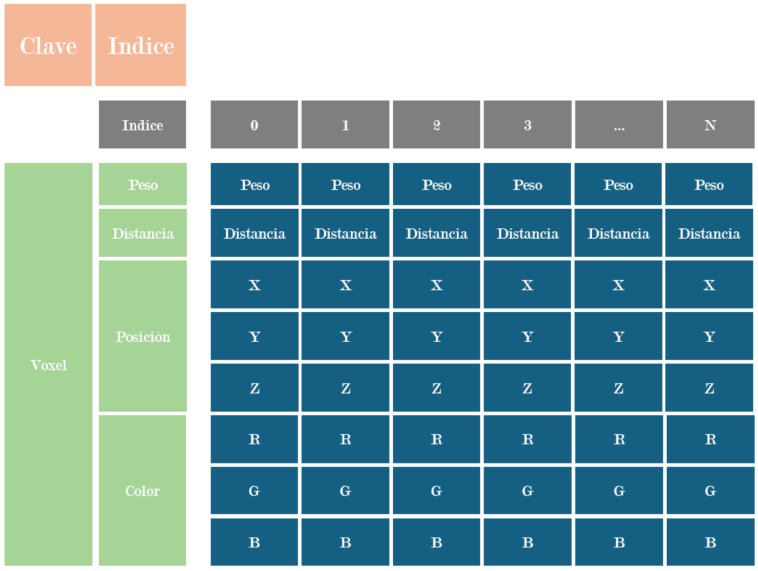

Fig. 4.5: Disposición de memoria en la tabla de hash de la estructura de datos del Voxel ESDF.

con la implementación de la tabla de hash permite que todas las operaciones de busqueda, inserción, actualización y eliminación de datos dentro de la tabla dependan únicamente de la cantidad de datos involucrados, por más que estos solo representen un pequeño porcentaje de la estructura del dato original. Por otro lado, la disposición en memoria SoA no garantiza que los datos se encuentren consecutivos en memoria cuando se realice una busqueda por rango. Esto depende de la función de hash utilizada.

4.3.2. Particionamiento espacial del entorno

En el sistema coVoxSLAM el mapa se representa mediante tres niveles de particionamiento. El primero es una lista de submapas superpuestos, donde cada submapa tiene una tabla hash de tamaño dinámico para los índices de los bloques, y dentro de cada bloque hay un arreglo tridimensional de vóxeles de tamaño fijo. Tanto las colecciones de submapas como de bloques son dispersas, lo que proporciona flexibilidad al sistema para representar únicamente las secciones del entorno de las cuales ha recopilado datos de sus sensores. Esta jerarquía de dos niveles, que utiliza una tabla hash de tamaño dinámico, hace uso del enfoque de hashing de vóxeles introducido por [12].

Los submapas y bloques se indexan por su posición 3D en el mapa. Para el caso de los submapas es la posición global y para el caso de los bloques, esta es relativa a la posición del submapa al cual pertenecen. El índice devuelto por la tabla de hash se utiliza para acceder a los datos en

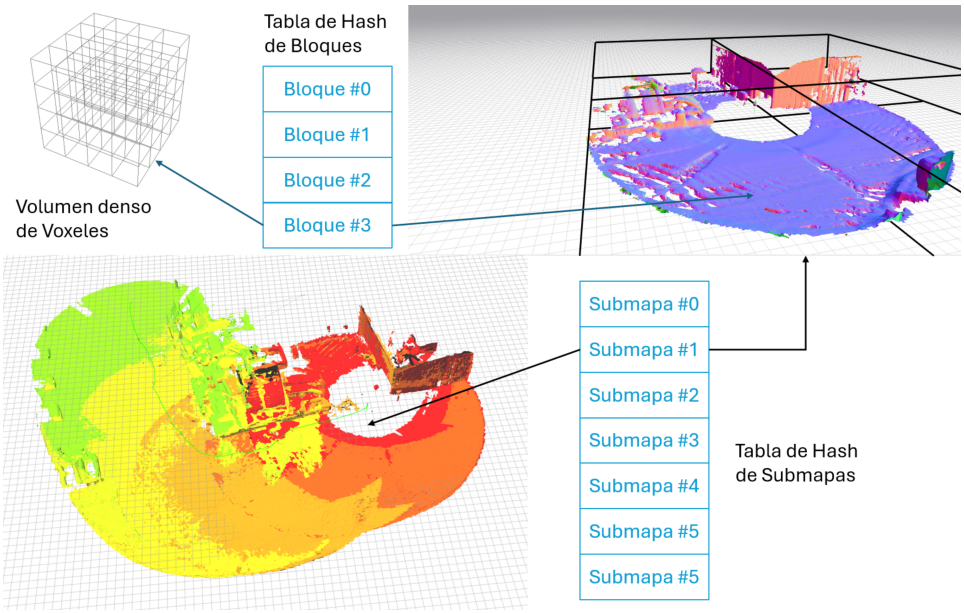

Fig. 4.6: Visualización de la tabla de hash de coVoxSLAM de tres niveles. Se divide el submapa en unicamente 4 bloques para simplificar la visualización.

memoria, por lo que no es necesario realizar una doble indirección buscando el puntero en una tabla hash secundaria. Los datos de los vóxeles, como el peso o la distancia, o cualquier otro dato declarado en una estructura definida por el usuario, se empaquetan densamente en memoria utilizando el contenedor SoA, lo que maximiza el número de accesos fusionados en la memoria de la GPU y también facilita la sincronización de las escrituras en memoria no coherentes.

La representación del particionamiento espacial del entorno en coVoxSLAM puede visualizarse en la Fig. 4.6. Dos tablas de hash se utilizan para indexar tanto los submapas como los bloques. Los valores de cada miembro de la estructura de los bloques y voxeles se encuentran continuos dentro del mismo bufer de memoria. Cada submapa asigna un búffer de memoria distinto para la tabla de hash de bloques, de manera de permitir un tamaño dinámico sin necesidad de reasignar los valores de los bloques del resto de los submapas cada vez que se elimina o se crea uno nuevo.

4.3.3. Ponderación y fusión de submapas

El ponderado ayuda a minimizar el impacto del ruido del sensor y las inexactitudes en las mediciones de profundidad. Al asignar pesos más altos a las mediciones más confiables, la calidad general del mapa TSDF mejora. Por ejemplo, un sensor LiDAR de larga distancia podría tener una precisión de $\pm 10\,\%$ a corta distancia, mientras que un sensor de corta distancia podría

comenzar a perder precisión pasados los 6 o 10 metros, comenzando a degradarse a medida que aumenta la distancia. El ponderado ayuda a aumentar la confiabilidad del mapa al asignar un nivel de confianza apropiado para cada medición tomando en cuenta diferentes pesos dependiendo de la curva de rendimiento del sensor.

Como se describe en [71], la elección de la función de ponderación puede impactar fuertemente la precisión de la reconstrucción resultante, especialmente para vóxeles grandes o superficies cercanas al sensor, donde miles de puntos pueden fusionarse en el mismo vóxel, así como para vóxeles distantes donde la calidad del sensor puede verse degradada.

Sin embargo, no todas las representaciones de mapas y sus métodos de construcción permiten el uso de diferentes técnicas de ponderación. Por ejemplo, nvBlox utiliza una técnica de interpolación durante la ponderación para aproximar el nivel de confianza de los datos de cada vóxel, lo que resulta en una pérdida de precisión.

Casteo de rayos vs. mapeo de proyecciones

El problema al integrar una nube de puntos en un mapa volumétrico de vóxeles ocurre cuando se repiten puntos dentro de un mismo voxel. Para solucionar esto es necesario por un lado identificar los repetidos y por otro lograr fusionar sus valores dentro del voxel en el que se encuentran.

En cuanto a la manera en la que se fusionan los datos del sensor en el mapa TSDF, existen dos métodos principales: casteo de rayos (*raycasting*) [71] y mapeo de proyecciones (*projection mapping*) [17, 91]. El método de casteo de rayos proyecta rayos desde el centro óptico de la cámara hacia cada punto observado y actualiza todos los vóxeles que intersectan con este rayo. La principal desventaja de esta técnica es que resulta desafiante de implementar en GPUs. El mapeo de proyecciones, en cambio, proyecta los vóxeles en el campo visual en la imagen de profundidad y calcula su distancia a partir de la diferencia entre el centro del vóxel y el valor de profundidad en la imagen. Esto puede llevar a fuertes efectos de aliasing para vóxeles de mayores tamaños más grandes. Además, puede devenir en reconstrucciones incompletas por pérdida de detalles.

Proyectar vóxeles puede no capturar con precisión las variaciones de profundidad en superficies con pendientes pronunciadas y puede descuidar grandes cantidades de datos de entrada, especialmente cuando se trabaja con tamaños grandes de vóxel. Por otro lado, en cuanto a las operaciónes necesarias, proyectar cada vóxel en la imagen de la cámara y asociarlo con el píxel más cercano puede ser costoso en términos computacionales, especialmente cuando se trabaja con grillas de resolución alta. Además, los métodos de proyección pueden ser sensibles al ruido del sensor, lo que puede introducir errores en el mapa TSDF [91, 92].

Para superar estas limitaciones, y a diferencia de nvBlox [63], el sistema

de coVoxSLAM no utiliza el método de mapeo de proyecciones, sino que implementa una versión en GPU de casteo de rayos, que considera cada punto del sensor al evaluar un vóxel, facilitando la integración de diferentes técnicas de ponderación sin perder precisión.

4.3.4. Generación de mapas TSDF

A medida que el robot explora el entorno, el módulo de integración de nube de puntos envia una grilla de mediciones a una frecuencia determinada por el hardware del sensor, junto con una estimación de la posición en tiempo real de la cámara. Estos datos pueden ser procesados de manera incremental a medida que se generan, o sea, en la finalización de cada ciclo de medición del sensor, o también pueden ser almacenados por lotes para ser integrados todos juntos luego de un intervalo de tiempo. Dependiendo de la arquitectura del sistema, una u otra alternativa podria resultar mas conveniente.

El sensor LiDAR genera continuamente nubes de puntos que deben ser convertidos e integrados en tiempo real para minimizar la pérdida de mediciones y asegurar el funcionamiento fluido del sistema. Se generan nuevos submapas a intervalos regulares, basados en el tiempo transcurrido o la distancia recorrida, evitando que los errores de odometría se acumulen en exceso. El área cubierta por la trayectoria del sensor se aproxima considerando el número de bloques utilizados por el submapa actual. Esta aproximación es computacionalmente económica y maximiza la consistencia en el número de bloques para cada submapa.

El sistema coVoxSLAM emplea un enfoque de múltiples pasos para mejorar el rendimiento y maximizar el paralelismo en algoritmos de casteo de rayos. Todas las sincronizaciones necesarias para el algoritmo se realizan entre las etapas, por lo que la mayor prioridad es mantener a la GPU con un nivel de ocupación alto durante la mayoria del tiempo que dure la ejecucion de cada paso. A continuación detallamos cada una de estas etapas.

Conteo preliminar

Durante la etapa de conteo preliminar, el algoritmo calcula los nuevos vóxeles y los bloques que contendrán estos vóxeles. Este paso es fácilmente paralelizable porque cada hilo puede procesar de manera independiente diferentes segmentos de los datos del sensor para determinar el número requerido de nuevos vóxeles y bloques. Además, se crean dos listas de vóxeles nuevos y únicos durante este paso. Estas listas se mantienen en búferes pre asignados y la inserción es una operación atómica para minimizar el impacto de la sincronización que se necesita al insertar en las listas.

El conteo generalmente se desaconseja en contextos de GPU porque puede llevar a problemas de contención. En lugar de depender de un único contador global, nuestro método utiliza técnicas de reducción donde cada hilo

o bloque mantiene su contador local, y los resultados se combinan únicamente al final. Este enfoque reduce la contención y mejora la escalabilidad. Además, colocar contadores en memoria compartida, que es más rápida que la memoria global, puede mejorar el rendimiento, especialmente cuando los hilos acceden frecuentemente al contador dentro del mismo bloque.

Asignación de memoria

Durante la etapa de asignación de memoria, un búfer pre-asignado maneja los nuevos vóxeles y bloques. Los hilos solicitan porciones de este búfer e incrementan sus índices utilizando operaciones atómicas. Este método permite a los hilos asignar memoria de manera segura y eficiente sin depender de mecanismos de bloqueo tradicionales. Los búferes asignados no son usados durante esta etapa, por lo que no es necesario calcular la posición definitiva de cada nuevo bloque o voxel. Esto permite que puedan calcularse los requerimientos de manera incremental y en paralelo, similar a la etapa anterior, y calcular al final de la etapa las asignaciones finales para cada nuevo voxel o bloque.

Actualización de vóxeles

Durante la etapa de actualización, el sistema actualiza los valores del mapa TSDF para los vóxeles únicos. Dado que los cálculos de peso y distancia son conmutativos y distributivos es que se puede utilizar el método de reducción segmentada (*segmented reduce* en la Fig. 4.1) para distribuir la carga de trabajo entre los hilos de la GPU y fusionar los resultados intermedios de manera incremental. Además, el algoritmo ordena los vóxeles por sus direcciones de memoria para prevenir condiciones de carrera al escribir los resultados finales en la cuadrícula. Esto se puede ver con más detalle en el frontend de la Fig. 4.1. Este ordenamiento asegura escrituras de memoria ordenadas y minimiza la necesidad de sincronización en la fase subsiguiente.

Procesamiento de Bloques

En la última etapa, los valores finales se escriben en la memoria de la GPU; cada warp en la GPU utiliza un rango de direcciones predefinido para evitar condiciones de carrera. Para lograr esto la memoria se ordena utilizando los índices dentro del bloque principal. Esto implica que por cada vóxel se realizan dos escrituras, una para almacenar el valor en un arreglo desordenado y otra para guardarlo definitivamente en memoria global. A pesar de tener el doble que escrituras que existirían si se hiciese directamente en memoria global, la mejora en los tiempo de ejecución se alcanza máximizando el rendimiento en paralelo libre de sincronizaciones.

Una observación importante es que los algoritmos de construcción de mapas SDF en los sistemas nvBlox, Voxgraph y Voxblox, utilizan tablas de

hash para representar y almacenar el mapa en memoria de manera eficiente, sin embargo sigue siendo necesario que el mapa completo se encuentre alojado en memoria para el correcto funcionamiento, lo cual acota los límites de crecimiento del mapa global. Por el contrario, en coVoxSLAM utilizamos la GPU como un mapa global parcial que se encuentra pre-alojado en memoria y que denota la "memoria a corto plazo" del sistema. Es posible transferir bloques del mapa a una memoria secundaria para fusionarlos al mapa global total. Al momento de iniciar el recorrido, el sistema carga los bloques de vóxeles que existan en un vecindario acotado, luego, al recorrer y alejarse del punto de origen, descarta los bloques que superan dicha cota y carga nuevos bloques en caso de que existan.

4.3.5. Mejoras en el casteo de rayos

En coVoxSLAM, cada hilo de la GPU maneja una cantidad similar de trabajo, enfocándose solo en los vóxeles intersectados por un rayo del sensor. El sistema maximiza la eficiencia de la GPU al igualar la carga de trabajo, manteniendo más hilos activos y utilizándolos de manera efectiva. Este equilibrio asegura que ningún hilo esté sobrecargado mientras otros están inactivos. Esto resulta en un mejor rendimiento general y uso de recursos, ya que la GPU puede procesar más datos en paralelo sin retrasos causados por una distribución desigual del trabajo.

Por su parte, el mapeo de proyección desconoce si todos los vóxeles fueron realmente observados por el sensor, dado que aproxima a la superficie utilizando bloques de vóxeles. Está técnica necesita proyectar cada bloque para asegurarse si intersecta o no el plano del sensor. A medida que aumenta la resolución del mapa, proyectar los vóxeles de una cuadrícula volumétrica en el sensor se vuelve menos eficiente. Esto se puede observar en la Fig.6 del artículo que presenta nvBlox [63], donde los tiempos de ejecución en la comparación con Voxblox disminuyen a medida que aumenta la resolución. La razón es que los bloques (llamados $VoxelBlocks$ en nvBlox) no son lo suficientemente pequeños como para ajustarse solo al área intersectada por los rayos del sensor, y en su lugar, forman volúmenes similares a prismas rectangulares que se forman en el área más cercana al punto de origen de los rayos. Esta área crece en tamaño a medida que la resolución disminuye, pero en términos de cantidad de bloques se mantiene relativamente constante. Esto produce que, dentro de estos bloques, la cantidad de vóxeles intersectados por un rayo disminuya a medida que aumenta la resolución. Usualmente, por cuestión de calidad en la medición del sensor, es justamente esta área cercana al sensor la que involucra los vóxeles y bloques que participan en la reconstruccción del mapa.

Por otro lado, el caso más extremo cuando los bloques son lo suficientemente pequeños como para que cada bloque intersecte solo un rayo se da cuando el tamaño de bloque es igual al tamaño del voxel, lo que produce

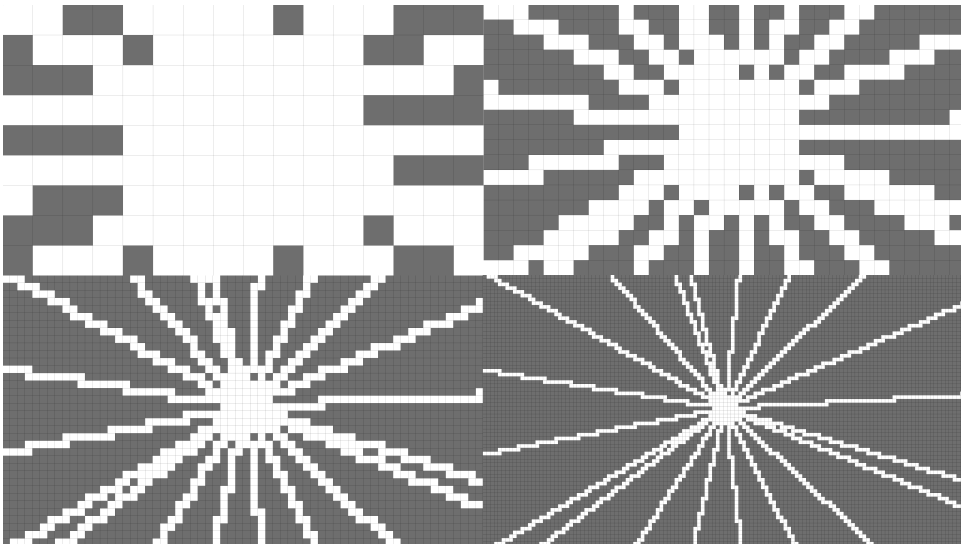

Fig. 4.7: Visualización de la intersección entre rayos y bloques en un ejemplo sintetico 2D.

que el mapeo de proyecciones opere como un casteo de rayos pero de manera suboptima debido al procesamiento extra que implica proyectar cada voxel.

En la Figura. 4.7 se representa la intersección de rayos en una grilla 2D con bloques de distinto tamaño. La grilla aumenta de 16×16, 32×32, 64×64 hasta 128×128 vóxeles de resolución. Como puede observarse, a medida que la resolución decrese, el bloque de vóxeles central aumenta de tamaño. Utilizando un mecanismo de pasos multiples se pueden evitar las sincronización necesarias del algoritmo clásico de casteo de rayos y mantener al mismo tiempo elevados índices de ocupación en la GPU durante la ejecución del método.

Adicionalmente, las operaciones requeridas para reproyectar los vóxeles de la cuadrícula de vuelta a la imagen del sensor son computacionalmente más costosas que las involucradas en el casteo de rayos. Esto enfatiza aún más la ineficiencia del mapeo por proyección en comparación con el casteo de rayos, que procesa directamente los vóxeles intersectados, lo que conduce a un mejor rendimiento y escalabilidad.

4.3.6. Generación de mapas ESDF

Los Mapas de Distancia con Signo Euclidianos (ESDFs) proporcionan mediciones precisas de distancia a los obstáculos, lo cual es esencial para navegar en entornos complejos. Se utilizan para tareas como la verificación de colisiones, donde el robot debe asegurarse de no chocar con obstáculos, y para inferir distancias y gradientes hacia objetos, lo que ayuda a planificar trayectorias suaves y eficientes.

La llegada de cámaras RGB-D, como el Kinect [18], ha popularizado el uso de TSDFs. Estas cámaras proporcionan información de profundidad que se puede utilizar para crear mapas 3D detallados. Esta técnica utiliza TSDFs para crear una representación de mapa rápida y flexible, calculando la posición de las superficies mediante cruces por cero, que son puntos donde el campo de distancia cambia de signo, indicando la presencia de una superficie.

Aunque ambas representaciones son campos de distancia con signo (SDFs), para cada voxel en el mapa ESDF, la distancia representa la distancia euclidiana al voxel ocupado más cercano. Esto significa que la distancia para los vóxeles libres es hacia el obstáculo más cercano, y para los vóxeles ocupados, es hacia el espacio libre más cercano. En contaste con los mapas TSDF donde la distancia desde el sensor a cada voxel se calcula únicamente a lo largo de la dirección del rayo.

Los mapas ESDF ayudan a que los algoritmos de planificación de trayectorias funcionen de manera más eficiente al proporcionar acceso rápido a la información de distancia y gradiente en relación con los obstáculos. Sin embargo, el desafío radica en mantener estos mapas actualizados en tiempo real en entornos dinámicos que cambian constantemente.

El cálculo de los mapas de Campo de Distancia con Signo Euclidiana (ESDF) comparte similitudes con los algoritmos de propagación de ondas, particularmente en cómo manejan los cálculos de distancia y las actualizaciones. Tanto los mapas ESDF como los algoritmos de propagación de ondas a menudo implican calcular distancias desde un conjunto de puntos (como obstáculos) hasta un punto objetivo. En el caso de ESDF, esto es crucial para determinar qué tan lejos está un punto del obstáculo más cercano, lo cual es esencial para tareas como la planificación de movimientos.

Los algoritmos de propagación de ondas típicamente utilizan un enfoque de frente de onda (wavefront) para actualizar las distancias. De manera similar, los cálculos de ESDF pueden emplear métodos como la Búsqueda en Amplitud (BFS) para propagar valores de distancia a través de una cuadrícula o espacio de vóxeles, asegurando que cada punto refleje la distancia más corta a un obstáculo.

En problemas de propagación de ondas, especialmente aquellos modelados por métodos de diferencias finitas o elementos finitos, el sistema de ecuaciones resultante a menudo conduce a matrices en banda. Estas matrices surgen de discretizar las ecuaciones de onda, donde solo un número limitado de puntos vecinos influye en cada punto de la cuadrícula.

Ambas técnicas pueden diseñarse para actualizar sus valores de manera incremental. Por ejemplo, cuando se detectan nuevos obstáculos, el ESDF puede actualizarse en tiempo real, al igual que los algoritmos de propagación de ondas ajustan sus cálculos en función de nueva información. Ambos enfoques buscan optimizar la eficiencia computacional, especialmente en aplicaciones en tiempo real. Por ejemplo, el ESDF puede calcularse a partir de

TSDFs utilizando técnicas de propagación de frente de onda, lo que permite actualizaciones eficientes y mediciones de distancia precisas.

Implementar estos algoritmos en GPU no es trivial, dado que este tipo de recursión requiere sincronización y de un determinado orden para actualizar el valor de los vóxeles de la grilla. Por ese motivo, en coVoxLSLAM el cálculo de ESDF se basa en el Algoritmo de Banda Paralela (PBA) [79]. El algoritmo PBA evita problemas de sincronización al dividir la grilla (o imagen para el caso del paper original) en bandas independientes que se pueden procesar de manera concurrente sin necesidad de comunicarse entre sí durante la fase de actualización de los valores. Es similar a la implementación presentada en nvBlox [72], excepto que dividimos la ejecución en regiones, y cada región se comunica utilizando colas de vóxeles y operaciones de solo lectura. Cada banda se trata como una tarea separada, permitiendo que los hilos de la GPU trabajen en paralelo. Este diseño minimiza la necesidad de sincronización.

Usar hilos para manejar cada voxel o bloque puede aumentar teóricamente el paralelismo, ya que pueden trabajar simultáneamente en diferentes partes del mapa, acelerando potencialmente los tiempos de procesamiento. Sin embargo, este enfoque puede llevar a un patrón completamente aleatorio de operaciones de escritura. Cada hilo podría intentar escribir en diferentes partes de la memoria simultáneamente, causando conflictos y condiciones de carrera. Las GPUs actuales están optimizadas para patrones ordenados de acceso a la memoria. Las operaciones de escritura aleatorias pueden degradar el rendimiento porque no se alinean bien con la arquitectura de la GPU, lo que lleva a problemas como la contención de memoria y un uso ineficiente de la caché. Los patrones de acceso a la memoria deben optimizarse para lograr un mejor rendimiento, lo que podría implicar organizar los datos para minimizar conflictos. En nuestra implementación, el procesamiento se divide en bloques, ya que son una matriz densa de n^3 vóxeles, típicamente $n = 8$ o $n = 16$. Cada warp en la GPU actualiza un bloque a la vez y actualiza una tabla hash global de valores de solo lectura dentro de los límites del bloque terminado.

Cuando las estimaciones de profundidad están sincronizadas, se genera una nube de puntos de obstáculos. Esta nube de puntos representa las posiciones 3D de los obstáculos en relación con el robot. Este método implica lanzar rayos desde el sensor hacia los puntos detectados en la nube de puntos, actualizando el estado de ocupación de los vóxeles a lo largo del camino del rayo. Después del casteo de rayos, el mapa se actualiza con todos los vóxeles recién observados (es decir, previamente desconocidos pero ahora detectados como ocupados o libres). Cada vez que un voxel pasa de ocupado a libre, el integrador ESDF necesita propagar esta información alrededor de sus vóxeles vecinos. Por el contrario, cuando un voxel pasa de libre a ocupado, el cálculo de actualizaciones se realiza alrededor de sus vóxeles vecinos para cambiar las distancias anteriores por posibles nuevos valores en los casos en que la distancia al voxel recién ocupado es menor.

Cuando un voxel actualiza su distancia, también debe propagar esta nueva información a sus vecinos. Este es el tercer caso en el que más vóxeles pueden necesitar ser encolados para su procesamiento. Subdividimos el mapa ESDF en regiones utilizando un bloque por región. Los bloques son matrices densas de n^3 vóxeles, típicamente $n = 8$ o $n = 16$. Cada warp en la GPU actualiza un bloque a la vez. Estas regiones se eligen por su índice, por lo que cada bloque en cada región es contiguo en memoria. Cada región utiliza su propia cola, y cada vez que un voxel recién ocupado o libre necesita encolar sus vóxeles vecinos, lo hace en la cola de la región a la que pertenecen.

De manera similar a nvBlox, procesamos cada dirección de eje dentro de cada bloque encolado en paralelo. Después de eso, se actualizan los límites de cada bloque vecino si registran un valor de una superficie más distante. Este proceso se repite hasta que no hay más vóxeles que cumplan con la condición mencionada.

4.4. Back-End

El sistema coVoxSLAM computa la aproximación por cuadrados mínimos no lineal para el grafo de poses cada vez que se finaliza la generación de un submapa nuevo. Este proceso es el primer paso del Back-end para ajustar el error acumulado en un modelo esparzo por definición, dado que los submapas creados son el resultado de la intersección entre la trayectoria del robot y la representación interna del mapa global.

Para resolver el problema de aproximación por cuadrados mínimos resultante, los sistemas SLAM utilizan usualmente la biblioteca Ceres Solver [38], la cual solo ofrece una versión acelerada en GPU para problemas de aproximación de sistemas lineales. Por este motivo, presentamos como parte de este trabajo una implementación del algoritmo de Levenberg-Marquardt para GPU basada en implementación en CPU de Ceres y optimizado para el problema de optimización del grafo de poses.

A pesar que la estructura principal del algoritmo en GPU esta inspirada en la implementación en CPU, varias consideraciones son necesarias para evitar penalizaciones en el rendimiento y la precisión final del método. A diferencia de otras implementaciones que buscan ejecutar en múltiples GPU, para este trabajo solo buscamos resolver la aproximación utilizando una única GPU, por lo que técnicas de subdivisión del grafo de poses no fueron consideradas.

A continuación se detallan las otras decisiones que se tomaron para la implementación del Back-end del sistema en GPU:

4.4.1. Implementación del Jacobiano

Es posible utilizar tanto los datos del mapa TSDF como los del mapa ESDF para la optimización del grafo ya que los nodos representan a los

submapas vistos como una pose con orientación y posición, ademas de su AABB (Axis aligned Bounding Box) y es indistinta el formato de sus datos.

El primer paso es sub-muestrear los puntos relevantes de los datos de cada nodo del grafo de poses. Esto se realiza tomando en cuenta el trabajo de Voxgraph [22] en donde se muestra que un subconjunto de los datos de los submapas es suficiente para lograr una óptima representación del mapa. Sin embargo, no se utiliza un subconjunto cualquiera sino que se pondera utilizando el peso de los vóxeles que fue calculado previamente por el Front-end.

El segundo paso es el cálculo de la matriz de Jacobianos y el vector de reciduales. El tamaño de esta matriz depende de la cantidad de datos sub-muestreados en cada submapa, esto además debe realizarse cada vez que se optimiza el grafo de poses. En coVoxSLAM, cada submapa tiene pre-asignado un buffer de tamaño máximo, que limitará la cantidad máxima de puntos que se puedan subsamplear. Sin embargo, este límite se puede configurar desde un parámetro global del sistema, dejando así la posibilidad de asignar recursos de acuerdo al problema particular sobre el que se este trabajando. Mantener un buffer pre-alojado permite que los valores de los Jacobianos y residuales se puedan calculan por cada hilo de la GPU en paralelo, sin necesidad de utilizar mecanismos de sincronización, maximizando de esta forma la utilización de la misma. El pseudo-código2 detalla el procedimiento de cálculo de error de registración utilizando la tecnica de subsampleo para cada par de submapas superpuestos.

Algorithm 2 Algoritmo de costo de registración entre dos submapas utilizando subsampleo ponderado.

Require: Conjunto de $N_{\mathcal{S}_i}$ puntos $\mathcal{U}_{\mathcal{S}_i}$ del Submapa $\mathcal{S}_i$,
con pesos $\omega_{\mathcal{S}_i}$
Submapa $\mathcal{S}_j$ con SDF $\Phi_{\mathcal{S}_j}$
Ratio de sampleo α
Transformada relativa entre Submapas $T_{S^j S^i}$

1: **function** ErrorDeRegistración
2: $e^{ij} \leftarrow 0$
3: **for** $m \leftarrow 1$ to $\alpha N_{\mathcal{S}_i}$ **do**
4: mover v^m desde $\mathcal{U}_{\mathcal{S}_i}$ con probabilidad $\propto \omega_{\mathcal{S}_i}(v^m)$
5: $e^{ij} \leftarrow e^{ij} + \Phi_{\mathcal{S}_j}(T_{S^j S^i}\ v^m)^2$
6: **end for**
7: $e^{ij} \leftarrow \frac{1}{\alpha} e^{ij}$
8: **return** e^{ij}
9: **end function**

4.4.2. Manejo de memoria reservada

Durante cada iteración, el algoritmo de LM aproxima la función no lineal alrededor de las estimaciones actuales de los parámetros mediante una expansión de Taylor de primer orden. Este proceso requiere calcular la matriz Jacobiana, que incluye las primeras derivadas de los residuos con respecto a los parámetros. En coVoxSLAM, tanto la matriz Jacobiana, los residuos, el gradiente derivado de los residuos y la matriz de condicionamiento se calculan por completo en la GPU. Esto requiere el uso de búferes para almacenar resultados hasta que el algoritmo converja a una solución. Dado el considerable tamaño de estos búferes, este trabajo emplea búferes preasignados temporalmente que pueden ser utilizados simultáneamente por el Front-end y el Back-end. Este enfoque ayuda a distribuir los costos de memoria entre múltiples procesos, optimizando el uso de recursos y evitando realizar sucesivas asignaciones de memoria en GPU que requeriría de sincronización entre los distintos kernels y de comunicación con la CPU.

4.4.3. Formato de las matrices

El formato de Fila Dispersa Comprimida por Bloques (BSR) es una variación del formato de Fila Dispersa Comprimida (CSR), diseñado específicamente para almacenar matrices dispersas que contienen sub-matrices densas. En lugar de almacenar elementos no nulos individuales, BSR almacena bloques de elementos. Cada bloque tiene un tamaño fijo, definido por las dimensiones de las filaes y columnas.Esto es particularmente útil para matrices donde muchos elementos no nulos están agrupados.

El formato BSR utiliza típicamente tres arreglos: el Arreglo de Datos, que contiene los bloques no nulos de la matriz; los índices de columna, que almacenan los índices de columnas para cada bloque; y un puntero que indica el índice de inicio de cada fila de bloque en el arreglo de datos.

Los beneficios del procesamiento por bloques durante el acondicionamiento también se aplican a operaciones generales. Basado en la estructura de bloques de la matriz Jacobiana, este formato no solo ahorra memoria, sino que también aumenta la velocidad si las operaciones de matriz se ejecutan por bloques. Este tipo de formato permite que J_p se convierta en una matriz diagonal por bloques, y tanto J_p como J_p^T comparten la misma representación BCSR, eliminando así la necesidad de almacenar J_p^T de forma separada.

4.4.4. Números duales

Los números duales son una extensión de los números reales análoga a los números complejos: mientras que los números complejos aumentan los reales al introducir una unidad imaginaria i tal que $i^2 = -1$, los números duales introducen una unidad infinitesimal ϵ tal que $\epsilon^2 = 0$. Un número dual $\alpha + v\epsilon$

tiene dos componentes: la componente real α y la componente infinitesimal v. Este cambio conduce a un método conveniente para calcular derivadas exactas sin necesidad de manipular expresiones simbólicas complicadas.

Esta extensión es la que se utiliza en la biblioteca Ceres para calcular la diferenciación automática. Sin embargo, la disposición en memoria de este tipo de datos utilizado por Ceres a través de la biblioteca JET no resulta la más conveniente para maximizar el rendimiendo de la caché de la GPU. Para este trabajo implementamos un conjunto de funcionalidades para la diferenciación automática que hace uso de la biblioteca de manejo de memoria SoA presentada previamente para convertir la disposición de memoria de la bilioteca JET de forma de poder realizar accesos a memoria fusionados, lo cual maximiza la utilización y disminuye los tiempos de carga de datos en cada kernel del Back-end.

4.4.5. Optimización de accesos a memoria

La caché de la memoria de la GPU no es coherente, lo que significa que de no mantener un orden en las escrituras y lecturas en memoria podríamos caer en una condición de carrera. Por otro lado, sincronizar lecturas y escrituras puede ser costoso al punto de desaprovechar la optimización que se obtiene al ejecutar en paralelo. Para contener estos problemas, los accesos a memoria, especialmente a las matrices de Jacobianos y a los vectores residuales se ordenan de forma tal de maximizar el uso de caché y evitar conflictos de escritura. Por ejemplo, el error de registración se calcula sobre cada par de submapas que se superpongan A, B y se mide el error de manera reciproca tanto desde A hacia B como de B hacia A. Para optimizar los accesos a memoria se ordena la lista de pares de submapas superpuestos primero por su primer componente y luego por el segundo. Por otro lado, cada hilo conoce el índice del elemento dentro de la matriz Jacobiana que actualizará, a sí mismo, cada warp de la GPU actualiza elementos continuos dentro del Jacobiano, de manera de evitar errores de propagación en los valores temporales de la caché de cada grupo de hilos.

En un sistema lineal representado como $\mathbf{Ax} = \mathbf{b}$, donde $\mathbf{A}$ es una matriz simétrica definida positiva, la matriz de condicionamiento (a menudo aproximada como la Hessiana) permanece constante a lo largo de las iteraciones. Esto se debe a que la matriz no cambia a medida que se actualiza la solución $\mathbf{x}$. En problemas de optimización no lineales, la función objetivo y su Hessiana asociada (o matriz de condicionamiento) son típicamente funciones de los parámetros que se están optimizando. A medida que se actualizan los parámetros en cada iteración, se recalculan el Jacobiano y, en consecuencia, la matriz de condicionamiento. Esta discrepancia no menor agrega el requerimiento de escritura sobre el búffer que contiene a la matriz del Hessiano, así como también un patrón de escritura que evite la necesidad de sincronización en arquitecturas GPU donde no existe la coherencia de caché. Para

lograr esto se utiliza una representación de la matriz por bloques, donde cada warp de la GPU actualiza individualmente cada bloque, dada la naturaleza de la operación, los cálculos son completamente independientes entre sí por lo que no se necesita de mecanismos de sincronización o comunicación entre warps.

4.4.6. Implementación en micro-kernels

Un kernel de GPU es una función o programa que se ejecuta en los núcleos de procesamiento. Este bloque básico de código se ejecuta en paralelo en cientos o miles de threads, aprovechando su arquitectura masivamente paralela. En general, dos razones principales reducen la ocupación en la GPU, por un lado se encuentra la insuficiencia de recursos y por otro lado la divergencia creciente en el flujo de ejecución de los kernels. Para minimizar el impacto de ambas en el rendimiento del módulo del Back-end, es necesario abordar el código de los kernels del minimizador de cuadrados mínimos de forma conjunta.

En este trabajo se siguió el enfoque conocido como *micro-kernels*, que son pequeños módulos de ejecución en GPU que resuelven de manera independiente cada uno de los subpasos del algoritmo LM, así como también la inicialización de las matrices y vectores. Este enfoque permite por un lado minimizar la cantidad de registros y memoria necesarios para cada ejecución, logrando a su vez minimizar o evitar una cantidad de hilos de GPU inactivos por falta de recursos al lanzamiento de cada kernel. A su vez, este enforque permite homogeneizar el flujo de ejecución, limitando la cantidad de saltos condicionales por medio de la resolución estática de saltos condicionales, esto es, dividir grandes porciones de código que ejecutan de manera disjunta en kernels separados.

Los micro-kernels también aumentan la posibilidad de optimizaciones por parte del compilador, como es el desenrrollado de ciclos, que aumenta el rendimiento de caché pero que solo es posible cuando existen registros disponibles para hacerlo. A su vez evitan otras técnicas que impactan negativamente en el rendimiento como es el derramado de registros (*register spilling*) que ocurre cuando el compilador del kernel no tiene suficiente cantidad de registros y por consecuente recae en la utilización de memoria de caché L1 o incluso de memoria global para alojar las variables que necesita.

5. RESULTADOS

En este capítulo presentamos los experimentos realizados para evaluar el sistema coVoxSLAM. Para esto, utilizamos diferentes datasets de dominio público: por un lado, el que corresponde a 4 vuelos de 400m de longitud aproximadamente, realizados con un microvehículo aéreo de tipo hexacóptero en una zona exterior de grandes dimensiones que se utiliza en pruebas de búsqueda y rescate de personas. Este dataset se utilizó para evaluar el sistema Voxgraph[1]. El microvehículo aéreo está equipado con un LiDAR Ouster OS1 de 64 canales, una cámara monocular, una IMU sincronizada y un RTK-GNSS (utilizado únicamente como datos de referencia para la evaluación de la trayectoria). Por otro lado, utilizamos dos datasets sintéticos de ambientes interiores, del tamaño de una habitación que fueron utilizados para evaluar el sistema nvBlox: Replica [93] y Redwood [94].

Las unidades de procesamiento utilizadas para realizar los experimentos fueron una PC de escritorio estándar (CPU con un procesador AMD Ryzen 9 5950× y una GPU Nvidia GeForce RTX 2060) y la placa Nvidia Jetson Xavier AGX.

5.1. Generación de mapas TSDF

El primer experimento consistió en comparar coVoxSLAM con Voxblox para el módulo de integración TSDF. La Figura 5.1 muestra la aceleración alcanzada para varias nubes de puntos. Los vuelos 1 a 4 corresponden a cuatro escenarios de gran escala, donde se obtuvo una mejora de entre 30× y 140× con un promedio de 100×. En lo dos últimos datasets, Replica y Redwood, que corresponden a pequeños ambientes interiores; se obtuvo una una aceleración de 50× en promedio. Esto representa un incremento de 2× y 1,5× respectivamente frente a nvBlox, que informa 38× y 25× frente a Voxblox y para estos dos últimos datasets, respectivamente. Esto prueba que el uso de la técnica de proyección de rayos (*raycasting*) para integrar TSDF supera el mapeo de proyección utilizado en nvBlox. Esto se hace considerando la generación del mapa TSDF + Color.

En la Figura 5.2 muestra cómo el método escala al incrementar el número de vóxeles. Se utilizó el dataset correspondiente al Vuelo 1. Nuestro objetivo es mostrar que el método de raycasting escala pseudo-linealmente a medida que aumenta el número de vóxeles.

[1] Disponible en: http://robotics.ethz.ch/~asl-datasets/2020_voxgraph_arche

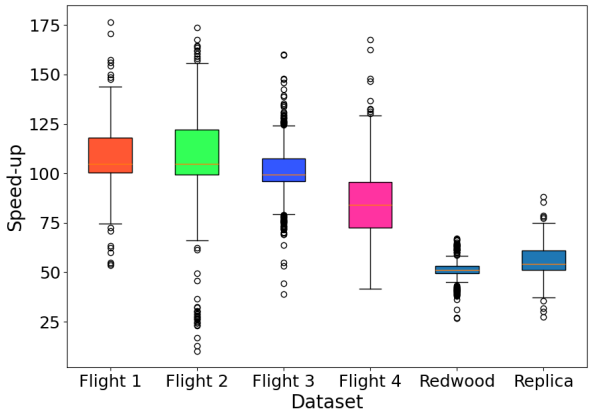

Fig. 5.1: Aceleración de los tiempos de ejecución de la generación del mapa TSDF en seis conjuntos de datos diferentes.

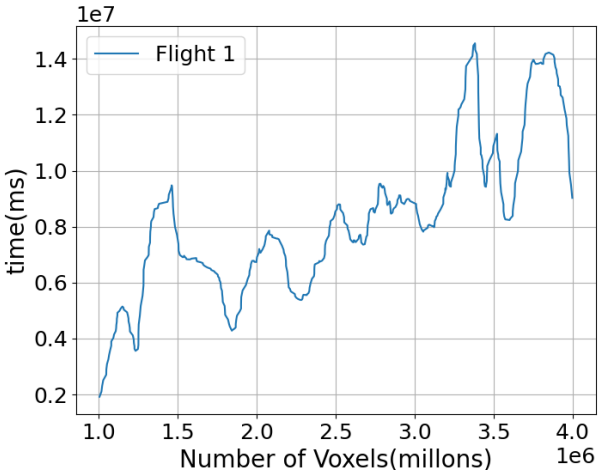

Fig. 5.2: Tiempo de ejecución del dataset correspondiente al vuelo 1, aumentando el número de vóxeles de 1 a 4 millones.

5.2. Generación de mapas ESDF

Evaluamos coVoxSLAM frente a Voxgraph para el módulo de integración ESDF. La Figura 5.3 muestra la aceleración obtenida en varias nubes de puntos integradas en 4 datasets. Se puede observar una mejora de entre 10× y 50×.

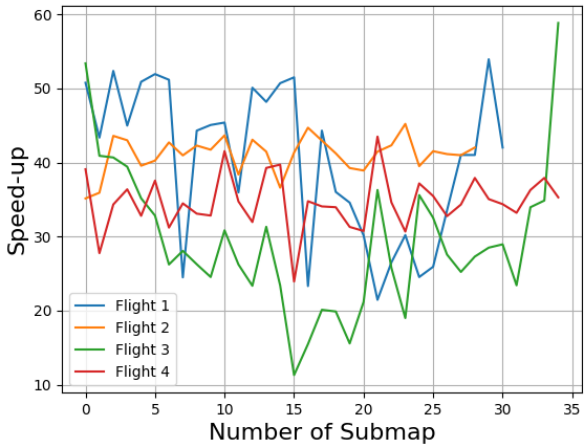

Fig. 5.3: Aceleración de los tiempos de ejecución de la generación del mapa ESDF en 4 datasets diferentes.

5.3. Back-end

Evaluamos el back-end de coVoxSLAM contra Voxgraph, que utiliza Ceres en la CPU para computar la optimización del grafo de poses. La Figura 5.4 muestra la aceleración entre las implementaciones de coVoxSLAM y Voxgraph para varias ejecuciones en 4 datasets. Como puede observarse se registró una mejora de entre 2× y 14×.

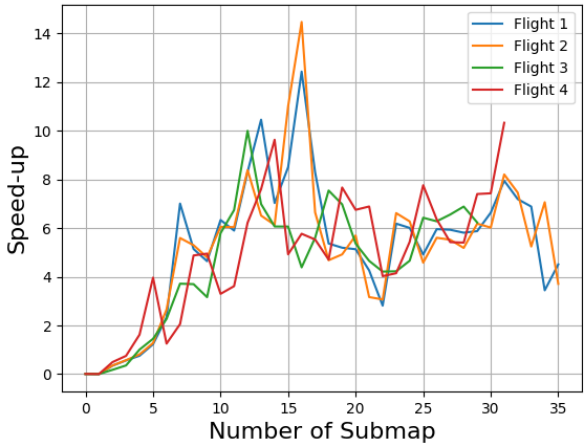

Fig. 5.4: Aceleración de los tiempos de ejecución del Back-end entre coVoxSLAM y Voxgraph en 4 datasets diferentes.

5.4. Prueba de estrés de la tabla hash

Utilizamos un conjunto de datos sintéticos para realizar pruebas de estrés a la tabla de hash de forma aislada. Nuestro objetivo es demostrar la solidez de su rendimiento bajo diferentes factores de carga. Como se muestra en la Figura 5.5, funciona sin inconvenientes cuando el factor de carga es inferior al 85 %, lo que es mucho más de lo que necesitamos, ya que el factor de carga en modo operativo de coVoxSLAM nunca supera el 50 % . La prueba realiza 50 millones de inserciones en la tabla hash y logra un tiempo constante de diez milisegundos para cada inserción.

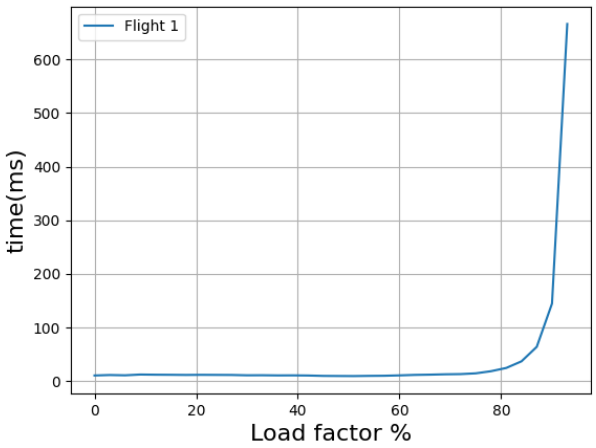

Fig. 5.5: Tiempo de ejecución de la Tabla Hash por factor de carga, al insertar 50 millones de claves.

5.5. Evaluación de la trayectoria

Se calculó el error cuadrático medio (RMSE) a partir de la trayectoria del robot y en comparación con los valores de referencia (*ground-truth*) de los datasets exteriores de gran escala. La Tabla 5.1 muestra errores similares para Voxgraph y coVoxSLAM ejecutándose en PC. También muestra la precisión del sistema ejecutándose en dispositivos integrados como Jetson Xavier AGX.

Version / Flight	Standard Desktop PC		Nvidia Jetson Xavier	
	Voxgraph	coVoxSLAM	Voxgraph	coVoxSLAM
1	0.93	**0.89**	—	**1.20**
2	**0.64**	0.92	—	**0.97**
3	**0.82**	1.14	—	**1.21**
4	**0.79**	1.02	—	**1.03**

Tab. 5.1: Error cuadrático medio (RMSE) (en metros) del error absoluto de trayectoria (ATE) para Voxgraph y coVoxSLAM en diferentes unidades de procesamiento.

Finalmente, en la Figura 5.6 se muestran las trayectorias estimadas del dataset correspondiente al Vuelo 1 por Voxgraph (azul), coVoxSLAM en PC (naranja), coVoxSLAM en Jetson Xavier AGX (verde) y las mediciones RTK-GNSS (rojo). Como puede observarse resultan todas muy similares, lo cual revela que el sistema coVoxSLAM mantiene una precisión similar a Voxgraph, a pesar de acelerar en varios órdenes de magnitud los tiempos de ejecución.

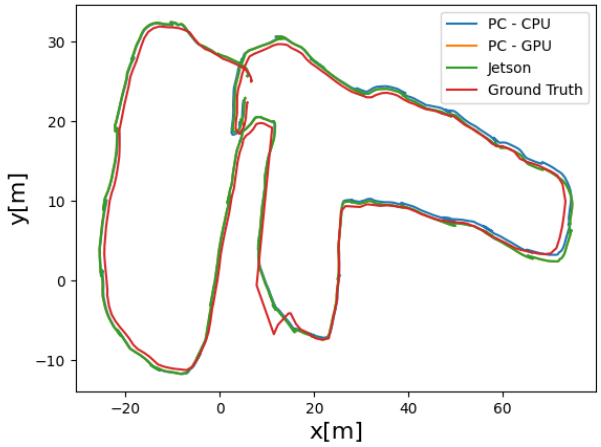

Fig. 5.6: Trayectorias estimadas del dataset correspondiente al Vuelo 1 por Voxgraph (azul), coVoxSLAM en PC (naranja), coVoxSLAM en Jetson Xavier AGX (verde) y las mediciones RTK-GNSS (rojo) utilizadas como valores de referencia para evaluar el sistema.

6. CONCLUSIONES

En este trabajo, presentamos un nuevo sistema de SLAM acelerado por GPU, acuñado coVoxSLAM, para construir mapas volumétricos globalmente consistentes en tiempo real. Tanto el Front-end como el Back-end están diseñados para ejecutarse completamente en la GPU maximizando la eficiencia y evitando transferencias de datos innecesarias y conflictivas entre la CPU y la GPU. Hasta donde sabemos es el primer sistema de SLAM basado en grafo de poses que está implementado íntegramente en GPU, tanto el Front-End como el Back-end, incluída la optimización del grafo de poses mediante el método de mínimos cuadrados siguiendo el algoritmo LM.

Los resultados obtenidos utilizando una GPU discreta y una GPU embebida muestran que el sistema coVoxSLAM mejora sustancialmente los tiempos de ejecución manteniendo una precisión similar a otros sistemas de SLAM del estado del arte. coVoxSLAM logra superar otros sistemas de SLAM acelerados por GPU de última generación, como la biblioteca nvBlox lanzada recientemente, logrando una mejora de entre $1,5\times$ a $2\times$. Nuestro algoritmo de fusión de rayos (*raycasting*) supera las alternativas de mapeo proyectivo actuales al tiempo que facilita la adopción de mejores opciones para técnicas de ponderación y fusión de mapas para el módulo integrador de TSDF.

Por otro lado, la adopción de una tabla de hash utilizando la disposición de datos SoA en memoria resulto de gran adaptabilidad a los distintos modulos del sistema. Permitiendo reducir la cantidad de datos transferidos entre distintas consultas. Tal como se ve en la Fig. 5.5 el rendimiento se mantiene muy estable hasta factores de carga de $85\,\%$ esto es significativamente superior al $50-60\,\%$ que suele ser el factor de carga durante los experimentos que realizamos. Aún así creemos que podrían mejorarse estos resultados analizando nuevas estrategias para la resolución de colisiones que se adapten mejor a datos espaciales.

Logramos demostrar que tal como se viene mencionando en distintos trabajos [14, 15, 81] la adopción de tablas de hash y disposición SoA resulta conveniente y eficiente para sistemas SLAM debido a la necesidad de cooperación subyacente entre sus módulos tan heterogeneos.

Finalmente, como una contribución de esta tesis, hemos publicado todo el código fuente del proyecto en un repositorio de acceso público de GitHub [1], con ejemplos e instrucciones para facilitar el uso por parte de la comunidad científica y asegurar la repetibilidad de los experimentos realizados.

[1] https://github.com/lrse-uba/coVoxSLAM

7. APÉNDICE

7.1. Transformaciones de cuerpos rígidos

Es esencial para cada aplicación de robótica móvil construir una noción del cuerpo del robot, su ubicación y modelar su movimiento a través del espacio. Sin tener en cuenta las restricciones sobre la capacidad de movimiento, un cuerpo rígido se puede trasladar y rotar libremente. La configuración geométrica se conoce como la pose (posición y orientación) del robot, la cual puede tener 3 grados de libertad en el caso de aplicaciones 2D (2 para traslación más 1 orientación en ángulo), o 6 grados de libertad en el caso 3D (3 para la traslación y 3 más para la orientación). Una plataforma robótica puede constar de múltiples cuerpos rígidos conectados entre sí a través de articulaciones, cada uno con su propia pose como puede observarse en la Figura 7.1. Por ello, resulta útil definir un marco de referencia local que servirá como sistema de coordenadas de referencia para el cual especificar todas sus partes. El movimiento a través del espacio del cuerpo rígido compuesto puede entonces explicarse por el movimiento de su marco de referencia y servirá de manera equivalente como modelo de la pose. Cada marco es siempre relativo a otro, definido por una convención de ejes, la posición de su origen de coordenadas y su orientación con respecto al marco externo, generalmente llamado marco inercial. El padre de todos los marcos se lo conoce en la literatura como marco global o marco del mundo, que normalmente se considera estático.

Las poses de un cuerpo rígido no viven en un espacio vectorial euclidiano o, de manera equivalente, no pueden ser representadas por un vector euclídeo, principalmente debido a la presencia de componentes angulares que no son euclidianos. El conjunto de movimientos rígidos y composiciones que pueden realizarse forman un grupo matemático conocido como Grupo Euclidiano Especial SE(n). Los elementos en SE(n) tienen un grado de libertad $n(n+1)/2$, donde n se atribuyen al subgrupo de simetría traslacional T(n), y el $n(n-1)/2$ restante a un subgrupo de simetría rotacional, el Grupo Ortogonal Especial SO(n). En aplicaciones de 3D SLAM las poses de robot pertenecen al grupo SE(3), el conjunto de poses y movimientos rígidos en 3D, y tienen seis grados de libertad: tres para las traslaciones (T(3)), y tres para las rotaciones (SO(3)). Tanto SO(n) y SE(n) forman grupos de Lie compactos, de dimensiones 3 y 6 respectivamente.

Un grupo de Lie es un manifold diferenciable, un espacio topológico que se asemeja localmente a un espacio euclidiano lineal. Cada grupo de Lie tiene un álgebra de Lie asociada, la cual se define como un espacio vectorial tangente centrado en el elemento identidad del grupo. La importancia del

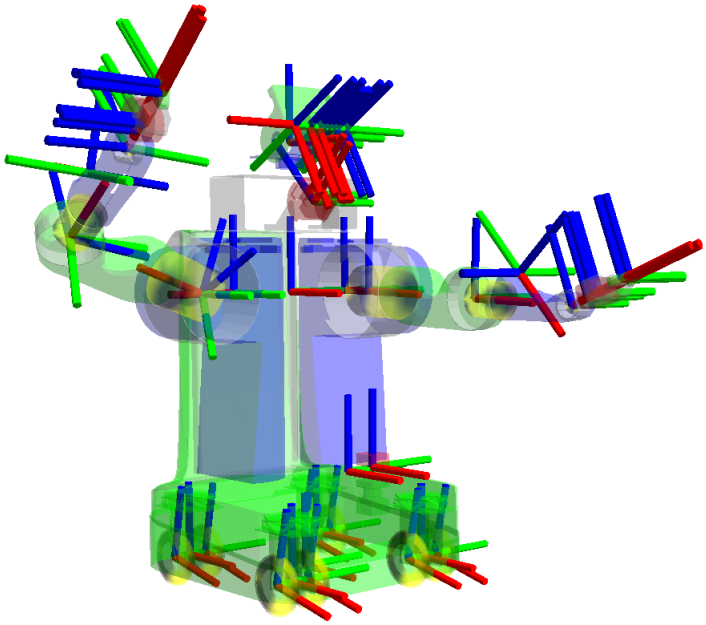

Fig. 7.1: Una vista del icónico robot RP2 de Willow Garage, modelado como un conjunto de cuerpos rígidos. Los ejes de los marcos de coordenadas se representan como X en rojo, Y en verde y Z en azul. Imagen perteneciente a la biblioteca de transformaciones [95].

álgebra de Lie radica en que proporciona un espacio vectorial en el que se pueden representar los elementos del grupo (las poses). Este espacio vectorial es siempre de representación mínima coincidiendo con los grados de libertad de los elementos del grupo, y permite aplicar métodos de cálculo lineal conservando las propiedades originales de los grupos.

Los elementos de movimiento rígido de SE(n) pueden representarse como

$$ {}^{w}_{r}\mathbf{T} = \begin{pmatrix} {}^{w}_{r}\mathbf{R} & {}^{w}_{r}\mathbf{t} \\ \mathbf{0} & 1 \end{pmatrix} \in SE(n), \tag{7.1} $$

donde ${}^{w}_{r}\mathbf{T}$ representa la pose del robot r (el marco de referencia del robot) con respecto al marco del mundo w, como una transformación donde ${}^{w}_{r}\mathbf{R} \in$ SO(n) es una matriz de rotación y ${}^{w}_{r}\mathbf{t} \in \mathbb{R}^n$ un vector traslación. Es posible definir dos operadores muy útiles sobre elementos SE(n), la composición $\oplus : SE(n) \mapsto SE(n)$ y su inversa $\ominus : SE(n) \mapsto SE(n)$,

$$ {}^{r_1}_{r_2}\mathbf{T} = {}^{w}_{r_2}\mathbf{T} \ominus {}^{w}_{r_1}\mathbf{T} \triangleq ({}^{w}_{r_1}\mathbf{T})^{-1} \cdot {}^{w}_{r_2}\mathbf{T} \tag{7.2} $$

$$ {}^{w}_{r_2}\mathbf{T} = {}^{w}_{r_1}\mathbf{T} \oplus {}^{r_1}_{r_2}\mathbf{T} \triangleq {}^{w}_{r_1}\mathbf{T} \cdot {}^{r_1}_{r_2}\mathbf{T} \tag{7.3} $$

,donde ${}^{r_1}_{r_2}\mathbf{T}$ describe el movimiento rígido relativo entre las poses de robot r_1 y r_2, y ambos operadores se resuelven por medio de multiplicaciones de

matrices simples.

Las álgebras de Lie para los grupos SO(n) y SE(n) se conocen como so(n) y se(n) respectivamente, y existen dos funciones de mapeo complementarias que relacionan elementos de ambos espacios. El mapa logarítmico Log:SE(n)$\mapsto$ se(n) toma elementos del grupo y los convierte al espacio tangente (tangent space), el cual utiliza estructuras no triviales (matrices simétricas sesgadas, skew-symmetric matrices), mientras que el mapa exponencial Exp:se(n)$\mapsto$ SE(n) es la operación contraria que convierte elementos del álgebra de Lie en elementos del grupo.

La teoría de Lie es extensa y sirve para varios propósitos en robótica móvil, no profundizaremos mucho más en el contexto de esta tesis pero mediante el uso del álgebra de Lie es posible calcular de manera eficiente las derivadas parciales de los operadores de composición (Eq. 7.3). Las matrices Jacobianas resultantes permiten realizar el manejo y propagación de incertidumbres en grupos de Lie, aplicando métodos análogos a los aplicados en espacios vectoriales, y por lo tanto eficientes. Existen varias bibliotecas de código abierto que implementan estos conceptos como *manif*[1] [96] o *Sophus*[2]. El reporte técnico de Joan Solà [97] centrado específicamente en la derivación de los Jacobianos es una introducción gentil al tema. Y la exhaustiva investigación de Timothy D. Barfoot [98, 99] sobre propagación y manejo de la incertidumbre en poses tridimensionales es altamente recomendable.

[1] https://github.com/artivis/manif
[2] https://github.com/strasdat/Sophus

Bibliografía

[1] C. Cadena, L. Carlone, H. Carrillo, Y. Latif, D. Scaramuzza, J. Neira, I. Reid, and J. J. Leonard, "Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age," *IEEE Transactions on robotics*, vol. 32, no. 6, pp. 1309–1332, 2016.

[2] R. Mur-Artal and J. D. Tardós, "Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras," *IEEE transactions on robotics*, vol. 33, no. 5, pp. 1255–1262, 2017.

[3] M. Labbé and F. Michaud, "Rtab-map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation," *Journal of field robotics*, vol. 36, no. 2, pp. 416–446, 2019.

[4] A. Cramariuc, L. Bernreiter, F. Tschopp, M. Fehr, V. Reijgwart, J. Nieto, R. Siegwart, and C. Cadena, "maplab 2.0–a modular and multimodal mapping framework," *IEEE Robotics and Automation Letters*, vol. 8, no. 2, pp. 520–527, 2022.

[5] R. A. Newcombe, S. J. Lovegrove, and A. J. Davison, "Dtam: Dense tracking and mapping in real-time," in *2011 international conference on computer vision*. IEEE, 2011, pp. 2320–2327.

[6] C. Kerl, J. Sturm, and D. Cremers, "Dense visual slam for rgb-d cameras," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2013, pp. 2100–2106.

[7] T. Whelan, S. Leutenegger, R. F. Salas-Moreno, B. Glocker, and A. J. Davison, "Elasticfusion: Dense slam without a pose graph." in *Robotics: science and systems*, vol. 11. Rome, Italy, 2015, p. 3.

[8] J. Czarnowski, T. Laidlow, R. Clark, and A. J. Davison, "Deepfactors: Real-time probabilistic dense monocular slam," *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 721–728, 2020.

[9] L. Koestler, N. Yang, N. Zeller, and D. Cremers, "Tandem: Tracking and dense mapping in real-time using deep multi-view stereo," in *Conference on Robot Learning*. PMLR, 2022, pp. 34–45.

[10] X. Yang, H. Li, H. Zhai, Y. Ming, Y. Liu, and G. Zhang, "Vox-fusion: Dense tracking and mapping with voxel-based neural implicit representation," in *2022 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*. IEEE, 2022, pp. 499–507.

[11] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "Octomap: An efficient probabilistic 3d mapping framework based on octrees," *Autonomous robots*, vol. 34, pp. 189–206, 2013.

[12] M. Nießner, M. Zollhöfer, S. Izadi, and M. Stamminger, "Real-time 3d reconstruction at scale using voxel hashing," *ACM Transactions on Graphics (ToG)*, vol. 32, no. 6, pp. 1–11, 2013.

[13] L. Buckley, J. Byrne, and D. Moloney, "Investigating the impact of suboptimal hashing functions," in *2018 IEEE Games, Entertainment, Media Conference (GEM)*. IEEE, 2018, pp. 324–331.

[14] D. Jünger, R. Kobus, A. Müller, C. Hundt, K. Xu, W. Liu, and B. Schmidt, "Warpcore: A library for fast hash tables on gpus," in *2020 IEEE 27th international conference on high performance computing, data, and analytics (HiPC)*. IEEE, 2020, pp. 11–20.

[15] W. Dong, Y. Lao, M. Kaess, and V. Koltun, "Ash: A modern framework for parallel spatial hashing in 3d perception," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, no. 5, pp. 5417–5435, 2022.

[16] B. Curless and M. Levoy, "A volumetric method for building complex models from range images," in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, 1996, pp. 303–312.

[17] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohi, J. Shotton, S. Hodges, and A. Fitzgibbon, "Kinectfusion: Real-time dense surface mapping and tracking," in *2011 10th IEEE international symposium on mixed and augmented reality*. Ieee, 2011, pp. 127–136.

[18] S. Izadi, D. Kim, O. Hilliges, D. Molyneaux, R. Newcombe, P. Kohli, J. Shotton, S. Hodges, D. Freeman, A. Davison, *et al.*, "Kinectfusion: real-time 3d reconstruction and interaction using a moving depth camera," in *Proceedings of the 24th annual ACM symposium on User interface software and technology*, 2011, pp. 559–568.

[19] R. Wagner, U. Frese, and B. Bäuml, "Graph slam with signed distance function maps on a humanoid robot," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2014, pp. 2691–2698.

[20] H. Oleynikova, Z. Taylor, R. Siegwart, and J. Nieto, "Safe local exploration for replanning in cluttered unknown environments for microaerial vehicles," *IEEE Robotics and Automation Letters*, vol. 3, no. 3, pp. 1474–1481, 2018.

[21] N. Fioraio, J. Taylor, A. Fitzgibbon, L. Di Stefano, and S. Izadi, "Large-scale and drift-free surface reconstruction using online subvolume registration," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 4475–4483.

[22] V. Reijgwart, A. Millane, H. Oleynikova, R. Siegwart, C. Cadena, and J. Nieto, "Voxgraph: Globally consistent, volumetric mapping using signed distance function submaps," *IEEE Robotics and Automation Letters*, vol. 5, no. 1, pp. 227–234, 2019.

[23] S. Aldegheri, N. Bombieri, D. D. Bloisi, and A. Farinelli, "Data flow orb-slam for real-time performance on embedded gpu boards," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2019, pp. 5370–5375.

[24] R. Giubilato, S. Chiodini, M. Pertile, and S. Debei, "An evaluation of ros-compatible stereo visual slam methods on a nvidia jetson tx2," *Measurement*, vol. 140, pp. 161–170, 2019.

[25] J. Song, J. Wang, L. Zhao, S. Huang, and G. Dissanayake, "Mis-slam: Real-time large-scale dense deformable slam system in minimal invasive surgery based on heterogeneous computing," *IEEE Robotics and Automation Letters*, vol. 3, no. 4, pp. 4068–4075, 2018.

[26] T. Schops, T. Sattler, and M. Pollefeys, "Bad slam: Bundle adjusted direct rgb-d slam," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 134–144.

[27] J. Behley and C. Stachniss, "Efficient surfel-based slam using 3d laser range data in urban environments." in *Robotics: Science and Systems*, vol. 2018, 2018, p. 59.

[28] K. Koide, M. Yokozuka, S. Oishi, and A. Banno, "Globally consistent 3d lidar mapping with gpu-accelerated gicp matching cost factors," *IEEE Robotics and Automation Letters*, vol. 6, no. 4, pp. 8591–8598, 2021.

[29] K. O. Arras, "An introduction to error propagation: derivation, meaning and examples of equation cy= fx cx fxt," ETH Zurich, Tech. Rep., 1998.

[30] S. Thrun, W. Burgard, and D. Fox, "Probalistic robotics," *Kybernetes*, vol. 35, no. 7/8, pp. 1299–1300, 2006.

[31] F. Dellaert, M. Kaess, *et al.*, "Factor graphs for robot perception," *Foundations and Trends® in Robotics*, vol. 6, no. 1-2, pp. 1–139, 2017.

[32] T. D. Barfoot, *State estimation for robotics*. Cambridge University Press, 2024.

[33] J. Nocedal and S. Wright, *Numerical Optimization*, ser. Springer Series in Operations Research and Financial Engineering. Springer New York, 2006. [Online]. Available: https://books.google.com.ar/books?id=VbHYoSyelFcC

[34] K. Levenberg, "A method for the solution of certain non-linear problems in least squares," *Quarterly of applied mathematics*, vol. 2, no. 2, pp. 164–168, 1944.

[35] D. W. Marquardt, "An algorithm for least-squares estimation of nonlinear parameters," *Journal of the society for Industrial and Applied Mathematics*, vol. 11, no. 2, pp. 431–441, 1963.

[36] M. J. Powell, "A hybrid method for nonlinear equations," *Numerical methods for nonlinear algebraic equations*, pp. 87–161, 1970.

[37] R. Kümmerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard, "g 2 o: A general framework for graph optimization," in *2011 IEEE international conference on robotics and automation*. IEEE, 2011, pp. 3607–3613.

[38] S. Agarwal, K. Mierle, and T. C. S. Team, "Ceres Solver," 10 2023. [Online]. Available: https://github.com/ceres-solver/ceres-solver

[39] F. Dellaert and G. Contributors, "borglab/gtsam," May 2022. [Online]. Available: https://github.com/borglab/gtsam)

[40] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast bvh construction on gpus," in *Computer Graphics Forum*, vol. 28, no. 2. Wiley Online Library, 2009, pp. 375–384.

[41] J. L. Bentley, "K-d trees for semidynamic point sets," in *Proceedings of the sixth annual symposium on Computational geometry*, 1990, pp. 187–197.

[42] R. B. Rusu and S. Cousins, "3d is here: Point cloud library (pcl)," in *2011 IEEE international conference on robotics and automation*. IEEE, 2011, pp. 1–4.

[43] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time kd-tree construction on graphics hardware," *ACM Transactions on Graphics (TOG)*, vol. 27, no. 5, pp. 1–11, 2008.

[44] I. Wald, S. Boulos, and P. Shirley, "Ray tracing deformable scenes using dynamic bounding volume hierarchies," *ACM Transactions on Graphics (TOG)*, vol. 26, no. 1, pp. 6–es, 2007.

[45] J. Gunther, S. Popov, H.-P. Seidel, and P. Slusallek, "Realtime ray tracing on gpu with bvh-based packet traversal," in *2007 IEEE Symposium on Interactive Ray Tracing*. IEEE, 2007, pp. 113–118.

[46] D. Meagher, "Geometric modeling using octree encoding," *Computer graphics and image processing*, vol. 19, no. 2, pp. 129–147, 1982.

[47] K. Museth, J. Lait, J. Johanson, J. Budsberg, R. Henderson, M. Alden, P. Cucka, D. Hill, and A. Pearce, "Openvdb: an open-source data structure and toolkit for high-resolution volumes," in *Acm siggraph 2013 courses*, 2013, pp. 1–1.

[48] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, "Real-time parallel hashing on the gpu," in *ACM SIGGRAPH asia 2009 papers*, 2009, pp. 1–9.

[49] I. García, S. Lefebvre, S. Hornus, and A. Lasram, "Coherent parallel hashing," *ACM Transactions on Graphics (TOG)*, vol. 30, no. 6, pp. 1–8, 2011.

[50] P. Stotko, "stdgpu: Efficient stl-like data structures on the gpu," *arXiv preprint arXiv:1908.05936*, 2019.

[51] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *2012 IEEE conference on computer vision and pattern recognition*. IEEE, 2012, pp. 3354–3361.

[52] B. Bodin, H. Wagstaff, S. Saecdi, L. Nardi, E. Vespa, J. Mawer, A. Nisbet, M. Luján, S. Furber, A. J. Davison, *et al.*, "Slambench2: Multi-objective head-to-head benchmarking for visual slam," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2018, pp. 3637–3644.

[53] R. Kümmerle, B. Steder, C. Dornhege, M. Ruhnke, G. Grisetti, C. Stachniss, and A. Kleiner, "On measuring the accuracy of slam algorithms," *Autonomous Robots*, vol. 27, pp. 387–407, 2009.

[54] R. O. Castle, D. J. Gawley, G. Klein, and D. W. Murray, "Towards simultaneous recognition, localization and mapping for hand-held and wearable cameras," in *Proceedings 2007 IEEE International Conference on Robotics and Automation*. IEEE, 2007, pp. 4102–4107.

[55] M. Kaess, "Simultaneous localization and mapping with infinite planes," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2015, pp. 4605–4611.

[56] J. C. Carr, R. K. Beatson, J. B. Cherrie, T. J. Mitchell, W. R. Fright, B. C. McCallum, and T. R. Evans, "Reconstruction and representation

of 3d objects with radial basis functions," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, 2001, pp. 67–76.

[57] C. Zach, T. Pock, and H. Bischof, "A globally optimal algorithm for robust tv-l 1 range image integration," in *2007 IEEE 11th International Conference on Computer Vision*. IEEE, 2007, pp. 1–8.

[58] J. D. Foley, *Computer graphics: principles and practice*. Addison-Wesley Professional, 1996, vol. 12110.

[59] N. Fairfield, G. Kantor, and D. Wettergreen, "Real-time slam with octree evidence grids for exploration in underwater tunnels," *Journal of Field Robotics*, vol. 24, no. 1-2, pp. 03–21, 2007.

[60] J. Engel, T. Schöps, and D. Cremers, "Lsd-slam: Large-scale direct monocular slam," in *European conference on computer vision*. Springer, 2014, pp. 834–849.

[61] T. Whelan, M. Kaess, M. Fallon, H. Johannsson, J. Leonard, and J. McDonald, "Kintinuous: Spatially extended kinectfusion," 2012.

[62] A. Rosinol, M. Abate, Y. Chang, and L. Carlone, "Kimera: an open-source library for real-time metric-semantic localization and mapping," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2020, pp. 1689–1696.

[63] A. Millane, H. Oleynikova, E. Wirbel, R. Steiner, V. Ramasamy, D. Tingdahl, and R. Siegwart, "nvblox: Gpu-accelerated incremental signed distance field mapping," *arXiv preprint arXiv:2311.00626*, 2023.

[64] W. C. Thibault and B. F. Naylor, "Set operations on polyhedra using binary space partitioning trees," in *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, 1987, pp. 153–162.

[65] A. Fahim, M. E. Ali, and M. A. Cheema, "Unsupervised space partitioning for nearest neighbor search," *arXiv preprint arXiv:2206.08091*, 2022.

[66] H. S. Chhabra, A. K. Srivastava, and R. Nijhawan, "A hybrid deep learning approach for automatic fish classification," in *Proceedings of ICETIT 2019: Emerging Trends in Information Technology*. Springer, 2020, pp. 427–436.

[67] T. Müller, A. Evans, C. Schied, and A. Keller, "Instant neural graphics primitives with a multiresolution hash encoding," *ACM transactions on graphics (TOG)*, vol. 41, no. 4, pp. 1–15, 2022.

[68] H. Moravec and A. Elfes, "High resolution maps from wide angle sonar," in *Proceedings. 1985 IEEE international conference on robotics and automation*, vol. 2. IEEE, 1985, pp. 116–121.

[69] J. J. Park, P. Florence, J. Straub, R. Newcombe, and S. Lovegrove, "Deepsdf: Learning continuous signed distance functions for shape representation," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 165–174.

[70] X. Zheng, Y. Liu, P. Wang, and X. Tong, "Sdf-stylegan: Implicit sdf-based stylegan for 3d shape generation," in *Computer Graphics Forum*, vol. 41, no. 5. Wiley Online Library, 2022, pp. 52–63.

[71] H. Oleynikova, Z. Taylor, M. Fehr, R. Siegwart, and J. Nieto, "Voxblox: Incremental 3d euclidean signed distance fields for on-board mav planning," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017, pp. 1366–1373.

[72] A. Millane, H. Oleynikova, E. Wirbel, R. Steiner, V. Ramasamy, D. Tingdahl, and R. Siegwart, "nvblox: Gpu-accelerated incremental signed distance field mapping," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2024, pp. 2698–2705.

[73] A. Millane, Z. Taylor, H. Oleynikova, J. Nieto, R. Siegwart, and C. Cadena, "C-blox: A scalable and consistent tsdf-based dense mapping approach," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2018, pp. 995–1002.

[74] H. R. Kang, *Computational color technology*. Spie Press Bellingham, WA, 2006.

[75] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear, *The Art of Multiprocessor Programming*. Elsevier Science, 2020. [Online]. Available: https://books.google.com.ar/books?id=7MqcBAAAQBAJ

[76] L. Nardi, B. Bodin, M. Z. Zia, J. Mawer, A. Nisbet, P. H. Kelly, A. J. Davison, M. Luján, M. F. O'Boyle, G. Riley, *et al.*, "Introducing slambench, a performance and accuracy benchmarking methodology for slam," in *2015 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2015, pp. 5783–5790.

[77] W. Xu and F. Zhang, "Fast-lio: A fast, robust lidar-inertial odometry package by tightly-coupled iterated kalman filter," *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 3317–3324, 2021.

[78] J. Amanatides, A. Woo, *et al.*, "A fast voxel traversal algorithm for ray tracing." in *Eurographics*, vol. 87, no. 3. Citeseer, 1987, pp. 3–10.

[79] T.-T. Cao, K. Tang, A. Mohamed, and T.-S. Tan, "Parallel banding algorithm to compute exact distance transform with the gpu," in *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, 2010, pp. 83–90.

[80] A. Kumar, J. Park, and L. Behera, "High-speed stereo visual slam for low-powered computing devices," *IEEE Robotics and Automation Letters*, 2023.

[81] H. Homann and F. Laenen, "Soax: A generic c++ structure of arrays for handling particles in hpc codes," *Computer Physics Communications*, vol. 224, pp. 325–332, 2018.

[82] B. M. Gruber, G. Amadio, J. Blomer, A. Matthes, R. Widera, and M. Bussmann, "Llama: The low-level abstraction for memory access," *Software: Practice and Experience*, vol. 53, no. 1, pp. 115–141, 2023.

[83] R. Strzodka, "Abstraction for aos and soa layout in c++," in *GPU computing gems Jade edition*. Elsevier, 2012, pp. 429–441.

[84] D. Weber, J. Bender, M. Schnoes, A. Stork, and D. Fellner, "Efficient gpu data structures and methods to solve sparse linear systems in dynamics applications," in *Computer graphics forum*, vol. 32, no. 1. Wiley Online Library, 2013, pp. 16–26.

[85] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August, "Dynamically managed data for cpu-gpu architectures," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, 2012, pp. 165–174.

[86] A. E. Lefohn, S. Sengupta, J. Kniss, R. Strzodka, and J. D. Owens, "Glift: Generic, efficient, random-access gpu data structures," *ACM Transactions on Graphics (TOG)*, vol. 25, no. 1, pp. 60–99, 2006.

[87] A. A. Stepanov and P. McJones, *Elements of programming*. Addison-Wesley Professional, 2009.

[88] S. Jubertie, I. Masliah, and J. Falcou, "Data layout and simd abstraction layers: decoupling interfaces from implementations," in *2018 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2018, pp. 531–538.

[89] W.-m. Hwu, *GPU computing gems jade edition*. Elsevier, 2011, vol. 2.

[90] G. Herschlag, S. Lee, J. S. Vetter, and A. Randles, "Gpu data access on complex geometries for d3q19 lattice boltzmann method," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 825–834.

[91] M. Klingensmith, I. Dryanovski, S. S. Srinivasa, and J. Xiao, "Chisel: Real time large scale 3d reconstruction onboard a mobile device using spatially hashed signed distance fields." in *Robotics: science and systems*, vol. 4, no. 1, 2015.

[92] D. Werner, A. Al-Hamadi, and P. Werner, "Truncated signed distance function: experiments on voxel size," in *Image Analysis and Recognition: 11th International Conference, ICIAR 2014, Vilamoura, Portugal, October 22-24, 2014, Proceedings, Part II 11*. Springer, 2014, pp. 357–364.

[93] J. Straub, T. Whelan, L. Ma, Y. Chen, E. Wijmans, S. Green, J. J. Engel, R. Mur-Artal, C. Ren, S. Verma, *et al.*, "The replica dataset: A digital replica of indoor spaces," *arXiv preprint arXiv:1906.05797*, 2019.

[94] J. Park, Q.-Y. Zhou, and V. Koltun, "Colored point cloud registration revisited," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 143–152.

[95] T. Foote, "tf: The transform library," in *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*, ser. Open-Source Software workshop, April 2013, pp. 1–6.

[96] J. Deray and J. Solà, "Manif: A micro Lie theory library for state estimation in robotics applications," *Journal of Open Source Software*, vol. 5, no. 46, p. 1371, 2020. [Online]. Available: https://doi.org/10.21105/joss.01371

[97] J. Solà, J. Deray, and D. Atchuthan, "A micro Lie theory for state estimation in robotics," http://arxiv.org/abs/1812.01537, Institut de Robòtica i Informàtica Industrial, Barcelona, Tech. Rep. IRI-TR-18-01, 2018.

[98] T. D. Barfoot, *State Estimation for Robotics*. Cambridge University Press, 2017.

[99] T. D. Barfoot and P. T. Furgale, "Associating Uncertainty With Three-Dimensional Poses for Use in Estimation Problems," *IEEE Transactions on Robotics*, vol. 30, no. 3, pp. 679–693, 2014.