NanoSLAM: Enabling Fully Onboard SLAM for Tiny Robots

Vlad Niculescu, Tommaso Polonelli, *Member, IEEE,* Michele Magno, *Senior Member, IEEE,* and Luca Benini, *Fellow, IEEE*

Abstract—Perceiving and mapping the surroundings are essential for enabling autonomous navigation in any robotic platform. The algorithm class that enables accurate mapping while correcting the odometry errors present in most robotics systems is Simultaneous Localization and Mapping (SLAM). Today, fully onboard mapping is only achievable on robotic platforms that can host high-wattage processors, mainly due to the significant computational load and memory demands required for executing SLAM algorithms. For this reason, pocket-size hardware-constrained robots offload the execution of SLAM to external infrastructures. To address the challenge of enabling SLAM algorithms on resource-constrained processors, this paper proposes NanoSLAM, a lightweight and optimized end-to-end SLAM approach specifically designed to operate on centimeter-size robots at a power budget of only $87.9\,\mathrm{mW}$. We demonstrate the mapping capabilities in real-world scenarios and deploy NanoSLAM on a nano-drone weighing $44\,\mathrm{g}$ and equipped with a novel commercial RISC-V low-power parallel processor called GAP9. The algorithm is designed to leverage the parallel capabilities of the RISC-V processing cores and enables mapping of a general environment with an accuracy of $4.5\,\mathrm{cm}$ and an end-to-end execution time of less than $250\,\mathrm{ms}$.

Index Terms—SLAM, Mapping, Nano-Drone, UAV, Constrained Devices.

Supplementary Material

Supplementary video at https://youtu.be/XUSVLHJ87J0.

I. Introduction

The field of autonomous pocket-size robotics systems and Unmanned Aerial Vehicles (UAVs) experienced rapid growth in the past years due to the advancement and miniaturization of capable embedded computing platforms creating new possibilities for IoT applications [1]–[4]. Nano-robots, and especially palm-size UAVs, weigh only a few tens of grams and benefit from increased agility compared to their standard-size counterparts, enabling them to fly in narrow spaces reliably [5], [6]. Furthermore, their reduced dimensions make nano-UAVs perfect candidates for safely operating near humans, especially in cramped indoor environments [7], [8]. In most practical applications, the mission of the nano-UAV is to follow a path through the environment that is predefined or

V. Niculescu and L. Benini are with the Integrated Systems Laboratory of ETH Zürich, ETZ, Gloriastrasse 35, 8092 Zürich, Switzerland (e-mail: vladn@iis.ee.ethz.ch, lbenini@iis.ee.ethz.ch).

T. Polonelli and M. Magno are with the Center for Project-Based Learning of ETH Zürich, ETZ, Gloriastrasse 35, 8092 Zürich, Switzerland (e-mail: tommaso.polonelli@pbl.ee.ethz.ch, michele.magno@pbl.ee.ethz.ch).

L. Benini is also with the Department of Electrical, Electronic and Information Engineering of University of Bologna, Viale del Risorgimento 2, 40136 Bologna, Italy.

adjusted dynamically during the mission [5], [7]. For instance, finding the source of gas leaks [9] or localizing and reaching sensor nodes for data acquisition [10] are only a few examples of such applications.

The environments where nano-UAVs typically fly are filled with walls and obstacles, and thus, optimal path planning requires good knowledge of the surroundings map [5]. Furthermore, in a wide range of applications, the map can change over time, so preprogramming the map into the nano-UAVs is not an ideal solution [11]. In smart buildings, for instance, where the layout of reconfigurable walls can change [12], or simply in crowded offices where tables, chairs, and furniture are often moved. Moreover, the arrangement of pallets and shelves in warehouses can change from one day to another, and therefore, the UAVs used for inventory need a constantly updated map for reliable navigation [13]–[15].

In the scenarios mentioned so far, the drone needs an accurate environmental map and the ability to localize itself within the map [11]. The algorithm class that performs both tasks is called Simultaneous Localization and Mapping (SLAM). Among the existing SLAM algorithms, graph-based SLAM [16], [17] is one of the most adopted variations of the algorithm due to its high accuracy and capability to refine the complete trajectory. Moreover, graph-based SLAM models each trajectory pose (i.e., position and heading) as a graph node and the odometry measurements as graph edges. Due to the odometry errors that typically characterize any robotic platform, the uncertainty in the poses grows as the drone moves [2], [18]. Hence, upon revisiting a location (i.e., loop closure), the pose error is higher than at the initial visit. To mitigate this issue, the robot also acquires environmental observations (i.e., depth measurements) during the flight [19]. By comparing the observations associated with two different poses, an accurate rigid body transformation can be derived between the two, using an approach called scan-matching [19].

While the transformation provided by scan-matching allows correcting the current pose, graph-based SLAM propagates this information back to the previously added nodes in the graph (i.e., graph optimization) and corrects the whole trajectory [16]. In conclusion, the accuracy of the corrected trajectory depends on the accuracy of the scan-matching and, therefore, on the observations' accuracy [20]. In most common applications, the observations consist of depth measurements, typically provided by LiDARs or stereo cameras [21], [22]. Although SLAM paired with LiDARs is widely used in applications with standard-size UAVs, these solutions require large amounts of computational resources and memory, which

are not available on nano-UAVs [6], [19]. Furthermore, even the most compact LiDARs used with standard-size UAVs are about one order of magnitude heavier[1] than the maximum payload of nano-UAVs [23].

The recent release of lightweight (i.e., $42\,\mathrm{mg}$), low-resolution, and energy-efficient depth sensors based on Time of Flight (ToF) technology has changed the status quo in the feasibility of SLAM for nano-UAVs [24]. With the aid of such sensors, recent works demonstrated SLAM on nano-UAVs, but only under the assumption that the complex SLAM could be offloaded to an external base station [19]. This approach reduces the flight time due to the significant power consumption introduced by the radio communication with the base station [7]. Even more serious issues are the latency associated with the wireless communication protocol and the limited radio link range, which typically constrains the operating area within a few tens of meters in indoor environments [7]. Furthermore, because of the limited measurement capabilities of the early ToF sensors (i.e., a single distance value per sensor and a narrow FoV) [5], the existing systems that enable SLAM with nano-UAVs can only map simple-geometry environments such as long flat corridors. In contrast to the existing works, we exploit the novel VL53L5CX ToF sensor, which features an $8{\times}8$ resolution and provides a 64-pixel depth map with a Field of View (FoV) of $45°$. By mounting four such sensors on the nano-UAV (i.e., front, rear, left, right), we achieve a cumulative FoV of $180°$. Furthermore, spinning the drone by $45°$ results in a full angular coverage (i.e., $360°$), providing superior loop-closure performance compared to the previous ToF-based solutions and achieving centimeter-precision scan-matching accuracy, similar to the LiDAR-based approaches.

Despite the sparse information provided by the ToF sensors, scan-matching remains a computationally intense and memory-hungry problem. Furthermore, the computational requirements are further exacerbated by the graph optimization performed by the graph-based SLAM, which is independent of the depth observations. Standard-size UAV systems used for SLAM typically employ powerful embedded computers such as Qualcomm Snapdragon, Nvidia Jetson TX2, or Xavier [22], which have a power consumption of a few tens of watts, about two orders of magnitude higher than the power budget nano-UAVs typically have for computation. Recent trends in microcontroller design emphasize parallel processing, hardware accelerators, and energy efficiency. The GAP9 System on Chip (SoC) from GreenWaves Technologies[2] exemplifies these trends, being suited for specialized applications with nano-UAVs. With advanced parallel capabilities provided by the RISC-V cores, power optimization, and sensor integration, GAP9 empowers nano-UAVs with real-time edge computing, extended flight times, and enhanced data processing. GAP9 is based on the Parallel Ultra-Low-Power (PULP) computing paradigm [25], has a small form factor, and a power consumption below $180\,\mathrm{mW}$.

This paper proposes NanoSLAM, the first fully deployable framework that enables SLAM onboard nano-UAVs, per-forming the whole computation and environmental perception without relying on any external infrastructure or computation offload. Furthermore, by exploiting novel and low-power depth sensors in combination with the parallel capabilities of GAP9 SoC, our system achieves accurate indoor mapping comparable with SoA results from bigger and more computationally capable drones, commonly referred to as MAV or standard-size UAV [1]. Exploiting the parallel capability and energy efficiency of GAP9, we executed scan-matching and SLAM in real-time onboard the nano-UAV, which was not performed by any previous work. The contribution of this paper can be summarized as follows:

(i) an optimized parallel implementation of the graph-based SLAM algorithm that runs in the GAP9 SoC in real-time in less than $250\,\mathrm{ms}$. We comprehensively examine the various stages of the SLAM algorithm, providing an in-depth analysis of the optimizations made to each stage. Additionally, we evaluate the algorithm's execution time and memory requirements. *(ii)* a parallel implementation and evaluation of the Iterative Closest Point (ICP) algorithm, an SoA in scan-matching, running onboard in $55\,\mathrm{ms}$. *(iii)* a custom plug-in companion board for the commercial Crazyflie 2.1 nano-UAV that extends the sensing capabilities of the drone with 4 ToF matrix sensors, allowing it to perform scan-matching and autonomous navigation. *(iv)* a communication protocol that orchestrates the integration and data exchange between the drone's stock MCU and the GAP9 SoC, dictating how to store that graph, exchange graph poses, add edges, and perform graph optimization. *(v)* an extensive in-field experimental evaluation that proves our system's closed-loop mapping functionality, which exploits NanoSLAM to achieve a trajectory error reduction by up to 67% and a mapping accuracy of $4.5\,\mathrm{cm}$.

II. RELATED WORK

In the field of robotics, several essential components are indispensable for facilitating autonomous navigation on diverse unmanned vehicles. These components encompass real-time environment perception [26], onboard computational capabilities for prompt mission inference [11], [27], and, pertinent to the focus of this paper, the competence to map and explore unknown environments [28]. Mapping an environment is generally done by employing different combinations of sensors [2], such as LiDARs, stereo cameras, laser scanners, or radars. Subsequently, environmental and spatial information collected from these sensors is paired with estimation methods, including particle filters [11], Extended Kalman Filters (EKFs), covariance intersection that enables position estimation, and, finally, SLAM [29] that combines the position information with environmental observations to generate a layout of the environment. As discussed in the literature [30], [31], SLAM consists of two components: the front-end processing represented mainly by feature extraction and loop closure, which is largely dependent on the sensors used, and the sensor-agnostic pose-graph optimization, in charge of the back-end processing [32].

As the name suggests, visual SLAM (vSLAM) uses images to extract depth information [33]. It can use simple monocular

---

[1]The Crazyflye 2.1 weights $27\,\mathrm{g}$ and supports a maximum payload of $15\,\mathrm{g}$, while a lightweight LiDAR such as UST-10/20LX from Hokuyo weighs $130\,\mathrm{g}$.

[2]https://greenwaves-technologies.com/

cameras (e.g., wide angle, fish-eye, and spherical cameras), compound eye cameras (e.g., stereo and multi cameras), and RGB-D cameras such as depth or ToF cameras [33]. While SLAM can be enabled at a low cost with relatively inexpensive and limited cameras, the process involves large data volumes and is often marred with limited mapping accuracy [34]. On the other side, LiDARs are significantly more precise for depth estimation and are commonly used for applications involving high-speed moving vehicles such as self-driving cars and drones [35]. LiDAR-based systems typically provide sparse samples organized into high-precision 2D or 3D point clouds. Even if they yield accurate mapping results when combined with SLAM, LiDARS are generally expensive and heavy, weighing a few hundred grams [35].

Today, SLAM is useful in many applications [31] such as navigating a fleet of mobile robots to arrange shelves in a warehouse [13], parking self-driving cars in empty spots, autonomous race competitions [36], or delivering packages by navigating drones in unknown environments [6]. Many available tools already provide plug-and-play SLAM solutions that could be paired with other tasks such as sensor fusion, object tracking, path planning, and path following [36]. Although the mapping task seems to be a solved research problem in the literature, it relies on strong assumptions, such as memory availability of several gigabytes and powerful processor, e.g., the Intel i7 family [35], [37]. Moreover, carrying heavy and power-hungry 3D scanners, such as stereo cameras and LiDARs, is not considered a limitation for conventional robotic applications [35]–[37]. However, these assumptions do not hold for miniaturized and low-power robotic platforms, where the hardware cost is a concern, the payload is limited to a few tens of grams, and the computation power budget is limited to hundreds of mW [7], [11], [38]. Hence, enabling onboard mapping on this tiny class of devices is still an open problem.

This paper focuses on nano-UAVs as a specific application scenario to empirically validate the efficacy of our lightweight NanoSLAM approach. However, the challenges discussed in enabling mapping on nano-UAVs can be extended to the broader domain of micro-robotics and, more generally, to low-cost and resource-constrained devices [39]. Standard-size UAVs distinguish themselves from Micro-Aerial Vehicles (MAVs) and nano-UAVs in their physical dimensions, weight, total power consumption, and onboard processing capabilities [39]. For the latter two, the sensing and processing power budget represents about $\frac{1}{10}$ of the power consumed by the motors [7]. Presently, the majority of cutting-edge advancements in robotic perception and mapping have been showcased on standard-size UAVs and MAVs, which possess a power budget ranging from $50\,\text{W}$ to $100\,\text{W}$ and a total mass of $\geq 1\,\text{kg}$ [40]. Consequently, these vehicles can be equipped with high-performance onboard computing platforms, such as GPUs featuring gigabytes of memory [40]. Conversely, nano-UAVs, typically based on low power MCUs, weigh less than $50\,\text{g}$ with a power budget in the range of $5\,\text{W} - 10\,\text{W}$, with only $0.5\,\text{W} - 1\,\text{W}$ being allocated for powering the sensors, all the electronics, and the computational units [7], [39]. Low-power MCUs usually offer limited memory capacity, typically ranging from $100\,\text{kB}$ to $1\,\text{MB}$, posing a significant constraint

for visual-based perception and mapping [7], [11].

Previous studies conducted on MAVs and UAVs have commonly utilized miniature, conventional $360°$ LiDAR sensors [42] or depth stereo cameras [40] to perform mapping. For instance, Kumar *et al.* [43] integrated single-layer LiDAR sensors with inertial measurement units for indoor mapping tasks using a DJI Phantom 3 drone. This setup required an additional desktop-class Intel i5 processor onboard. The LiDAR sensor employed measures $62\,\text{mm} \times 62\,\text{mm} \times 87.5\,\text{mm}$, weighs $210\,\text{g}$, and consumes approximately $8.4\,\text{W}$. Similarly, Gao *et al.* [44] integrated a multi-layer LiDAR sensor with a desktop-class Intel i7 processor to enable 3D mapping of indoor environments. The LiDAR sensor they use consumes $8\,\text{W}$ and measures $103\,\text{mm} \times 103\,\text{mm} \times 72\,\text{mm}$ with a weight of $509\,\text{g}$. Another approach by Fang *et al.* [45] uses an RGBD camera combined with a particle filter to navigate through obstructed shipboard environments. Their platform is $58\,\text{cm} \times 58\,\text{cm} \times 32\,\text{cm}$ in size, carries over $500\,\text{g}$ of instrumentation, and is operated by a high-performance octa-core ARM processor. Table I provides an overview of SoA mapping strategies in the UAV field, encompassing sensor types, mapping accuracy, and power consumption of the computing platforms. For example, Causa *et al.* [35] proposed a scalable mapping strategy based on LiDAR and GNSS, utilizing a standard-size UAV weighing $3.6\,\text{kg}$ with off-board processing. Shen *et al.* [22] focused on onboard intelligence, utilizing a power-hungry Nvidia Xavier ($30\,\text{W}$) and a VLP-16 LiDAR. Huang *et al.* [2] entrusted the mapping algorithm and onboard processing to a Jetson TX2, equipped with a multi-core CPU and a GPU. Additionally, Chang *et al.* [37] proposed a robust multi-robot SLAM system designed to support swarms, but the results were validated offline using an Intel i7-8750H processor. Although these approaches demonstrated good mapping capabilities in the range of 5 to $20\,\text{cm}$, they involve large and heavy sensors that require power-intensive processing.

Implementing SLAM on nano-UAVs or any miniaturized and low-power hardware [46] is non-trivial due to the large memory and computation requirements typically associated with scan-matching or graph optimization. Moreover, alternatives such as offloading heavy computation tasks to an external computer is often an unpractical solution. In [2], authors show how the communication latency of a cloud-based multi-robot SLAM solution can reach up to $5\,\text{s}$, an unacceptable value in most nano-UAV uses cases. The severe limit imposed by continuous remote communication poses limits to the mapping speed and the overall system reliability [2], which further demonstrates the need for having a fully onboard SLAM even on resource-constrained nano-UAVs.

One approach to address the computational challenge involves parallelizing different processes on ultra-low power parallel SoCs [25], [46]. Utilizing embedded accelerators or multicore MCUs for processing, leveraging single instruction multiple data (SIMD) calculations, can enhance performance in certain scenarios [25], [47]. To this end, novel PULP SoCs have emerged in recent years, offering clusters of cores within $100\,\text{mW}$ of power consumption. Rossi *et al.* [25] present the basis of the commercial SoC GAP family from Greenwaves, which has already demonstrated its capabilities in the field

Work	On-board processing	Sensor	Latency	Map accuracy	Field test	Power Consumption	System Weight
			Nano-UAV and MAV				
This work	**Yes (Cortex-M4)**	**4× ToF 64-pixel VL53L5CX**	**247** ms	**4-8** cm	**Yes**	**350** mW	44 g
[6]	No	4× ToF VL53L1x	Post-processing	10-20 cm	Yes	-	27 g
[5]	Yes (Cortex-M4)	4× ToF VL53L1x	<10 ms	No Map	Yes	240 mW	31.7 g
[19]	No (Intel i7 station)	4× ToF VL53L1x	214 ms	5-15 cm	No	-	31.7 g
[41]	No	4× ToF VL53L1x	Post-processing	4.7 cm	No	-	401 g
			Standard-size UAV				
[35]	No	LiDAR	-	5-20 cm	Yes	-	3.6 kg
[22]	Yes (Xavier)	VLP-16 LiDAR	49 ms	2.14 m	No	30 W	>2 kg
[2]	Yes (Jetson TX2)	RP-LiDAR	∼1 s	-	Yes	>10 W	>2 kg
[37]	No (Intel i7 station)	LiDAR	Post-processing	15-20 cm	Yes	-	-
[40]	Yes (Jetson TX2)	Intel RealSense D435	∼120 ms	-	Yes	7.5 W	1.3 kg

TABLE I: System and performance comparison between this paper and the State-of-the-Art (SoA) works present in the literature. On-board processing, sensing elements, mapping accuracy, and system setups are compared.

of nano-UAVs for accurate localization [11] and autonomous navigation [7]. In particular, GAP9 is selected for the scope of this paper to carry the intensive computation.

To attain an optimal solution, the sensor selection needs to consider an optimal trade-off between power consumption, accuracy, and weight. In [48], the authors explore the possibility to use visual-based perception to enable obstacle avoidance and mapping on nano-UAVs. However, today, this direction does not seem to be promising due to the low performances of miniaturized RGB cameras and the large amounts of data they generate – which needs to be processed by resource-constrained processors [7], [49]. In their work [49], Tijmons *et al.* propose a stereo vision-based obstacle avoidance system for a flapping wing UAV, which demonstrates promising results with an onboard processing frequency of $15\,Hz$. This approach aligns with common methodologies employed in standard-size UAVs. However, their implementation needs an additional microcontroller (i.e., STM32F405) exclusively dedicated to image processing and the sensor board alone requires an energy consumption of $484\,mW$. It is worth noting that while the authors of [49] tested their system in real environments, they do not report any statistical analysis of the success rate. Furthermore, the authors acknowledge the limited robustness of their system in non-ideal flight conditions, such as the presence of small obstacles. Another practical example is provided by [7], where the authors introduce a grayscale camera-based navigation solution that is deployed onboard a nano-UAV to facilitate autonomous navigation and obstacle avoidance. A CNN is used for perception and exhibits reliable performance in detecting obstacles, allowing the drone to adjust its forward velocity or heading. However, in unfamiliar environments, particularly when executing $90°$ turns, the CNN's performance drops drastically, resulting in a high probability of collision when the drone exceeds $0.6\,m\,s^{-1}$ [7]. Additionally, their solution often struggles to avoid collisions with unknown obstacles placed in narrow environments such as corridors. Thus, vision-based approaches are not optimal solutions to enable onboard depth estimation with nano-UAVs, which is why we employ sensors that directly measure the depth.

Since the commercially available LiDAR exceeds the power and weight constraints of pocket-size UAVs, alternatives have been investigated. Recent studies have shown potential in enabling autonomous navigation with depth sensors based on the ToF technology. In [24], the authors investigate the possibility of using a commercial multi-zone ToF sensor that exhibits good measurement accuracy when measuring distances smaller than $2\,m$. Moreover, [26] used a lightweight 64-pixel ToF sensor for robust obstacle avoidance in indoor and outdoor scenarios, with a maximum speed of $1.5\,m/s$. At the time of writing, two commercially available depth sensors stand out: the VL53L5CX from ST Microelectronics and the ToF IRS2381C REAL3 from Infineon. The latter boasts an impressive resolution of 38,000 pixels and a maximum range of 4 meters. However, it requires an external illuminator, consumes up to $680\,mW$ for the entire circuitry, and has a weight exceeding $10\,g$. On the other hand, the VL53L5CX offers a lower resolution of 64 pixels but is significantly lighter, weighing only $42\,mg$. Additionally, its prior utilization in the nano-UAV field [11], [24] serves as a compelling motivation for selecting it for this paper.

As depicted in Table I, the existing literature offers only a limited number of studies proposing mapping solutions that use UAVs and have been successfully evaluated in field [2]. Notably, [37], [40] achieve their objectives without relying on external infrastructure. However, within the nano-UAV domain, even fewer works tackle the mapping challenge [6], [19], [41], and they offload the computation to an external base station. Furthermore, the existing works performing mapping with nano-UAVs are not able to reach the same level of accuracy as standard-size UAVs within the literature.

To the best of our knowledge, this paper introduces the first system that enables entirely onboard SLAM execution to enable accurate mapping of general environments, providing a comprehensive methodology, implementation, and field results. Our study demonstrates the system's functionality even with low-power miniaturized sensors that weigh only $44\,g$. The achieved accuracy aligns with the SoA for MAVs and standard-size UAVs, with a mapping error down to $4.5\,cm$. The proposed system facilitates advanced autonomous capabilities in nano-UAVs, paving the way for enabling additional features such as optimal path planning and multi-agent collaboration.

III. ALGORITHMS

This section presents a lightweight localization and mapping methodology that leverages the scan-matching and graph-based SLAM algorithms, targeting pocket-size robotic platforms and emerging low-power processors. Our solutions can enable any robotic platform of similar size or bigger to perform low latency SLAM in real-time, given depth measurement capabilities enabled by sensors such as 8×8 STMicroelectronics VL53L8CX described in Section II.

A. Scan Frames and Scans

Our objective is to conduct 2D localization and mapping utilizing depth sensors. Therefore, we assume a system equipped with n_s depth sensors (e.g., ToF) that provide measurements in the 2D plane with a resolution of n_z pixels (i.e., zones) per sensor. Figure 1 shows an example of such a system with $n_s = 4$ and $n_z = 8$, illustrating the drone, the ToF depth sensors, and how the distance measurements can be projected in the world frame. The world frame, body frame, and sensor frame are represented with W, D, and S, respectively.

Let $\boldsymbol{x}_k = (x_k, y_k, \psi_k)$ be the state of the drone (i.e., *pose*) expressed in the world coordinate frame at the discrete timestamp k. Furthermore, we use $\alpha \in \{1, 2, \ldots n_s\}$ to index among ToF sensors and $\beta \in \{1, 2, \ldots n_z\}$ to index among the zones of each sensor. The distance provided by sensor α for the zone β at instant k is marked as $d_k^{\alpha\beta}$. Equation 1 shows the distance measurement projection $d_k^{\alpha\beta}$ acquired at pose $\boldsymbol{x}_k$ into the world coordinate frame W. Indeed, the distance $d_k^{\alpha\beta}$ provided by the sensor is not the absolute distance to the object but the projection of the absolute distance on the OX axis of the sensor frame S^α. Thus, $tan(\theta_\beta) \cdot d_k^{\alpha\beta}$ represents the y-coordinate of the obstacle in the same sensor frame, where θ_β is the angle of each sensor zone. Translating the obstacle's coordinates to the origin of the drone's body frame B and rotating it to the world frame W leads to the second term of Equation 1. The translation is performed by adding the offset (o_x^α, o_y^α) to the obstacle's position – note that the offset is expressed in D, and it is different for each sensor. $\boldsymbol{R}$ represents the 2D rotation matrix and the sum $\psi_k + \gamma_\alpha$ represents the angle between S^α and W, where ψ_k is the heading angle between D and W; γ_α represents the rotation of the sensor frame w.r.t D and for the example in Figure 1, $\gamma_\alpha \in \{0°, 90°, 180°, 270°\}$. Lastly, we use the coordinates of the pose (x_k, y_k) to perform another translation and obtain the coordinates of the obstacle expressed in the world frame W. At every timestamp k, the ToF sensors provide at most $n_s n_z$ distance measurements – n_s sensors $\times$ n_z zones – as some distance measurements might be invalid and therefore not considered. Projecting the $n_s n_z$ points using Equation 1 leads to the collection $\{(p_x^{\alpha\beta}, p_y^{\alpha\beta}) \mid \alpha \leq n_s; \beta \leq n_z\}$ that we call a *scan frame*.

$$\begin{pmatrix} p_x^{\alpha\beta}(k) \\ p_y^{\alpha\beta}(k) \end{pmatrix} = \begin{pmatrix} x_k \\ y_k \end{pmatrix} + \boldsymbol{R}_{(\psi_k + \gamma_\alpha)} \begin{pmatrix} d_k^{\alpha\beta} + o_x^\alpha \\ tan(\theta_\beta) \cdot d_k^{\alpha\beta} + o_y^\alpha \end{pmatrix} \quad (1)$$

The 2D point collection in the scan frame could be further used as input for the scan-matching algorithm. However, the cardinality of a scan frame (i.e., the number of 2D points) is
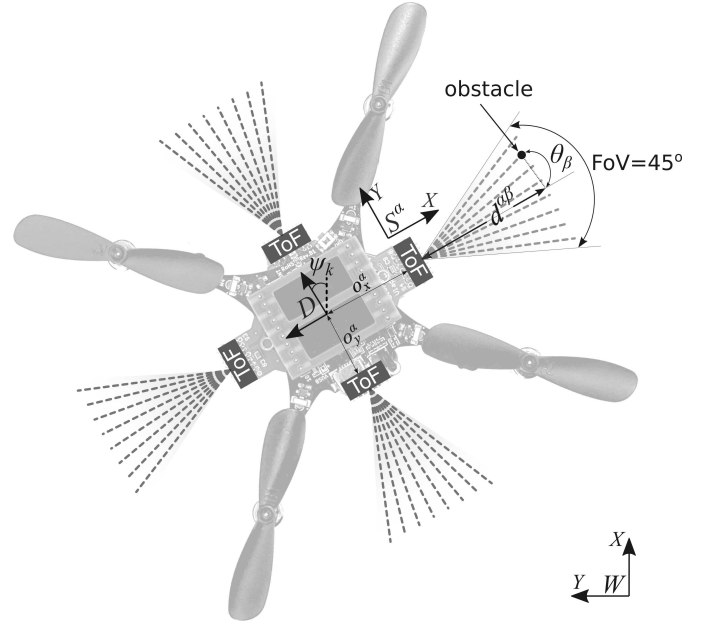

Fig. 1: Illustration of the four ToF sensors onboard the drone and the coordinate frames of the world, drone, and sensors.

still too small to enable accurate scan-matching. We overcome this issue by stacking n_{sf} consecutive scan frames in a set that we call a *scan* and define as $\boldsymbol{S_k} = \{(p_x^{\alpha\beta}(\tilde{k}), p_y^{\alpha\beta}(\tilde{k})) \mid \alpha \leq n_s; \beta \leq n_z; k \leq \tilde{k} < k + n_{sf}\}$. When the acquisition of a new scan is triggered, the robot starts appending new scan frames until it reaches the count of n_{sf}. The resulting cardinality of a scan is $n_{sf} n_s n_z$ points (minus the invalid pixels), which we call *the scan size*. Moreover, every scan $\boldsymbol{S}_k$ has an associated *scan pose* $\boldsymbol{x}_k$, which is the drone's pose when the scan acquisition starts.

Let f be the Field of View (FoV) of one ToF depth sensor, which leads to a cumulative FoV of $n_s f$, generally smaller than $360°$. To virtually increase the FoV and achieve full coverage, the drone also spins by $\frac{360 - n_s f}{n_s}$ degrees in place around the z-axis while acquiring the scan. For example, given the scenario in Figure 1 and assuming an FoV of $45°$ for each sensor, the drone should spin other $45°$ during the scan to cover the surroundings completely. With this mechanism, scan-matching can determine the transformation between two scans $\boldsymbol{S}_p$ and $\boldsymbol{S}_q$, which also applies to their associated scan poses $\boldsymbol{x}_p$ and $\boldsymbol{x}_q$. The scan size is a balance between scan-matching accuracy and memory usage, determined by the limitations of the system.

B. Scan-matching

Scan-matching is the process of determining the optimal rigid-body transformation between two scans. This transformation consists of rotation and translation, and with an ideal noise-free scenario, it should result in perfect overlapping with the other scan. Since scans and poses are strictly correlated, the transformation resulting from scan-matching also applies to the poses. When the drone is near a previously visited position, scan-matching can derive an accurate transformation w.r.t. a

Listing 1 The stages of the ICP algorithm. N_{iter}^{ICP} represents the number of iterations and $\boldsymbol{R}^*, \boldsymbol{t}^*$ the final solution after the algorithm executes.

```python
for k in range(N_iter^ICP):
    # Compute correspondences
    for i in len(S_p):
        correspondence[i] ← arg min_j‖p_i − q_j‖
    # Calculate the transformation
    R*,t* ← Equation 2
    # Apply transformation to scan S_p
    S_p ← R*S_p + t*
```

previously acquired pose in that location. In this way, scan-matching is used to correct the accumulated odometry errors. In this work, we implement and use ICP, an SoA algorithm in scan-matching [50].

We define two scans $\boldsymbol{S_p} = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots\}$ and $\boldsymbol{S_q} = \{\boldsymbol{q}_1, \boldsymbol{q}_2 \ldots\}$ where $\boldsymbol{p}_i$ and $\boldsymbol{q}_i$ are 2D points in the scans – we changed the initial indexing to enhance readability. Determining the optimal overlap between $\boldsymbol{S_p}$ and $\boldsymbol{S_q}$ can be formulated as a least squares problem, as shown in Equation 2 [50]. Note that Equation 2 requires to know what element $\boldsymbol{q}_i$ in scan $\boldsymbol{S_q}$ corresponds to the element $\boldsymbol{p}_i$ in scan $\boldsymbol{S_p}$. If the correspondences are known, a direct and optimal solution can be obtained by solving the optimization problem in Equation 2. This is typically done by offsetting each scan by its center of mass and then applying a rotational alignment based on the singular value decomposition method [50].

$$\boldsymbol{R}^*, \boldsymbol{t}^* = \arg\min_{\boldsymbol{R},\boldsymbol{t}} \sum \|\boldsymbol{q}_i - (\boldsymbol{R}\boldsymbol{p}_i + \boldsymbol{t})\|^2 \ . \tag{2}$$

However, the correspondences are unknown in our case and in most of real-world scan-matching applications. A common heuristic for determining the correspondences is to use the Euclidean distance – i.e., pairing each point $\boldsymbol{p}_i$ in $\boldsymbol{S_p}$ with the closest point $\boldsymbol{q}_j$ in $\boldsymbol{S_q}$ [50]. This implies solving the problem $\arg\min_j \|\boldsymbol{p}_i - \boldsymbol{q}_j\|$ for every point $\boldsymbol{p}_i$, using an exhaustive search over all elements in $\boldsymbol{S_q}$. Once these approximate correspondences are established, Equation 2 determines the transformation between the two scans, which is then applied to $\boldsymbol{S_p}$. Repeating this process until the two scans overlap represents the ICP algorithm, which we summarize in Listing 1.

C. Graph-based SLAM Algorithm

In most GPS-denied environments, such as indoor scenarios, the drone's internal state estimator computes the position and heading by integrating velocity and angular velocity measurements. However, the measurements are affected by sensor noise, and integrating noisy data over time results in drift. Equation 1 shows that projecting distance measurements in the world frame to obtain a scan or the map requires trajectory knowledge. Since the trajectory error impacts the accuracy of the map, we use SLAM to first correct the trajectory and then compute the map w.r.t. the corrected path. For this

purpose, we implement the graph-based SLAM introduced in [16], which can use scan-matching information to correct the trajectory. The graph-based SLAM represents the trajectory as a pose graph, where each pose (i.e., 2D position and heading) is modeled as a graph node, and the edges are relative constraints between the nodes. We distinguish two types of graph edges: *(i) the odometry edges* incorporating motion information between any two consecutive poses, and *(ii) the loop closure (LC) edges* which embody relative measurements derived by ICP.

Let N be the number of poses and n the number of LC edges. Moreover, let $\boldsymbol{X} = \{\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{N-1}\}$ be the graph nodes expressed in W, and $\boldsymbol{z}_{ij} = (z_x, z_y, z_\psi)$ the graph edge measurements, the latter being expressed in the coordinate frame of pose $\boldsymbol{x}_i$. We note as $\hat{\boldsymbol{z}}_{ij}$ the prediction of an edge measurement, or in other words, the edge measurement computed given two poses $\boldsymbol{x}_j$ and $\boldsymbol{x}_i$. Pose graph optimization (PGO) involves the estimation of optimal pose values that ensure consistency between the edge measurements zij and the predicted measurements $\hat{z}ij$. As shown in [16], this is done by minimizing the sum of the squared differences $\boldsymbol{e}_{ij} = \boldsymbol{z}_{ij} - \hat{\boldsymbol{z}}_{ij}$, where Equation 3 gives the maximum likelihood solution that requires the initial pose $\boldsymbol{x}_0$ and the edges $\boldsymbol{z}_{ij}$ to compute the optimal poses. The number of terms in the sum is equal to the number of edges in the graph, and Ω is the diagonal information matrix, which weighs the importance of each edge. Since ICP typically provides accurate results, the LC edge measurements are more precise than the odometry edge measurements.

$$\boldsymbol{e}_{ij} = \boldsymbol{z}_{ij} - \hat{\boldsymbol{z}}_{ij}(\boldsymbol{x}_i, \boldsymbol{x}_j) \ ,$$
$$\boldsymbol{X}^* = \arg\min_{\boldsymbol{X}} \sum_{i,j} \boldsymbol{e}_{ij}^T \Omega \boldsymbol{e}_{ij} \ . \tag{3}$$

Running SLAM onboard a resource-constrained device in real-time requires solving the optimization problem in Equation 3 efficiently. Since this is a non-linear problem, there is no closed-form solution, but iterative methods such as Gauss-Newton have been proven effective if a good initial guess is known. In every iteration, the error function $\boldsymbol{e}_{ij}(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is approximated with its first-order Taylor expansion, reducing the problem to a linear equation system. This paper provides an efficient implementation of the graph-based SLAM algorithm derived from [16], which is in charge of PGO. We summarize the algorithm in Listing 2 and discuss it in detail in Section V.

$$\begin{pmatrix} z_x \\ z_y \end{pmatrix} = \boldsymbol{R}_{-\psi_i} \begin{pmatrix} x_{i+1} - x_i \\ y_{i+1} - y_i \end{pmatrix} \ , \tag{4}$$

$$z_\psi = \psi_{i+1} - \psi_i \ . \tag{5}$$

The graph-based SLAM algorithm requires the initial pose $\boldsymbol{x}_0$, the edge measurements $\boldsymbol{z}_{ij}$, and an initial guess of the pose values. The initial pose $\boldsymbol{x}_0$ is the drone's pose right after take-off, and without loss of generality, it is always considered $(0, 0, 0)^T$. Since there is no additional information about the poses, the best initial guess is computed by forward integrating the odometry measurements w.r.t. $\boldsymbol{x}_0$. Consequently, the poses' initial guess encompasses the same information as the odometry edge measurements, and therefore, it suffices to

Listing 2 The graph-based SLAM algorithm that performs PGO. The outer *for* loop runs for N_{iter}^{SLAM} iterations.

```
# 1. Compute the odometry edge measurements
for i in range(N − 1):
    z_{i,i+1} ← Equations 4 − 5
# 2. Compute the LC edge measurements
for k in range(n):
    z_{ij} ← ICP(x_i, x_j)
# 3. Graph optimization
for k in range(N_{iter}^{SLAM}):
    # 3a. Compute H and b
    H ← 0, b ← 0, H_{11} ← I_3
    for edge in edges:
        # Compute the Jacobians
        A_{ij} ← ∂e_{ij}(x)/∂x_i |_{x=x*}    B_{ij} ← ∂e_{ij}(x)/∂x_j |_{x=x*}
        # Construct the linear system matrix
        H_{ii}+= A_{ij}^T Ω A_{ij}    H_{jj}+= A_{ij}^T Ω B_{ij}
        H_{ij}+= A_{ij}^T Ω B_{ij}    H_{ji}+= B_{ij}^T Ω B_{ij}
        # Construct the linear system vector
        b_i+= A_{ij}^T Ω e_{ij}    b_j+= B_{ij}^T Ω e_{ij}
    # 3b. Solve the linear system HΔx = −b
    Permutation H_P = PHP^T, b_P = Pb, Δx_P = PΔx
    # Solve H_P Δx_P = −b_P
    Cholesky decomposition H_P = L_P L_P^T
    Forward substitution L_P y = −b_P
    Backward substitution L_P^T Δx_P = y
    # Retrieve the solution Δx
    Inverse permutation Δx = P^{-1} Δx_P
    # 3c. Update the solution
    x* ← x* + Δx
```

store only the poses. This mechanism is convenient because, in many robotics applications, the robot's state estimator directly integrates the odometry measurements and provides the pose values. In this way, the odometry edge measurements are calculated right before the optimization is performed, as shown in the first *for* loop in Listing 2. Each measurement $z_{i,i+1}$ is expressed in a coordinate frame rotated by ψ_i and computed using Equations 4 – 5. The second *for* loop calculates the LC edges, using the ICP algorithm introduced in Section III-B.

Once all the edge measurements are calculated, the actual graph optimization can start, performed in the double *for* loop. Minimizing Equation 3 when $e_{ij}(x_i, x_j)$ is linearized around the current pose guess is equivalent to solving the linear equation system $H\Delta x = -b$ [16]. H and b are computed in the inner *for* loop. A_{ij} and B_{ij} are 3×3 matrices and represent the Jacobians obtained after linearization. Similarly, the 3×3 blocks H_{ii}, H_{ij}, H_{jj}, and H_{ji} represent the contribution on the H matrix of each graph edge from node i to node j. The dimension of the blocks b_i and b_j is 3×1, and they construct the system vector b. Given the constituent elements of matrix H and vector b, their dimension is $3N \times 3N$ and

$3N \times 1$, respectively. The $3N \times 1$ vector x^* serves as the ongoing estimate of the poses (stacked together), continuously refined during the iterative graph optimization process. Before the optimization starts, the initial guess is loaded into x^*.

The next step is to solve the linear system $H\Delta x = -b$. Inverting matrix H would demand significant memory and computational resources, inefficient for resource-constrained devices [51]. Nonetheless, more efficient alternatives have been suggested in the literature, which leverage the Cholesky decomposition [52], [53]. Since H is symmetric positive-definite, the decomposition calculates the lower triangular matrix L, such that $H = LL^T$. The equation system becomes $LL^T\Delta x = -b$. In addition, we make the notation $y = L^T\Delta x$. Since L is triangular, solving $Ly = -b$ is trivial using the forward substitution method. Having y, the solution Δx is easily calculated by solving $y = L^T\Delta x$ using backward substitutions. Lastly, the solution Δx is added to the current estimate of poses. Typically, the outer loop iterates until Δx reaches a sufficiently small value or becomes zero.

The ordering of the rows and columns of matrix H influences the non-zero count and computation time of matrix L [51]. Permuting both the rows and columns of H is done by the multiplication PHP^T, where P is the permutation matrix – i.e., an identity matrix with reordered rows [51]. Exploiting the property $P^{-1} = P^T$, the linear system is rewritten as $HP^TP\Delta x = -b$. Multiplying both sides by P on the left and making the substitutions $H_P = PHP^T$, $b_P = Pb$, and $\Delta x_P = P\Delta x$ leads to $H_P\Delta x_P = -b_P$. Lastly, Δx is retrieved from Δx_P, which implies a negligible overhead. Therefore, applying the permutation leads to the same mathematical problem, requiring the additional step of retrieving Δx from Δx_P, which comes with negligible overhead. The process of solving $H\Delta x = -b$ leveraging the Cholesky decomposition and the permutation mechanism is described in step 3b of Listing 2. In Section V, we discuss how the permutation matrix P is obtained.

D. SLAM in Real-world Scenarios

Figure 2 shows how a robot trajectory can be discretized into a pose graph. In this example, the drone flies along a square loop corridor, following the outer wall until it reaches the start point again. As the drone advances, it keeps adding new poses to the graph at fixed intervals using the information provided by the internal state estimator. The pose x_0 is the starting point and, therefore, error-free, but since the following poses are obtained based on integration w.r.t. x_0, they are affected by errors due to odometry drift. We note the poses as $x_0, x_1, \ldots x_{N-1}$ and represent them with empty circles in Figure 2, while the odometry constrains $z_{01}, z_{12}, \ldots$ are the edges connecting the circles. Performing the graph optimization at this point would lead to no change in the poses because any pose x_{i+1} is obtained by integrating the measurement $z_{i,i+1}$ w.r.t. x_i and therefore the poses and edge measurements are already in agreement – i.e., the sum from Equation 3 is already zero.

In Figure 2, the filled grey circles denoted as $x_0^R, x_1^R, \ldots$, represent the actual (i.e., the ground truth) position and heading of the poses, which are not known to the drone. At the
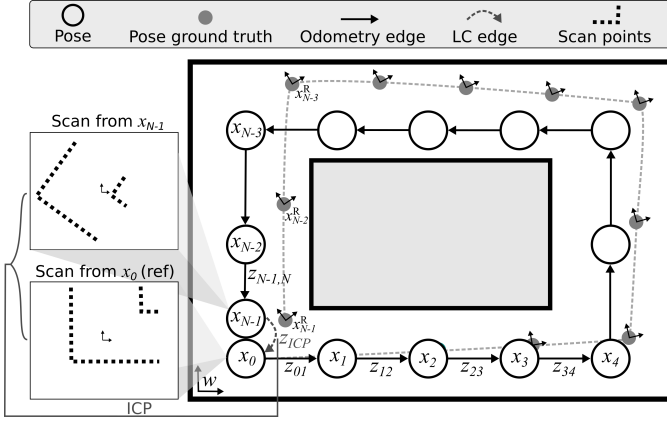
Fig. 2: The figure shows how a robot trajectory is discretized into a pose graph when passing through a square loop corridor. The drone keeps adding poses to the graph using information from the internal state estimator. The poses are affected by errors due to odometry drift.
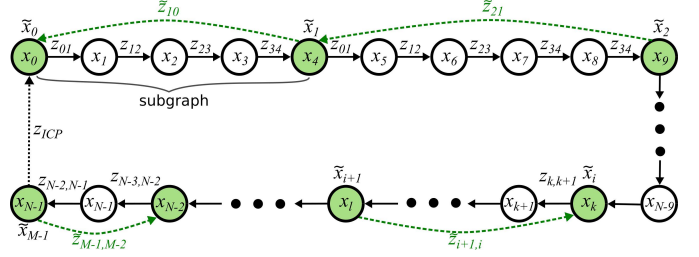

Fig. 3: Representation of the sparse graph (green) as a subset of the complete graph. The black arrows represent the odometry edges, the dashed black arrow represents the LC edge, and the dashed green arrows represent the additional constraints derived for optimizing the subgraphs.
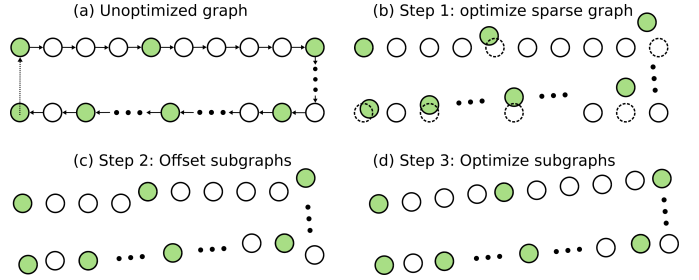

Fig. 4: A breakdown of the hierarchical graph-based SLAM.

end of the mission, even if the drone estimates that it crosses the starting point again (i.e., $\boldsymbol{x}_0 = \boldsymbol{x}_{N-1}$), its actual pose is $\boldsymbol{x}_{N-1}^R$. To mitigate the odometry errors, the drone acquires observations (i.e., a scan) in $\boldsymbol{x}_{N-1}$, which it compares with the scan acquired in $\boldsymbol{x}_0$, as shown in Figure 2. We call as *reference scan* the scan acquired when a place is visited for the first time – e.g., the scan acquired in $\boldsymbol{x}_0$. Furthermore, we define as *LC scan* the scan acquired when a place is revisited – e.g., the scan acquired in $\boldsymbol{x}_{N-1}$. An LC scan is always paired with a reference scan or another LC scan, and ICP is used to derive a transformation between the two. In the example from Figure 2, ICP is used to derive a transformation between $\boldsymbol{x}_0$ and $\boldsymbol{x}_{N-1}$, and therefore add a new LC edge to the graph – from node $N-1$ to node 0. Once there is at least one new LC edge in the graph, graph-based SLAM can run to correct the existing poses. After the optimization completes, the LC edges are typically kept in the graph. In the context of our approach, we assume an unchanging environment. Yet, should alterations occur within the environment that lead to scans that do not overlap, we identify these situations and discard the LC edge. The procedure for quantifying the degree of overlap between two scans is elaborated upon in Section VII.

E. Optimizing Large Graphs

The elements of graph-based SLAM were presented in a simple example in Figure 2, but they are representative of any graph and any number of poses or constraints. However, optimizing graphs larger than a few hundred poses with this method might be challenging because embedded platforms are typically constrained to a few hundred of kB of RAM. To address this problem, we implement a solution based on the hierarchical optimization approach introduced in [54]. The idea is to divide the graph into multiple subgraphs and apply the graph-based SLAM algorithm from Listing 2 on each subgraph – we refer to this approach as hierarchical graph-based SLAM. For this purpose, a *sparse graph* $\bar{\boldsymbol{X}} = \{\tilde{\boldsymbol{x}}_0, \ldots, \tilde{\boldsymbol{x}}_{M-1}\}$ is created first, whose poses are a subset (but still representative)

of the complete graph $\boldsymbol{X}$. We mention that the poses marked with a tilde are just an alternative notation for the poses already present in $\boldsymbol{X}$ to emphasize that we are referring to the sparse graph. We provide a graphical representation of such a hierarchical optimization problem in Figure 3, where the poses of the sparse graph are represented in green. Furthermore, Figure 4 shows a four-step breakdown of the hierarchical optimization. Using the scan-matching constraints (e.g., z_{ICP} in Figure 2), the sparse graph is optimized, resulting in the new set of poses $\{\tilde{\boldsymbol{x}}_1^{opt}, \ldots, \tilde{\boldsymbol{x}}_M^{opt}\}$ – as shown in Figure 4-(b). The idea now is to use the optimized poses of the sparse graph as constraints to correct the entire graph $\boldsymbol{X}$.

For each pair of consecutive poses in the sparse graph $(\tilde{\boldsymbol{x}}_i^{opt}, \tilde{\boldsymbol{x}}_{i+1}^{opt})$ we build a subgraph consisted of these poses and the in-between poses of the complete graph – e.g., $\{\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_4\}$ or $\{\boldsymbol{x}_4, \boldsymbol{x}_5, \ldots, \boldsymbol{x}_9\}$. To be more general, we consider the subgraph $\{\boldsymbol{x}_k, \boldsymbol{x}_{k+1}, \ldots, \boldsymbol{x}_l\}$. We recall that the $\boldsymbol{x}_k = \boldsymbol{x}_i$ and $\boldsymbol{x}_l = \boldsymbol{x}_{i+1}$. Since we have already corrected the extremes (i.e., $\tilde{\boldsymbol{x}}_i^{opt}$ and $\tilde{\boldsymbol{x}}_{i+1}^{opt}$), this information can be further used to derive the constraint that allows optimizing the whole subgraph. In this scope, we firstly offset every pose in the subgraph as shown in Figure 4-(c), so that $\boldsymbol{x}_k/\tilde{\boldsymbol{x}}_i$ matches $\tilde{\boldsymbol{x}}_i^{opt}$ – necessary because PGO never corrects the first pose. Then Equations 4 – 5 are used to derive a constraint $\tilde{\boldsymbol{z}}_{i+1,i}$ between poses $\tilde{\boldsymbol{x}}_{i+1}^{opt}$ and $\tilde{\boldsymbol{x}}_i^{opt}$, which is added to the subgraph as an LC edge from node l to k – this only simulates the effect of loop closure, as the LC edge is not provided by ICP directly. After these operations are performed on the $M-1$ subgraphs as shown in Figure 4-(d), the optimization of $\boldsymbol{X}$ is complete. This section, therefore, introduces two manners of performing PGO: directly applying graph-based SLAM on

the existing pose graph or dividing the graph into multiple smaller subgraphs and optimizing every subgraph individually. The advantages of every approach are discussed in Section V.

Sampling the poses of the sparse graph from the complete graph X is based on a threshold on the robot movement. The elements comprising the complete graph are chronologically traversed, and a new node is exclusively incorporated into the sparse graph if the Euclidean distance from the most recently added node exceeds a threshold value d_{min} or if the difference in heading surpasses a threshold value $\Delta\psi_{min}$. The sparse graph must also include all scan poses as a mandatory requirement, in addition to the threshold-based added poses. This is because the LC edges resulting from ICP only play a role in optimizing the sparse graph and not also the subgraphs. However, the number of scan poses is usually negligible compared to the sparse graph size.
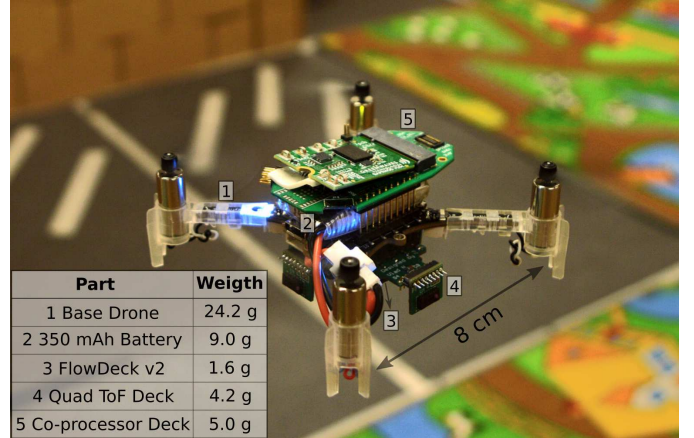
IV. NANO-UAV SYSTEM SETUP

Our mapping system is designed to be flexible and cover a large set of robotic platforms. The only prerequisite concerns the sensor, which has to be a depth camera. Thus the algorithm and the implementation can be adapted to support a different hardware setting, e.g., various processors or sensing elements. In this paper, we selected the Commercial-off-the-Shelf (COTS) nano-UAV Crazyflie 2.1 from Bitcraze to demonstrate the effectiveness of our solution in ultra-constrained platforms. In this way, our results can be easily replicated using commercially available hardware.
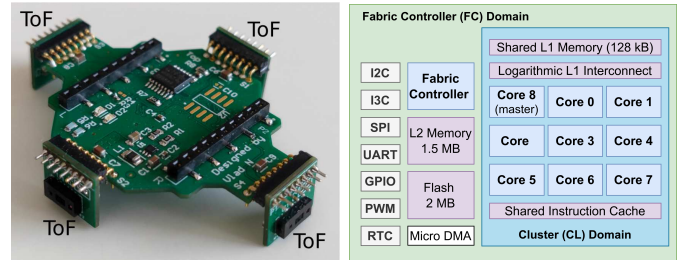
The open-source firmware of Crazyflie 2.1 provides capabilities for flight control, state estimation, radio communication, and setpoint commander. The drone's main PCB also acts as a frame, comprising the electronics such as an Inertial Measurement Unit (IMU), a radio transceiver (Nordic nRF51822), and an STM32F405 processor. The latter features a maximum clock frequency of $168\,\mathrm{MHz}$ and $192\,\mathrm{kB}$ of RAM, but over 70% of the resources are already used by the firmware to perform the control and estimation. Furthermore, the drone features extension headers that can be used to add additional decks (i.e., plug-in boards). We, therefore, also included the commercial Flow deck v2, which exploits a downward-facing optical flow camera and single-zone ToF ranging sensor to enable velocity and height measurements fused by the on-board Extended Kalman Filter (EKF) to perform position and heading estimation. In addition to the Flow deck, we equip the drone with two custom-designed boards: one containing four lateral depth ToF sensors to enhance the drone's capabilities to sense the surroundings, and the second deck contains the GAP9 SoC, used as a co-processor to extend the Crazyflie 2.1 computation capabilities. In this configuration, the total weight at take-off is $44\,\mathrm{g}$, including all the hardware used for the scope of this paper. The fully integrated system featuring our custom hardware is shown in Figure 5a.

A. Custom Quad ToF Deck

The VL53L5CX is a lightweight multi-zone 64-pixel ToF sensor, weighing only $42\,\mathrm{mg}$. Its suitability for nano-UAV applications was evaluated in a study by Niculescu *et al.* [24].

Part	Weigth
1 Base Drone	24.2 g
2 350 mAh Battery	9.0 g
3 FlowDeck v2	1.6 g
4 Quad ToF Deck	4.2 g
5 Co-processor Deck	5.0 g

(a) The fully integrated system.

(b) The custom quad ToF deck.   (c) Architecture of the GAP9.

Fig. 5: (a) Our prototype based on Crazyflie 2.1 extended with the ToF Deck and the Co-processor Deck. (b) The custom quad ToF deck featuring four ToF multi-zone sensors. (c) A simplified diagram showing the blocks of the GAP9 that are most relevant for this work.

This sensor offers a maximum ranging frequency of $15\,\mathrm{Hz}$ for an 8×8 pixel resolution, with a FoV of $45°$. Additionally, the VL53L5CX provides a pixel validity matrix alongside the 64-pixel measurement matrix, automatically identifying and flagging noisy or out-of-range measurements. To accommodate the use of multi-zone ranging sensors on the Crazyflie 2.1 platform, a custom deck was developed specifically for the VL53L5CX ToF sensors, as shown in Figure 5b. This deck can be used in conjunction with the Flow deck v2 and incorporates four VL53L5CX sensors positioned to face the front, back, left, and right directions, enabling obstacle detection from a cumulative FoV of $180°$. As a result, the final design of the custom deck weighs a mere $4.2\,\mathrm{g}$.

B. Co-processor Deck - GAP9 SoC

The second custom deck included in the system setup weighs $5\,\mathrm{g}$ and features the GAP9 SoC, the commercial embodiment of the PULP platform [25], produced by Green-waves Technologies. Figure 5c shows the main elements of the GAP9 architecture. The GAP9 SoC features 10 RISC-V-based cores, which are grouped into two power and frequency domains. The first domain is the fabric controller (FC), which features a single core operating at up to $400\,\mathrm{MHz}$ coupled with $1.5\,\mathrm{MB}$ of SRAM (L2 memory). The FC acts as the supervisor of the SoC, managing the communication with the

peripherals and orchestrating the on-chip memory operations. The second domain is the cluster (CL) consisting of nine RISC-V cores that can operate up to $400\,\text{MHz}$, specifically designed to handle highly parallelizable and computationally intensive workloads. Among the nine cores of the cluster, one acts as a "master core", receiving a job from the FC and delegating it to the other eight cores in the cluster, which carry the computation. The CL is coupled with $128\,\text{kB}$ of L1 memory, and the transfers between L2 and L1 are performed via the direct memory access (DMA) peripheral, requiring no involvement from the FC or CL during the transfers. To achieve an optimal execution time of a CL task, the data associated with the task should be transferred to L1 before the task is started. When the CL task completes, the result can be transferred back to L2 and further used by the FC. The GAP9 is interfaced with the STM32 via SPI and carries all the intensive computation required by PGO and scan-matching.

V. IMPLEMENTATION

Our system features two computational units: the STM32 MCU, part of the commercial Crazyflie 2.1 platform, and the more powerful GAP9 SoC, which extends the computational capabilities of the former. We extend the base firmware of the STM32 with our application – implemented through the Bitcraze Application Layer – containing only lightweight functionalities such as the ToF sensor data acquisition and the flight strategy, which have a negligible impact on the MCU load. Instead, we delegate the memory and computationally demanding tasks to the GAP9, which continuously communicates with the STM32 during the mission. Thus, computationally intensive solutions such as ICP, the graph-based SLAM, scan computation, or map generation run entirely on the GAP9. In the following, we provide the implementation details of NanoSLAM, which is based on the algorithms introduced in Section III.

A. Sensor Processing

As mentioned before, our system performs mapping in 2D. However, since each of the four ToF sensors provides an 8×8 distance matrix, we must process this information and reduce it to one plane (i.e., one row). For this reason, we discard the first two rows from the bottom and the top, leaving only the middle four rows that better represent the drone's plane. In the following, we select the median of the four remaining pixels for each column, obtaining a row vector of size eight for each ToF sensor. In case there are no valid pixels in a particular column (e.g., no obstacle within $4\,\text{m}$), the entire column is discarded. This approach ensures more robustness to outliers than simply selecting one of the middle rows from each matrix.

B. Scan-matching Implementation

Before detailing the actual scan-matching implementation, we provide the values of the scan parameters introduced in Section III-A. Indeed, our setup matches the configuration shown in Figure 1, featuring four ToF sensors of eight zones each – i.e., $n_s = 4$ and $n_z = 8$. During a scan, the drone undergoes a $45°$ rotation while adding new scan frames to the scan
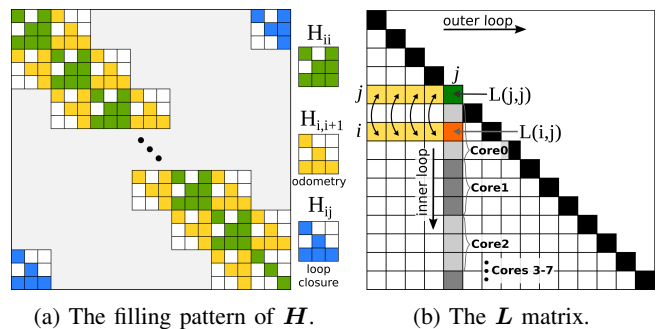

(a) The filling pattern of $\boldsymbol{H}$.　　(b) The $\boldsymbol{L}$ matrix.

Fig. 6: (a) The figure shows how the odometry and LC edges differently impact the sparsity of the $\boldsymbol{H}$ matrix. (b) The computation of the Cholesky decomposition and how each element's calculation is distributed among the CL cores.

with a frequency of $7.5\,\text{Hz}$. We empirically choose $n_{sf} = 20$ as a trade-off between scan-matching accuracy and memory footprint, resulting in a scan duration of about $2.7\,\text{s}$. Given these settings, the scan size is at most $n_{scan} = n_{sf}n_s n_z = 640$ points.

We recall that the ICP algorithm introduced in section III-B has two stages: determining the correspondences and calculating the transformation given the correspondence pairs. The latter exhibits a time complexity of $O(n_{scan})$ and is typically very fast. The correspondences calculation, represented by the inner *for* loop in Listing 1, takes more than 95% of execution time, operating with $O(n_{scan}^2)$ complexity. Furthermore, since the correspondences are calculated independently of each other, we leverage the parallel capabilities of GAP9, distributing the inner *for* loop from Listing 1 to eight cores of the CL in GAP9. In our implementation, we choose a fixed number of iterations N_{iter}^{ICP} to ensure a deterministic execution time. We empirically determined with in-field experiments that ICP always converges within $N_{iter}^{ICP} = 25$ iterations, and after that, the solution $(\boldsymbol{R}^*, \boldsymbol{t}^*)$ does not change anymore.

C. Graph-based SLAM Implementation

In the following, we provide the implementation details of the graph-based SLAM algorithm, which is presented in Listing 2. Having introduced how ICP is implemented, we now focus on step 3 from Listing 2, the heart of graph-based SLAM. Once all the odometry constraints are computed, each iteration of the algorithm consists of two main phases: *(i)* calculating $\boldsymbol{H}$ and $\boldsymbol{b}$, and *(ii)* solving the equation system $\boldsymbol{H}\Delta\boldsymbol{x} = -\boldsymbol{b}$. The main challenge is to enable the onboard execution, given the limited available amount of RAM. Storing all entries of the $3N \times 3N$ $\boldsymbol{H}$ matrix would result in about $1.44\,\text{MB}$ for a realistic pose number of 200 and a 4-byte float representation of the matrix entries. This requirement is infeasible for resource-constrained platforms – even for our capable target platform, GAP9, which would rapidly run out of memory storing such a matrix.

However, as Listing 2 shows, constructing matrix $\boldsymbol{H}$ implies looping through all edges and modifying the blocks $\boldsymbol{H}_{ii}$, $\boldsymbol{H}_{ij}$, $\boldsymbol{H}_{ji}$, and $\boldsymbol{H}_{jj}$, for each graph edge from i to j. Due to the highly accurate results offered by ICP, we experimentally

Listing 3 The Cholesky–Crout algorithm that performs the Cholesky decomposition column-by-column.

```
# Outer loop: iterating over columns
for j in range(N):
    sum0 = Σ_{k=0}^{j-1} L²(j,k)
    L(j,j) = √(H(j,j) - sum0)
        # Inner loop: iterating over rows
    for i in range(j+1, N):
        sum1 = Σ_{k=0}^{j-1} (L(i,k) · L(j,k))
        L(i,j) = (H(i,j) - sum1)/L(j,j)
```

TABLE II: The non-zero count of H, L and L_P. For the symmetric H matrix, only the non-zero elements from the main diagonal or lower are counted.

		40	80	120	160	200	240	280	320
non-zero	**H**	405	805	1205	1605	2005	2405	2787	3185
	L	1149	2349	3549	4749	5949	7149	8349	9549
	Lp	896	1817	2735	3657	4576	5496	6262	7155

selected an information matrix $\Omega = 20I$ for the LC edges and $\Omega = I$ for the odometry edges. This deliberate choice assigns greater significance to the LC edges during the optimization process. The number of odometry edges (i.e., $N-1$) is typically much larger than the number of LC edges, and for most of the blocks H_{ij}, it holds that $j = i+1$. Thus, most non-zero elements of H are concentrated around the main diagonal. Figure 6a, provides a graphical representation of the H matrix, where the contribution of blocks H_{ii} and $H_{i+1,i+1}$ is represented in green, and the contribution of $H_{i,i+1}$ and $H_{i+1,i}$ in yellow. Blocks in blue correspond to the LC edges, and their placement in the matrix does not follow a pattern.

Furthermore, by calculating the individual elements of each block with the equations from Listing 2, one could notice that some elements are always zero and represented in white in Figure 6a. This fact increases, even more, the sparsity of H, resulting in $17N - 13 + 10n$ non-zero elements according to the filling pattern of Figure 6a. Moreover, due to the fact that H is symmetric, it is sufficient only to store the elements below and including the main diagonal, implying $10N - 5 + 5n$ non-zero elements. As a numerical example, for the realistic values of $N = 200$ poses and $n = 10$ LC constraints, the ratio between non-zero elements and the total number of elements $3N \times 3N$ is about 0.56%, which proves that it is extremely memory inefficient to store a matrix in a dense form.

To exploit the high sparsity level, we propose storing H in a CSR sparse matrix representation [51]; we note the non-zero element count as nz. This representation uses three arrays: *(i) the values*: it has size nz and stores the non-zero elements; *(ii) the column index*: it has size nz and stores the column index associated with each value in the values array; *(iii) the row pointer*: it has size $3N + 1$, and its elements mark where in the values array a new row starts. Inserting a new element in the CSR matrix requires modifying the three arrays accordingly. Our software implementation solely utilizes static memory allocation to prevent memory leaks and overflows. Consequently, inserting elements in the sparse matrix must occur row-wise, in the ascending order of column indices – in this way, the arrays of the sparse matrix are never modified, only extended. Otherwise, a random insertion order would imply memory moves within the sparse matrix, slowing the execution. Since the filling pattern of H is deterministic given the graph, an ordered element insertion is possible.

The stages of graph-based SLAM exhibit a computational

complexity not exceeding $O(N)$, except for the Cholesky decomposition. To leverage the parallel capabilities of the system, we employ the Cholesky-Crout scheme [51]. This scheme efficiently computes the matrix L column by column, as outlined in Listing 3. To better illustrate the distribution of computation across eight cores of the GAP9 CL, we complement Listing 3 with Figure 6b. In each iteration for column j, the algorithm initially calculates the variable $sum0$, which represents the sum of squared elements from line j, excluding the diagonal. In Figure 6b, these elements are visually depicted by the upper yellow line, with each CL core responsible for computing the sum of $j/8$ elements. The value of $L(j,j)$ (highlighted in green) is subsequently determined based on $sum0$, and afterward, all column elements are computed within the inner loop. To offload the computation of the inner loop, we employ the CL, where each core performs the calculation for a predetermined number of $L(i,j)$ entries. Each element $L(i,j)$ (depicted in orange) is derived from $sum1$, which is computed as the dot product between row i and row j – considering only the elements to the left of column j.

It is imperative to note the direct dependence of $L(i,j)$ on $H(i,j)$, signifying that any non-zero element below the main diagonal in matrix H will correspondingly yield a non-zero element in matrix L with identical indices. Furthermore, each element $L(i,j)$ is contingent upon the elements located to its left within the same row. Consequently, the existence of a non-zero element $H(i,j)$ implies the existence of a non-zero element $L(i,j)$, which, in turn, can influence the non-zero status of all subsequent elements in the same row of L.

To analyze a concrete example, Figure 7-(a) illustrates the H matrix of a graph with six poses and two LC edges, where the black entries are the non-zero elements. For the symmetric H matrix, the figure provides nz_{low}, which is the non-zero count of the elements below or on the main diagonal. Figure 7-(b) illustrates the L matrix obtained from the Cholesky decomposition of H. The non-zero entries originating from the LC

(a) H, nz_{low}=65   (b) L, nz=129   (c) H$_P$, nz_{low}=65   (d) L$_P$, nz=103
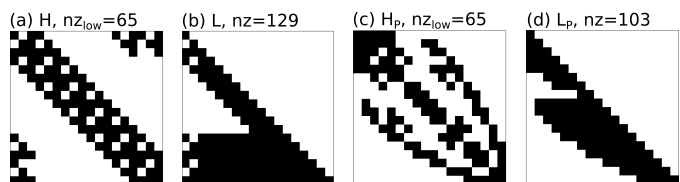

Fig. 7: An example of the H and L matrices for a graph with six poses and two LC edges (left) and how these matrices change when the RCM permutation is applied to H (right).

edges in the corner of matrix H resulted in corresponding non-zero elements in matrix L, subsequently leading to non-zero elements throughout the entire row to the right. Although one might argue that the LC edges have a negligible impact on the non-zero count of matrix H, this is obviously not the case for matrix L. The further the non-zero elements of H are from the main diagonal, the more non-zero entries they will yield in L.

To mitigate this problem, we employ the permutation solution introduced in Section III-C, which brings the elements of H closer to the main diagonal. The Reverse Cuthill–McKee (RCM) algorithm [55] computes the permutation vector π that defines the rearrangement of the rows of the identity matrix to obtain the permutation matrix P. The P obtained through RCM minimizes the bandwidth of a given matrix – i.e., how spread apart the elements are from the main diagonal. Note that accessing any element $H_P(i, j)$ is equivalent to accessing $H(\pi(i), \pi(j))$ and therefore it is not necessary to store and compute matrix H_P. Similarly, $b_P(i) = b(\pi(i))$.

Applying this algorithm to the example H matrix from Figure 7-(a) leads to the permuted H_P shown in Figure 7-(c), whose bandwidth is visibly reduced. Applying the Cholesky decomposition to H_P leads to the matrix L_P from Figure 7-(d) with 103 non-zero entries, about 20% less than L. Furthermore, Table II provides the resulting non-zero count of L and L_P for a graph with two LC edges, varying the number of poses in the range $40 - 320$. Also, in this case, the reduction in the non-zero count of L after permutation is at least 22%. Our system uses the RCM algorithm to determine the permutation matrix P and then solve the linear system as described in Listing 2. All steps involved in calculating Δx constitute a graph-based SLAM iteration. We empirically determined that the entries of $|\Delta x|$ are always smaller than 10^{-4} after three iterations, so we set $N_{iter}^{SLAM} = 3$.

D. Hierarchical SLAM Implementation

In Section III-E, the hierarchical graph-based SLAM method was introduced as an alternative approach to performing PGO, allowing to optimize graphs that exceed 440 poses. This approach utilizes the parameters d_{min} and $\Delta\psi_{min}$, which determine the inclusion of new poses in the sparse graph based on the robot's movement and rotation. Typically, in exploration scenarios, significant variations in the heading are rare, as the robot primarily rotates when encountering walls or obstacles. As a result, the parameter d_{min} has the greatest impact on the size of the sparse graph.

We investigate the influence of the d_{min} parameter on the accuracy of the optimized graph. Increasing the value of d_{min} reduces the size of the sparse graph, allowing for the mapping of larger environments. However, this also leads to a loss in capturing fine drone movements, resulting in decreased accuracy. To assess this impact, we conducted an experiment using a square loop corridor, creating an associated graph with 2000 poses, which significantly exceeds the limits of the graph-based SLAM algorithm when executed onboard. We varied the d_{min} parameter within the range of $0.1\,\mathrm{m}$ to $6.4\,\mathrm{m}$, as detailed in Table III, and measured the accuracy

TABLE III: Hierarchical graph-based SLAM.

$d_{min}(m)$	0.1	0.2	0.4	0.8	1.6	3.2	6.4
Correlation (%)	99.9	99.9	99.9	99.7	98.9	96.7	93.4
RMSE (cm)	0.04	0.12	0.27	0.59	24.3	46.7	84.0

of the resulting optimized poses for each case. To evaluate the accuracy, we computed the Pearson correlation coefficient between the optimized poses obtained using the hierarchical approach with varying d_{min} values and the poses derived from directly applying graph-based SLAM (performed on an external base station). As an additional metric, we also compare the root-mean-squared-error (RMSE) w.r.t. the directly optimized graph, considering only the x and y components of each pose. Table III shows that values of $d_{min} \leq 0.8$ provide almost the same accuracy as optimizing the graph directly with graph-based SLAM, leading to a correlation coefficient larger than 99% and an RMSE smaller than $1\,\mathrm{cm}$. Consequently, any value of d_{min} smaller than $0.8\,\mathrm{m}$ is appropriate for creating the sparse graph.

E. The Exploration Strategy and Corner Detection

In the following, we explain how the drone explores the environment and decides which areas are appropriate for acquiring scans. We mention that our mapping solution is completely independent of the type of trajectory and applicable to any environment. Thus, to demonstrate the capabilities of our system, we use a simple *exploration strategy* that drives a drone through the environment, always following the wall on the right. In case of no walls around, the drone moves forward. If a frontal wall or obstacle is detected, the drone changes direction to the left or right, depending on which direction is free – if both are free, the drone chooses left. Conversely, if a dead-end is detected, the drone lands, and the mission ends. This simple exploration strategy is configured with the target velocity of $0.5\,\mathrm{m/s}$ – correlated with the size of the rooms we explore.

Since our system should work autonomously in any environment, it must possess the ability to determine when a new scan should be acquired. Regions that exhibit rich textures, such as corners, are highly suitable for acquiring scans that facilitate precise scan-matching. To achieve this objective, we implemented a corner detector that takes a scan frame as input and utilizes the Hough transform [56] to identify all the straight lines defined by the points within the scan frame. The presence of any pair of lines that create an angle of at least $30°$ indicates that the scan frame represents a corner. The corner detector runs in the STM32 in less than $1\,\mathrm{ms}$.

F. The STM32 Application

The STM32 MCU is the manager of all the processes running onboard the nano-UAV. Even if it does not carry any heavy computation, it is responsible for off-loading it to the GAP9 via SPI communication. The application we developed on the STM32 is structured in three Free-RTOS tasks: *the mission task*, *the flight task*, and *the sensor task*. Overall, these tasks require less than $4\,\mathrm{kB}$ and only 2% of additional
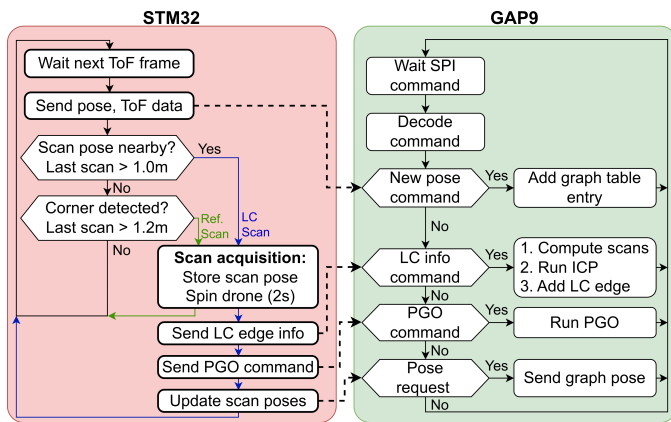
Fig. 8: The flow diagram of our software and the interaction between the STM32 and the GAP9.

CPU load in total. The sensor task communicates with the ToF sensors via I2C. It configures each sensor before the mission starts, fetches data from the ToF matrix sensors, and passes it to the other tasks. The flight task runs the wall-following exploration strategy introduced in Section V-E. However, other tasks can notify the flight process via a Free-RTOS queue to perform other maneuvers, such as stopping and spinning the drone to acquire a scan. The loops of the flight task and the sensor task run with a frequency of $15\,\mathrm{Hz}$.

The mission task manages the scan acquisition and the communication with the GAP9 using SPI packets. The flowchart from Figure 8–left presents a detailed illustration of the mission flow. In every iteration, the task fetches the ToF data from the sensor task and the current pose from the internal state estimator and sends this information to the GAP9. In the absence of any previous scans in the current location, if the current scan frame corresponds to a corner and the drone has traveled a minimum distance of $1.2\,\mathrm{m}$ from the last scan, the drone captures a reference scan. Then, the scan pose is stored in a structure called *scan pose list*, which stores the locations of all acquired scans. On the other hand, if the current location is actually revisited – i.e., the drone is closer than $0.6\,\mathrm{m}$ to one entry of the scan pose list – an LC scan is acquired, given at least $1\,\mathrm{m}$ from the last scan. The STM32 informs the GAP9 about the LC and then sends a PGO command. Lastly, the STM32 updates the scan pose list, fetching the updated scan pose values from the GAP9. The loop of a mission task runs at $7.5\,\mathrm{Hz}$, skipping every second ToF frame.

Note that the scan acquisition is identical for the reference and LC scans, and only the subsequent steps differ. During a scan, the flight task is spinning the drone by $45°$, while the mission task continues sending new poses to the GAP9 – this was omitted in Figure 8 for the sake of readability. Furthermore, it is important to impose a minimum distance between scans. Conducting consecutive scans at the same location does not enhance the system's efficacy but leads to a substantial surge in memory utilization. The distance thresholds were determined experimentally.

G. The GAP9 Application

The STM32 MCU performs several crucial functions, including managing sensor communication, controlling the drone, and determining when to acquire new scans. In contrast, the GAP9 processor assumes a subordinate role by handling computationally intensive tasks. The STM32 sends SPI packets to the GAP9, and each packet consists of a command and a corresponding data field, with the interpretation of the data contingent upon the specific command type. The GAP9 continually awaits the arrival of a new SPI packet, upon which it proceeds to decode and execute the command. Four possible SPI commands are defined in the system, distinguished by a command ID. The first is the *new pose* command, signaling that a new pose should be added to the graph, along with its associated ToF data. The graph is stored in memory as a table (i.e., *the graph table*), where each graph table entry contains the pose ID, the timestamp, the pose values, and the ToF data from the four sensors. The structure of each graph table entry is presented in Table IV, and the total size of an entry is $84\,\mathrm{B}$. Note that one graph table entry carries all the necessary information to compute one scan frame.

The second command is the *LC information* command. Within this command, the STM32 communicates that an LC edge should be added to the graph, from an id j to id i, communicated in the SPI packet. The GAP9 application fetches the graph table entries from i to $i + 19$ and from j to $j + 19$ and then calculates their associated scans $\boldsymbol{S}_i$ and $\boldsymbol{S}_j$. Having the two scans, it then computes the LC edge measurement using ICP and stores it into the *LC edge list*. The third command defined within our system is the *PGO* command, which informs the GAP9 to optimize the existing poses in the graph, given the LC edge list. Our system always used the hierarchical graph-based SLAM for PGO, as it allows for mapping larger environments. When PGO completes, the graph table is updated with the new pose values. Optionally, the map is regenerated by combining the scan frames computed from every graph pose entry. Lastly, the fourth command is the *pose request*, which enables the STM32 to obtain the value of a particular pose in the graph table by communicating its ID in the SPI packet. This is necessary because the STM32 application must update the scan pose list after every PGO. Furthermore, it must also update the drone's state estimator with the updated value of the most recent pose. Figure 8 illustrates the behavior of each command and the interaction with the STM32. NanoSLAM represents the whole logic running in the GAP9, which stores the graph, fetches scans, uses scan-matching to add new LC edges, and exploits hierarchical PGO to correct the poses.

TABLE IV: Structure of a graph table entry

Field	Representation	Size
Pose ID	*int32*	4 B
Timestamp	*int32*	4 B
Pose	$3 \times$ *float*	12 B
ToF Data	$4 \times 8 \times$ *int16*	64 B

TABLE V: Execution time in ms of the ICP algorithm w.r.t. the scan size. CL is the cluster with 8+1 cores, while the FC is the single-core fabric controller of the GAP9 SoC.

Scan size	128	256	384	512	640	768	896	1024
CL	3	10	21	36	55	79	107	138
FC	16	63	141	249	386	557	758	990
Speedup	**5.33**	**6.3**	**6.71**	**6.91**	**7.01**	**7.05**	**7.08**	**7.11**

TABLE VI: The execution time in ms of the Cholesky decomposition. CL is the cluster with 8+1 cores, while the FC is the single-core fabric controller of the GAP9 SoC.

Nr. of poses	20	80	140	200	260	320	380	440
CL (8 cores)	0.51	2.52	5.51	9.39	14.71	20.18	26.88	34.96
FC (1 core)	0.86	8.76	24.88	49.72	82.74	124.1	173.9	232.0
Speedup	**1.68**	**3.47**	**4.52**	**5.29**	**5.63**	**6.15**	**6.47**	**6.64**
Speedup SLAM	**1.28**	**2.24**	**2.97**	**3.55**	**4.04**	**4.43**	**4.79**	**5.08**

VI. PERFORMANCE ANALYSIS

In this section, we provide a breakdown of the execution time of our algorithms. We mainly evaluate the ICP, graph-based SLAM, and its hierarchical extension, emphasizing the benefits of the eight-core parallelization. Lastly, we also provide a power breakdown for the individual stages of graph-based SLAM. Power measurements are conducted upstream of the buck converter on the GAP9 co-processor deck, which receives a voltage supply of 4 V and generates an output voltage of 1.8 V intended for supplying the GAP9 SoC. The GAP9 always operates at the maximum frequency – i.e., 400 MHz for both FC and CL.

A. Execution Time of ICP

Table V shows the ICP execution time as a function of the scan size. The first line of the table provides the total execution time when the algorithm is parallelized and computed with the aid of the CL. To highlight the benefit of parallelization, we also provide the execution time when ICP runs entirely in the FC – i.e., second table line. We notice a speedup, defined as the ratio between the execution time on the FC and the CL, that increases with the scan size. This is due to the memory transfer overhead; for larger scan sizes, the computation time of the correspondences is significantly higher than the time necessary to transfer the scans from L2 to L1, which the CL can access. For a scan size larger than 640, the achieved speedup is above seven with eight cores. Furthermore, for the scan size used for the scope of this paper (i.e., 640), the ICP executes in 55 ms.

B. Execution Time of Graph-based SLAM

In the previous section, we have explained how every constitutive stage of graph-based SLAM is implemented. In the following, we provide the execution time and complexity of every stage. In this regard, we first analyze the Cholesky decomposition, which is the most complex and computationally intensive part of graph-based SLAM. As explained in Section V-C, the decomposition is offloaded to the CL of GAP9 to accelerate its execution through parallelization over eight cores. To highlight the advantages of parallelization, we
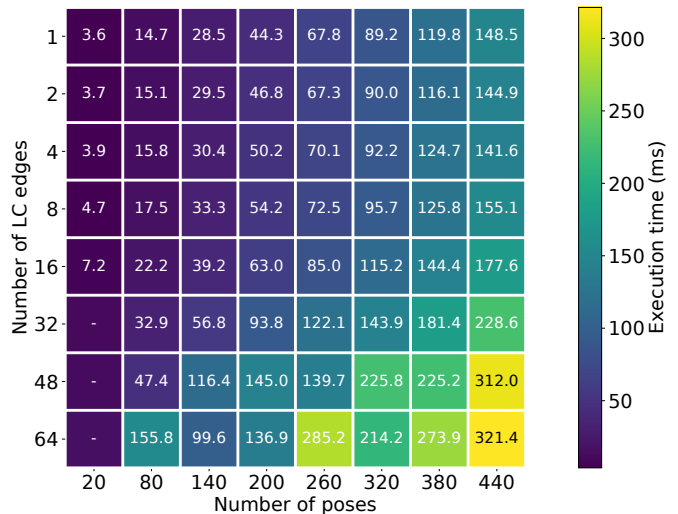

Fig. 9: The execution time in ms of graph-based SLAM for multiple configurations of the number of poses and LC edges.

also evaluate the execution time of the decomposition solely on the FC using a single core. Subsequently, we analyze the resulting measurements in comparison to those obtained on the CL. Similarly to the example analyzed in Section V-C, we consider a graph with two LC edges, and we vary the number of poses in the range of 20 – 440 with a step of 60. Table VI provides the results of this comparative analysis, showing the execution time as a function of the number of poses. The *Speedup* line gives the ratio between the execution time on the FC and the CL for each pose number. The maximum speedup is achieved for 440 poses, reducing the execution time from 232 ms to 34.96 ms and resulting in a speedup of 6.64. Overall, the table shows an increasing trend of the speedup with the number of poses. This is because the overhead for moving the input matrix to L1 (accessible by the CL) becomes more and more negligible w.r.t. the computation time for larger graphs. Furthermore, the *Speedup SLAM* line shows how many times the whole graph-based SLAM algorithm is accelerated when the Cholesky decomposition is offloaded to the CL. The maximum speed is 5.08, achieved with 440 poses.

Table VII presents the execution time analysis of the main stages of graph-based SLAM. The stages are paired with step 3 of Listing 2. The experiment involves a graph comprising two LC edges and a varying number of poses ranging from 20 to 440. The initial four rows of the table provide detailed information about the individual execution times for each stage within a single iteration, while the subsequent row presents the total iteration time. Notably, the Cholesky decomposition

TABLE VII: The execution time in ms of graph-based SLAM.

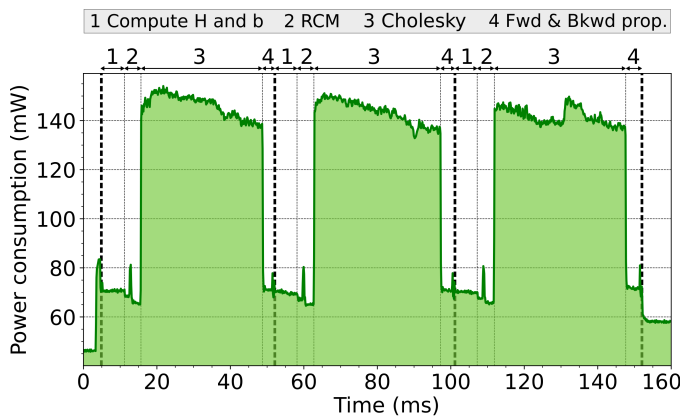| Nr. of poses	20	80	140	200	260	320	380	440
H and b	0.3	1.1	2	2.9	3.9	4.6	5.4	6.3
RCM	0.2	0.9	1.5	2.2	2.9	3.5	4.2	4.9
Cholesky	0.5	2.5	5.5	9.4	14.7	20.2	26.9	35.0
Fwd+Bwd	0.2	0.8	1.3	1.8	2.3	2.8	3.3	3.8
Iter. time	**1.3**	**5.4**	**10.4**	**16.3**	**23.7**	**31.1**	**39.9**	**50**
Total (3 iter.)	**3.7**	**15.1**	**29.5**	**46.8**	**67.3**	**90**	**116.1**	**144.9**

Fig. 10: The power curve associated with the graph-based SLAM execution for 440 poses, showing the power consumption during every main stage of the algorithm.

TABLE VIII: Power and energy consumption of ICP.

Scan Size	128	256	384	512	640	768	896	1024
Avg. power (mW)	121.5	149.5	165.0	169.9	172.5	175.2	176.3	177.6
Energy (mJ)	0.54	1.76	3.77	6.53	10.07	14.28	19.17	25.07

TABLE IX: Power and energy consumption of graph SLAM.

Nr. of poses	20	80	140	200	260	320	380	440
Avg. power (mW)	72.3	86.3	94.7	88.9	108.7	111.6	114.9	119.3
Energy (mJ)	0.31	1.5	3.1	4.38	7.8	10.73	14.29	18.2

where the size of the sparse graph is 162 poses. Assuming that the drone is flying with a constant velocity through the maze, the size of the subgraphs is about the same. Under the assumption of a uniform subgraph size, the total execution time is $t_{sg} + (M-1)t_{subgraph}$, where t_{sg} is the time required to optimize the sparse graph and $t_{subgraph}$ is constant.

C. Power Analysis

Table VIII shows the power and energy consumption of the ICP as a function of the scan size. An observable rising pattern in the average power is observed, attributed to the correspondence calculation occupying a larger proportion of the overall execution time for larger scan sizes. The maximum power consumption is $177.6\,\mathrm{mW}$ for a scan size of 1024. However, for the scan size that we use (i.e., 640), the average power and energy consumption are $172.5\,\mathrm{mW}$ and $10.07\,\mathrm{mJ}$, respectively. In conclusion, for every LC, the system consumes about $10\,\mathrm{mJ}$ plus the energy consumed to optimize the graph.

In Figure 10, the power trace of the GAP9 deck during the execution of graph-based SLAM is presented. The experiment involves 440 poses and 2 LC edges. The labels positioned above the plot represent the four primary stages of the algorithm: calculation of matrices $\boldsymbol{H}$ and $\boldsymbol{b}$, computation of the RCM permutation, Cholesky decomposition, and solution computed through forward and backward propagation. An observation can be made that the power consumption is notably higher during the Cholesky decomposition phase, primarily due to the activity of the CL. The peak of the power curve reaches $153\,\mathrm{mW}$, while the average power value amounts to $119.3\,\mathrm{mW}$. When the CL is inactive, and the FC solely handles the computation, the power instant tends to remain below $80\,\mathrm{mW}$. The total energy consumed for executing the graph-based SLAM is calculated to be $18.2\,\mathrm{mJ}$. In Table IX, the average power and energy are tabulated for the experiment conducted with various numbers of poses ranging from 20 to 440. The average power shows a monotonic trend, decreasing with a smaller number of poses due to reduced Cholesky decomposition execution time.

accounts for approximately 40% to 70% of the total iteration time. Additionally, it is observed that all stages, except for the Cholesky decomposition, exhibit linear complexity. Although the conventional decomposition has a complexity of $O(N^3)$, the numbers in Table VII demonstrate a purely quadratic relationship with the number of poses, showing a correlation of 99.9% with a second-order polynomial fit. This is due to our efficient implementation that exploits the sparsity properties. The last row provides the total execution time of the three iterations. This is approximately equal to the iteration time multiplied by three, but inter-iteration differences are possible due to different non-zero counts of the $\boldsymbol{H}$ and $\boldsymbol{L}$ matrices.

In the next experiment, we investigate the impact of varying both the number of poses and LC edges on the execution time of graph-based SLAM. The ranges considered for the number of poses and LC edges are 20 – 440 and 1 – 64, respectively, as depicted in Figure 9. The figure illustrates that increasing either parameter leads to an increase in the execution time, although the relationship is not strictly monotonic. Interestingly, for instance, the scenario with 64 LC edges demonstrates a faster execution time with 320 poses compared to 260 poses. This behavior can be attributed to the Cholesky decomposition's execution time, which is affected by the non-zero count of the matrix $\boldsymbol{L}$, determined by the permutation obtained through RCM. Since RCM does not guarantee the same non-zero reduction for all matrices, some configurations could benefit from a higher non-zero reduction in $\boldsymbol{L}$ after applying the permutation to $\boldsymbol{H}$. Given the $128\,\mathrm{kB}$ of L1 available to the CL, the graph-based SLAM algorithm can optimize at most 440 poses at a time, requiring $321\,\mathrm{ms}$ with 64 LC edges and $148.5\,\mathrm{ms}$ with one LC edge.

The execution time of the hierarchical graph-based SLAM strongly depends on the structure of the sparse graph and subgraphs. Using a large value for d_{min} would result in a small, sparse graph and few large subgraphs. On the other hand, using a small d_{min} would result in a large sparse graph and many small subgraphs. As a numerical example, for a graph of 2000 poses associated with a square loop corridor, a $d_{min} = 0.3\,\mathrm{m}$ results in a total execution time of $406\,\mathrm{ms}$,

VII. IN-FIELD EXPERIMENTS

In this section, we evaluate the algorithms introduced in Section III and recall that NanoSLAM is the framework that leverages hierarchical PGO to optimize the graph and correct the drone's trajectory while considering the LC edges provided by ICP. We, therefore, present three main classes of results: *(i)* an evaluation of the rotation and translation error
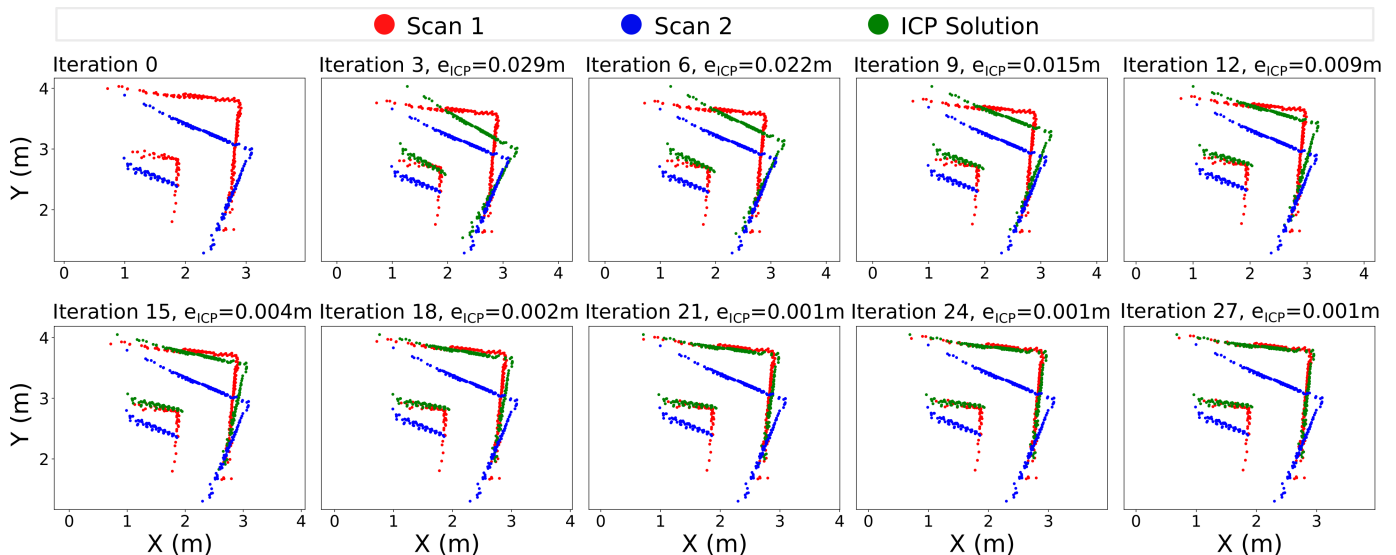
Fig. 11: A breakdown of the ICP algorithm, indicating the solution found by the algorithm after every three iterations. Assuming a target overlapping error of $\approx 0.001\,\mathrm{m}$, the algorithm converges in about 20 iterations.

achieved by the scan-matching algorithm; *(ii)* an investigation on how NanoSLAM improves the trajectory estimation; *(iii)* coherent maps generated out of the pose graph and the ToF measurements. Our results are experimentally acquired and demonstrate the effectiveness of our closed-loop system that leverages NanoSLAM and carries the computation entirely onboard. The ground truth (GT) used in our evaluation is provided by the Vicon Vero 2.2 motion capture system (mocap) installed in our testing arena. To assess our system's localization and mapping capabilities, we build mazes of different complexities out of $1\,\mathrm{m} \times 0.8\,\mathrm{m}$ chipboard panels.

A. Scan-matching Evaluation

In the following, we analyze the scan-matching capabilities of the ICP algorithm. In this scope, we position the drone inside a $90°$ pipe of $1\,\mathrm{m}$ width made out of chipboard panels. The drone is then commanded to take off and acquire a scan – i.e., *Scan 1*. Then, we manually change the position of the drone by about $30\,\mathrm{cm}$ and $30°$ and repeat the same procedure to obtain *Scan 2*. The two scans are shown in *Iteration 0* of Figure 11. The drone position is changed to simulate the odometry drift that the drone normally acquires when it revisits a location and evaluates the ICP performance in matching two non-overlapping scans. Therefore, the resulting rotation and translation values by the ICP are compared with the ground truth – obtained out of the ground truth of each individual pose. We obtain a translation error of $e_T = 3.5\,\mathrm{cm}$ and a rotation error of $e_R = 2.3°$ – the reported translation error e_T represents the norm of the two components of the error – i.e., on x and y. To ensure the validity of the results, we repeated the experiment multiple times, always obtaining a translation error $e_T < 6\,\mathrm{cm}$ and a rotation error $e_R < 5°$.

Figure 11 shows the result of the ICP algorithm after every three iterations. We recall that ICP aims to determine the rotation and translation that, once applied to *Scan 2*, results in an optimal overlapping with *Scan 1*. This is represented

by the green curve, which represents the scan obtained by applying the current ICP estimate to *Scan 2*. Furthermore, e_{ICP} represents the arithmetic mean of the Euclidean distances between each correspondence pair of the red and green curves. This metric evaluates the overlapping degree between *Scan 1* and the ICP solution applied to *Scan 2*, and it is a good indicator of when the algorithm should stop. We observe that in this case, as well as in other experiments we conducted, the ICP solution that leads to $e_{ICP} \approx 0.001\,\mathrm{m}$ is found in about 20 iterations. We note that the rotation and translation errors (e_T, e_R) provide a quantitative indication of the precision of the solution found by ICP. Conversely, e_{ICP} is only an intrinsic parameter indicating the convergence progress. For example, if the input scans are affected by large amounts of noise or biases, e_{ICP} could still indicate a small value, while the actual transformation found by ICP is inaccurate.

B. SLAM Results

In the following, we demonstrate our system's capabilities to correct trajectories and generate coherent maps in three different mazes of increasing complexity. The first maze
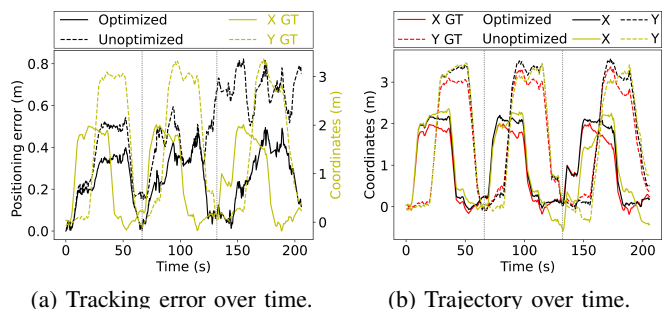
(a) Tracking error over time. (b) Trajectory over time.

Fig. 12: (a) The tracking error on both the x and y axis and (b) the evolution of the optimized (i.e., with NanoSLAM) and unoptimized trajectories represented against the ground truth.

consists of a square loop corridor similar to the one used as an example in Section III-D, whereas the latter two mazes exhibit greater complexity and are illustrated in Figure 13.

1) Maze 1: We start with a simple circular maze (i.e., *Maze 1*), shown in Figure 15a. The drone's mission starts from the bottom left corner and flies three laps – i.e., cycle through the maze three times, relying on the wall following strategy, introduced in Section V-E. During the first lap, the drone identifies every corner and acquires a reference scan in each of the four. In the second and third laps, the drone acquires new scans in the revisited corners, creating LC constraints with the reference scans and adding new LC edges to the graph. To allow for a consistent comparison between the unoptimized poses (i.e., no drift correction) and the optimized poses (i.e., with NanoSLAM), we only perform graph optimization at the end of the mission. In this way, we show the benefits of using NanoSLAM on data from the same mission.

We define as *trajectory* the set of all poses acquired during a mission and as positioning error the pose-wise Euclidean distance between the poses and their GT. Figure 12a shows the positioning error (in black) of the optimized and unoptimized trajectories, calculated by subtracting the GT from the optimized and unoptimized poses, respectively. The vertical grey lines indicate when the drone passes again through the starting point. Furthermore, in the same figure, we represent the x and y components of the trajectory over time (in yellow) to highlight a pattern between the positioning error and the trajectory. While the error of the unoptimized trajectory drifts unbounded, the optimized trajectory shows a rather repetitive pattern, with the error being zero every time the drone crosses the starting point. This is because the reference scan (of *pose 0*) acquired right after take-off is error-free and, therefore, any LC constraint between a *pose k* and *pose 0* will correct the positioning error of *pose k* almost completely – the correction is only bounded by the accuracy of the ICP.

Despite using NanoSLAM, the positioning error takes considerable values of about $0.4\,\mathrm{m}$ throughout one lap. A fundamental assumption when using SLAM is that the odometry errors are increasing slowly [57], and therefore the poses associated with the reference scans acquired at the beginning of the mission are accurate. However, this assumption does not hold in our case, as the poses of the second and third reference scans already have errors of up to $0.4\,\mathrm{m}$, and these errors will represent a lower bound for a future LC correction. Throughout a lap, the positioning error increases while the drone moves toward the top left corner of the maze and decreases during the
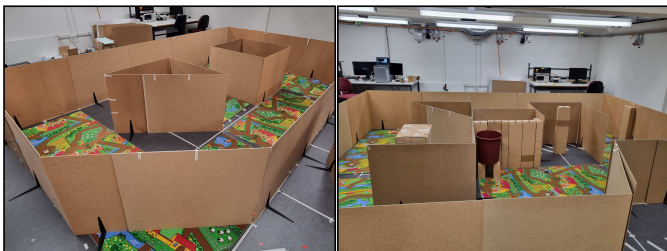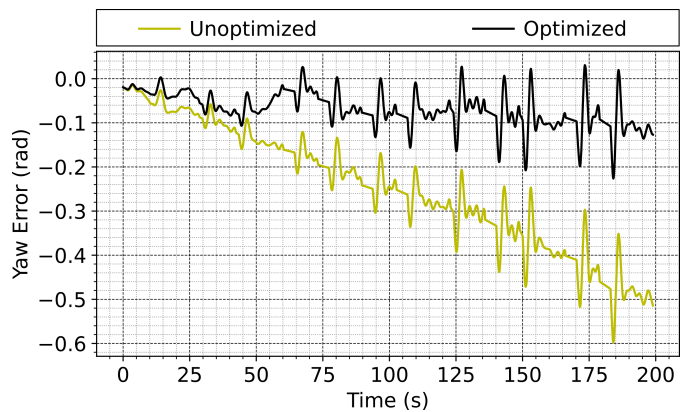

Fig. 14: The heading error with and without NanoSLAM.

last half of every lap. In other words, forward movement on the x and y axis increases the positioning error, while a backward movement along one axis decreases the positioning error. This translates into a direction-dependent odometry bias that is positive for forward movements and negative for backward movements. Since the takeoff position is always $(0, 0)$, the effect of the bias results in a scaled trajectory w.r.t. the GT.

The scaling effect can also be seen in Figure 12b, which shows information from the same mission but represents the x and y components of the optimized and unoptimized trajectories as well as the GT. The optimized trajectory (black) is very similar to the ground truth (red), but scaled by a factor which we determined to be $\approx 11\%$. On the other hand, we stress that the shape of the unoptimized trajectory (yellow) is often very different compared to the ground truth, which proves the effectiveness of NanoSLAM in correcting the trajectory and making it match the ground truth. So far, we have proved that the errors uncorrectable by NanoSLAM (i.e., the direction-dependent drift) are deterministic, and simply scaling the poses obtained from the drone's state estimator by 0.9 mitigates their effect. To demonstrate that this scaling actor generalizes for any environment, we apply the same correction for all mazes considered for our experiments and presented later in this section. The most likely cause of the direction-dependent errors is the down-pointing optical flow camera onboard the drone, which estimates the drone's velocity and enables the state estimator to determine the position by integrating the velocity. However, since the drone tilts when moving in a particular direction, this also rotates the camera frame, which is no longer parallel to the ground and leads to errors.

Next, we also analyze the capability of NanoSLAM to correct the heading estimation error (i.e., yaw). Figure 14 shows the yaw estimation error for the optimized and unoptimized poses over time. The error curves were again computed with the aid of the GT. One can notice that the heading estimation tends to drift unbounded for the unoptimized trajectory. Furthermore, the spikes in the error curves are associated with the scan acquisition when the drone rotates by $45°$ and subsequently returns to its initial heading. While these error spikes are also visible in the yaw corrected with NanoSLAM, the estimate's mean is stable, only exhibiting a steady state error mainly bounded by the precision of ICP.

Fig. 13: Maze 2 (left) and Maze 3 (right).

(a) Maze 1 layout.

(b) Trajectory with / without NanoSLAM.

(c) Trajectory with 1-LC NanoSLAM.

(d) Map without optimization.

(e) Map with NanoSLAM.
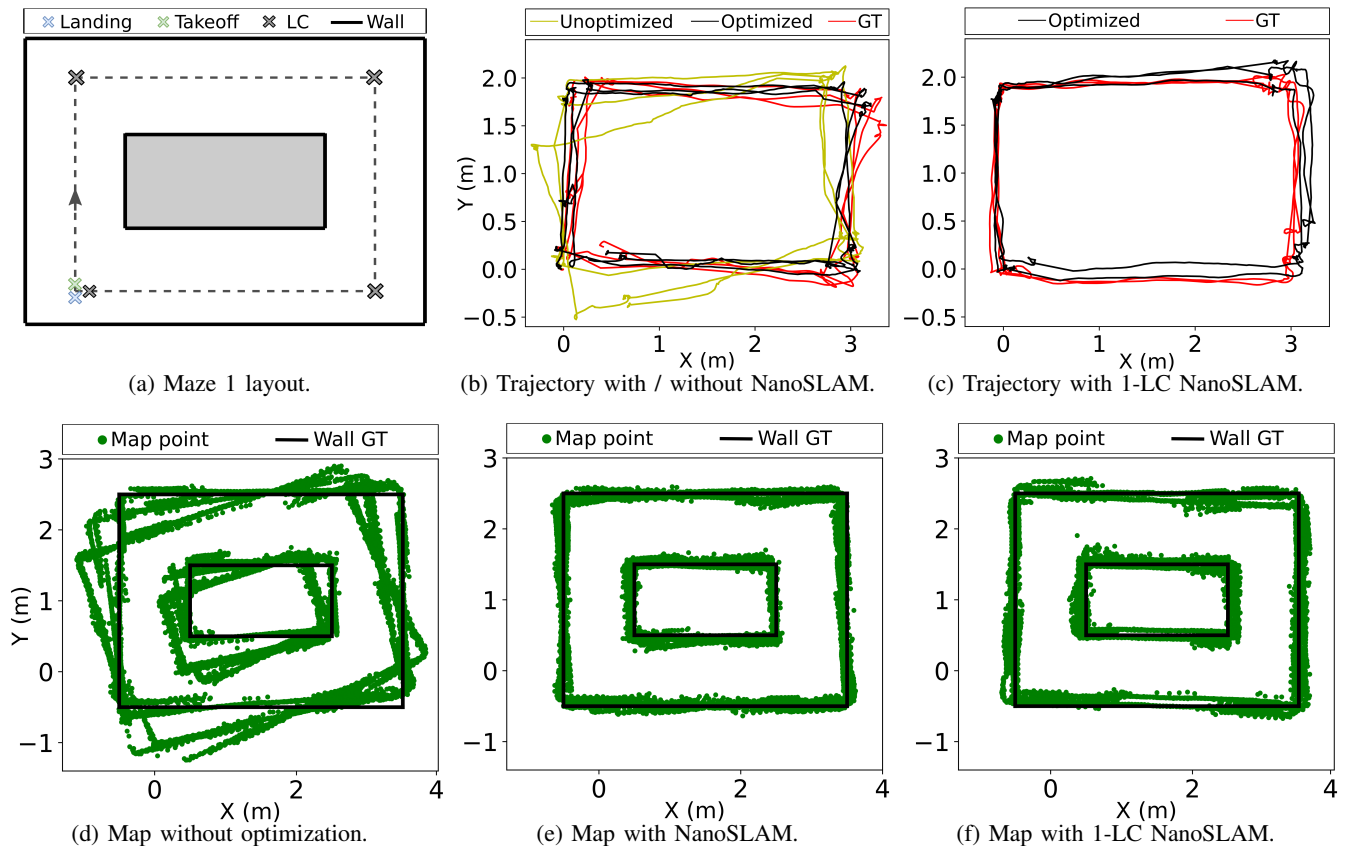
(f) Map with 1-LC NanoSLAM.

Fig. 15: Maze 1: (a) Illustration of the maze layout, showing the takeoff and landing locations and where an LC is performed. (b)-(c) The trajectories obtained with the two optimization approaches (in black) and the GT (in red). (d)-(f) The dense maps generated based on the unoptimized, NanoSLAM-corrected, and 1-LC NanoSLAM-corrected trajectories.

In the presented experiments within this section, we conducted optimization according to the approach detailed in Section III, wherein an LC edge is incorporated into the graph upon acquisition of an LC scan. With the NanoSLAM methodology that we introduced, the LC edges integrated into the graph are not subject to removal. Consequently, the LC edges count can only increase throughout the mission, and as shown in the experiments conducted in Section VI, this can even triple the execution time of PGO. For this purpose, we also analyze the *1-LC NanoSLAM*, which always discards the LC edge after optimization. Therefore, within the 1-LC NanoSLAM, the graph will always have one LC edge. Although this lightweight approach discards the prior constraints and consequently cannot guarantee future alignment of previously matched regions, we consider it a compelling exploration due to its superior scalability when many LCs are performed. In the following, we analyze the trajectory correction and mapping performance of both optimization techniques: NanoSLAM and 1-LC NanoSLAM.

$$RMSE_{pos} = \sqrt{\frac{\sum_{i=0}^{N-1} \|\boldsymbol{x}_i - \boldsymbol{x}_i^{GT}\|^2}{N}} \qquad (6)$$

Figure 15b shows the unoptimized trajectory, the trajectory optimized with NanoSLAM and how they compare to the GT. Although the unoptimized trajectory exhibits a noticeable deviation from the GT, we observe a substantial alignment

between the optimized trajectory and the GT, providing compelling evidence of the efficacy of NanoSLAM. While the 1-LC NanoSLAM trajectory shown in Figure 15c also leads to satisfactory results, we notice a trajectory misalignment within the laps due to the LC constraint relaxation. We introduce a quantitative metric for evaluating how close each trajectory is to the ground truth, which we call positioning root-mean-squared-error (RMSE). Given an arbitrary trajectory represented by the poses $\boldsymbol{x}_0 \ldots \boldsymbol{x}_{N-1}$, and the corresponding set of ground truth poses $\boldsymbol{x}_0^{GT} \ldots \boldsymbol{x}_{N-1}^{GT}$, the positioning RMSE is calculated as in Equation 6. The formula uses a reduced pose representation, considering only each pose's x and y components. Applying Equation 6 for the trajectories in Figure 15b, we obtain a positioning RMSE of $0.46\,\mathrm{m}$ for the unoptimized trajectory and $0.146\,\mathrm{m}$ for the trajectory optimized with NanoSLAM, showing a reduction in the positioning RMSE of about three times. Despite the inter-lap trajectory misalignment in Figure 15c, it leads to a positioning RMSE of $0.18\,\mathrm{m}$, which is 23% higher than the trajectory optimized with NanoSLAM.

After proving the capability of NanoSLAM to correct trajectories, we further to explore the mapping performance. Applying Equation 1 for each trajectories from Figures 15b – 15c, we obtain the maps shown in Figures Figures 15d – 15f. The maps in Figure 15d and Figure 15e are associated with the unoptimized and optimized trajectories in Figure 15b, while the map in Figure 15f corresponds to the trajectory optimized

(a) Maze 2 layout.

(b) Trajectory with / without NanoSLAM.

(c) Trajectory with 1-LC NanoSLAM.

(d) Map without optimization.

(e) Map with NanoSLAM.
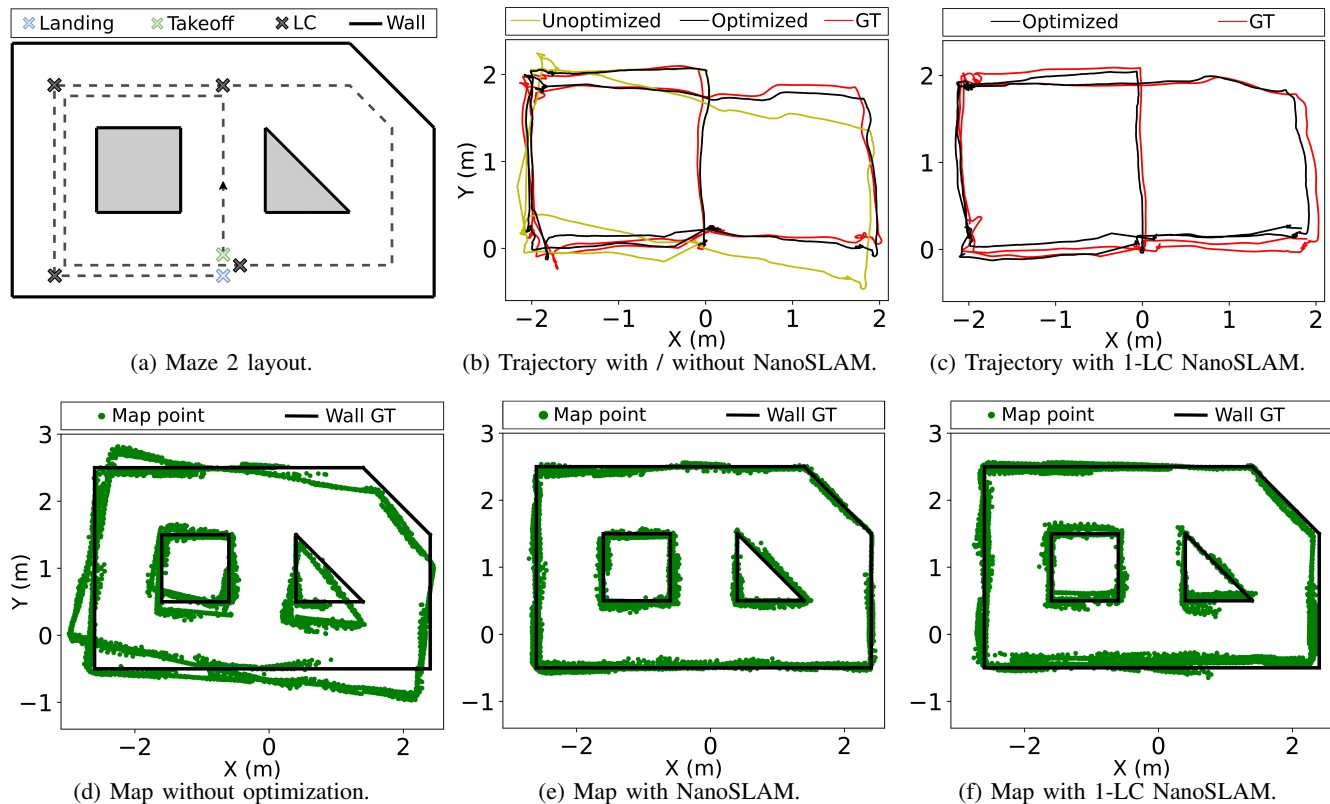
(f) Map with 1-LC NanoSLAM.

Fig. 16: Maze 2: (a) The maze layout. (b)-(c) The trajectories obtained with the two optimization approaches (in black) and the GT (in red). (d)-(f) The dense maps generated based on the unoptimized, NanoSLAM-based, and 1-LC NanoSLAM-based trajectories.

with 1-LC NanoSLAM. Visibly, Figure 15d shows the poorest accuracy, as no graph optimization is used, and therefore the position and heading drift heavily impact the alignment of the three laps. The map corrected with NanoSLAM is the most accurate w.r.t. the GT. Similarly, the map computed from the 1-LC NanoSLAM-optimized trajectory is definitely usable, but due to imposing one constraint at a time in the optimization process, not all corners corresponding to the three laps are perfectly aligned. We also propose a quantitative metric to analyze the mapping accuracy, which we call *the mapping RMSE*. This metric first calculates the length of the projection to the closest straight line for every point in a given dense map – the extensions of the maze lines are also considered. The mapping RMSE represents the RMSE of all projection lengths, as shown in Equation 7. This metric penalizes how far each map point is from a wall, and in a noise-free case, when all map points are on a maze line, the mapping RMSE is zero. Applying Equation 7 on the three maps from Figures 15d – 15f leads to $21.5\,\mathrm{cm}$, $5.8\,\mathrm{cm}$, and $7.3\,\mathrm{cm}$ for the cases with no optimization, NanoSLAM and 1-LC NanoSLAM, respectively. This proves that NanoSLAM reduces the mapping RMSE by about 3.7 times.

$$RMSE_{map} = \sqrt{\frac{\sum_{i=0}^{N-1} \left(\min_{\forall w \in W} dist(w, \boldsymbol{p}_i)\right)^2}{N}} \quad (7)$$

2) Maze 2: In the following, we present the trajectories and maps obtained by employing identical optimization method-

ologies on a marginally more intricate maze, as illustrated in Figure 13-(left). A top-view layout of the maze depicted in Figure 16a reveals the presence of non-straight-angle walls, distinguishing it from Maze 1. The trajectory followed by the drone is represented in Figure 16a, where the green cross marks the take-off point, and the arrow indicates the flying direction. Since only the left half of the maze is revisited, the LC is only performed four times, as indicated by the grey crosses. Table X presents the positioning and mapping RMSE for the experiments in Maze 2. We report a reduction of 67% and 64% in the positioning RMSE when applying NanoSLAM and 1-LC NanoSLAM, respectively, compared to the scenario without any optimization. Looking at the maps from Figures 16d-Figures 16f, we notice a significant heading drift that rotates the map w.r.t. the GT when no optimization is performed.

Mapping with the 1-LC NanoSLAM approach leads to a mapping RMSE of $7.5\,\mathrm{cm}$, about 53% smaller than the case without optimization. Employing NanoSLAM reduces the mapping RMSE even further, to $4.5\,\mathrm{cm}$, representing a reduction of 72% compared to the case without optimization.

Metric	No SLAM	NanoSLAM	1-LC NanoSLAM
Positioning RMSE	32.6 cm	10.7 cm	11.9 cm
Mapping RMSE	16.0 cm	4.5 cm	7.5 cm

TABLE X: Positioning and mapping RMSE for Maze 2

(a) Maze 3 layout.

(b) Trajectory with / without NanoSLAM.

(c) Trajectory with 1-LC NanoSLAM.

(d) Map without optimization.

(e) Map with NanoSLAM.
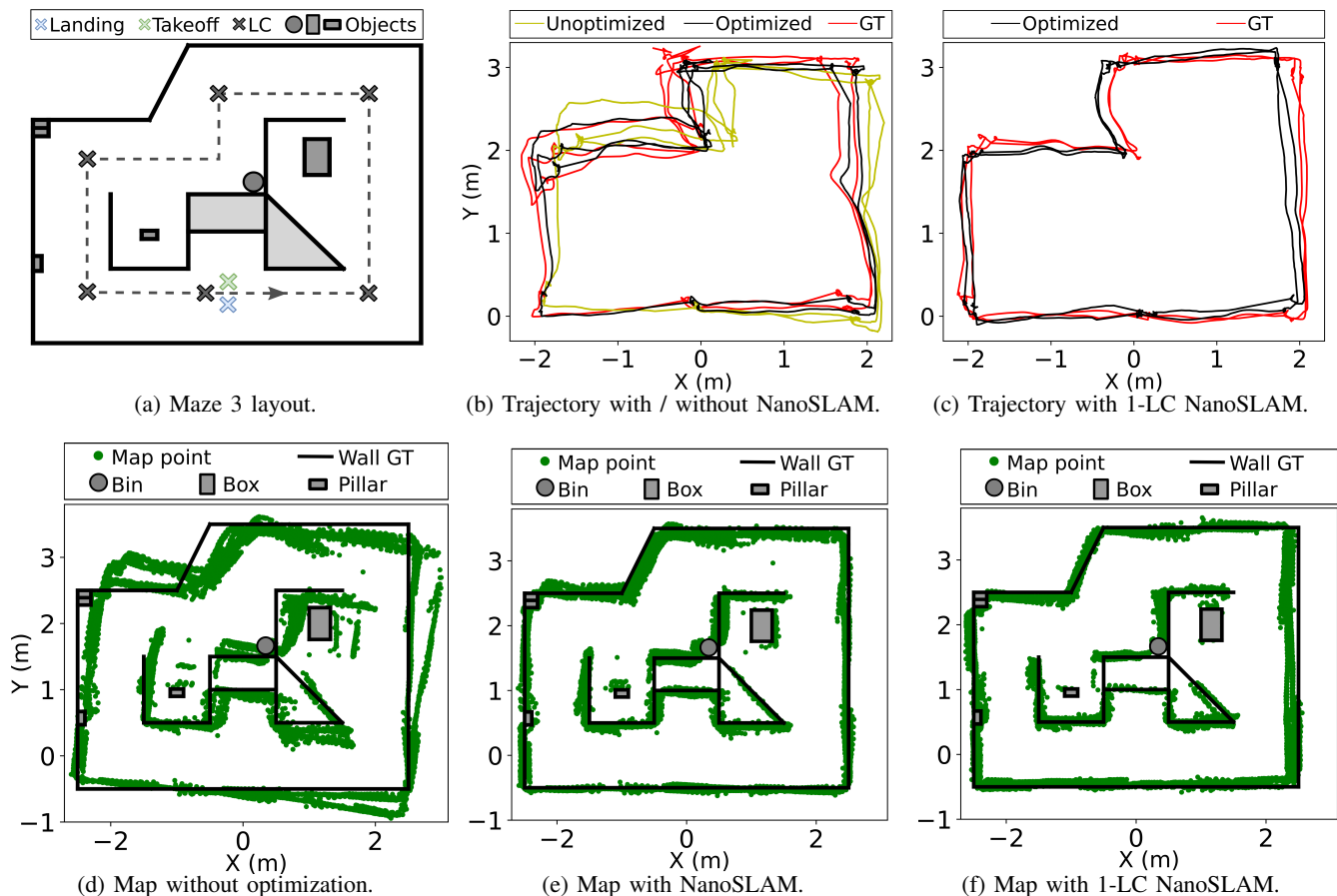
(f) Map with 1-LC NanoSLAM.

Fig. 17: Maze 3: (a) The maze layout that contains various objects. (b)-(c) The trajectories obtained with the two optimization approaches (in black) and the GT (in red). (d)-(f) The dense maps generated based on the unoptimized, NanoSLAM-corrected, and 1-LC NanoSLAM-corrected trajectories.

While the positioning RMSE is somewhat similar for the NanoSLAM and 1-LC NanoSLAM approaches, the relative difference in the mapping RMSE is more significant – about 40%. This observation is also evident in Figure 16f, where a comparison with the map depicted in Figure 16e reveals the presence of certain artifacts. For instance, the bottom side of the triangle-shaped wall is distorted, or the bottom maze wall appears thicker. This is again the effect of dropping previous LC edges and therefore failing to satisfy the constraints associated with all corners.

3) Maze 3: The last maze we propose is the most complex among the three because it also contains obstacles such as pillars, boxes, or a bin, and therefore it better generalizes a real-world indoor environment. Figure 13-(right) shows an image of the maze, while the top-view layout is shown in Figure 17a. The drone starts from the middle position marked with the green cross and performs two maze loops flying in a counterclockwise direction. Similar to the previous cases, throughout the first lap, the drone only acquires reference scans, while in the second lap, it closes the loop in every corner, as indicated by the grey crosses in Figure 17a. Table XI shows the positioning and mapping RMSE obtained with Maze 3. The optimized trajectories depicted in Figures 17b – 17c exhibit a positioning RMSE reduced by 65% and

60%, respectively, compared to the unoptimized trajectory. Figures 17d – 17f show the three maps, and unlike the experiments in the first two mazes, the maps generated with NanoSLAM and 1-LC NanoSLAM are very similar. One can still notice a misalignment on the right side of the map in Figure 17f, but it is not significant. This is also visible in the mapping RMSE, where the error of the NanoSLAM-based map is only 14% smaller than in the map obtained with the 1-LC NanoSLAM approach. Both approaches bring a significant improvement w.r.t. the map in Figure 17d (i.e., without optimization), reducing the mapping RMSE by 63% with NanoSLAM and 57% with 1-LC NanoSLAM.

In the NanoSLAM experiment, the optimization problem results in a graph with 1355 poses and five LC edges. The subgraph has a size of 109 poses, and the total hierarchical optimization requires 247 ms. For the experiment employing 1-LC NanoSLAM, the graph has 1293 poses, and the final optimization requires 223 ms. The graphs associated with

Metric	No SLAM	NanoSLAM	1-LC NanoSLAM
Positioning RMSE	44.1 cm	15.4 cm	17.3 cm
Mapping RMSE	20.3 cm	7.5 cm	8.7 cm

TABLE XI: Positioning and mapping RMSE for Maze 3

Maze 3 are larger than those of Maze 1 ($\approx$ 1200 poses) and Maze 2 ($\approx$ 850 poses). For the experiment in Maze 3, running graph optimization results in an average power consumption of $69.1\,\mathrm{mW}$ and a total energy consumption of $17.06\,\mathrm{mJ}$ – note how optimizing a 1355 pose graph with the hierarchical graph-based SLAM results in the same energy consumption as optimizing a 440 pose graph with the direct graph-based SLAM. Overall, for every LC, NanoSLAM implies an average power consumption of $87.9\,\mathrm{mW}$ and an energy consumption of $27.13\,\mathrm{mJ}$ – accounting for both PGO and ICP.

C. Discussion

We discuss the limitations of our system and possible future improvements. We first recall the importance of odometry calibration to maximize mapping accuracy. The calibration approach depends on the type of sensors available on the platform and, thus, the odometry calibration is a platform-specific tuning. In the following, we estimate the maximum area that can be mapped with our system. This limitation mainly comes from the maximum size (i.e., 440 poses) of the sparse graph used by the hierarchical PGO. We attempt to perform such an estimation starting from Maze 3, the most complex environment we map in our work. Note that when mapping Maze 3, we let the drone fly two laps to acquire more maze details. On the other hand, this is not mandatory, and flying just one lap and closing the loop when the drone crosses again through the start would as well lead to good results. Furthermore, in our experiments, we employed a $d_{min} = 0.3\,\mathrm{m}$ when adding poses to the sparse graph to demonstrate that our system can work with large graphs. However, as shown in Table III, setting $d_{min} = 0.8\,\mathrm{m}$ results in almost the same optimization accuracy. Mapping one lap of Maze 3 implies the drone to travel about $14\,\mathrm{m}$, which results in 18 poses for the sparse graph when using $d_{min} = 0.8\,\mathrm{m}$. Therefore, it is possible to map an environment of about $13\,\mathrm{m}^2$ using 18 poses. By extrapolation, 440 poses would allow our system to map an environment of about $317\,\mathrm{m}^2$.

Another limitation of our system comes from the ToF sensors, which replace the conventional LiDARs. This substitution introduces several distinctions between the two technologies. Firstly, LiDARs typically possess a greater operational range, enabling robots to map distant areas of the environment. In contrast, our ToF sensor is restricted by a maximum range of $4\,\mathrm{m}$. Additionally, LiDARs often exhibit a higher angular resolution, resulting in measurement accuracy that is less reliant on the distance magnitude. Conversely, our employed ToF sensor operates with distinct zones, assuming that any detected obstacle within a zone is located at the zone's center. Considering the angle of one zone of $\theta_{zone} = 45°/8 = 5.625°$, the maximum distance error induced by this effect is approximately $e_{max} = d \cdot \tan(\theta_{zone}/2) \approx 0.05 \cdot d$. While this error is negligible for short distances, it exceeds $5\,\mathrm{cm}$ for distances beyond $1\,\mathrm{m}$. Consequently, these errors can lead to map misalignments, regardless of the effectiveness of PGO.

A potential extension that we mention involves expanding our system's capabilities to accommodate a swarm of drones. Given that ICP can derive transformations between scans captured by different drones, it would be feasible to merge pose

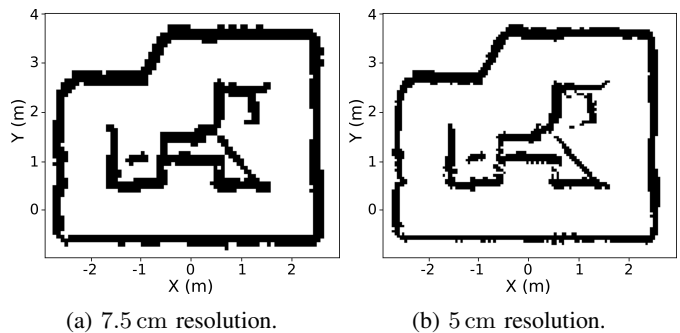(a) 7.5 cm resolution.   (b) 5 cm resolution.

Fig. 18: Binary occupancy maps resulted from Maze 3.

graphs from multiple drones and optimize them collectively to align their trajectories. This would enable faster mapping of an environment through parallel sensing of distinct areas by the swarm of drones, consequently reducing the overall mapping time. Another area of future work is to perform mapping in 3D. While our system assumes a flat environment, enabling absolute altitude estimation would allow to create a 3D map by acquiring depth measurements at various heights. Furthermore, implementing a mechanism that discards the depth measurements associated with moving people or obstacles would enable our system to operate even in dynamic environments. Lastly, another possible exploration entails leveraging the map generated by our approach to enable additional functionalities, such as optimal path planning. Converting the dense maps produced by our approach into binary maps – using the approach from [58] – would further reduce the memory footprint of the maps. For example, Figure 18 depicts the binary representation of the dense map from Figure 17e for resolutions of $7.5\,\mathrm{cm}$ and $5\,\mathrm{cm}$.

VIII. CONCLUSIONS

The paper presented NanoSLAM, a lightweight SLAM for autonomous drones, and the methodology to enable fully onboard mapping for small robotic platforms, which before was only possible with larger and more power-intensive computational platforms. NanoSLAM is the first system that enables SLAM for autonomous nano-UAVs and performs the computation entirely onboard exploring a novel RISC-V parallel low-power processor. We demonstrated the effectiveness of NanoSLAM by mapping three different real-world mazes, achieving a mapping error down to $4.5\,\mathrm{cm}$ and reducing the trajectory estimation error by up to 67%. The SLAM algorithm runs onboard in less than $250\,\mathrm{ms}$ and the whole mapping pipeline requires less than $500\,\mathrm{kB}$ of RAM. In spite of its remarkably lightweight configuration ($44\,\mathrm{g}$), the system introduced in this study achieves mapping accuracy on par with SoA approaches developed for standard-size UAVs, but consumes only $87.9\,\mathrm{mW}$. The system presented in this paper sets the foundation for increased autonomy in small form-factor robots with highly constrained hardware, thus introducing novel technology to the field of nano-UAVs. By enabling a comprehensive environmental map, this advancement opens up possibilities for advanced navigation solutions, including enhanced flight autonomy through optimal path planning.

Acknowledgments

References

[1] C. Wang, J. Wang, J. Wang, and X. Zhang, "Deep-reinforcement-learning-based autonomous uav navigation with sparse rewards," *IEEE Internet of Things Journal*, vol. 7, no. 7, pp. 6180–6190, 2020.

[2] P. Huang, L. Zeng, X. Chen, K. Luo, Z. Zhou, and S. Yu, "Edge robotics: Edge-computing-accelerated multirobot simultaneous localization and mapping," *IEEE Internet of Things Journal*, vol. 9, no. 15, pp. 14 087–14 102, 2022.

[3] L. Yang, J. Yu, S. Yang, B. Wang, B. J. Nelson, and L. Zhang, "A survey on swarm microrobotics," *IEEE Transactions on Robotics*, vol. 38, no. 3, pp. 1531–1551, 2021.

[4] O. Vermesan, R. Bahr, M. Ottella, M. Serrano, T. Karlsen, T. Wahlstrøm, H. E. Sand, M. Ashwathnarayan, and M. T. Gamba, "Internet of robotic things intelligent connectivity and platforms," *Frontiers in Robotics and AI*, vol. 7, p. 104, 2020.

[5] B. P. Duisterhof, S. Krishnan, J. J. Cruz, C. R. Banbury, W. Fu, A. Faust, G. C. de Croon, and V. J. Reddi, "Tiny robot learning (tinyrl) for source seeking on a nano quadcopter," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2021, pp. 7242–7248.

[6] S. Karam, F. Nex, O. Karlsson, J. Rydell, E. Bilock, M. Tulldahl, M. Holmberg, and N. Kerle, "Micro and macro quadcopter drones for indoor mapping to support disaster management," *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. 1, pp. 203–210, 2022.

[7] V. Niculescu, L. Lamberti, F. Conti, L. Benini, and D. Palossi, "Improving autonomous nano-drones performance via automated end-to-end optimization and deployment of dnns," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 11, no. 4, pp. 548–562, 2021.

[8] D. Liu, W. Bao, X. Zhu, B. Fei, T. Men, and Z. Xiao, "Cooperative path optimization for multiple uavs surveillance in uncertain environment," *IEEE Internet of Things Journal*, vol. 9, no. 13, pp. 10 676–10 692, 2021.

[9] B. P. Duisterhof, S. Li, J. Burgués, V. J. Reddi, and G. C. de Croon, "Sniffy bug: A fully autonomous swarm of gas-seeking nano quadcopters in cluttered environments," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2021, pp. 9099–9106.

[10] V. Niculescu, D. Palossi, M. Magno, and L. Benini, "Energy-efficient, precise uwb-based 3-d localization of sensor nodes with a nano-uav," *IEEE Internet of Things Journal*, 2022.

[11] H. Müller, N. Zimmerman, T. Polonelli, M. Magno, J. Behley, C. Stachniss, and L. Benini, "Fully on-board low-power localization with multizone time-of-flight sensors on nano-uavs," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023, pp. 1–6.

[12] S. Kisseleff, S. Chatzinotas, and B. Ottersten, "Reconfigurable intelligent surfaces in challenging environments: Underwater, underground, industrial and disaster," *IEEE Access*, vol. 9, pp. 150 214–150 233, 2021.

[13] D. Cristiani, F. Bottonelli, A. Trotta, and M. Di Felice, "Inventory management through mini-drones: Architecture and proof-of-concept implementation," in *2020 IEEE 21st International Symposium on" A World of Wireless, Mobile and Multimedia Networks"(WoWMoM)*. IEEE, 2020, pp. 317–322.

[14] S. Chen, W. Meng, W. Xu, Z. Liu, J. Liu, and F. Wu, "A warehouse management system with uav based on digital twin and 5g technologies," in *2020 7th International Conference on Information, Cybernetics, and Computational Social Systems (ICCSS)*. IEEE, 2020, pp. 864–869.

[15] A. Moura, J. Antunes, A. Dias, A. Martins, and J. Almeida, "Graph-slam approach for indoor uav localization in warehouse logistics applications," in *2021 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*. IEEE, 2021, pp. 4–11.

[16] G. Grisetti, R. Kümmerle, C. Stachniss, and W. Burgard, "A tutorial on graph-based slam," *IEEE Intelligent Transportation Systems Magazine*, vol. 2, no. 4, pp. 31–43, 2010.

[17] Y. Chen, S. Huang, L. Zhao, and G. Dissanayake, "Cramér–rao bounds and optimal design metrics for pose-graph slam," *IEEE Transactions on Robotics*, vol. 37, no. 2, pp. 627–641, 2021.

[18] H. Chao, Y. Cao, and Y. Chen, "Autopilots for small unmanned aerial vehicles: A survey," *International Journal of Control, Automation and Systems*, vol. 8, pp. 36–44, 2010. [Online]. Available: https://doi.org/10.1007/s12555-010-0105-z

[19] H. Zhou, Z. Hu, S. Liu, and S. Khan, "Efficient 2d graph slam for sparse sensing," in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2022, pp. 6404–6411.

[20] S. Srinara, C.-M. Lee, S. Tsai, G.-J. Tsai, and K.-W. Chiang, "Performance analysis of 3d ndt scan matching for autonomous vehicles using ins/gnss/3d lidar-slam integration scheme," in *2021 IEEE International Symposium on Inertial Sensors and Systems (INERTIAL)*. IEEE, 2021, pp. 1–4.

[21] R. Eyvazpour, M. Shoaran, and G. Karimian, "Hardware implementation of slam algorithms: a survey on implementation approaches and platforms," *Artificial Intelligence Review*, pp. 1–53, 2022.

[22] H. Shen, Q. Zong, B. Tian, X. Zhang, and H. Lu, "Pgo-liom: Tightly-coupled lidar-inertial odometry and mapping via parallel and gradient-free optimization," *IEEE Transactions on Industrial Electronics*, 2022.

[23] N. Chen, F. Kong, W. Xu, Y. Cai, H. Li, D. He, Y. Qin, and F. Zhang, "A self-rotating, single-actuated uav with extended sensor field of view for autonomous navigation," *Science Robotics*, vol. 8, no. 76, p. eade4538, 2023.

[24] V. Niculescu, H. Müller, I. Ostovar, T. Polonelli, M. Magno, and L. Benini, "Towards a multi-pixel time-of-flight indoor navigation system for nano-drone applications," in *2022 IEEE International Instrumentation and Measurement Technology Conference (I2MTC)*. IEEE, 2022, pp. 1–6.

[25] D. Rossi, F. Conti, M. Eggiman, A. Di Mauro, G. Tagliavini, S. Mach, M. Guermandi, A. Pullini, I. Loi, J. Chen *et al.*, "Vega: A ten-core soc for iot endnodes with dnn acceleration and cognitive wake-up from mram-based state-retentive sleep mode," *IEEE Journal of Solid-State Circuits*, vol. 57, no. 1, pp. 127–139, 2021.

[26] H. Müller, V. Niculescu, T. Polonelli, M. Magno, and L. Benini, "Robust and efficient depth-based obstacle avoidance for autonomous miniaturized uavs," *arXiv preprint arXiv:2208.12624*, 2022.

[27] G. Cerutti, R. Andri, L. Cavigelli, E. Farella, M. Magno, and L. Benini, "Sound event detection with binary neural networks on tightly power-constrained iot devices," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, 2020, pp. 19–24.

[28] N. Dilshad, A. Ullah, J. Kim, and J. Seo, "Locateuav: Unmanned aerial vehicle location estimation via contextual analysis in an iot environment," *IEEE Internet of Things Journal*, 2022.

[29] Z. Li, X. Zhao, Z. Zhao, and T. Braun, "Crowdfusion: Multi-signal fusion slam positioning leveraging visible light," *IEEE Internet of Things Journal*, 2023.

[30] R. Latif and A. Saddik, "Slam algorithms implementation in a uav, based on a heterogeneous system: A survey," in *2019 4th World Conference on Complex Systems (WCCS)*. IEEE, 2019, pp. 1–6.

[31] J. A. Placed, J. Strader, H. Carrillo, N. Atanasov, V. Indelman, L. Carlone, and J. A. Castellanos, "A survey on active simultaneous localization and mapping: State of the art and new frontiers," *IEEE Transactions on Robotics*, 2023.

[32] T. Suzuki, "Time-relative rtk-gnss: Gnss loop closure in pose graph optimization," *IEEE Robotics and Automation Letters*, vol. 5, no. 3, pp. 4735–4742, 2020.

[33] H. Cao, J. Xu, D. Li, L. Shangguan, Y. Liu, and Z. Yang, "Edge assisted mobile semantic visual slam," *IEEE Transactions on Mobile Computing*, 2022.

[34] M. Kasper, S. McGuire, and C. Heckman, "A benchmark for visual-inertial odometry systems employing onboard illumination," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2019, pp. 5256–5263.

[35] F. Causa, M. Asciolla, R. Opromolla, P. Molina, A. Mennella, M. Nisi, and G. Fasano, "Uav-based lidar mapping with galileo-gps ppp processing and cooperative navigation," in *2022 International Conference on Unmanned Aircraft Systems (ICUAS)*. IEEE, 2022, pp. 938–947.

[36] E. Ghignone, N. Baumann, and M. Magno, "TC-driver: A trajectory conditioned reinforcement learning approach to zero-shot autonomous racing," *Field Robotics*, vol. 3, no. 1, pp. 637–651, Jan. 2023. [Online]. Available: https://doi.org/10.55417/fr.2023020

[37] Y. Chang, K. Ebadi, C. E. Denniston, M. F. Ginting, A. Rosinol, A. Reinke, M. Palieri, J. Shi, A. Chatterjee, B. Morrell *et al.*, "Lamp 2.0: A robust multi-robot slam system for operation in challenging large-scale underground environments," *IEEE Robotics and Automation Letters*, vol. 7, no. 4, pp. 9175–9182, 2022.

[38] P. Mayer, M. Magno, and L. Benini, "Smart power unit—mw-to-nw power management and control for self-sustainable iot devices," *IEEE Transactions on Power Electronics*, vol. 36, no. 5, pp. 5700–5710, 2020.

[39] Y. Alghamdi, A. Munir, and H. M. La, "Architecture, classification, and applications of contemporary unmanned aerial vehicles," *IEEE Consumer Electronics Magazine*, vol. 10, no. 6, pp. 9–20, 2021.

[40] B. Zhou, H. Xu, and S. Shen, "Racer: Rapid collaborative exploration with a decentralized multi-uav system," *IEEE Transactions on Robotics*, 2023.

[41] S. Karam, F. Nex, B. T. Chidura, and N. Kerle, "Microdrone-based indoor mapping with graph slam," *Drones*, vol. 6, no. 11, p. 352, 2022.

[42] E. Jeong, S. Kang, D. Lee, and P. Kim, "Parsing indoor manhattan scenes using four-point lidar on a micro uav," in *2022 22nd International Conference on Control, Automation and Systems (ICCAS)*. IEEE, 2022, pp. 708–713.

[43] G. A. Kumar, A. K. Patil, R. Patil, S. S. Park, and Y. H. Chai, "A lidar and imu integrated indoor navigation system for uavs and its application in real-time pipeline classification," *Sensors*, vol. 17, no. 6, 2017. [Online]. Available: https://www.mdpi.com/1424-8220/17/6/1268

[44] F. Gao, W. Wu, W. Gao, and S. Shen, "Flying on point clouds: Online trajectory generation and autonomous navigation for quadrotors in cluttered environments," *Journal of Field Robotics*, vol. 36, no. 4, pp. 710–733, 2019. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.21842

[45] Z. Fang, S. Yang, S. Jain, G. Dubey, S. Roth, S. Maeta, S. Nuske, Y. Zhang, and S. Scherer, "Robust autonomous flight in constrained and visually degraded shipboard environments," *Journal of Field Robotics*, vol. 34, no. 1, pp. 25–52, 2017. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.21670

[46] H. Han and J. Siebert, "Tinyml: A systematic review and synthesis of existing research," in *2022 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*. IEEE, 2022, pp. 269–274.

[47] E. Tabanelli, G. Tagliavini, and L. Benini, "Dnn is not all you need: Parallelizing non-neural ml algorithms on ultra-low-power iot processors," *ACM Transactions on Embedded Computing Systems*, vol. 22, no. 3, pp. 1–33, 2023.

[48] J. Kühne, M. Magno, and L. Benini, "Parallelizing optical flow estimation on an ultra-low power risc-v cluster for nano-uav navigation," in *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2022, pp. 301–305.

[49] S. Tijmons, G. C. H. E. de Croon, B. D. W. Remes, C. De Wagter, and M. Mulder, "Obstacle avoidance strategy using onboard stereo vision on a flapping wing mav," *IEEE Transactions on Robotics*, vol. 33, no. 4, pp. 858–874, 2017.

[50] M. Gross and H. Pfister, *Point-based graphics*. Elsevier, 2011.

[51] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, 2003.

[52] S. Touchette, W. Gueaieb, and E. Lanteigne, "Efficient cholesky factor recovery for column reordering in simultaneous localisation and mapping," *Journal of Intelligent & Robotic Systems*, vol. 84, pp. 859–875, 2016.

[53] E. S. Quintana, G. Quintana, X. Sun, and R. van de Geijn, "A note on parallel matrix inversion," *SIAM Journal on Scientific Computing*, vol. 22, no. 5, pp. 1762–1771, 2001.

[54] G. Grisetti, R. Kümmerle, C. Stachniss, U. Frese, and C. Hertzberg, "Hierarchical optimization on manifolds for online 2d and 3d mapping," in *2010 IEEE International Conference on Robotics and Automation*. IEEE, 2010, pp. 273–278.

[55] A. Azad, M. Jacquelin, A. Buluç, and E. G. Ng, "The reverse cuthill-mckee algorithm in distributed-memory," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 22–31.

[56] P. Mukhopadhyay and B. B. Chaudhuri, "A survey of hough transform," *Pattern Recognition*, vol. 48, no. 3, pp. 993–1010, 2015.

[57] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza, *Introduction to autonomous mobile robots*. MIT press, 2011.

[58] T. Polonelli, C. Feldmann, V. Niculescu, H. Müller, M. Magno, and L. Benini, "Towards robust and efficient on-board mapping for autonomous miniaturized uavs," in *2023 9th International Workshop on Advances in Sensors and Interfaces (IWASI)*. IEEE, 2023, pp. 9–14.