Multi-Agent Collaborative Map Exploration

UoB Intelligent Robotics, *Group 10*

***Abstract*—We present the foundations for a combative tank game played between two teams of autonomous robots. We start by describing an algorithm for simulating robot detection, followed by a collaborative SLAM approach using map merging and sporadic rendezvous. This lays the groundwork needed for the teamwork required to seek out enemy tanks once a member of the team communicates the enemy tank's position. We also describe how robots will explore and travel around the map using a combination of local, mapless navigation with a finite state machine, and A* pathfinding through a (shared) map. We experiment and test these approaches in order to home in on accurate parameters for our use case.**

***Index Terms*—SLAM, map merging, navigation, A*.**

I. Introduction

THE simultaneous localisation and mapping (SLAM) problem has been a key area of research in robotics over the past decade. With the increase in availability of increasingly complex robots, demand for faster and more accurate SLAM techniques has increased, and has contributed to an estimated market size for SLAM Robots of USD 388.2 million in 2022 [1]. An approach that is proving to be successful has been to solve the SLAM problem using collaboration between multiple robots, enabling the system to autonomously map out unknown areas more quickly. This can also achieve higher robustness and accuracy [9].

In this paper, we lay the foundation for a team-based tank game, in which robots must work together in order to localise, coordinate fire at, and subsequently eliminate, enemies. This foundation consists of a few key components, namely: simulating robot detection, collaborative SLAM using map merging techniques, and exploration of the map through local and global navigation. We also include a visualisation tool for ROS bagfiles as part of our project.

II. Related Work

A. Collaborative SLAM

Simultaneous Localization and Mapping (SLAM) systems for multiple agents have garnered a lot of attention over the past ten years [2]. In the early stages of multi-agent SLAM, the majority of studies concentrate on using range sensors, such as sonar and lidar, combined with relative metrics like bearing and distance between different robots in order to get a clear estimation of the state of every robot in the scene. One approach to SLAM that can incorporate collaborative aspects is monocular visual SLAM. Due to the rapid development and increasingly robust nature of this approach to the SLAM problem, it's natural to consider the use of visual SLAM across multiple cameras, particularly across a set of cameras that move independently from one another [3]. Most collaborative visual slam systems use a centralized architecture, consisting of a front end on the agent side and a back end on the server side. The backend is usually on a separate server and is responsible for map fusion, simplification, and optimisation. The frontend is completely separated from the backend and is responsible for computing the real-time states of all agents, which is crucial for online applications [3]. As opposed to decentralised architectures, centralised ones are easier to implement and deploy.

B. Occupancy grid map merging

The main challenge of the flavour of collaborative SLAM we have chosen to implement is the topic of map merging. Map merging is the method by which partial maps are created by separate robots on a team or several runs of a single agent/robot in order to build the environment's global map more quickly and with more coverage [5], and it does not require the use of cameras as with monocular visual SLAM. "[Map merging] is an interesting and tough subject that has not earned the same attention as localization and map building has," says Konolige *et al.* [4]. Despite the fact that some articles on the subject have since been published, this area of research is still relatively underrepresented. The majority of methods that have been proposed thus far presume that maps are displayed as occupancy grid maps, which is the approach we adopt in this report. According to Carpin and Birk [6], the map merging problem can be viewed as an optimisation problem involving stochastic search. The space of possible transformations is investigated, the objective being to search for ones that can overlap two grid maps. The function being optimised is a measure of how much overlap there is. Birk and Carpin later improved their design by using more sophisticated algorithms and introducing methods to identify failures [7]. When the number of attempts tends to infinity, both methods are assured to find the best answer; however due to their repetitive nature, their computational requirements are significant, making them unsuitable for real-time use.

The map merging problem is also addressed by Howard *et al.* through sporadic robot rendezvous [8]. During the course of the exploring process, two robots may occasionally meet up. When this occurs, the relative pose of the two robots is estimated, and their separate maps are combined. Although it has been shown to be effective in the real world, this method has one disadvantage: merging maps requires computing the relative positions. Robots won't be capable of combining their maps when they can interact but cannot see each other if relative localization needs a visual line of sight, such as when it is dependent on vision. The same concept is examined and improved upon by Dieter *et al.* [9]. The main distinction is that robots purposefully seek out interactions with one another in order to exchange and merge their maps, as opposed to randomly meeting by chance. Additionally, relative localisation can be established by transmitting sensor data once two robots are able to communicate with one another, eliminating the need for sight lines.

In this paper, we try to build upon the work by Carpin, Birk [5]–[7] and Howard *et al.* [8] by combining their ideas into a robust mapping and communication framework for our use case.

III. Algorithms and Framework

A. Simulating robot detection

A this stage, our project is entirely run in a simulation. Therefore, we need an algorithm to simulate robot detection when a robot enters another's LIDAR range. Of course, this hides the complexity of implementing such a system in the real world; one can imagine a multitude of different ways for robots to detect each other, from visual object-recognition to more specialised communication systems. Algorithm 1 describes how we simulate such a system, taking advantage of ground truth information that the simulation gives us. It runs on every robot on the scene, and loops over every other one, giving an $O(N_r^2)$ complexity, where N_r is the number of robots in the scene. In cases where only a subset of robots have the capability to detect others, the complexity becomes $O(N_d * N_r)$, where N_d is the number of 'detector' robots. Evidently, $N_d \leq N_r$. The algorithm is otherwise not computationally expensive, and runs very fast for any reasonable number of robots in the scene.

Algorithm 1 Simulating robot detection

```
1: DetectRobots($R, L(\theta), L(\theta)_{max}, \theta_{max}, \theta_{min}$)
2: let $R$ be the set of all robots on the stage
3: let $L(\theta)$ be the range measured by our LIDAR at angle $\theta$
4:       where $L(\theta)_{max}$ is the LIDAR's max range
5: let $\theta_{max} = -\theta_{min}$ be the LIDAR's min and max mea-
   surement angles
6: initialise $D$ to an empty array
7: for $r \in R$ do
8:     let $d$ be the distance from our robot to $r$
9:     let $\theta$ be the angle between $r$ and our current heading
10:    if $d \leq L(\theta)_{max}$ and $|\theta| \leq \theta_{max}$ then
11:        if $d \leq L(\theta)$ then
12:            Insert $r$ into $D$
13: return $D$
```

B. Map merging

The algorithm used for map merging is described in detail in Carpin [5]. It uses Hough transforms to find straight lines in the two maps M_1 & M_2, and cross correlates them to generate candidate rotations ψ_i of which to rotate one map by, in order to match the other. The justification for this approach is that most real-world buildings exhibit many straight lines, making this a very sensible and robust approach in most situations. For every hypothesised angle ψ_i, some additional work is done using spectral information to figure out the best translation Δ_x^i, Δ_y^i. The result of the algorithm is a number of affine transformations $T_i(\psi_i, \Delta_x^i, \Delta_y^i)$ that can be used to superimpose the two maps. A score ω_i ("acceptance index") is given for how accurate the result is, which is defined as follows:

$$\omega(M_1, M_2) = \begin{cases} 0 & \text{if } agr(M_1, M_2) = 0 \\ \frac{agr(M_1,M_2)}{agr(M_1,M_2)+dis(M_1,M_2)} & \text{if } agr(M_1, M_2) \neq 0 \end{cases} \quad (1)$$

Where $agr(M_1, M_2)$ is defined as the *agreement* between the number of cells, meaning those that are both free or both occupied. The *disagreement*, $dis(M_1, M_2)$ is the number of cells in which one is occupied and the other is free. Note that we ignore cells where either map contains an unknown value. Following Eqn. 1 we have therefore:

$$0 \leq \omega \leq 1 \quad (2)$$

where $\omega = 1$ if and only if $M_1 = M_2$.

The algorithm used is outlined in Algorithm 2. An example run through of two robots exploring the map with the map merging algorithm running is shown in Figure 1.

Algorithm 2 Map merging

```
1: ComputeHypothesis($M_1, M_2, N, \epsilon$)
2: $HS_{M_1} \leftarrow HoughSpectrum(M_1)$
3: $\psi_r \leftarrow LocalMaxima(HS_{M_1}, 1)$
4: $M_1 \leftarrow T(0, 0, \psi_r)M_1$
5: $HS_{M_1} \leftarrow HoughSpectrum(M_1)$
6: $HS_{M_2} \leftarrow HoughSpectrum(M_2)$
7: $CC_{M_1M_2} \leftarrow CircularCrossCorrelation(HS_{M_1}, HS_{M_2})$
8: $\psi_{1...N} \leftarrow LocalMaxima(CC_{M_1M_2}, N)$
9: if $\epsilon \neq 0$ then
10:    for $\psi_i \in \psi_{1...N}$ do
11:        Append $\psi_i + \epsilon$, $\psi_i - \epsilon$ to $\psi_{1...3N}$
12:    $N \leftarrow 3N$
13: $SX_{M_1} \leftarrow XSpectrum(M_1)$
14: $SY_{M_1} \leftarrow YSpectrum(M_1)$
15: for $i \leftarrow 1$ to $N$ do
16:    $M_3 \leftarrow T(0, 0, \psi_i)M_2$
17:    $SX_{M_3} \leftarrow XSpectrum(M_3)$
18:    $SY_{M_3} \leftarrow YSpectrum(M_3)$
19:    $\Delta_x^i \leftarrow$ arg max$_\tau CCX_{M_1M_3}(\tau)$
20:    $\Delta_y^i \leftarrow$ arg max$_\tau CCY_{M_1M_3}(\tau)$
21:    $T_i \leftarrow \Delta_x^i \Delta_y^i \psi_i$
22:    $\omega_i \leftarrow AcceptanceIndex(M_1, TM_2)$
23: return $T_1 \dots T_N, \omega_1 \dots \omega_N$
```

The algorithm is essentially the same as Algorithm 1 in Carpin [5], but we include a couple of important additions. Notably, we start the algorithm by first rotating M_1 in order to align its walls with the x, y-axes (line 4), in order to improve performance. Note that line 5 can also be achieved by shifting over the previously found value of HS_{M_1}, since M_1 is a rotation of $T(0, 0, \psi_r)M_1$ and the Hough spectrum is periodic over 2π. This avoids an extra call to $HoughSpectrum()$ if desired. We also implement the option to use an extra parameter ϵ,

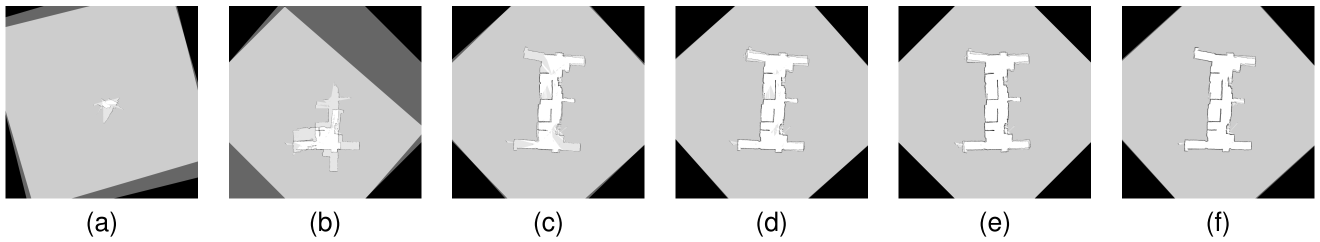

Fig. 1. The evolution of the merged map as the simulation runs, and more of the map is explored. Two robots were included in this simulation.

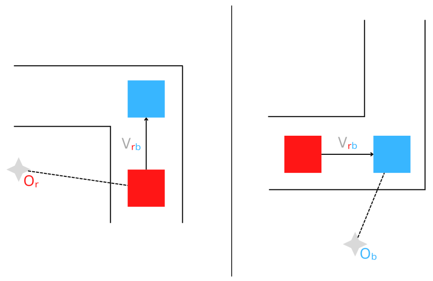

Fig. 2. *Left:* The situation when the red robot detects the blue robot. *Right:* The same situation, but in the blue robot's reference frame. When such a detection occurs, we can use the fact that the vector between the two, V_{rb}, are equivalent in the world frame. This information can be used to calculate an accurate affine transformation $T_i(\psi_i, \Delta x_i, \Delta y_i)$ between the two frames.

that adds two extra hypotheses for every hypothesised angle, $\psi_i \pm \epsilon$ (line 11), as suggested by Carpin [5]. This increases the number of tracked hypotheses from N to $3N$, as made clear in line 12. On this note, it's clear that the algorithm runs in $O(N)$, and that the runtime is thrice as long for non-zero values of ϵ.

The specifics of the function calls used in Algorithm 2, such as $HoughSpectrum$, $CircleCrossCorrelation$ and $XSpectrum$, are explained in more detail in Carpin [5].

Our implementation takes further advantage of a multi-agent environment, by calculating additional, accurate candidate transformations T_i when a robot detects another in it's field of view, as described by Figure 2. When the red robot detects a blue robot, the vector from itself to the blue robot $\mathbf{v}_{rb}$ can be calculated in the red robot's reference frame $\{\mathbf{v}_{rb}\}_r$. Assuming the blue robot has the same capabilities, it can do the same measurement in it's own reference frame: $\{\mathbf{v}_{rb}\}_b$ =$\{$-$\mathbf{v}_{br}\}_b$.

We can then calculate the corresponding affine transformation T, with $\{\mathbf{v}_{rb}\}_b \equiv \mathbf{y}$ and $\{\mathbf{v}_{rb}\}_r \equiv \mathbf{x}$ as follows:

$$\begin{bmatrix}\mathbf{y}\\1\end{bmatrix} = T\begin{bmatrix}\mathbf{x}\\1\end{bmatrix} \tag{3}$$

as such

$$T = \begin{bmatrix}\mathbf{y}_1 & \dots & \mathbf{y}_{n+1}\\ 1 & \dots & 1\end{bmatrix}\begin{bmatrix}\mathbf{x}_1 & \dots & \mathbf{x}_{n+1}\\ 1 & \dots & 1\end{bmatrix}^{-1} \tag{4}$$

This candidate transformation matrix T can then be incorporated in Algorithm 2, which will calculate an appropriate acceptance index ω.

C. Local Navigation

For exploring local surroundings, the robot need only to rely on its LIDAR sensors to navigate around. We build a simple AI that avoids walls by turning the other way, and attempts to free itself when it's odometry detects that it hasn't been able to move around for some time. This results in a relatively robust and autonomous system that can explore small parts of the map with ease. However, with no knowledge of the map as a whole, it can get stuck going back and forth, especially in an enclosed area. In order to more reliable map out larger areas, we therefore implement a more global navigation system that can seek out areas of interest on the robot's map.

D. Global Navigation

Every robot runs its own instance of SLAM using ROS's gmapping package. Every robot also has access to its team's combined map. Our global navigation implementation algorithm extends the local system by picking unexplored areas of the map and calculating a valid path towards them using an A* algorithm. The algorithm makes sure the robot doesn't get into areas where it could get stuck by calculating a costmap along its path. The resulting path is translated into movement commands for the robot, which are fed into its motion controller. After reaching the end destination, the robot switches back to the local exploration mode for a while before picking a new location to uncover. The movement algorithm is e

Algorithm 3 Movement control

```
1:  RunNavigation()
2:  while $True$ do
3:      if $last\_state \neq$ EXPLORE then
4:          for $N$ seconds do
5:              $run\_local\_navigation()$
6:              $last\_state \leftarrow$ EXPLORE
7:      else
8:          $goal \leftarrow find\_unexplored(G, P)$
9:          $path \leftarrow a\_star(goal)$
10:         for $waypoint \in path$ do
11:             $velocity \leftarrow calculate\_velocity(waypoint)$
12:             $move(velocity)$
13:         $last\_state \leftarrow$ NAVIGATE
```

Evidently, the navigation is run as an endless loop, as we want the robots to move continuously. A flag $last\_state$ is used to track the state of the last action, in order to switch between local and global navigation. Local exploration is done for a given amount of time, while a navigation cycle consists of finding a close, unexplored grid cell, given a grid G and a robot pose P. An optimal path is calculated using A*, and the robot is issued move commands by calculating the appropriate linear and angular velocities needed to move from waypoint to waypoint.

E. Battlefield visualisation

Our work also includes a visualiser to be used for future tank battles. The visualisation is achieved by parsing ROS bagfiles and leveraging the pygame library to display relevant information on screen.

IV. Experimental Results

A. Parameter experimentation for map merging

As seen in Algorithm 2, there are two main parameters to tweak. The first is the number of hypotheses N, and the second is ϵ. We are presented with a usual trade-off between runtime and accuracy, as the bulk of computation is done inside the **for** loop on line 15. As touched on previously, a non-zero value of ϵ will triple the number of times the **for** loop is ran. This means that running the algorithm with parameters $N = 12, \epsilon = 0$ and $N = 4, \epsilon \neq 0$ would have a similar runtimes.

To get a better idea of the value of the trade-offs resulting from tweaking the parameters, we measure the runtime of one iteration of the **for** loop. The **for** loop is measured multiple times across a handful of starting maps M_1, M_2. All maps used have the same size (1000 by 1000), although the level to which they have been explored varies. The result of the measurements ($n = 100$) are:

$$\Delta t = 4.7396s \pm 0.08551 \tag{5}$$

Since our implementation was made in Python, the vast majority of the runtime is spent while iterating over the maps with doubly nested **for** loops. Operations such as $HoughSpectrum()$, which use the very optimised OpenCV library, are much faster, suggesting that transpilation from Python to C/C++ in the future could be very beneficial for our use case. Regardless, the relatively high runtime per **for** loop suggests that we should be very cautious of increasing N, or introducing ϵ, as it will have significant impacts on runtime.

The results of our experimentation with the N and ϵ parameters are summarised in Figure 3. Note that in order to make fair comparisons between runs where $\epsilon = 0$ and those where $\epsilon \neq 0$, we can only compare the best acceptance indices when the number of iterations $N_{iter} \equiv 0 \mod 3$.

We can see how increasing N_{iter} can reasonably increase the best acceptance index ω at all stages of exploration. It's also useful to see that $N_{iter} > 15$ does not have any significant effect (if at all) on increasing ω in the three cases. This is to be expected, as the algorithm will start with the most 'likely' hypotheses, i.e. the ones with the highest maxima given by the circular cross correlation of the two Hough spectra, resulting in diminishing returns as we sample from lower and lower maxima. This is also the reason why we notice that the acceptance parameter is already reasonably good even in the very first iteration of the program.

Interesting to note is that, especially in the later stages of the runtime of the algorithm, there does not seem to be a great benefit to having ϵ included in the final calculations. While having the option of getting more precise angular resolution can be beneficial, as evidenced in (a) of Figure 3, one may choose to instead opt for exploring more hypotheses N returned by $localMaxima()$, and choose $\epsilon = 0$, at least in the map used for our experiments. This may, of course, be different for maps of a different nature, and one could spend an eternity exploring how different maps give rise to different optimal parameter values for the task at hand.

B. Parameter tuning for velocity calculation

As per Algorithm 3, we can see that the movement of the agents is executed by calculating and broadcasting velocity commands to the robot. The robot first needs to turn to the right angle, then go forward for the right amount of time. However, there isn't an easy way to tell how long we need to perform a rotation or forward motion for, therefore the most reliable way is to just keep broadcasting the movement command until we reach a desired position, up to a constant. However, the difference between the desired and actual angle can't be reliably measured with high enough frequency to ensure we will catch the point of perfect alignment. We therefore need to choose our action parameters carefully, otherwise the robot may overshoot the waypoint.

Instead of trying to achieve a perfect match, we decided find a precision value δ_θ such that if $\|current_\theta - desired_\theta\| < \delta_\theta$, we stop the movement. δ_θ needs to be small enough to avoid errors in our movement, but large enough so that we don't miss the window in which the measured angle is near enough to the desired angle. The initial value of δ_θ that we experimented with was 0.4, with an angular velocity of 1 rad/sec. This δ_θ proved to be too large, and the incurred error caused the robot to end up at the wrong location or crash into walls. Lowering it wasn't enough to solve the issue, because now the robot was overshooting its targets as the sampling frequency of the robot's pose wasn't high enough. Indeed, we can observe that:

$$\delta_\theta = \omega f \tag{6}$$

where f is the sampling frequency at which angles are measured, and ω the angular velocity. Since we don't have much control over f, it follows from here that we can improve our chance by lowering velocities as well, since slowing the movement increases the time spent in the goal region. The trade-off, however, is that it also increases the total time of movement. Our final implementation uses a δ_θ of 0.15 with the robot's angular velocity gradually decreasing as the robot approaches the desired pose, striking a balance between speed and accuracy. We use the same approach for linear velocity, using $\delta_{xy} = 0.03$.

V. Limitations and Future Work

With the base components necessary for the team-based tank game completed, future work would evidently be combining

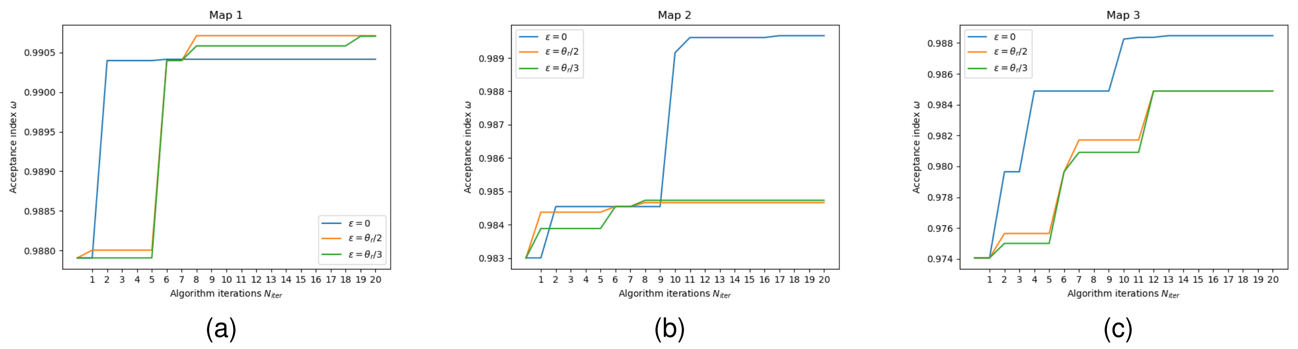

Fig. 3. The evolution of the best acceptance index as more iterations are run through the for loop. (a) (b) and (c) represent the same map in different stages of exploration by two robots, with (a) corresponding to a partially explored map by both robots (with some overlap to allow for map merging), and (c) corresponding to M_1 and M_2 being fully explored by both robots. θ_r corresponds to the angular resolution used by the HoughTransforms. Having $\epsilon < \theta_r$ would therefore allow for potentially more precise rotation angles ψ, since ψ orginates from a Hough transform discretised by θ_r. Images of the maps used can be viewed in the 'merging-experiments' repository of the appendix.

these elements, along with adding in basic shooting and health mechanics, in order to set the stage for the game. Once this is completed, one could also start to perform experiments on tank AI. Optimal strategies could be found by adjusting parameters and tweaking AI, using for example, genetic algorithms or other optimisation techniques.

There are also other improvements to be made on our implementation - the main one being transpiling hot sections from Python to a more low-level language such as C++, which would provide some significant speed ups that can be crucial especially for real world scenarios.

As this project was mainly aimed at collaborative SLAM, the navigation of robots didn't get as much focus. In the future it could be improved by a better state-automata system, that can handle robots getting stuck better. Another way to improve efficiency with navigation would be to have a shared costmap that updates in real-time, as the current A* algorithm uses a locally managed one, wasting valuable computational power on revisiting cells already uncovered before. The concept of publishing enemy tank locations and attacking/navigating toward them through the shared world map would also clearly be a very interesting addition.

Finally, our work is also limited in scope due to it occurring entirely on a simulation. While a real-world robot tank game sounds exciting, it's not entirely realistic. However, it would be interesting to see how, for example, a robot detector could be implemented. One can imagine complex solutions such as visual object recognition, or simpler ones by having robots communicate directly with each other wirelessly.

VI. Conclusion

In this paper, we introduce an efficient way to simulate a robot detector system. Additionally, we explore map merging techniques and, by tweaking and experimenting with parameters, find them to be suitable for our use case in a team-based autonomous robot tank game.

We also explore solutions for autonomous robot navigation in an unknown map, and find solutions for exploring unknown regions within it. The combination of such a navigation system with the map merging component allows for robots to work together and map out larger maps much faster than if they were operating alone.

Appendix

The main repository is hosted online at:
https://github.com/uob-ir-g10/collab-slam

The following repository was also used for the map merging experiments:
https://github.com/uob-ir-g10/merging-experiments

References

[1] Research Reports World. "SLAM Robots Market Size, Global 2023 Share, Growth, Segments, Revenue, Manufacturers and 2028 Forecast Research Report" MarketWatch, https://www.marketwatch.com/press-release/slam-robots-market-size-global-2023-share-growth-segments-revenue-manufacturers-and-2028-forecast-research-report-2022-11-18 (accessed Nov. 28, 2022).

[2] T. Arai, E. Pagello, and L. E. Parker, "Editorial: Advances in Multi-Robot Systems (584 cites)," *IEEE Transactions on Robotics and Automation 18* , no. 5, 2002.

[3] P. Schmuck and M. Chli,"CCM-SLAM: Robust and efficient centralized collaborative monocular simultaneous localization and mapping for robotic teams," *J Field Robot 36*, no. 4, 2019, doi: 10.1002/rob.21854.

[4] K. Konolige, D. Fox, B. Limketkai, J. Ko, and B. Stewart, "Map Merging for Distributed Robot Navigation," *in IEEE International Conference on Intelligent Robots and Systems 1*, 2003, doi: 10.1109/iros.2003.1250630.

[5] S. Carpin, "Fast and accurate map merging for multi-robot systems," *Auton Robot 25*,305–316 (2008). https://doi.org/10.1007/s10514-008- 9097-4

[6] S. Carpin, A. Birk, and V. Jucikas, "On map merging," *Rob Auton Syst 53*, no. 1, 2005, doi: 10.1016/j.robot.2005.07.001.

[7] A. Birk and S. Carpin, "Merging occupancy grid maps from multiple robots," *Proceedings of the IEEE 94*, no. 7, 2006, doi: 10.1109/JPROC.2006.876965.

[8] A. Howard, L. E. Parker, and G. S. Sukhatme, "Experiments with a large heterogeneous mobile robot team: Exploration, mapping, deployment and detection," *in International Journal of Robotics Research 25*, no. 5–6 , 2006, doi: 10.1177/0278364906065378.

[9] D. Fox, J. Ko, K. Konolige, B. Limketkai, D. Schulz, and B. Stewart, "Distributed multirobot exploration and mapping," *Proceedings of the IEEE, 94*, no. 7, 2006, doi: 10.1109/JPROC.2006.876927.