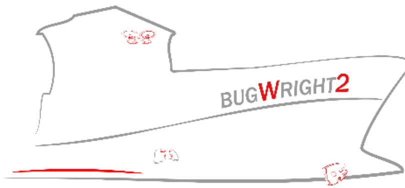


Autonomous Robotic Inspection and Maintenance on Ship Hulls and Storage Tanks

Deliverable report – D4.2

Context	
Deliverable title	Local Mapping and Obstacle Perception
Lead beneficiary	UIB
Author(s)	Partners involved
Work Package	WP04
Deliverable due date	March 2023 (M39)
Document status	
Version No.	1
Type	REPORT
Dissemination level	**Public**
Last modified	17 April 2023
Status	RELEASED
Date approved	17 April 2023
Approved by Coordinator	Prof. Cédric Pradalier (CNRS) Signature:
Declaration	Any work or result described therein is genuinely a result of the BUGWRIGHT2 project. Any other source will be properly referenced where and when relevant.

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under Grant Agreement No. 871260

TABLE OF CONTENTS

LIST OF FIGURES 1
LIST OF TABLES 2
ABBREVIATIONS 2
REFERENCED DOCUMENTS 3
HISTORY OF CHANGES 3
Executive summary 4
I. Introduction 4
II. Local Mapping and Obstacle Perception in the Aerial Inspection Drone 5
II.1. Brief overview of the sensor suite and the control architecture 5
II.2. Local mapping for motion estimation 7
II.3. Local mapping for obstacle perception 11
III. Local Mapping and Obstacle Perception for the Underwater Inspection Drone 14
III.1. Setup overview 14
III.2. Multibeam sonar-based generation of inspection maps 14
III.3. Front-plane depth and scale perception 15
IV. Local Mapping and Obstacle Perception for the Inspection Crawler 18
IV.1. Stop and map procedure 18
IV.2. Obstacle perception 19
IV.3. Pose correction and ICP 19
IV.4. From point clouds to texture maps 20
V. Conclusions 21

LIST OF FIGURES

Figure 1: Control software of the aerial inspection drone shown as a layered architecture. 6
Figure 2: Full-state estimator module (aerial inspection drone). 6
Figure 3: Picture of the aerial inspection drone, with indication of the main hardware components that support the control architecture. 7
Figure 4: Overview of LiODOM. 8
Figure 5: Example of map produced by LiODOM (for the *KITTI 05 sequence*), comprising an unoptimised global map generated during navigation (in white) and a local map (in red) that is retrieved according to the position of the vehicle, to be used for next pose estimation optimisation. 9
Figure 6: Example of cells and points produced by LiODOM (for the *KITTI 05 sequence*). 10

Figure 7: Distances involved in the collision prevention mechanism (aerial inspection drone). 12
Figure 8: Frustum used to select the points involved in the attenuation of the user's desired velocity (aerial inspection drone)..13
Figure 9: Illustration of the computation of the repulsion vector (aerial inspection drone). 13
Figure 10: Underwater inspection drone sensor footprints on a simulated hull. 16
Figure 11: Features detected in a sonar scan (underwater inspection drone)...16
Figure 12: Inspection maps represented as a voxel map (underwater inspection drone). 17
Figure 13: Perception of feature size by means of the laser-camera setup (underwater inspection drone). 17
Figure 14: Flow chart of the mapper running onboard the aerial crawler: grayed-out rectangles denote repeated behaviour. 18
Figure 15: RGB vs Intensity map, showing the metal plates used to test the aerial crawler: (a) Accumulated intensity point cloud, taken from the Livox Mid-70 sensor, and (b) RGB camera feed. 19

LIST OF TABLES

Table 1 : List of ICP constraints (aerial crawler)..20
Table 2: List of ICP parameters used for pose correction (aerial crawler)..20

ABBREVIATIONS

BW2	*BUGWRIGHT2*
EKF	*Extended Kalman Filter*
FL-MBS	*Forward-Looking MultiBeam Sonar*
FMU	*Flight Management Unit*
FOV	*Field of View*
FSE	*Full State Estimator*
IMU	*Inertial Measurement Unit*
LiDAR	*Light Detection And Ranging (or Laser Imaging Detection and Ranging)*
LiODOM	*Lidar-only ODOMetry*
PF	*Particle Filter*
UWB	*Ultra-Wide Band*

REFERENCED DOCUMENTS

- Deliverable D2.1 – *Crawler adaptation to BUGWRIGHT2's requirements*
- Deliverable D2.2 – *AUV adaptation to BUGWRIGHT2's requirements*
- Deliverable D2.3 – *MAV adaptation to BUGWRIGHT2's requirements*
- Deliverable D4.1 – *Localisation*
- Deliverable D5.1 – *Unified Control Interfaces*
- Deliverable D5.2 – *Autonomous Trajectory Tracking and Obstacle Avoidance*
- Deliverable D5.3 – *Single-Robot Planning, Coverage and Mission Execution*

These documents are stored on the file sharing site hosted by CNRS.

HISTORY OF CHANGES

Date	Written by	Description of change	Approver	Version No.
19/01/2023	UIB	Starting document		v0.0
01/03/2023	UIB	Section II		v0.1
01/03/2023	NTNU	Section III		v0.2
21/03/2023	CNRS	Section IV		v0.3
27/03/2023	UPORTO	Sections III & IV		v0.4
30/03/2023	UIB	Sections I, V & document finalisation		v0.5
31/03/2023	CNRS	Proofreading		v0.6
13/04/2023		Validation	CNRS	v1.0

Executive summary

Deliverable D4.2 focuses on the mapping functionalities running on the BUGWRIGHT2 (BW2) platforms that require representations of the immediate surroundings for their operation, i.e. local maps, being the typical use obstacle representation and motion planning for collision avoidance, though other uses also take place, e.g. motion estimation.

I Introduction

The project BW2 aims at the development of several robotic platforms oriented to making (ship) inspections easier and faster for the inspection crew. To this end, the robots need a suitable control architecture to solve the respective operating cases with the required level of autonomy. Among others, the control architecture may need representations of the environment at different levels to achieve the intended goals. These representations can be regarded as containers where the corresponding platform stores processed data collected by means of the onboard sensors as well as possibly updates in accordance to new findings.

Deliverable D4.2 focuses on the mapping functionalities running on the BW2 platforms that make use of representations of the immediate surroundings during their operation, i.e. local maps, being the typical use that of obstacle representation and motion planning for collision avoidance, though other uses can also take place, as described along the following sections.

Deliverables D4.1 – *Localisation* (already released), D5.1 – *Unified Control Interfaces* (already released), D5.2 – *Autonomous Trajectory Tracking and Obstacle Avoidance* (due M39) and D5.3 – *Single-Robot Planning, Coverage and Mission Execution* (due M45) describe complementary aspects of the control architectures of the respective BW2 platforms. Consequently, a certain level of overlap can be expected between the contents of the aforementioned deliverables and this document, as a result of trying to make every deliverable a reasonably self-contained report.

The rest of this document is organised as follows: Section II is devoted to the aerial inspection drone, where Section II.1 overviews very briefly the control architecture where the aforementioned local mapping functionalities reside, Section II.2 describes local mapping oriented to LiDAR-based motion estimation, and Section II.3 refers to local maps for obstacle representation and collision prevention; Section III deals with local mapping and depth perception issues as solved by the underwater inspection drone, reviewing the sensor suite in Section III.1 and the perception-related functionalities in Sections III.2 and III.3; Section IV is for the inspection crawler, detailing obstacle perception in Sections IV.1 and IV.2, and mapping in Sections IV.3 and IV.4; finally, Section V summarises and concludes deliverable D4.2.

Local Mapping and Obstacle Perception in the Aerial Inspection Drone

This section describes local mapping functionalities that are available in the control architecture of the UIB aerial inspection drone. These functionalities serve to different purposes but share the feature of building local representations of the surrounding environment in order to produce the corresponding estimations. On the one hand, local maps are built for platform motion estimation on the basis of the laser scans supplied by a 3D LiDAR. On the other hand, the same sensor supplies information on the surrounding obstacles, which have to be perceived in order to be able to navigate safely throughout the environment.

In order to situate correctly these two functionalities in the control software of the platform, the vehicle control architecture is briefly overviewed in the next section, to describe in the posterior sections the processes of building the respective local maps involved in motion estimation and obstacle perception.

II.1. Brief overview of the sensor suite and the control architecture

Figure 1 shows the control architecture of the aerial inspection drone assimilated to a hierarchical layered structure, where each layer implements a different level of control. Moreover, mid- and high-level control layers run different robot behaviours that contribute to the generation of the final motion control command. In more detail:

- The low-level control layer comprises attitude and thrust control, as well as the behaviours that check the viability of the flight. We make use of the DJI FMU services, through the DJI SDK, to make available this functionality.
- The mid-level control layer accommodates safety-oriented control by including the *safety manager* module, which comprises several robot behaviours that combine the user desired speed command with the available sensor data to obtain final and safe speed and height set-points that are sent to, respectively, the horizontal speed and height controllers.
- At the highest level of the hierarchy, the application-oriented control layer allows the execution of predefined missions by means of the *mission manager* module, which is in charge of executing higher autonomy behaviours that implement missions defined as a set of way-points to attain. The corresponding motion commands are generated in sequence and issued to the corresponding position controller, while monitoring way-point achievement and overall correct mission execution.

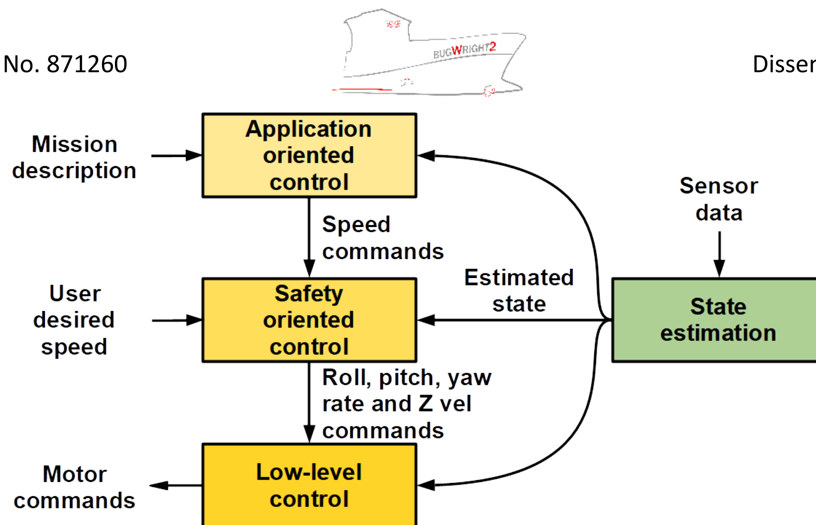


Figure 1: Control software of the aerial inspection drone shown as a layered architecture.

As shown in Figure 1, a state estimation module is transverse to all layers. This module is in charge of processing and fusing all the sensor data available on-board, to estimate with enough accuracy the platform state. The state estimate is employed in all control layers, as could be expected.

Given the nature of the missions handled by the *mission manager*, defined in terms of sets of way-points to be achieved, the platform state comprises the platform pose $(x, y, z, \varphi, \theta, \psi)$, the linear velocities $(\dot{x}, \dot{y}, \dot{z})$ and accelerations $(\ddot{x}, \ddot{y}, \ddot{z})$, and the angular velocities $(\dot{\varphi}, \dot{\theta}, \dot{\psi})$. The *full state estimator* (FSE) module shown in Figure 2 supplies these state estimates by fusing the available navigation data (once processed) and a number of different positioning sources. More precisely, the FSE module comprises two cascaded *Extended Kalman Filters* (EKF): the *local EKF* fuses navigation data from motion and relative-position sources, while the *global EKF* combines estimates from the *local EKF* with positioning data from global localisation sources such as GPS and/or UWB-based position estimators, as well as from SLAM algorithms (whichever are available) fed by the on-board sensors.

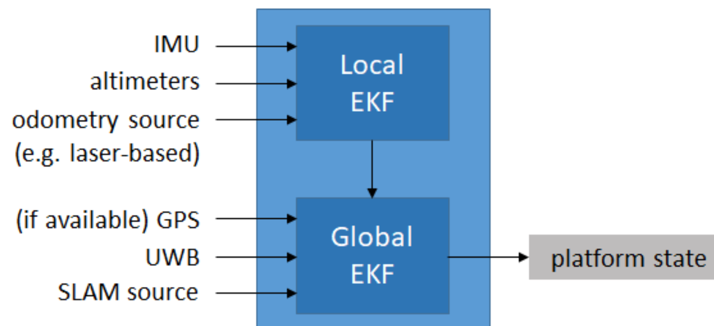


Figure 2: Full-state estimator module (aerial inspection drone).

In its current configuration, the sensor suite of the aerial inspection drone comprises (see Figure 3, and deliverable D2.3 – *MAV adaptation to BUGWRIGHT2's requirements*):

- A 3D laser scanner from Ouster, model *Ouster OS1-64 Gen 2*, which supplies structured 3D point clouds organised into 64 channels, with 2048 bins per channel (maximum), able to operate at 10 or 20 Hz. The maximum range is 100 m, for a Field of View (FOV) of +22.5° to -22.5° (vertical) and 360° (horizontal).
- A downward-looking LIDAR-Lite v3 single-beam laser range finder used to supply height data for a maximum range of 40 meters. This sensor is complemented with a barometric pressure sensor that is included in the *Flight Management Unit* (FMU) of the drone.

- (optionally) An upward-looking TeraRanger Evo 60m Time-of-Flight (ToF) infrared range finder to supply the distance to the ceiling up to a maximum range of 60 meters.
- An imaging system which can interchangeably consist of an RGB-D or a lighter RGB camera (Figure 3 shows the configuration including an RGB-D camera from Intel, model *Intel Realsense D435i*, which also integrates a 6 DOF IMU).
- An Ultra-Wide Band (UWB) receiver/tag, used as part of a UWB-based global localisation system.
- GNSS and RTK GNSS receivers.

In addition to motion estimation, the 3D structured point cloud provided by the 3D laser scanner is used within the control architecture to prevent collisions with surrounding obstacles, while attenuating the speed of the platform based on the proximity to them. From the mapping perspective, both procedures include building a representation of the local environment of the platform. More details are given in the following sections.

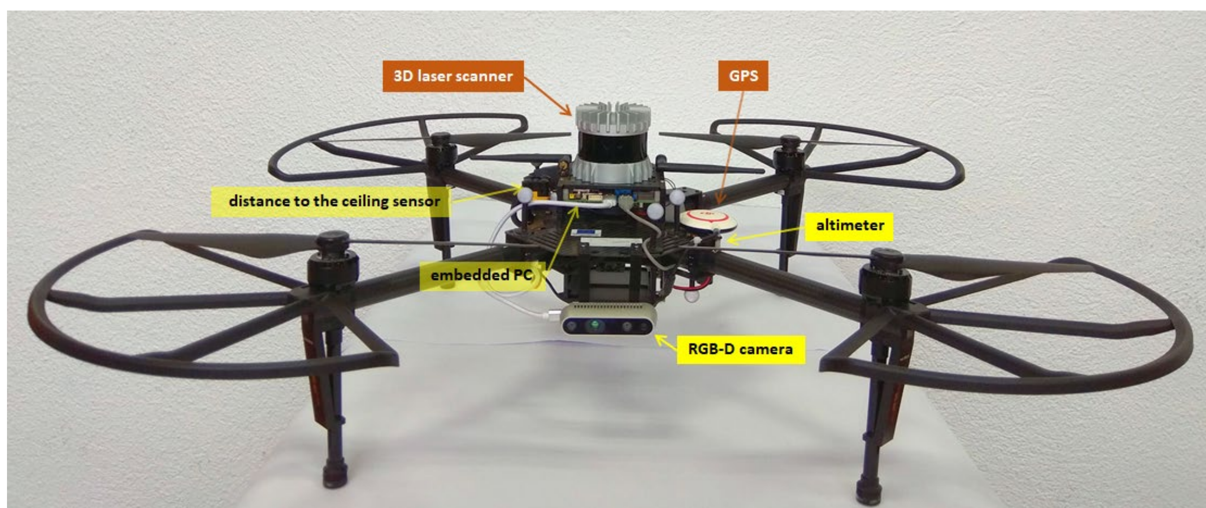


Figure 3: Picture of the aerial inspection drone,
with indication of the main hardware components that support the control architecture.

II.2. Local mapping for motion estimation

LiODOM (*LiDAR-only ODOMetry*) is a laser-based motion estimation module developed within the framework of project BW2 by the UIB. Currently, it is the odometry source that feeds the local EKF of the full state estimator of the aerial inspection drone (Figure 2). As already said, in the current configuration of the drone, the input for LiODOM comes from an *Ouster OS1-64* 3D laser scanner, although it is independent of the sensor. LiODOM works naturally with structured point clouds, as the ones provided by the Ouster LiDAR; in case the input sensor is not of this class, a specific module to transform from unstructured to structured point clouds must be employed. Next sections provide relevant data about the local mapping functionalities.

LiDAR odometry

Figure 4 illustrates LiODOM. As its name suggests, LiODOM can compute the pose of the platform without the help of any other sensor, such as IMU, GPS, etc. It is fundamentally organised into two main

components, *odometry* and *mapping*, which are executed in parallel during task execution. The *mapping module*, which is the fundamental topic covered in this deliverable, builds an unoptimised global representation of the environment $\mathrm{M_i}$ i.e. loop closures are not detected nor considered. From this global map, a local representation m_i is extracted according to the current pose of the platform. This local map is further used to find correspondences with the current point cloud in the next estimation stage. Each of these components is detailed next.

The *odometry module* is further divided into two synchronised threads to decouple the feature extraction from the pose estimation. In the feature extraction stage, the received sweep S_i at time step i is divided into its different scans but discarding points which do not fall within an interval $[r_{min}, r_{max}]$. Next, a set of edge features E_i^L (as defined in LOAM[1]) are detected and selected in accordance to the local curvature within the input point cloud. Each scan is additionally split into sectors and a maximum number of points per sector is chosen to ensure an even distribution of point data throughout the environment for better optimisation conditioning.

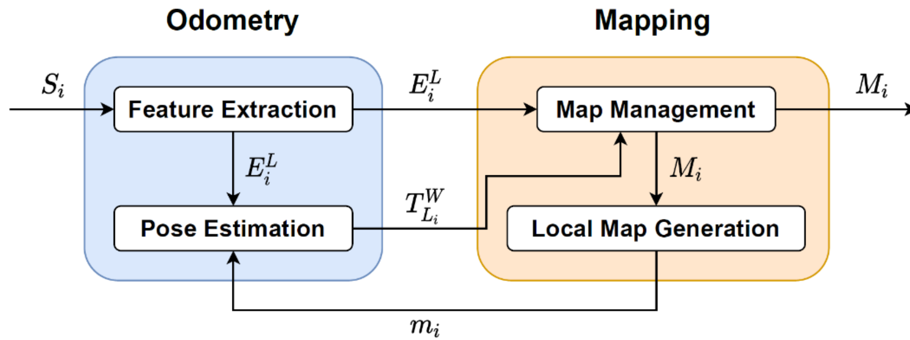


Figure 4: Overview of LiODOM.

The resulting edges are then processed by the pose estimation thread. Each edge $p_j^{L_i}$ in the LiDAR coordinate frame L is transformed into world coordinates as $p_j^W = T_{L_i}^W p_j^{L_i}$, being $T_{L_i}^W$ the transformation from LiDAR to world at time t_i. A constant velocity motion model is initially assumed for this transformation.

Despite LiODOM is able to provide an estimation of the pose using only LiDAR data, at this point we can take advantage of the IMU and the height estimated by the FSE to enrich the initial guess on the platform motion. Next, a set of point-to-line correspondences are established between each point p_j^W and a local map m_i. The latter is built by the *mapping module*, which is described later.

Let us now denote the set $\mathrm{N}(p_j^W)$ as the k-th nearest points in the local map of the point p_j^W and $N_n(p_j^W)$ as the n-th nearest neighbour. We first check whether $\mathrm{N}(p_j^W)$ really conforms a line and, if this is the case, we compute the point to line distance from p_j^W to the line $l(p_j^W)$, resulting from $N_1(p_j^W)$ and $N_2(p_j^W)$, as:

$$d_e\left(p_j^W, l(p_j^W)\right) = \frac{|\left(p_j^W - N_1(p_j^W)\right) \times N_{12}|}{|N_{12}|},$$

[1] Ji Zhang and Sanjiv Singh, LOAM: Lidar Odometry and Mapping in Real-time, in Proc. Robotics : Science and Systems Conference (RSS), 2014

with $N_{12} = N_1(p_j^W) - N_2(p_j^W)$. Next, the optimal pose of the LiDAR $T_{L_i}^W$ is computed by means of non-linear optimisation. To this end, each valid correspondence provides a constraint whose residual is defined as:

$$\varrho_e\left(p_j^W, l(p_j^W)\right) = \omega_j \, d_e\left(p_j^W, l(p_j^W)\right),$$

where ω_j is a weighting term defined by:

$$\omega_j = 1 - \frac{r_j - r_{\min}}{r_{\max} - r_{\min}},$$

being r_j the range returned by the LiDAR for edge $p_j^{L_i}$. Then, the optimal pose is computed by minimising the loss function $J\left(\widetilde{T_{L_i}^W}, \Upsilon\right)$:

$$\mathrm{J}\left(\widetilde{T_{L_i}^W}, \Upsilon\right) = \frac{1}{2}\sum_{j\in\Upsilon} \rho\left(||\varrho_e\left(\widetilde{T_{L_i}^W} p_j^{L_i}, l\left(\widetilde{T_{L_i}^W} p_j^{L_i}\right)\right)||^2\right),$$

and, hence, the optimal pose can be stated as:

$$T_{L_i}^W = \min_{\widetilde{T_{L_i}^W}} J\left(\widetilde{T_{L_i}^W}, \Upsilon\right),$$

where Υ is the set of correspondences established between the detected edges and the local map, and ρ is a Huber loss function.

The registration of the extracted edges E_i^L on the global map M_i is performed by the *mapping module* using the last optimised pose $T_{L_i}^W$. This module also generates the corresponding local map m_i. Figure 5 shows an example of the maps resulting from LiODOM.

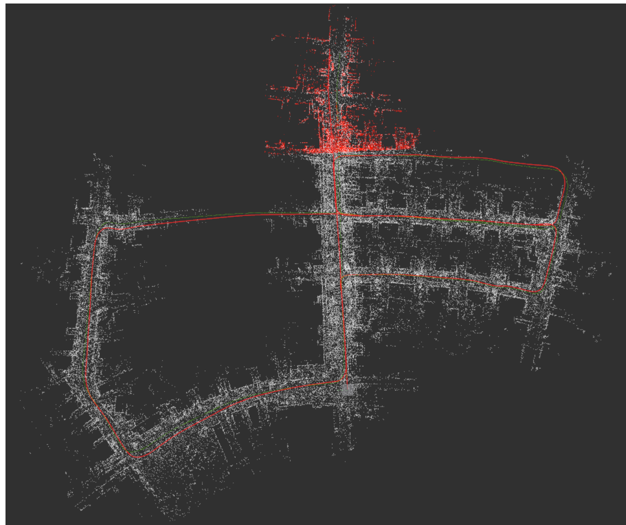


Figure 5: Example of map produced by LiODOM (for the *KITTI 05 sequence*), comprising an unoptimised global map generated during navigation (in white) and a local map (in red) that is retrieved according to the position of the vehicle, to be used for next pose estimation optimisation.

Map Representation

Given the high frequency at which the map must be accessed, the type of data structure chosen to represent 3D space becomes crucial for fast operation. A single KD-tree has been typically used to this end. However, this option presents several drawbacks: on the one hand, the full tree tends to change as points are added or deleted to/from the tree, and, on the other hand, the KD-tree performance decreases as more points need to be managed. To overcome these issues, in LiODOM we introduce an efficient hashing data structure for representing the map, taking inspiration from other recent works. To be more specific, the 3D space is partitioned into a set of disjoint cuboids of fixed size that we name *cells*. A cell C_j is represented by its geometric center, denoted by (c_{jx}, c_{jy}, c_{jz}), and includes all 3D points whose coordinates fall into its limits. Examples of cells and points can be found in Figure 6. We define a map at time t_i as $M_i = \{H_i, C_i\}$, where H_i is a hash table and C_i is the set of existing cells up to time t_i. Table H_i allows us to rapidly get access to a specific cell C_j using a hash function based on its coordinates, being defined by:

$$\mathrm{H}(\mathrm{C_j}) = \left(\mathrm{c_{jx}} \oplus \left(\mathrm{c_{jy}} \ll 1\right)\right) \oplus \left(\mathrm{c_{jz}} \ll 2\right),$$

where $\oplus$ and $\ll$ are, respectively, the bitwise XOR and the left shift operators. This function has been selected in order to minimise, as much as possible, hash collisions. That is to say, if bits of a binary word have roughly 50% chance of being 0 or 1, i.e. as randomly distributed as possible, the bitwise XOR between such binary words results into another word also following a random distribution. Furthermore, since the bitwise XOR is a symmetric operation, the order of the elements in the hash code is lost. To break this symmetry, we use the shift operator, at a limited computational cost.

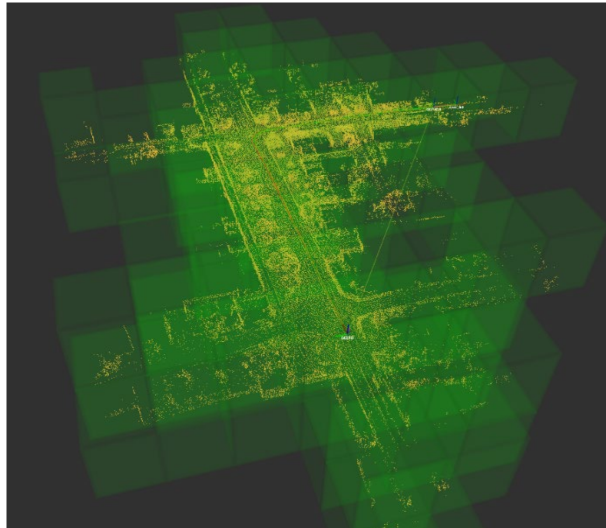


Figure 6: Example of cells and points produced by LiODOM (for the *KITTI 05 sequence*).

Map Updates

In LiODOM, map updates are performed once per sweep, being the input data the set of edges E_i^L extracted from S_i and the last optimised transformation $T_{L_i}^W$. Unlike other approaches, where the raw point cloud is used for mapping, in our approach, the map is built using directly the edges to speed up the mapping procedure, resulting into more sparse maps. Initially, every point $p_j^{L_i} \in E_i^L$ is transformed into world

coordinates. Next, for each point $p_j^W = (x, y, z)$, we compute the geometric center of the cell C_q in which the point should be stored as:

$$\begin{bmatrix} \mathrm{C_{qx}} \\ \mathrm{C_{qy}} \\ \mathrm{C_{qz}} \end{bmatrix} = \begin{bmatrix} \lfloor x/s_{xy} \rfloor s_{xy} + \frac{1}{2} s_{xy} \\ \lfloor y/s_{xy} \rfloor s_{xy} + \frac{1}{2} s_{xy} \\ \lfloor z/s_z \rfloor s_z + \frac{1}{2} s_z \end{bmatrix}$$

where s_{xy} and s_z are the metric cell sizes for the corresponding dimension. We next check if the cell C_q is already in the map by querying the hash table H using the key $H(C_q)$. If this is the case, the point is added to the existing cell. Otherwise, a new cell C_n is created with point p_j^W as seed, to be added next to C and indexed on H by $H(C_n)$.

Finally, modified cells exceeding a certain number of points are filtered using a 3D voxel grid approach. Note that our data structure allows us to rapidly update just the required areas of the environment, avoiding the update of the whole map at each iteration. This fact contributes to speeding up the mapping procedure.

Adaptive Local Map Computation

Lastly, the mapping module generates a local map m_i, which contains the points of M_i within a certain range from the current LiDAR pose. Assuming a moderate motion between two consecutive sweeps, these points are enough to find correspondences for the next pose estimation step. To build the local map, we first retrieve the cell C_{L_i} where the LiDAR is located at that moment using its current position $T_{L_i}^W$. Next, assuming a 3D grid arranged over M_i, neighbouring cells of C_{L_i} up to a certain distance are further retrieved from M_i, and their corresponding points are merged to form the local map m_i. This operation results to be very fast due to the proposed hashing structure. Points on m_i are finally organised into a KD-tree to speed up nearest neighbour search. Note that this tree is very simple, as it just contains a small subset of the total map points, in contrast to managing the whole global map.

On the other side, we refer to this local map as *adaptive* since it always covers a specific area of the environment, contrary to a local map built by aggregation of a sliding window. Besides, it provides us with correspondences with revisited areas of the environment in a natural way. Additionally, the availability of m_i avoids us to search for correspondences against the whole map, as done by other solutions. Finally, to avoid reduced amounts of points from unexplored areas, we always add the last three sweeps to m_i.

II.3. Local mapping for obstacle perception

As already mentioned, inside the aerial inspection drone, the LiDAR sensor is also used for perceiving the environment and preventing collisions with the surrounding obstacles. To be precise, the collision prevention mechanism makes use of the 3D point-cloud in two different ways:

- On the one hand, the user's desired speed is attenuated towards zero when the platform approaches an obstacle. The closer the obstacle, the greater the attenuation, so that the user's desired speed is completely attenuated when the platform is very close to the obstacle.
- On the other hand, all obstacles surrounding the platform create a repulsion vector for the vehicle to move away from them. The closer the obstacle is, the greater the repulsion vector that is generated. Once all the vectors have been calculated (for all surrounding obstacles) they are added together to obtain a single repulsion vector/velocity.

Two distances are involved in the collision prevention mechanism: the attenuation distance d_{att} and the minimum distance allowed to obstacles d_{min}, where $d_{att} > d_{min}$. Given these two distances and considering a single obstacle situated at a distance d_o from the aerial platform, three different situations may arise:

1. $d_o > d_{att}$: the obstacle is not deemed a threat for the platform and hence it is not considered by the collision prevention mechanism.
2. $d_{min} < d_o < d_{att}$: the obstacle is not used to create a repulsion vector to move the aerial platform away from it, but d_o is used to attenuate the user's desired speed in the case this points towards the obstacle. As a result, the platform can still approach the obstacle, but with decreasing speed.
3. $d_o < d_{min}$: the vehicle is too close to the obstacle so that d_o is used to create a repulsion vector to move the platform away. The user's desired speed is fully attenuated (i.e., it becomes 0) in the case this points towards the obstacle.

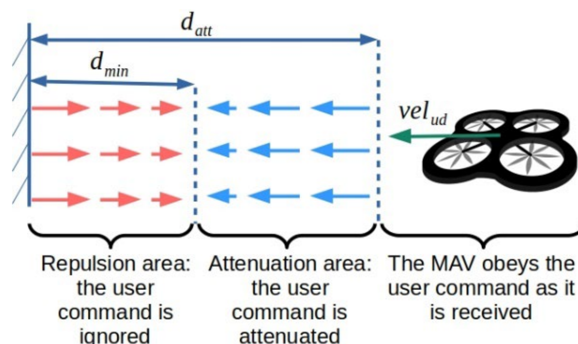


Figure 7: Distances involved in the collision prevention mechanism (aerial inspection drone).

Figure 7 shows the three situations depending on where the aerial platform is situated with regard to the two distance parameters. In the following, we detail the velocity attenuation and repulsion mechanisms.

Velocity attenuation

For a given user's desired velocity vel_{ud} to move the vehicle towards an obstacle situated at distance d_o, the velocity is attenuated by a factor $\lambda \in [0, 1]$ which is computed as:

$$\lambda = min\left(1.0, max\left(0.0, \left(\frac{d_o - d_{min}}{d_{att} - d_{min}}\right)\right)\right)$$

so that the attenuated desired velocity results into:

$$vel_{att} = \lambda \cdot vel_{ud}$$

Notice that if there is no obstacle in the direction of the desired velocity, or this is farther than d_{att}, the desired velocity is not attenuated.

The distance d_o is computed considering all the points of the point cloud situated within a pyramidal frustum oriented in the direction of the user's desired speed (see Figure 8). The FOV of the frustum is $45°$ in both the vertical and horizontal directions, with the near and far planes situated at, respectively, a distance equal to the robot radius d_r and at a distance $d_{att} > d_r$. Then, d_o is calculated as the minimum of the distances to the points within the frustum (in red in Figure 8).

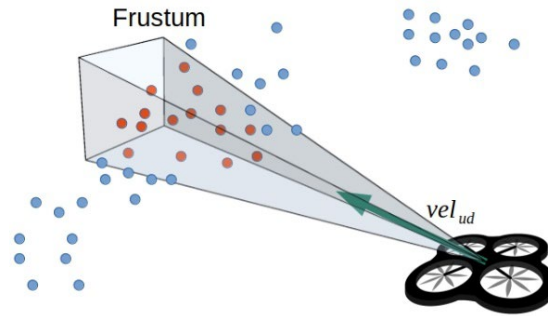


Figure 8: Frustum used to select the points involved in the attenuation of the user's desired velocity (aerial inspection drone).

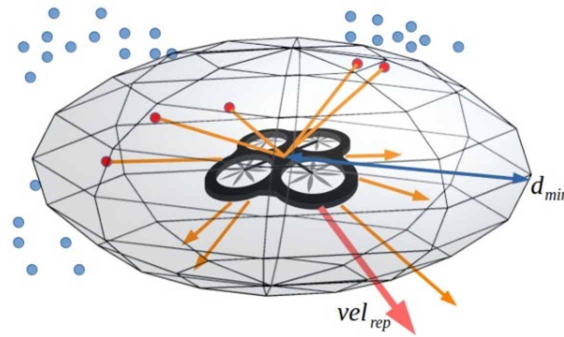


Figure 9: Illustration of the computation of the repulsion vector (aerial inspection drone).

Calculation of the repulsion vector

A repulsion speed vector rep_p is created for each 3D point p within the point cloud and situated at a distance $d_p < d_{min}$ (see Figure 9). The magnitude of the repulsion vector is then computed as:

$$\|rep_p\| = K \cdot (d_{min} - d_p)$$

where K is the repulsion factor and the direction of the resulting vector is pointing away from p. The final repulsion vector vel_{rep} is computed as the mean of all the repulsion vectors defined for each point separately. In addition, the magnitude of vel_{rep} is limited to the maximum speed allowed.

Considering both the velocity attenuation and the repulsion mechanisms, the final velocity command results:

$$vel_{cmd} = vel_{att} + vel_{rep}$$

██ Local Mapping and Obstacle Perception for the Underwater Inspection Drone

In this section, we describe the methods developed for local mapping and obstacle/depth perception for the underwater platform under development in BW2. As already described in deliverable D2.2 – *AUV adaptation to BUGWRIGHT2's requirements*, the underwater vehicle developed jointly by Blueye, NTNU and UPORTO is based on the Blueye X3 ROV, with contributions from UNI-KLU on the platform localisation side and hence also on mapping. In the following, Section III.1 revises the platform setup, while Section III.2 describes the process related to the generation of a local map used to represent the progress during the inspection of the submerged part of a vessel hull, and Section III.3 deals with feature scale perception.

III.1. Setup overview

In the X3 small ROV, the default integrated sensor payload consists of two Inertial Measurement Units (IMUs), a pressure sensor providing depth measurements, and a camera inside a glass dome with (approx.) $48°$ degrees vertical FOV and (approx.) $77°$ horizontal FOV. Additionally, three external sensors are connected to the vehicle's guest ports: a GPS on a stick for synchronisation with the global navigation frame, a *forward-looking multibeam sonar* (FL-MBS), and a DVL oriented towards the sea-bottom to measure the speed over ground in the vehicle's frame. The FL-MBS has $130°$ horizontal and $20°$ vertical apertures, features a configurable sight distance, and points in the same direction as the camera, with a slight vertical offset. The footprints of the camera and the forward-looking sonar are depicted in Figure 10, with the Blueye ROV facing a simulated ship hull.

An additional, lower-cost obstacle perception sensor setup has been under investigation from the side of UPORTO, apart from the aforementioned setup based on the FL-MBS. This setup adopts a triangulation-based visual approach and comprises two laser pointers combined with an imaging sensor.

The ROV is actuated in surge, sway, heave, and yaw. The localisation, guidance, and control algorithms all run fully onboard the vehicle, whereas the optical imagery and the sonar data are processed at the surface, within an external computer due to the limited computational capacity onboard the vehicle. The operator has the possibility to interact with the vehicle for safety reasons based on the online data feed and to provide high-level input.

III.2. Multibeam sonar-based generation of inspection maps

To keep track of the inspection progress during the operation, an inspection map is built online. Using sonar and navigation data, everything seen by the vehicle is registered with position, and point clouds are established to form an occupancy map. The map contains local uncertainty information that the operator can use to assess the reliability of the inspection. Automatic detection of coverage holes is added to allow the autonomous system to find them and (re)inspect the area. The map is referred here as inspection map

because it is solely used for the drone to keep track of the mission and not for any other tasks such as the reconstruction of the ship hull, which is an independent task.

Using the assumption that the surface in front of the ROV has a flat shape, the closest detected feature for each sonar beam provides a set of information about the surroundings that allows the creation of the inspection map. Therefore, to build the inspection map, features are detected for each sonar scan. An example is displayed in Figure 11, with the detected features highlighted as green dots.

However, before building the map, the sonar features must be filtered for noise and outliers. To achieve this, a method based on averaged point distances is proposed. It measures the average distance between the surrounding points in a window for each point at a time. For objects that are present, the corresponding feature points of consecutive beams are expected to be close to each other. The window size *s* selects a total of *s+1* consecutive beams, with half of *s* beams before and half of *s* beams after the focus point. The average distance value for a point is then calculated to have a better understanding of its neighbourhood.

This method is more robust and less prone to true positive rejection compared to methods such as bin-based evaluations using the direct sonar distances to the points. However, a cluster of noisy points will anyway create a local bias and might increase rejection of true positives in that area. The rejection of the points is then based on a threshold test.

Over time, points for the sonar scans accumulate and create a dense point cloud and enables the creation of the voxel map. A voxel is created only if there are enough reliable points inside the area related to the voxel. To assess and validate the proposed methods, full-scale ship hulls have been mapped. An approximately 30 x 5 m^2 section of a ship was autonomously inspected twice using different inspection patterns. The first survey contains horizontal slices taken at 1 m distance to the hull. The second survey contains vertical slices for a 1.3 m distance to the hull. In both cases, the inspection starts at the water surface and ends at the keel. The results of experiments on full-scale ship can be found in Figure 12.

III.3. Front-plane depth and scale perception

A lower-cost sensor setup comprising two laser pointers and an imaging sensor is used to provide the operator with depth data regarding the (locally) planar surface in front of the robot, as well as scale perception, as shown in Figure 13. With the aforementioned setup, depth and scale can be calculated by processing the camera frames after locating the laser dots reflections and triangulating them using the camera-laser pointers intrinsics.

All this allows obstacle detection and automatic hull distance. Additionally, because of the tilt actuator on the camera, we can adjust its tilt using the pitch calculated relative to the hull, to be applied where the hull curvature is more pronounced. This information can be overlayed onto the video feed to provide the operator with features scale/size perception directly onto the camera image (Figure 13). This development adds to the data the FL-MBS provides.

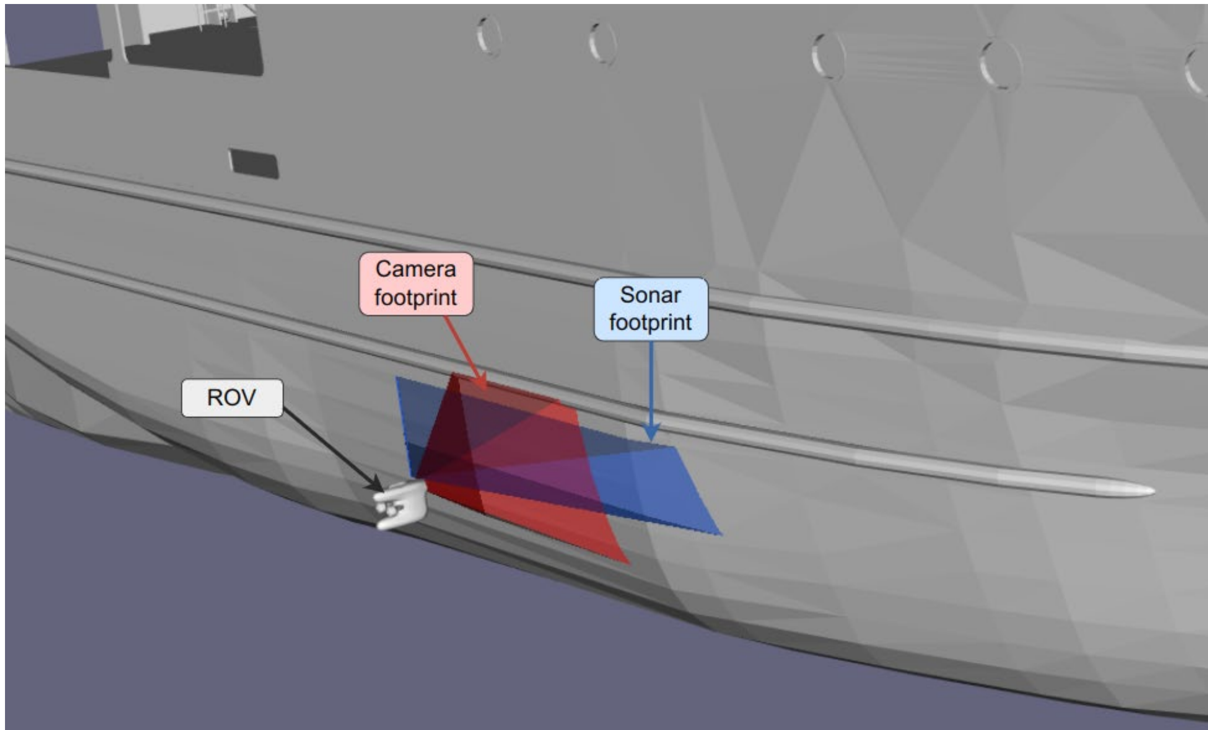


Figure 10: Underwater inspection drone sensor footprints on a simulated hull.

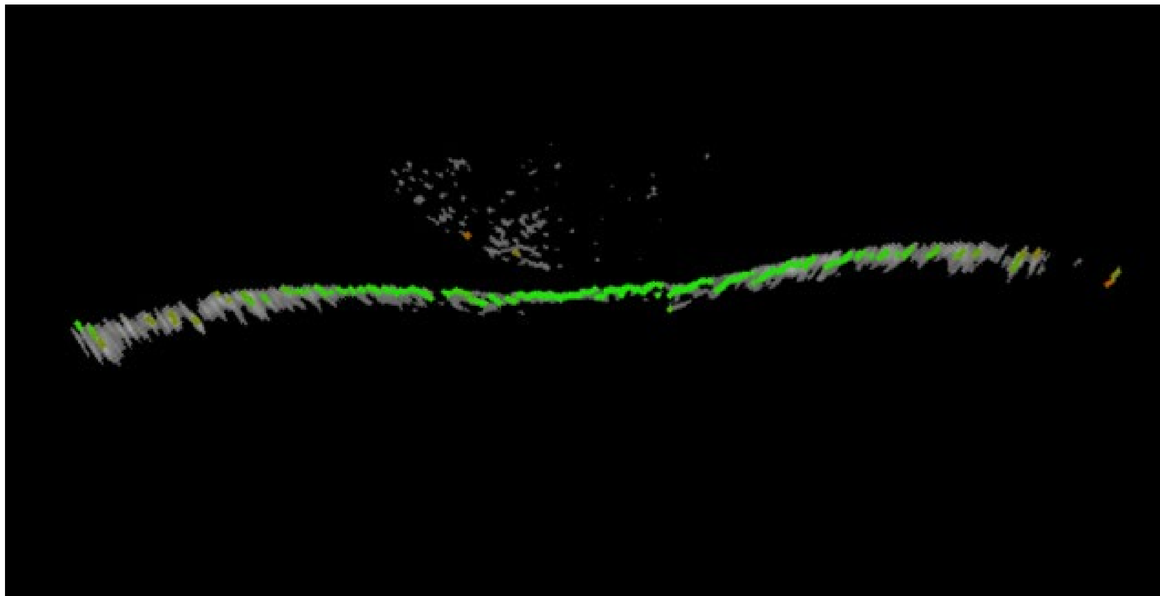


Figure 11: Features detected in a sonar scan (underwater inspection drone).

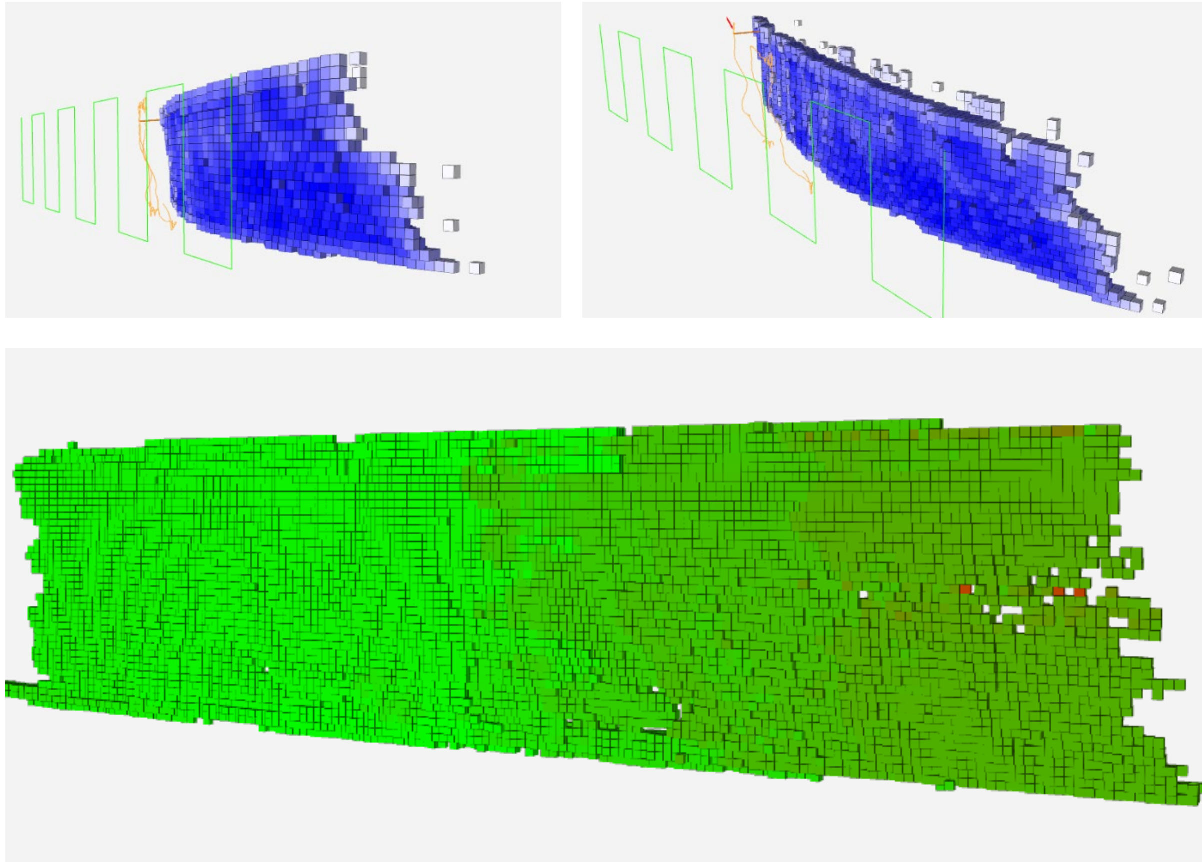


Figure 12: Inspection maps represented as a voxel map (underwater inspection drone).

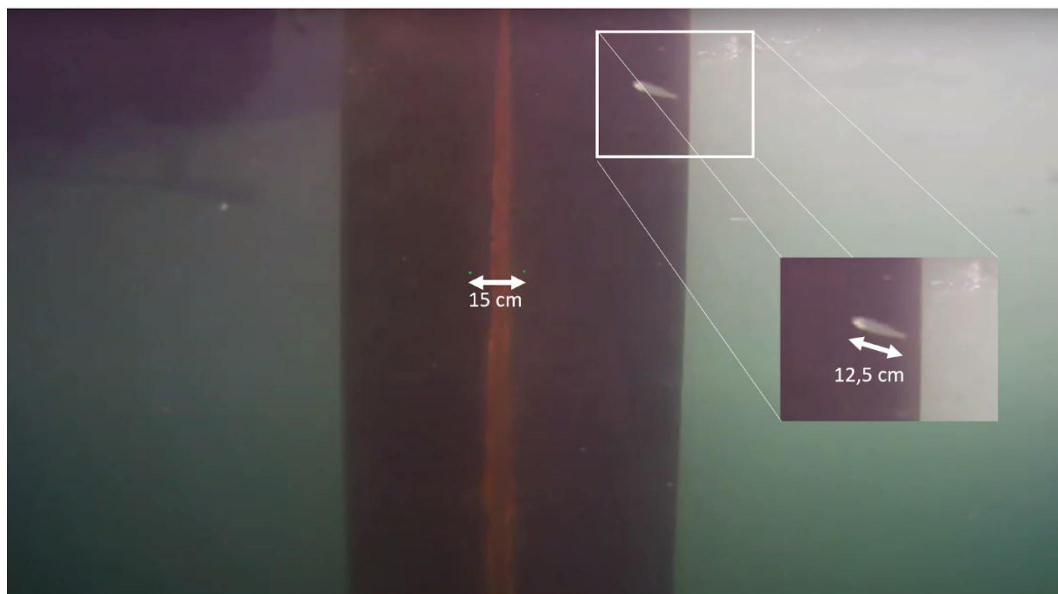


Figure 13: Perception of feature size by means of the laser-camera setup (underwater inspection drone).

Local Mapping and Obstacle Perception for the Inspection Crawler

In this section, we present the crawler's approach to the mapping problem (with CNRS, Roboplanet and UPORTO as partners involved), while taking into account contextual constraints such as obstacles, and the need to detect free space. Maps are important to robots, as long as they are useful for obstacle avoidance, path planning, or to constrain the attitude of the robotic system, among others. Maps become also important to human operators, as a way to provide visual feedback from the robot's perspective.

The crawler is equipped with an RGB camera, a 3D LiDAR, and an IMU. In addition, optical sensors capture wheel odometry. Further, a *Particle Filter* (PF) is used for localisation purposes by fusing IMU data, UWB range measurements and wheel odometry data. The pose estimated by the PF is further referenced as the PF pose. To compensate for motion uncertainties (due to e.g. drift), pose correction is performed using a constrained version of ICP, discussed in more detail below.

IV.1. Stop and map procedure

The majority of filtering techniques, such as PFs, introduce time delays, namely between the filter estimates and the actual observations. In that sense, the generated estimate was found to satisfy control requirements for autonomous driving, though, nevertheless, mapping proved to be more challenging. To solve this problem, a stop and map approach was implemented in the autonomous planner, i.e. the robot task manager.

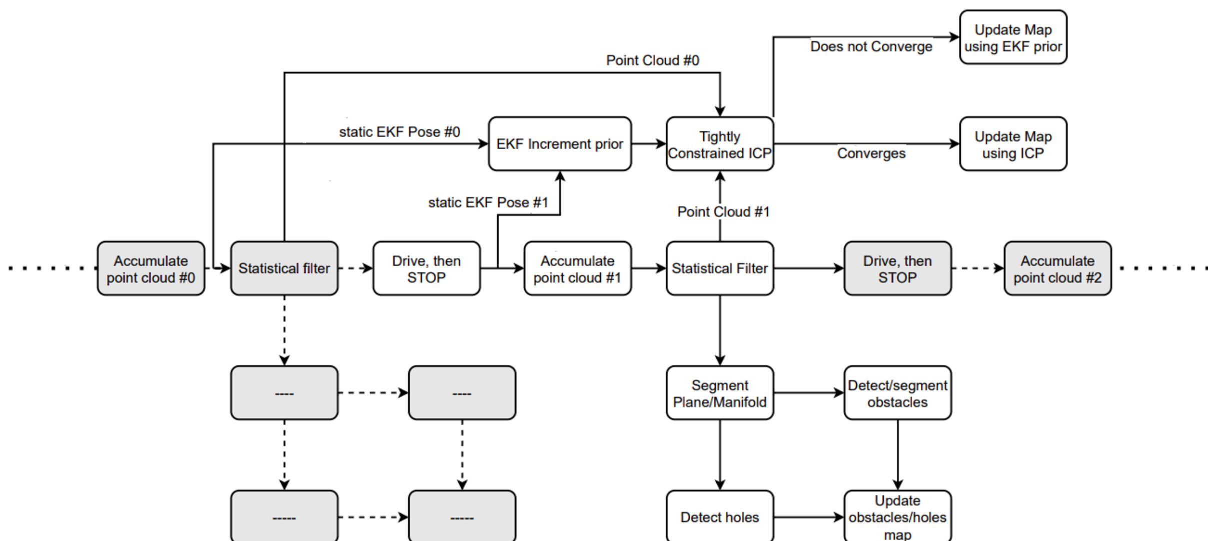


Figure 14: Flow chart of the mapper running onboard the aerial crawler: grayed-out rectangles denote repeated behaviour.

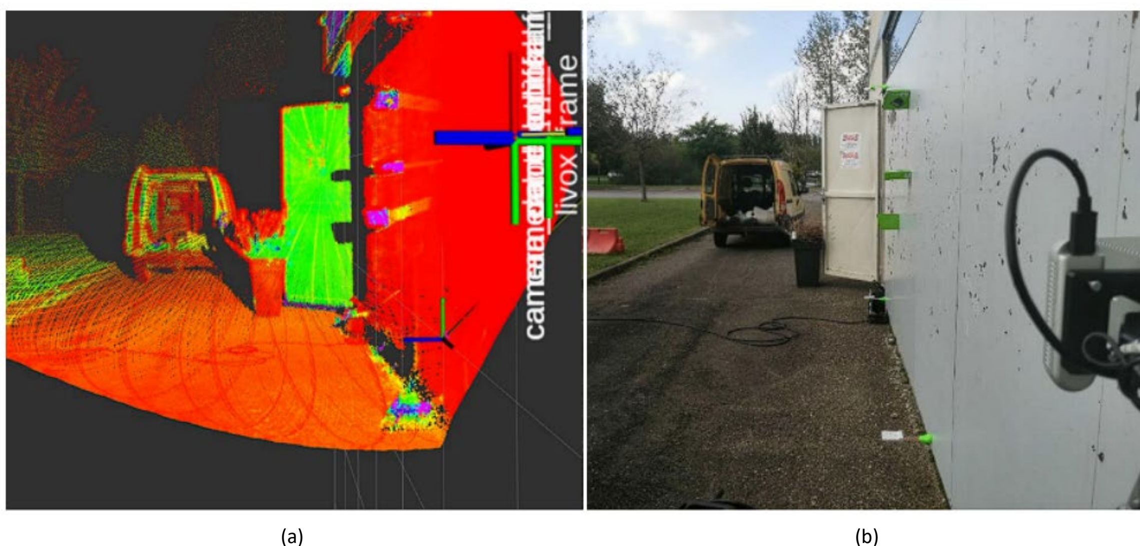
As shown in Figure 14, the mapper proposed is idle while the robot is moving and only captures data when the robot stops. Once static, the point clouds accumulate and the pose is captured.

The latter approach therefore does not suffer from data delays. The PF pose is then captured around half a second after the robot stops. Minimum stop time is set to 3 seconds, to allow the on-board laser scanner to accumulate sufficient points for data fitting. This is especially useful with 3D LiDARs equipped with a scanning unit, such as the Livox Mid-70 (see Deliverable D2.1 – *Crawler adaptation to BUGWRIGHT2's requirements* for the sensor suite). A sample accumulated cloud can be seen in Figure 15.

IV.2. Obstacle perception

After we have the data from the stop and map process, the accumulated point cloud is voxelized, and processed through RANSAC, by fitting a second-degree manifold. The choice of a second-degree manifold is rooted in the application in which the mapper will be used: ship hulls and storage tanks using autonomous robots for inspection are often significant in size; as a result, non-flat surfaces have a significant radius. The curvature is therefore locally negligible, i.e. the surface around the current position of the robot can be represented as a plane. Nevertheless, a second-degree manifold captures better the surface geometry at unique places with an important curvature, such as at the tip of the ship structure.

Finally, RANSAC inliers denote free, observable space that belongs to the detected manifold, while outliers denote positive obstacles such as protruding objects, and negative obstacles such as holes.



(a) (b)

Figure 15: RGB vs Intensity map, showing the metal plates used to test the aerial crawler: (a) Accumulated intensity point cloud, taken from the Livox Mid-70 sensor, and (b) RGB camera feed.

IV.3. Pose correction and ICP

ICP (*Iterative Closest Point*) is an algorithm used to stitch overlapping point clouds. It works by iteratively finding the transformation that better aligns point cloud pairs. An ICP prior on the transformation to-be-found improves the chances of converging to a valid solution.

Odometry-based PF still suffers from translational drift. To counteract that, ICP is used, in-between accumulated point clouds to reduce drift between successive stops. Nevertheless, ICP does not always properly converge on featureless surfaces. To overcome this issue, a constrained version of ICP is implemented. The purpose of these constraints is to prevent ICP from reducing the quality of the estimated PF pose when it does not properly converge. The list of constraints can be found in Table 1.

Table 1 : List of ICP constraints (aerial crawler)

Constraint type	**Value**
2D constraint	$\phi = \theta = Z = 0$
Maximum rotation norm	0.05 rad
Maximum translation norm	0.35 m
Minimum differential rotation error	0.01 rad
Minimum differential translation error	0.01 rad

After running few ICP iterations, and due to point cloud overlap, the density of points has to be standardised for both the newly accumulated point cloud and the previous ICP map. To that end, a density filter is applied to both inputs. Although the filter value depends on the point cloud density, the true purpose of it is to have the same density (value) for both inputs. The full list of values for the ICP parameters can be found in Table 2.

Table 2: List of ICP parameters used for pose correction (aerial crawler)

Parameter	**Mapper**
Matcher	KD tree matcher
Matcher KNN size	15
Error minimiser	Point to plane
Maximum number of iterations	25
Octree grid filter	0.01
Maximum input point density	400000
Maximum ICP map point density	400000

Finally, the map pose is corrected according to $P_{new} = P_{prev} P_{c_{prev}}^{-1} P_{c_{new}} C$, where C is the ICP correction, inferred by matching the current accumulated cloud to the previously accumulated point cloud, P_{prev} is the current pose in the reference frame of the map, $P_{c_{prev}}$ is the previously captured PF pose when the robot was still static, and $P_{c_{new}}$ is the most recently captured pose, with the robot also being static.

IV.4. From point clouds to texture maps

Up to this point, the proposed framework still lacks a high-level visual component, to be used by the system operator for visual feedback, manual driving, and debugging a possible snag. So far, point clouds have proven to be versatile data containers, and they are the precursors to creating maps. Nevertheless, there

is a need for a representation that is finite in space, and intelligible for people who are not point cloud experts. To that end, a multi-layer texture map was conceived.

The generated texture is a projection of the RGB image on the robot surface. In the latter context, we will assume that the ground is flat. Ground pixels are now projected onto the camera frame, for color extraction. To do so, we will use the pinhole model: $p = A[R|t]P_g$ where P_g is a 3D ground point, $[R|t]$ is the extrinsic matrix that provides the geometric connection between the LiDAR and camera frames, and A is the camera intrinsic matrix, obtained by checkerboard calibration. Finally, the colors of ground points $\mathrm{P_g}$ are inferred by copying the colors of the nearest pixel after projection, i.e., those of $\mathrm{p}(\mathrm{u}, \mathrm{v}, 1)$.

We have now projected the RGB image onto the ground surrounding the robot. What follows is the fusion of relevant semantic information, such as free spaces and obstacles, extracted from point cloud data. As such, pixels not seen by the LiDAR, i.e. unobservable space, will be marked in black, pixels belonging to obstacles will be marked in red, and free space will keep the original RGB colors. The texture map has 3 layers: (a) a bottom layer, consisting of a dynamically updated projection of the ground portion of the image, drawn at the estimated pose; (b) a middle layer, that overwrites the bottom layer using a clean representation, updated every time the robot stops; and (c) a top layer, consisting of meta data such as grid resolution.

Conclusions

Deliverable D4.2 summarises the developments regarding local mapping included in the control architectures of the BW2 robotic platforms that need them to attain their goals as part of the inspection framework. As has been described, local maps serve to different purposes in the different cases: the inspection crawler builds them using accumulated laser scans and navigation data, and use them to plan their own motion and map positive and negative obstacles; the underwater inspection drone builds local occupancy maps using sonar and navigation data, and use them to keep track of the inspection progress during the operation so that the operator can assess the reliability of the inspection; finally, the aerial inspection drone builds specific local maps (i.e. they are not the standard occupancy maps) also using 3D laser scans and navigation data, and make use of them for both efficient motion estimation and obstacle perception and collision avoidance during navigation.