ERDC TR-23-12

US Army Corps of Engineers®
Engineer Research and Development Center

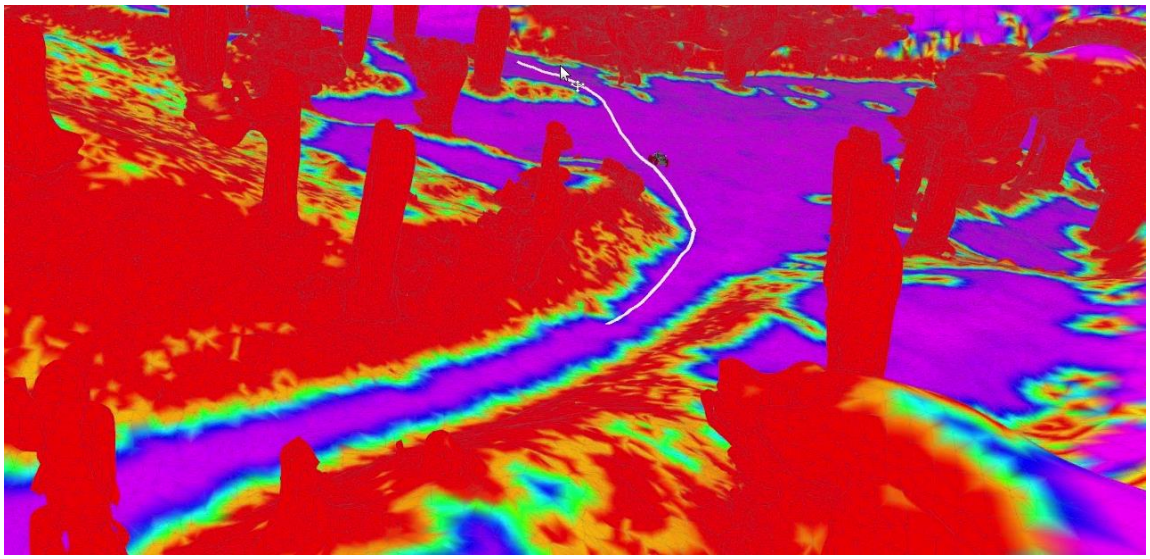
ERDC
ENGINEER RESEARCH & DEVELOPMENT CENTER

UGV-Localization in 3D and Path Planning (U-L3AP)

Unmanned Ground Vehicle (UGV) Path Planning in 2.5D and 3D

Osama Ennasr, Charles Ellison, Anton Netchaev,
Ahmet Soylemezoglu, and Garry Glaspell

August 2023



Engineer Research and Development Center

The US Army Engineer Research and Development Center (ERDC) solves the nation's toughest engineering and environmental challenges. ERDC develops innovative solutions in civil and military engineering, geospatial sciences, water resources, and environmental sciences for the Army, the Department of Defense, civilian agencies, and our nation's public good. Find out more at www.erdc.usace.army.mil.

To search for other technical reports published by ERDC, visit the ERDC online library at https://erdclibrary.on.worldcat.org/discovery.

ERDC TR-23-12
August 2023

Unmanned Ground Vehicle (UGV) Path Planning in 2.5D and 3D

Osama Ennasr and Gary Glaspell

US Army Engineer Research and Development Center (ERDC)
Geospatial Research Laboratory (GRL)
7701 Telegraph Road
Alexandria, VA 22315-3864

Charles Ellison and Anton Netchaev

US Army Engineer Research and Development Center (ERDC)
Information Technology Laboratory (ITL)
3909 Halls Ferry Road
Vicksburg, MS 39180-6199

Ahmet Soylemezoglu

US Army Engineer Research and Development Center (ERDC)
Construction Engineering Research Laboratory (CERL)
2902 Newmark Drive
Champaign, IL 61822

Final Technical Report (TR)

Prepared for US Army Engineer Research and Development Center (ERDC)
3909 Halls Ferry Road, Vicksburg, MS 39180-6199

Under FLEX-4 funding

Abstract

Herein, we explored path planning in 2.5D and 3D for unmanned ground vehicle (UGV) applications. For real-time 2.5D navigation, we investigated generating 2.5D occupancy grids using either elevation or traversability to determine path costs. Compared to elevation, traversability, which used a layered approach generated from surface normals, was more robust for the tested environments. A layered approached was also used for 3D path planning. While it was possible to use the 3D approach in real time, the time required to generate 3D meshes meant that the only way to effectively path plan was to use a preexisting point cloud environment. As a result, we explored generating 3D meshes from a variety of sources, including handheld sensors, UGVs, UAVs, and aerial lidar.

Contents

Figures

Preface

This study was conducted for the US Army Engineer Research and Development Center (ERDC) of the US Army Corps of Engineers. It was funded by ERDC under FLEX-4.

The work was performed by the Data Representation Branch, Topography Imagery and Geospatial Research Division of the ERDC Geospatial Research Laboratory (ERDC-GRL); the Computational Science and Engineering Division of ERDC, Information Technology Laboratory (ERDC-ITL); and the Warfighter Engineering Branch of ERDC, Construction Engineering Research Laboratory (ERDC-CERL). At the time of publication, Mr. Vineet Gupta was branch chief, Mr. Jeff Murphy was division chief, and Dr. Austin Davis was the technical director of GRL. The deputy director of ERDC-GRL was Ms. Valerie L. Carney, and the director was Mr. David R. Hibner. Dr. Jeffrey Hensly was division chief, and Dr. Rob Wallace and Mr. Ken Pathak were technical directors of ERDC-ITL. The deputy director of ERDC-ITL was Dr. Jacqueline Pettway, and the director was Dr. David Horner. Mr. Jeff Burkhalter was branch chief, Dr. George Calfas was division chief, and Mr. Jim Allen was the technical direction of ERDC-CERL. The deputy director of ERDC-CERL was Ms. Michelle Hanson, and the director was Dr. Andrew Nelson.

The authors acknowledge the following individuals for their contributions to this project: Mr. Steven Bunkley, Mr. Rich Curran, and Mr. Phil Devine, for technical guidance and point cloud data that were used to generate the various 3D traversability meshes.

The commander of ERDC was COL Christian Patterson, and the director was Dr. David W. Pittman.

This page intentionally left blank.

1 Introduction

1.1 Background

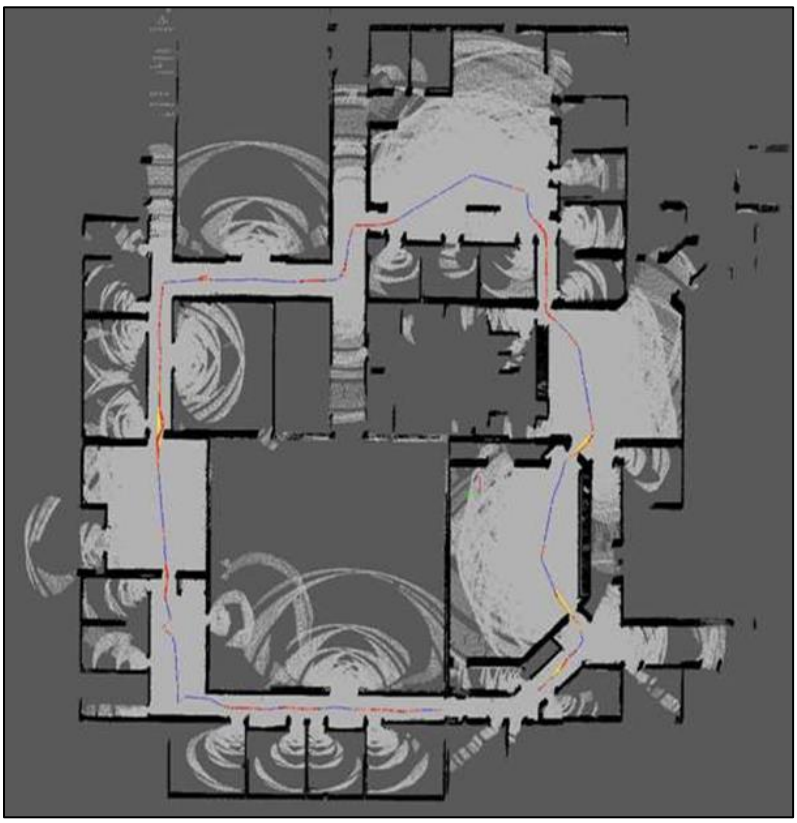
Robot navigation can typically be broken down into three categories: *teleoperation*, *waypoint navigation*, or *full autonomy*. In the simplest form of teleoperation, a user remotely controls the robot with a joystick. Waypoint navigation still requires a user to set the waypoint or provide a list of waypoints, but the robot does the path planning and obstacle avoidance necessary to reach its goal. In contrast, full autonomy allows the robot to choose its own waypoints (Christie et al. 2021).

Both waypoint navigation and full autonomy require a map for operation. This map can be provided before the robot begins navigating or can be generated on the fly using simultaneous localization and mapping (SLAM). This map, henceforth referred to as an *occupancy grid*, has historically been rendered in 2D and is typically a grayscale image in the portable gray map (PGM) format. Grayscale pixel values range from 0 (for white) to 255 (for black) and can be used in 2D planning, which is generally trinary in nature. Specifically, the occupancy grid is used to track *known*, *unknown*, and *obstacles*. The known space is generally assigned the lowest values, and obstacles are typically assigned the highest value (Glaspell et al. 2020). Figure 1 is an example of a 2D occupancy grid of a building. The walls are black and are considered obstacles. The floor is light gray and is considered known space. Areas the robot has not mapped are darker gray and are considered unknown space. As the robot explores, unknown areas are converted to known areas via ray tracing. The multicolored red, blue, and yellow indicates the path of the robot. We typically use a 3D lidar to determine obstacles. The navigation stack, comprised of open source software and commercial off-the-shelf sensors, plans paths in the known space while simultaneously avoiding both static and dynamic obstacles.

The current Robot Operating System (ROS) navigation stack ingests 2D occupancy grids and uses them to plan routes. The navigation stack uses a layered approach to achieve this goal. At a minimum, we typically specify three layers: *static*, *obstacle*, and *inflation*. The static layer uses stored maps, created by SLAM or previously generated by the user, to define obstacles the robot will encounter while moving from point A to point B. The

obstacle layer uses live sensor input, such as a lidar, to add real-time obstacles to the occupancy grid using the parameters `neutral_cost` and `lethal_cost`, which we set to 50 and 253, respectively. The inflation layer adds padding to obstacles to keep the robot from getting too close to them.

Figure 1. A 2D occupancy grid of a building.



We can combine these layers in various ways to generate costmaps. Costmaps are occupancy grids with costs assigned to each cell. We typically use two costmaps for navigation: a *global* costmap and a *local* costmap. Further, each costmap has its own planner. Planners attempt to find the path with the lowest costs. Global planner is used for long-term routes, and local planner is used for short-term collision avoidance. The global costmap typically uses static and inflated layers. The local costmap typically uses the obstacle and inflation layers. While it is possible to include the obstacle layer in the global costmap, it is not strictly necessary to do so. This is because the local costmap or planner is given the highest priority. This is especially helpful in a dynamic environment. If, for example, a dynamic obstacle was included in the static layer but was moved, the local costmap would indicate the change.

While this simple 2D approach works well in most circumstances, there are cases in which higher dimensional navigation can improve performance. For instance, consider a scenario that has both a paved road and loose gravel. We would prefer the robot to drive on the road, especially if we are using wheel encoders for odometry. However, a typical 2D costmap would label both the paved and loose-gravel roads as known spaces with zero cost. As a result, the robot would simply take the shortest path to its destination. However, with a 2.5D occupancy grid, the loose-gravel road could have an increased, but not lethal, cost, and the global path planner would plan along the road as much as possible. Another scenario that might require higher dimensional navigation is placing waypoints on a bridge. With a 2D, or even a 2.5D, occupancy grid, it is difficult to determine if the waypoint is supposed to be on or below the bridge. However, navigation in 3D makes it possible to distinguish whether the waypoint is on or under the bridge. This report explores both 2.5D and 3D navigation.

1.2 Objective

This report addresses the *Army Multi-Domain Intelligence: FY21–22 S&T Focus Areas* (Office of the Deputy Chief of Staff 2020). Specifically, in the sensors section, this report correlates with the following need: "Novel combinations of sensors and robotic platforms that can not only move across terrain, but maneuver to sense" (4). Traditionally, an unmanned ground vehicle (UGV) navigates in 2D and divides the world into known, unknown, and obstacles. This report attempts to expand navigation into 2.5D and 3D by including information about the actual ground surface in the maps. Higher dimensions result in more efficient navigation.

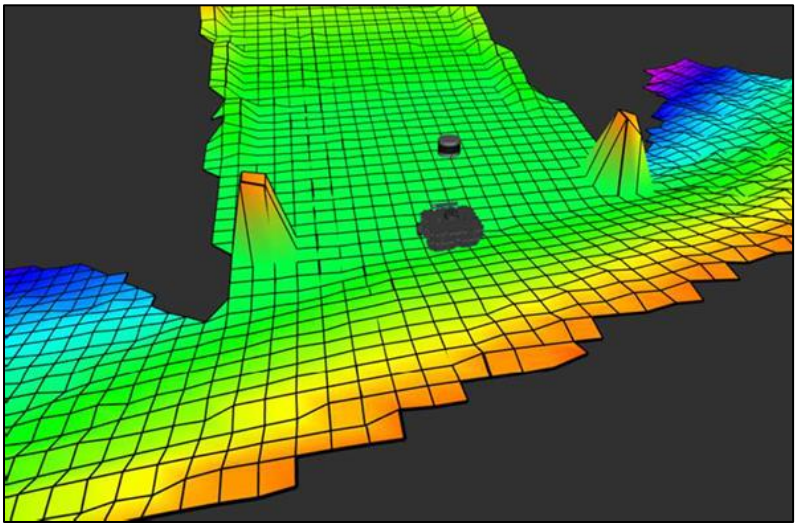
This work also addresses the statement, "Wars will be fought at hyper speed and scale, dominated by technologies such as robotics and autonomous systems (RAS), machine learning (ML), and AI [artificial intelligence] capabilities, which are widely available, packaged, and ready for use" (Office of the Deputy Chief of Staff 2020, 5). While outside the scope of this article, the 2D frontier-based exploration packages (Christie et al. 2021) will work out of the box with our 2.5D approach.

1.3 Approach and Scope

Herein, we explore two ROS packages specific to 2.5D and 3D navigation. For the generation of 2.5D occupancy grids, we used the `grid_map` package (Fankhauser 2019; Fankhauser and Hutter 2016). Although the `grid_map`

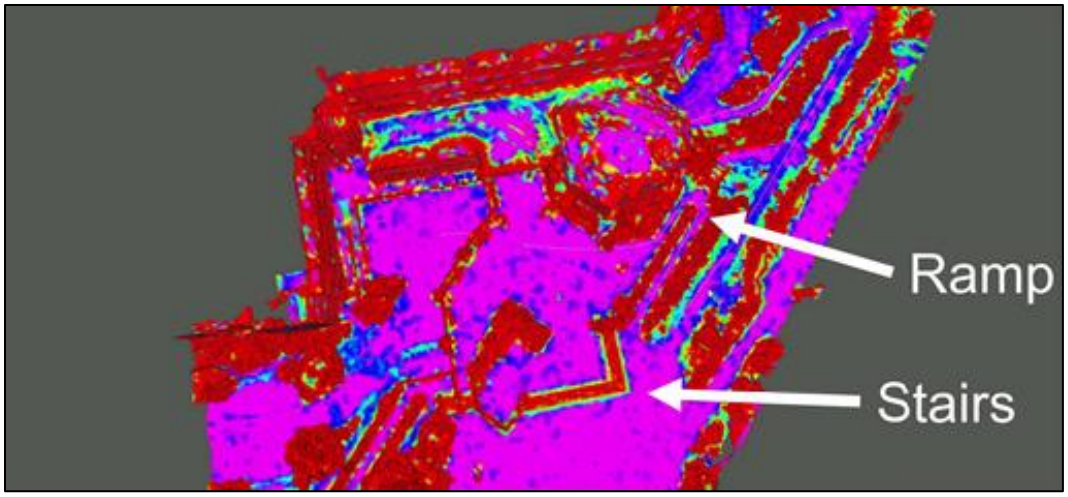
package was originally designed for legged robots to navigate various terrains, it can also be used on UGV platforms to maintain information about the ground surface. Figure 2 shows an example of a grid map. Typically, we use 3D lidar to generate the grid map. However, the grid map here is 2.5D, which means that we represented the *z*-direction with a single value. If the point cloud provides two different coordinates with the same *x* and *y* values but different *z* values, the grid map keeps the higher *z* value. Cooler colors, such as purple and blue, indicate lower elevation, while warmer colors, such as red and orange, indicate higher elevation. The `grid_map` package provides useful conversions to a variety of formats, including costmap_2d, OpenCV, OctoMap, Point Cloud Library (PCL), and even Signed Distance Field (SDF). The costmap_2d conversion was the most pertinent for our navigation. However, the ability to view the pointcloud2, Vectors, and GridCells data is especially helpful when tuning parameters because they each have their own parameter file that can be adjusted for specific scenarios.

Figure 2. Example of a 2.5D grid map.



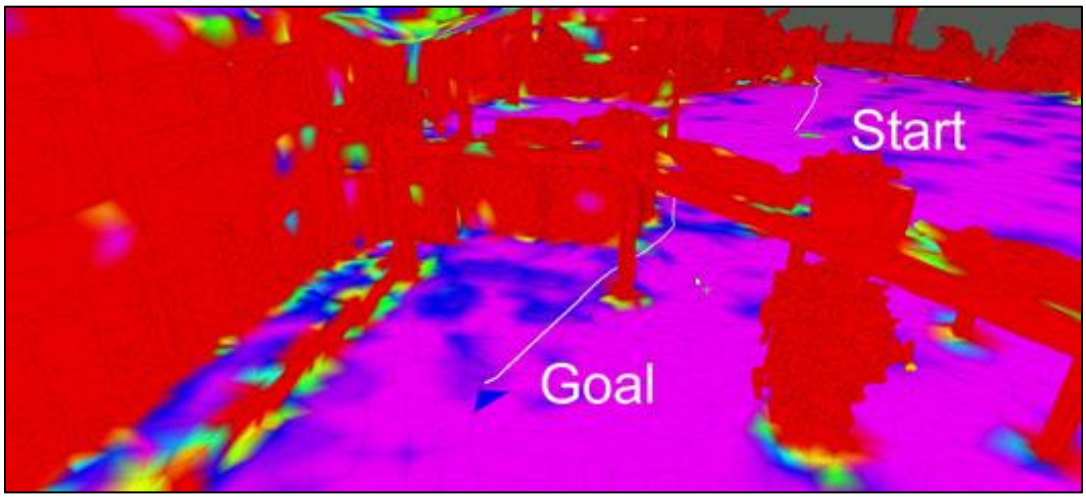
We also investigated `mesh_navigation` for 3D path planning (Pütz 2019; Pütz et al. 2021). Figure 3 contains an example of mesh navigation. In this example, cooler colors, such as pink and blue, indicate traversable terrain, while red is reserved for obstacles. One advantage of using mesh navigation is that steep slopes (i.e., negative obstacles), such as stairs, are labeled red and avoided. As a result, the robot's path is planned down the ramp.

Figure 3. Example of a 3D-generated mesh.



Another advantage of using mesh navigation is that paths are planned along the actual mesh, so gaps and steep slopes are easily recognized and avoided. In addition, waypoints can have different *z* values, and the planner explicitly considers this. In Figure 4, for example, the starting location is higher in elevation, and the goal is lower in elevation. As a result, the mesh navigation can successfully direct the robot to the goal position using a route that best achieves the change in elevation. This is not possible with a 2D or 2.5D approach.

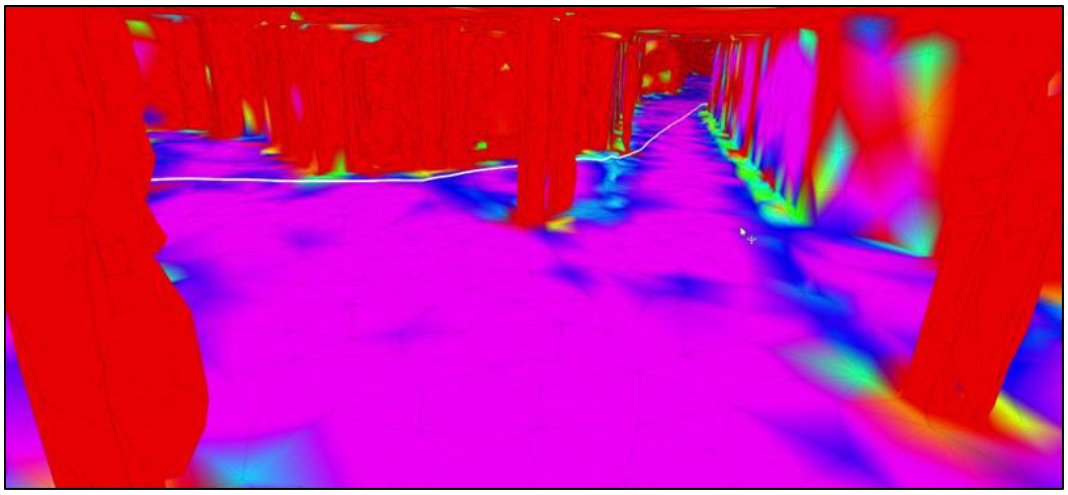
Figure 4. Example of path planning on a 3D mesh.



Additionally, with a mesh, multiple points can exist along the same *z*-axis, as shown in Figure 5. This is also not possible with a 2D or 2.5D approach. Figure 5 also demonstrates that the `mesh_navigation` package can

efficiently use the 3D mesh to plan a path through the tunnel to reach its intended goal.

Figure 5. Example of 3D path planning through a tunnel.



The figures in this section were generated using an online dataset (University of Osnabrück 2020). However, the remainder of this report will focus on the generation of traversable meshes from a variety of input sources.

2 Research and Discussion

2.1 2.5D Navigation

2.1.1 Installing Elevation Mapping

The ROS package `elevation_mapping` uses the robot's pose information, a distance sensor (e.g., 3D lidar), and the underlying `grid_map` package to create 2.5D grid maps that the robot can use for navigation (Fankhauser et al. 2014, 2018). The GitHub site for elevation mapping (Fankhauser and Wulf 2019) links all the necessary dependencies to compile the package, including Grid Map (Fankhauser 2019), Kindr (Gehring, Bellicoso, et al. 2019), Kindr ROS (Gehring, Fankhauser, et al. 2019), PCL (Rusu and Cousins 2011), and Eigen (Jacob and Guennebaud, n.d.). Detailed instructions on compiling the project are also provided, including how to clone a project into a catkin workspace and compile it with catkin. The catkin workspace in ROS-based software is where you build, modify, and install ROS-based packages. Note that, when compiling, ensure that the flag `-DCMAKE_BUILD_TYPE=Release` is passed for better performance.

2.1.2 Running Elevation Mapping

Elevation mapping subscribes to three topics: points (`sensor_msgs/PointCloud2`), transforms (`tf/tfMessage`), and pose (`geometry_msgs/PoseWithCovarianceStamped`). The transforms describe the locations of sensors on the robot, and pose is the representation of the robot in free space. We used a 3D lidar to generate the points topic, and static transforms were used to generate the transforms tree. For the pose topic, we used the following node.

```
<!-- Publish tf base_footprint as pose.-->
<node pkg="elevation_mapping_demos" type="
 tf_to_pose_publisher.py" name="waffle_pose_publisher">
 <param name="from_frame" type="string" value="odom"/>
 <param name="to_frame"   type="string" value="
 base_footprint" />
</node>
```

In our specific use case, we used fast lidar odometry and mapping (FLOAM; Flynn 2021) to generate the odom frame. The odom frame is a short-term local reference that tracks the robot's position as it moves. Alternatively, the laser scan matcher package (CCNY Robotics Lab, n.d.) could also be used to generate the `PoseWithCovarianceStamped` message.

Laser scan matcher needs to be compiled from the source, and the following parameter is required.

```
<param name="publish_pose_with_covariance\_stamped"
   value="true"/>
```

Once pose was established, the elevation mapping node was launched with the following launch script. The node itself calls three separate YAML files, which are included in Appendix A. The YAML defines the parameters required by the node.

```
<!-- Launch elevation mapping node. -->
<node pkg="elevation_mapping" type="elevation_mapping"
 name="elevation_mapping" output="screen">
  <rosparam command="load" file="/home/garry/
 Downloads/tb3/elevation_mapping/tb3.yaml" />
  <rosparam command="load" file="$(find_
 elevation_mapping)/config/sensor_processors/
 velodyne_HDL-32E.yaml" />
  <rosparam command="load" file="/home/garry/
 Downloads/tb3/elevation_mapping/
 postprocessor_pipeline.yaml" />
</node>
```

The complete tb3.yaml file is listed in Appendix A, Section A.1. We made several modifications. The original `map_frame_id` parameter was set to odom. However, we wanted to pass the occupancy grid to `move_base`, which required it to be in the map frame. As a result, we set `map_frame_id` to map. Furthermore, we reduced `min_update_rate` to 1 to match the rate of our global planner. Because the resulting occupancy grid was going to be 2.5D, it was important to set `sensor_processor/ignore_points_above` to match the height of the robot. This kept overhead obstacles that the robot could drive under from showing up as obstacles in the global cost map. We also increased the parameters for `length_in_x` and `length_in_y` to 50.0 for a significantly larger map. Elevation map uses a circular buffer. As a result, the map moves with the robot. However, the resulting occupancy grid can still be passed to the static layer of the global cost map, achieving persistence. The YAML files `velodyne_HDL-32E.yaml` and `postprocessor_pipeline.yaml`, found in Appendix A, Sections A.2 and A.3 respectively, were used without modification. If using a depth sensor other than a 3D lidar, the GitHub repository lists other configurations for 2D lidars and red-green-blue and depth (RGB-D) cameras (ANYbotics 2021).

The elevation mapping package outputs two topics, specifically `elevation_map` and `elevation_map_raw`, both in a custom message type (`grid_map_msgs/GridMap`). To generate the 2.5D occupancy grid from the 2.5D grid map, we used the code that follows. The YAML file is provided in Appendix A, Section A.4. The values `data_min` and `data_max` should be tuned to the environment.

```
  <node pkg="grid_map_visualization" type="grid_map_visualization"
   name="grid_map_visualization" output="screen">
     <rosparam command="load" file="/home/garry/
   Downloads/tb3/elevation_mapping/costmap.yaml" />"
  </node>
```

2.1.3 Testing Elevation Mapping in Simulation

With the ability to generate 2.5D occupancy grids established, we began looking for simulated worlds to use for testing. ROS uses Gazebo to simulate the physics of a robot in an environment. For this particular application, we used the Clearpath additional simulation worlds, specifically Office Construction World (Clearpath Robotics 2020a) and the Inspection World (Clearpath Robotics 2020b).

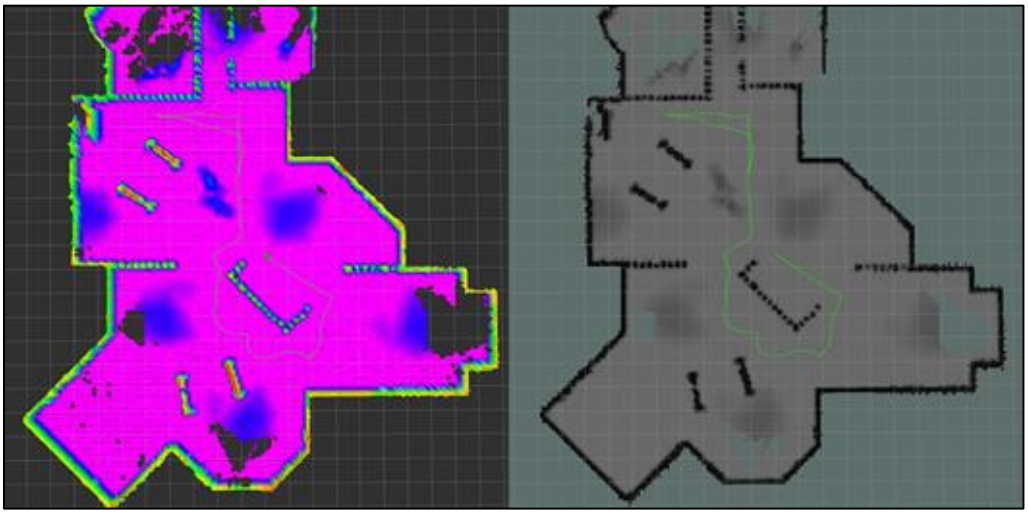
The simulated Office Construction World is shown in Figure 6. This world is particularly interesting because it has building materials scattered along the ground. Reflecting back on the 2D occupancy grid shown in Figure 1, there are three possible cell designations: obstacles, known space, and unknown space. In the traditional 2D occupancy grid, all obstacles are considered lethal and thus avoided, and known space is equally free to traverse. The advantage of a 2.5D occupancy grid is that we can maintain information about the ground surface and identify the sand piles as nonlethal obstacles. This means that the sand piles should be avoided if a safer path is identified but could be traversed if no other path existed.

Figure 7 shows the resulting 2.5D grid map (on the left) and the converted occupancy grid (on the right) of the Office Construction World. In the grid map, the floor is pink, the walls are green, and the construction materials, including the sand piles, are blue. The occupancy grid generated from the grid map is shown on the right. In the occupancy grid, the sand piles can be easily identified and are considered nonlethal obstacles. In this example, the walls, shown in black, would be considered lethal obstacles. The robot's path is shown in green, and when setting waypoints, the sand piles were avoided because a lower-cost path, going around them, was

calculated. In contrast, a traditional height-based ground filtering would identify the top of the pile as an obstacle and the base as free space. This would result in the robot edging around the base of the pile with one set of wheels on the sand. The 2.5D occupancy grid avoids these issues by having the global planner plan around the sand piles entirely.

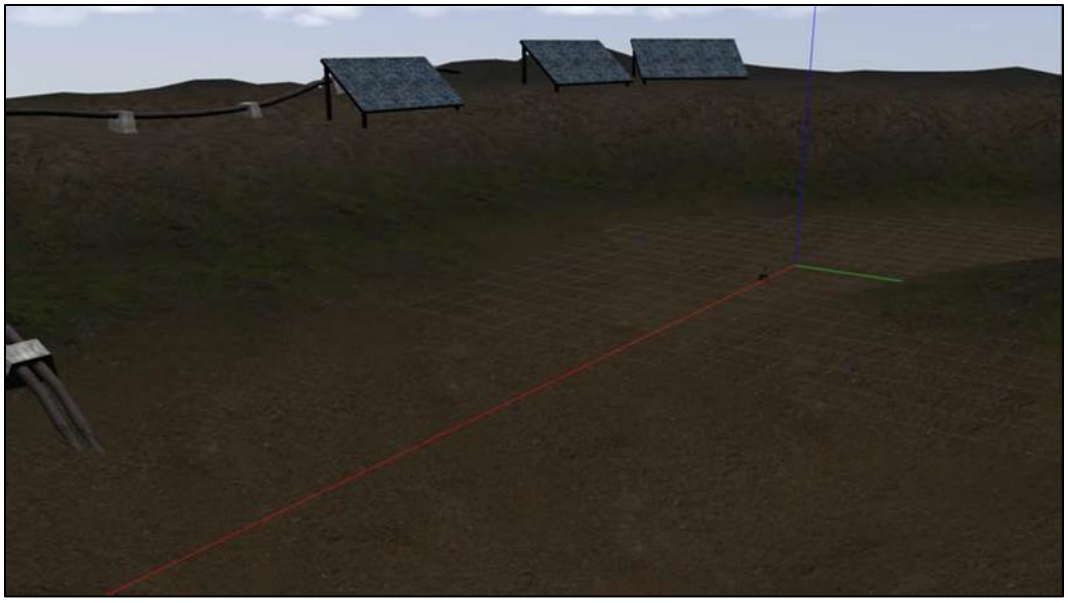
Figure 6. Clearpath Office Construction World.

Figure 7. Elevation-based 2.5D grid map (*left*) and 2.5D occupancy grid (*right*) of the Office Construction World.



Next, we tested the Clearpath Inspection World. Inspection World is unique because it has 3D terrain with gently sloping hills, compared to the relatively flat Office Construction World. Figure 8 shows the Clearpath Inspection World.

Figure 8. Clearpath Inspection World.

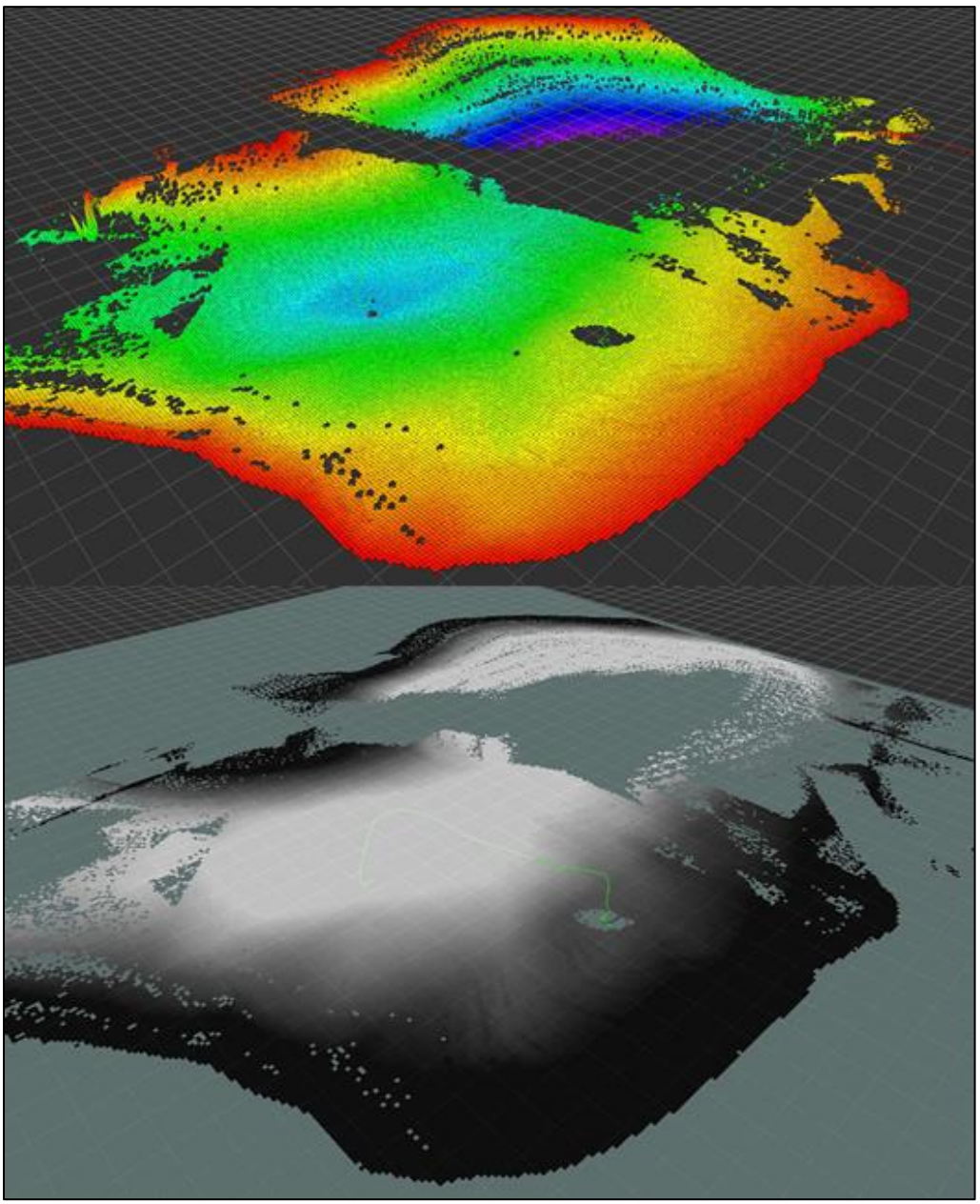


One thing we noticed, while exploring this world, was that the ground quickly became a lethal obstacle when driving up a hill. This is a result of `data_min` and `data_max` being too restrictive. While the local costmap attempts to clear these obstacles, the global planner fails or plans unnecessarily longer routes to navigate around them. As a result, we significantly increased the `data_min` and `data_max`. Figure 9 shows the resulting 2.5D grid map (top) and occupancy grid (bottom). While these elevated values increase the navigable surface area of the world, obstacles relevant to the robot's scale are no longer registered. This is due to the elevation factor functioning as a ratio. For example, with the inflated values, the sand piles in the previous example would not be evident in the global costmap. Also, when setting waypoints along the sides of the hill, the robot plans its path by dropping into the valley (i.e., lower cost) before traversing back up the hill to reach the waypoint (i.e., higher cost). This is because the cost is based solely on elevation.

Based on the aforementioned limitation when using the elevation layer to generate 2.5D occupancy grids, we began investigating traversability grid maps. Instead of elevation, we used the slope, edges, and roughness layers. These layers were calculated using surface normals and can be combined to generate the traversability grid map. To convert the traversability grid map to a 2.5D occupancy grid, we used the same method as before. To generate the traversability grid map, we used the following node:

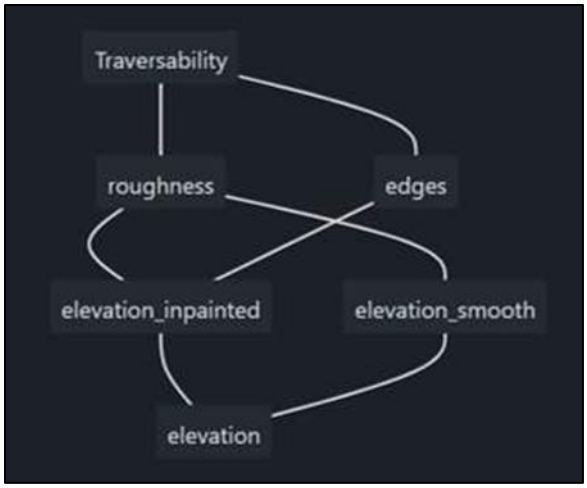
```
<node pkg="grid_map_demos" type="filters_demo" name="
 grid_map_filter_demo" output="screen">
 <!-- Input topic-->
     <param name="input_topic" value="/
 elevation_mapping/elevation_map" />
     <!-- Output topic-->
     <param name="output_topic" value="filtered_map"/>
     <!-- Load grid map filter chain configuration-->
 <rosparam command="load" file="/home/garry/
 Downloads/tb3/elevation_mapping/
 filters_demo_filter_chain.yaml" />
</node>
```

Figure 9. Elevation-based 2.5D grid map (*top*) and 2.5D occupancy grid (*bottom*) of the Inspection World.



Appendix A, Section A.5, contains the `filters_demo_filter_chain.yaml`. The flowchart shown in Figure 10 depicts the layers that were used to generate the traversability layer. In the flowchart and corresponding YAML file, traversability was generated using the `MathExpressionFilter` using the sum of the *roughness* and *edge* layers, shown in the code that follows.

Figure 10. Traversability flowchart.




```
- name: traversability
  type: gridMapFilters/MathExpressionFilter
  params:
    output_layer: traversability
    # expression: 0.5 * (1.0 - (slope/ 0.6)) + 0.5 *
  (1.0 - (roughness/ 0.1))
    expression: ((roughness + edges) / 2)
```

The roughness layer is also `MathExpressionFilter`, but it takes the absolute value of the difference between the layers for the raw expression of elevation (i.e., `elevation_inpainted`) and a smoothed expression of elevation (i.e., `elevation_smooth`).

```
- name: roughness
  type: gridMapFilters/MathExpressionFilter
  params:
    output_layer: roughness
    expression: abs(elevation_inpainted -
  elevation_smooth)
```

The inpaint layer is an `InPaintFilter` and uses elevation as input.

```
    - name: inpaint
  type: gridMapCv/InpaintFilter
  params:
    input_layer: elevation
```

```
    output_layer: elevation_inpainted
    radius: 0.05
```

The smoothing layer is a `SlidingWindowMathExpressionFilter` and also uses elevation as input.

```
    - name: boxblur
  type: gridMapFilters/
SlidingWindowMathExpressionFilter
  params:
    input_layer: elevation
    output_layer: elevation_smooth
    expression: meanOfFinites(elevation)
    compute_empty_cells: true
    edge_handling: crop # options: inside, crop,
empty, mean
    window_size: 5 # optional
```

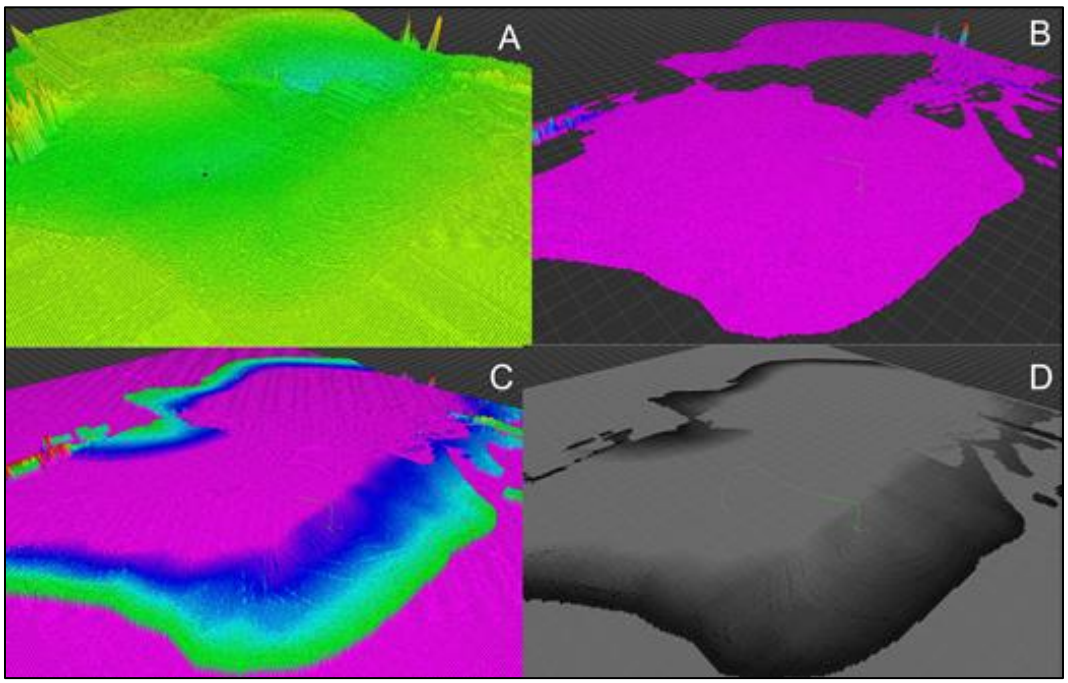
The edges layer is also a `SlidingWindowMathExpressionFilter` and also uses `elevation_inpainted` as input.

```
    - name: edge_detection
  type: gridMapFilters/
SlidingWindowMathExpressionFilter
  params:
    input_layer: elevation_inpainted
    output_layer: edges
    # expression: 'sumOfFinites
([0,1,0;1,-4,1;0,1,0] .*elevation_inpainted)' # Edge
detection.
    expression: 'sumOfFinites
([0, -1,0;-1,5,-1;0,-1,0].*elevation_inpainted)' #
Sharpen.
    compute_empty_cells: false
    edge_handling: mean # options: inside, crop,
empty, mean
    window_size: 3 # Make sure to make this
    compatible with the kernel matrix.
```

Figure 11 shows the results of using the traversability layer in the Inspection World. Figure 11*A* shows the results when applying the edge detection layer, and Figure 11*B* shows the results of the roughness layer. These two layers were combined to form the 2.5D traversability grid map shown in Figure 11*C*. Finally, the traversability grid map from *C* was converted to the 2.5D occupancy grid shown in Figure 11*D*. There are a few notable differences when comparing the 2.5D traversability occupancy grid with the 2.5D elevation based occupancy grid in Figure 9. First, the valley in the traversability grid map has the same cost (i.e., it is the same color). As a result, path planning was more efficient. Also, though they appear similar to

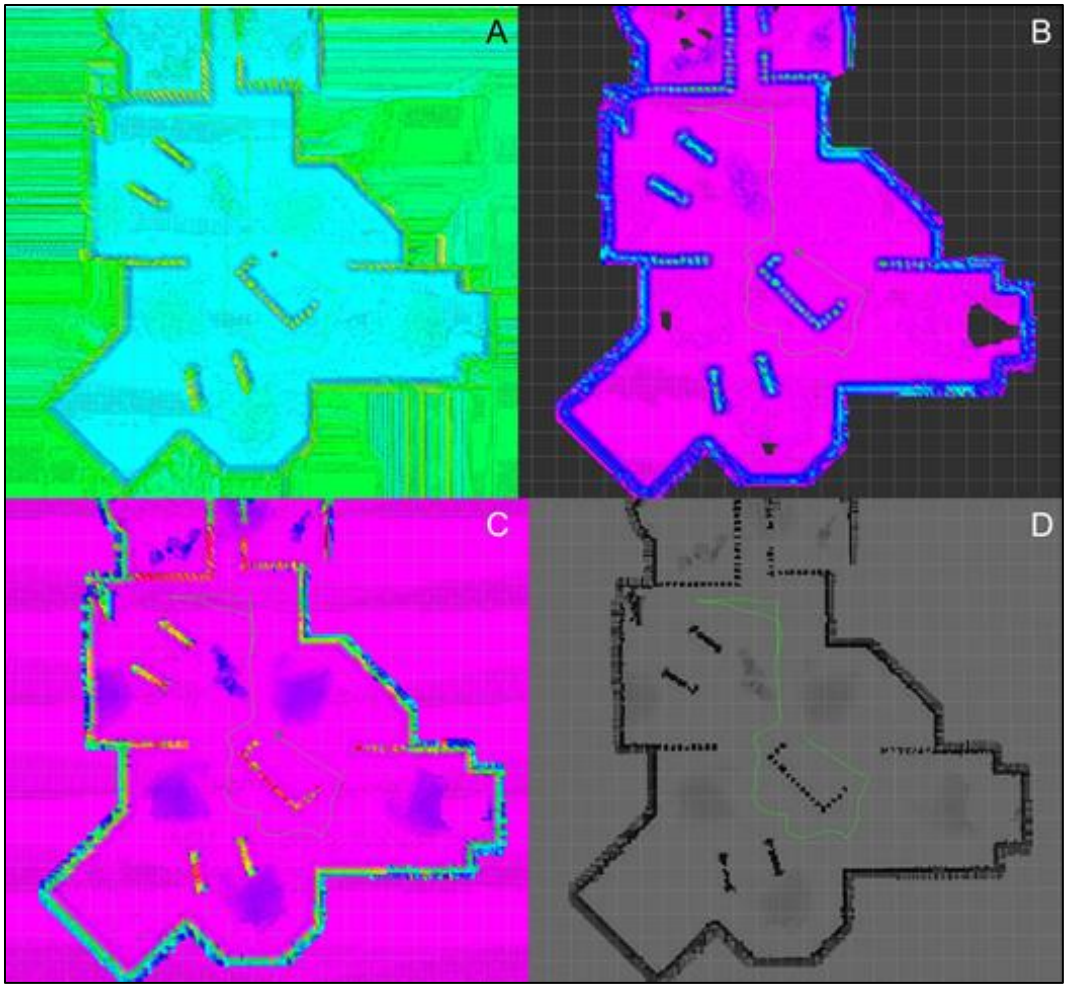
the obstacles in the elevation map, the lethal obstacles shown in the traversability grid map resulted from combining the edge and roughness layers. In the elevation approach, obstacles were marked solely on the basis of elevation. The values in the 2.5D occupancy grid range from O to 255. We set the lethal obstacles at 253; thus, the robot was able to navigate into the dark gray regions. Reducing the lethal obstacle value would diminish how far into the dark gray regions the robot would traverse.

Figure 11. Traversability grid map and 2.5D occupancy grid of the Inspection World. *A* shows the results when applying the edge detection layer, and *B* shows the results of the roughness layer. These two layers were combined to form the 2.5D traversability grid map shown in *C*. The traversability grid map from *C* was converted to the 2.5D occupancy grid shown in *D*.



To complete our analysis, we ran the same traversability layer, without any modifications, on the Inspection World. In the elevation example, we had to change the parameters so that the ground would not be considered a lethal obstacle in Inspection World. The generated grid maps for the Inspection World are shown in Figure 12. Again, the edges, roughness, and traversability are listed as *A*, *B*, and *C*, respectively. Note that the 2.5D occupancy layer, shown in *D*, looks similar to the 2.5D occupancy grid shown in the elevation example (Figure 7). In Figure 12*D*, the sand piles are identified as a nonlethal obstacle. Moving forward, we plan to use the traversability layer to generate 2.5D occupancy grids because it is more robust than the elevation-based approach.

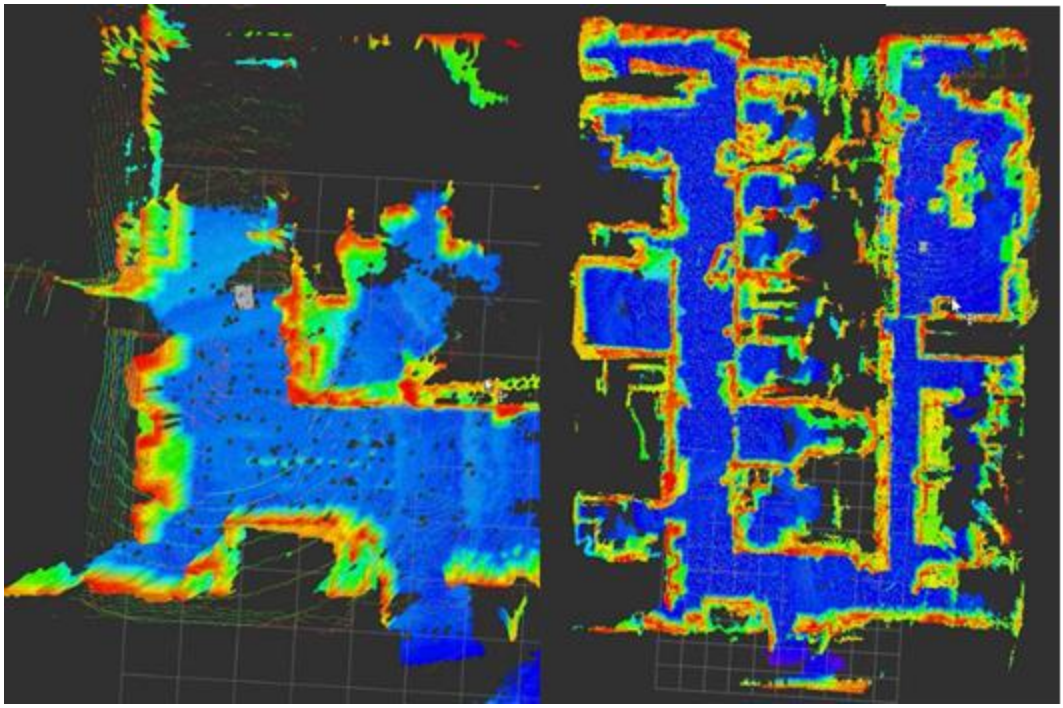
Figure 12. Traversability grid map and 2.5D occupancy grid of the Office Construction World. *A*, *B*, and *C*, respectively, show the edges, roughness, and traversability. *D* shows the occupancy grid.



2.1.4 Testing Elevation Mapping on the Physical Robot

With testing in simulation complete, we tried our modified version of elevation mapping on a physical robot. Figure 13 shows the 2.5D grid maps of a building. The image on the left is zoomed in to show detail, while the image on the right is of the entire grid map. As in the simulation, FLOAM was our source of odometry. In this test, we were able to set waypoints and explore the entire area. Moving forward, we plan to test this approach in much more rigorous environments.

Figure 13. Real-world example of a 2.5D grid map.



2.2 3D Navigation

2.2.1 Installing Mesh Navigation

The ROS package `mesh_navigation` uses a variety of packages, including `mbf_mesh_core` and `mbf_mesh_nav`. The `mbf_mesh_core` package contains the `MeshPlannerinterface`. This includes the `dijkstra_mesh_planner` and `cvp_mesh_planner` 3D path planners. The `mbf_mesh_nav` is the navigation stack written upon `move_base_flex` (Magazino 2018). The `move_base_flex` package was based off of the original `move_base` code, but it allows the use of other map representations, such as meshes created by Pütz et al. (2018). The ROS package `mesh_navigation` also uses `mesh_map`, which allows for a layered approach similar to the layered approach that is used in `elevation_mapping`. Specifically, `mesh_layers` allows for the use of HeightDiffLayer, RoughnessLayer, SteepnessLayer, RidgeLayer, and InflationLayer. Note that the `mesh_map` package typically accepts *HDF5* files as input. The HDF5 file format is specifically designed to contain large amounts of data. To make the `mesh_map` package more robust, we modified the `mesh_map` package to accept *PLY* files. The PLY file format was designed to store 3D data using flat polygons. Appendix B contains the modified files. The `mesh_navigation` packages have a number of dependencies, including the `lvr2` (University of Osnabrück 2018), `mesh_tools` (Nature Robots 2020), and `move_base_flex` (Magazino 2018). Detailed instructions

on compiling the project are provided on the GitHub site (Pütz 2019) and involved cloning the project into your catkin workspace and compiling it with catkin.

2.2.2 Running Mesh Navigation

Mesh navigation uses the Las Vegas Reconstruction Toolkit 2.0 (LVR2, n.d.) to represent meshes (Wiemann et al. 2015). These meshes use a half-edge mesh (HEM) structure (Wiemann et al. 2022). In addition to information about the faces, vertices, and edges, there is an additional element that relates to how the mesh elements are connected ("Introduction to Half-Edge" 2020). We explored two methods to covert a point cloud to this structure.

The first method to generate an HEM involved using the LVR2 toolkit to mesh the point cloud and then saving the data as an HEM. A description of the required steps follows.

First, navigate to the build directory using the following command:

```
cd ~/Libraries/lvr2/build
```

Next, use the `lvr2_reconstruct` tool to create a PLY mesh. The meshing parameters can be displayed using the following command:

```
bin/lvr2 reconstruct --h
```

An example of using the `lvr2_reconstruct` tool follows.

```
bin/lvr2_reconstruct --inputFile ~;path/to/pointcloud.
  ply --outputFile ~;path/to/output_mesh.ply -r -v
  0.2 -f 3 -o –useGPU
```

The second method for generating an HEM was a bit more involved, but it crashed less frequently than the aforementioned method. Specifically, it involved using different software (e.g., CloudCompare, n.d. or MeshLab, n.d.) to generate the PLY mesh, then using Open Flipper (Kobbelt, "Open Flipper," n.d.) for the conversion. Open Flipper uses Open Mesh (Kobbelt, "Open Mesh," n.d.) to perform the conversion. An outline of a use case example follows.

First, CloudCompare was used to convert the point cloud to a PLY mesh. The generated mesh file was imported into Open Flipper. Open Flipper performed the HEM conversion, and the resulting HEM file was saved as an OM file, which is OpenMesh's native format. Although this is technically an HEM mesh file, `mesh_navigation` does not support this file type. As a result, we had to convert it back into a PLY file format that could be understood by `mesh_navigation`. To accomplish this, we restarted Open Flipper (i.e., we closed it and opened it again), imported that newly saved OM, and then did a Save As to convert the mesh into a PLY file.

Regardless of the HEM meshing procedure used, the next step is to generate the H5 map files that are used for mesh navigation. The steps we took were as follows.

Navigate to the build directory with the following command:

```
cd ~;Libraries/lvr2/build
```

Next, use the `lvr2_hdf5_mesh_tool` tool to create an H5 map from the mesh file. The conversion parameters can be displayed using the command that follows:

```
bin/lvr2_hdf5_mesh_tool --h
```

An actual use case example is shown here:

```
bin/lvr2_hdf5_mesh_tool  -i ~/path/to/mesh.ply   -o ~/
   path/to/output_map.h5
```

The ability to generate HEMs allowed us to load these into the simulation for testing.

2.2.3 Testing Mesh Navigation Simulation

The quickest way to test mesh navigation in simulation is to use the `pluto_robot` (Pütz 2021) repository. The `pluto_robot` repository requires the following dependencies: `lvr2` (University of Osnabrück 2018), `move_base_flex` (Magazino 2018), `mesh_tools` (Nature Robots 2020), and `mesh_navigation` (Pütz 2019). The `pluto_robot` repository also contains a few data sets. The physics building data set, located at Westerberg Campus, Osnabrück University, was used to construct Figure 3. To launch the dataset, we used the command that follows.

```
roslaunch pluto_navigation physics_campus_westerberg.
  launch
```

The `physics_campus_westerberg.launch` file calls two nodes. The first node is the `move_base_flex` node; the code is shown here:

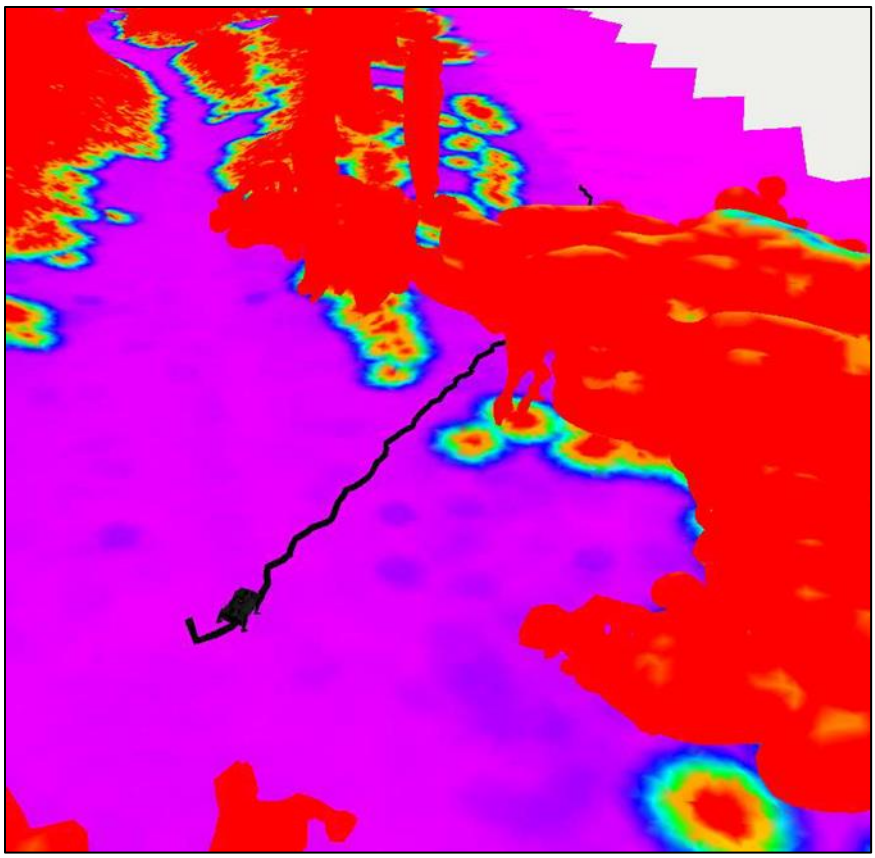
```
<arg name="rviz" default="false" />
<arg name="dataset" default="
 physics_campus_westerberg" />
<arg name="dir" default="maps/$(arg␣dataset)" />
<arg name="map_file" default="$(arg␣dir)/$(arg␣
 dataset).h5" />
<arg name="map_config" default="$(arg␣dir)/$(arg␣
 dataset)␣map.yaml" />
<arg name="nav_config" default="$(arg␣dir)/$(arg␣
 dataset)␣nav.yaml" />
<arg name="part" default="mesh" />
<arg name="planner_patience" default="10"/>

<node name="move_base_flex" pkg="mbf_mesh_nav" type="
 mbf_mesh_nav" output="screen">
  <rosparam command="load" file="$(find␣
 pluto_navigation)/$(arg␣nav_config) "/>
  <rosparam command="load"  ns="mesh_map"  file="$(find
 ␣pluto_navigation)/$(arg_map_config)"/>
  <param name="planner_patience" value="$(arg␣
 planner_patience)"/>
  <param name="mesh_map/mesh_file" value="$(find␣
 pluto_navigation)/$(arg␣map_file)" />
  <param name="mesh_map/mesh_part" value="$(arg␣part)
 "/>
</node>
```

The other node that is called is a state machine node, which follows. These two nodes together can be used to explore HEM meshes in simulation.

```
<node pkg="pluto_navigation" type="navigation_sm.py"
 name="navigation_smach" output="screen"/>
```

In addition to the `pluto_robot`, we also explored using the virtual CHAMP (2019) quadruped robots in conjunction with mesh navigation. Figure 14 shows the virtual quadruped robot navigating a 3D mesh.

Figure 14. Virtual quadruped robot navigating a 3D mesh.

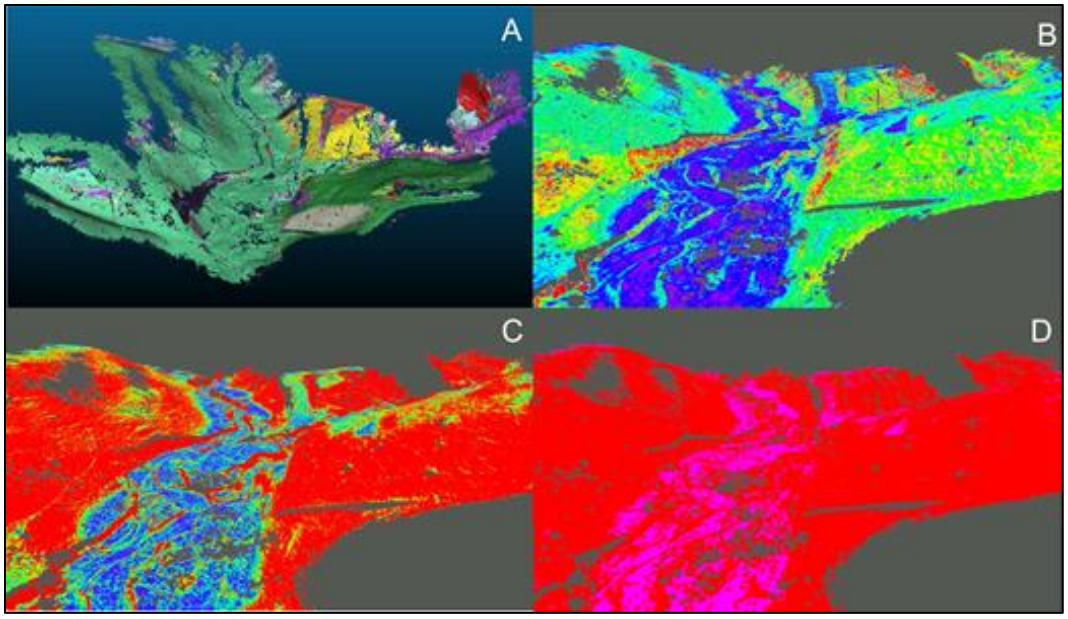


2.2.4 Mesh Navigation with Real-World Examples

With the ability to generate HEMs and ingest them into `mesh_navigation`, we explored generating 3D traversability meshes from a variety of sensors, including handheld, UGV, UAV, and even aerial lidar.

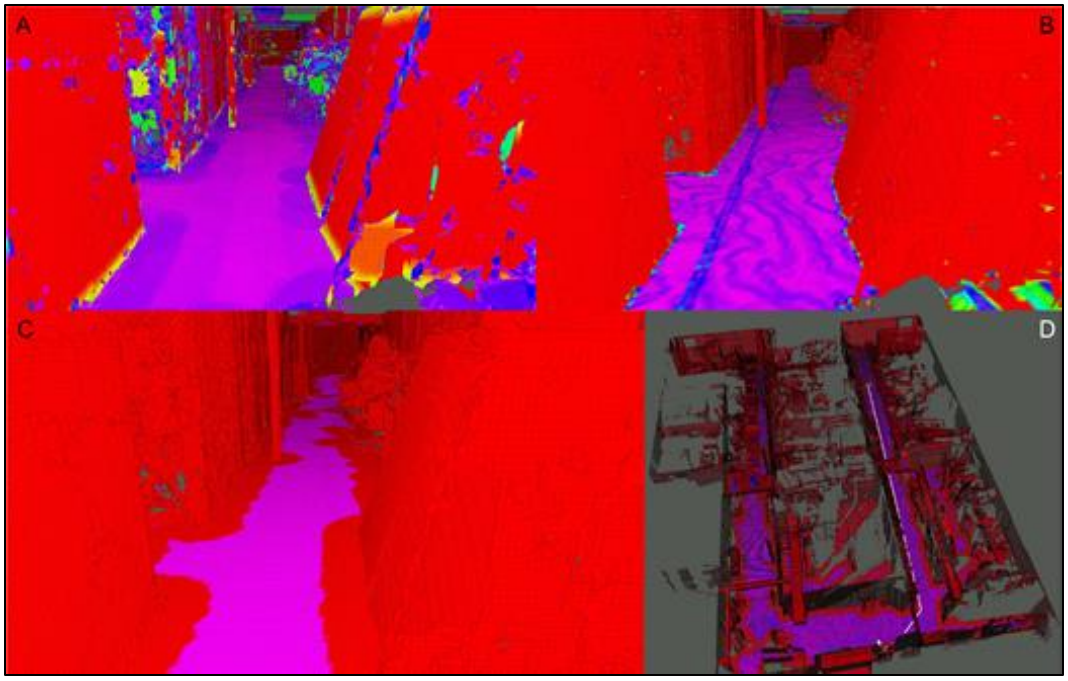
Figure 15 shows the generation of a 3D traversability mesh using a handheld sensor. In this example, a GeoSLAM ZEB Revo (GeoSLAM Ltd., Nottingham, United Kingdom) collected the initial point cloud. Using the steps mentioned in Section 2.2.2, the point cloud was converted to HEM, as shown in Figure 15*A*. The HeightDiffLayer and SteepnessLayer calculated for the HEM are shown in Figure 15*B* and *C*, respectively. These two layers were combined to generate the InflationLayer shown in *D*. In the InflationLayer, the pink area indicates terrain the UGV can safely traverse, while the red represents unnavigable terrain.

Figure 15. The 3D meshes generated from handheld lidar, showing the original mesh (*A*) and the height difference (*B*), steepness (*C*), and inflation (*D*) layers.



The previous example showcased an outside environment. Figure 16 demonstrates that mesh navigation can be used for path planning inside a building. The point cloud was collected with a Leica C 10 survey grade lidar (Leica Geosystems, Scarborough, Ontario, Canada). As in the previous example, the HeightDiffLayer and SteepnessLayer, shown in Figure 16*A* and *B*, respectively, were used to generate the InflationLayer shown in *C*. The full 3D traversability mesh of the building interior is shown in *D*, and the white line denotes the path of the robot as it navigated the building interior. Again, the pink areas indicate terrain that the UGV can safely traverse. Walls and obstacles are shown in red.

Figure 16. The 3D meshes generated from survey-grade lidar, showing the height difference (*A*), steepness (*B*), and inflation (*C*) layers. The full mesh is shown in *D*.



The next two images, Figure 17 and Figure 18, show point clouds collected with a UGV. Although the original point cloud, which was of a desert environment and is shown on the upper left of Figure 17, is relatively sparse in features, it is quite large. The resulting traversability mesh is shown on the bottom left of the figure, and the robot path planning on this large open space is shown on the right. Most of the terrain is navigable (i.e., pink), and the occasional obstacle is shown in red.

Although Figure 17 was meant to demonstrate that mesh navigation can be used on large point clouds, Figure 18 demonstrates a much more feature-rich outdoor environment. The bridge the UGV used to cross the ravine is shown in the upper right. In addition, the sides of the ravine around the bridge are marked red and highlight mesh navigation's ability to detect negative obstacles.

Figure 17. The 3D meshes generated from an unmanned ground vehicle (UGV), showing original mesh (*top left*), inflation layer (*bottom left*), and path planning along the mesh (*right*).

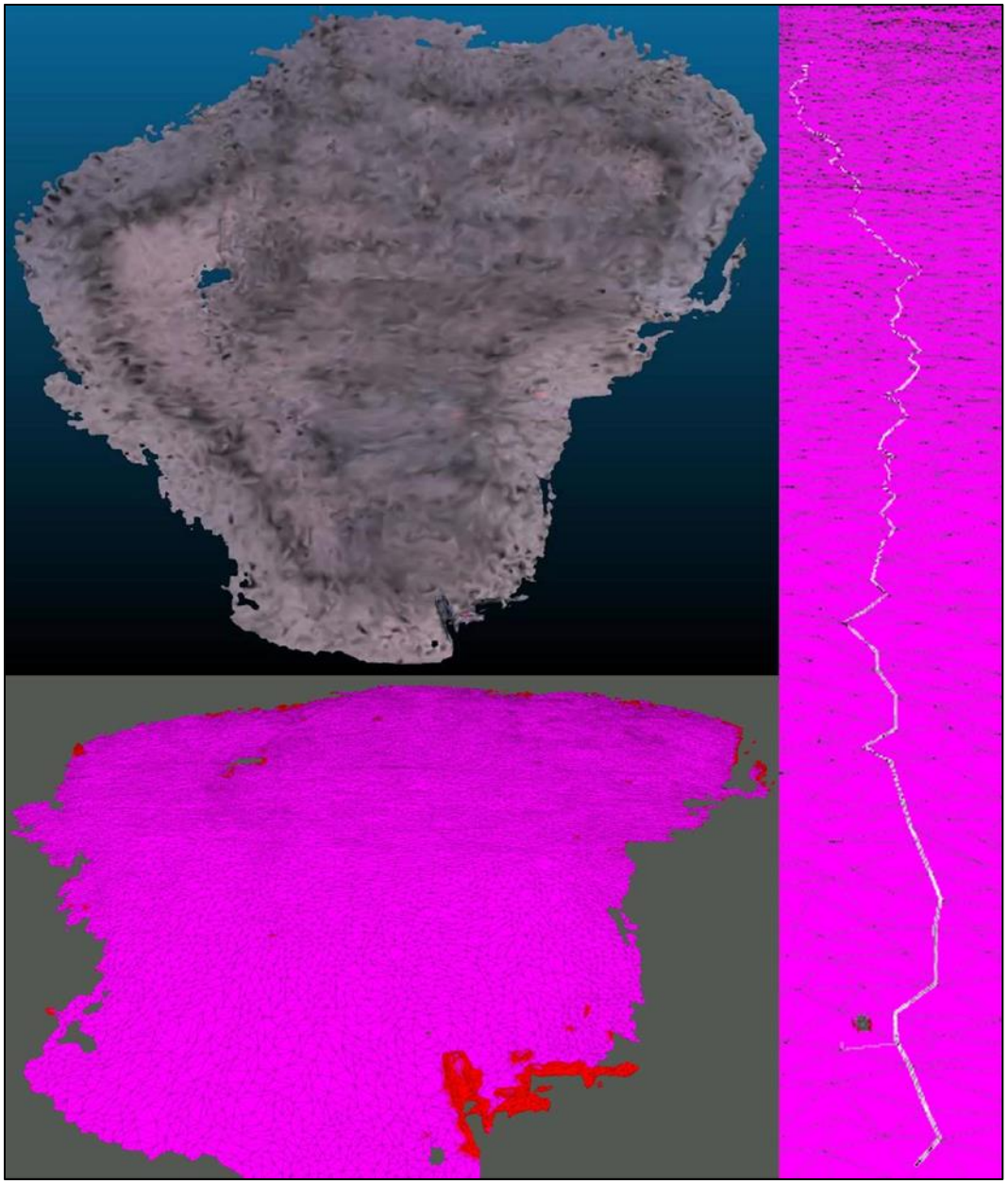


Figure 18. Feature-rich 3D mesh generated from a UGV.

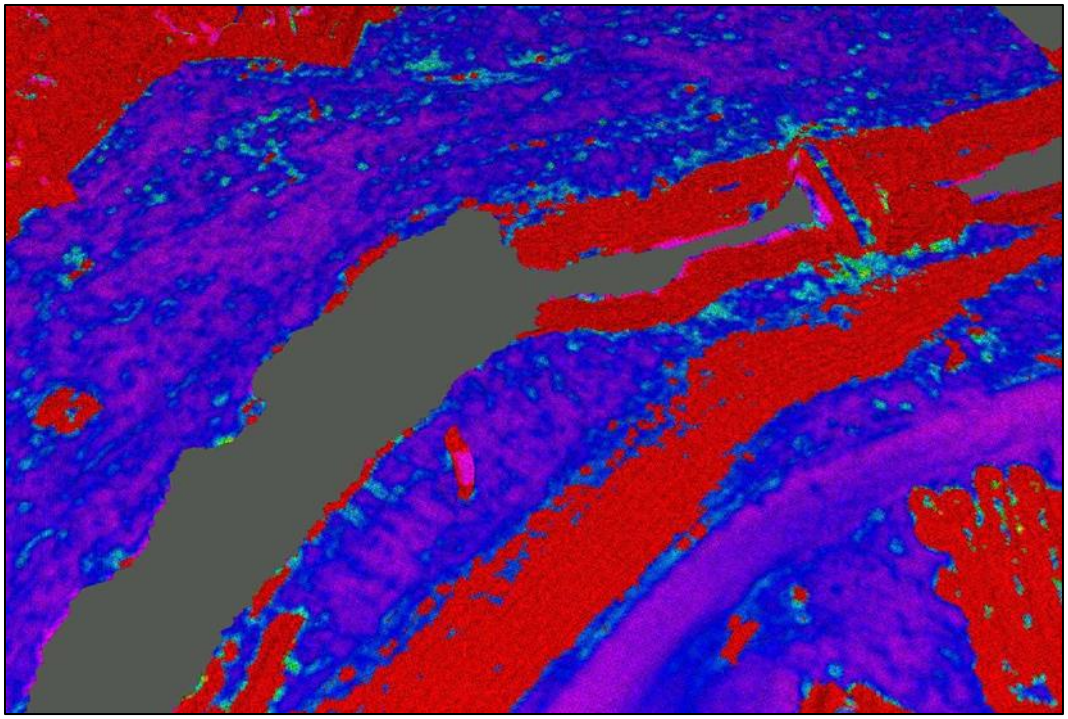


Figure 19*A* shows the only point cloud in this report that was not generated using lidar. In this example, the point cloud was generated from a UAV using photogrammetry. The HeightDiffLayer and RougnessLayer shown in Figure 19*B* and *C*, respectively, were used to generate the InflationLayer shown in *D*. Previous examples used steepness instead of roughness. The resulting InflationLayer, shown in *D*, identifies that it is safer to operate a UGV on the right, rather than left, side of the mesh. The HeightDiff and Roughness layers clearly identify features that are not easily discernible in the red-green-blue (RGB) point cloud.

Finally, to round out the sample set, Figure 20 shows a traversability mesh generated from an aerial lidar. The top image shows the HEM generated using the steps in Section 2.2.2, and the bottom image shows the resulting 3D traversability mesh. Again, the pink regions represent navigable terrain, and obstacles are shown in red.

Figure 19. The 3D meshes generated from a UAV, showing the original point cloud (*A*) and the height difference (*B*), roughness (*C*), and inflation (*D*) layers.

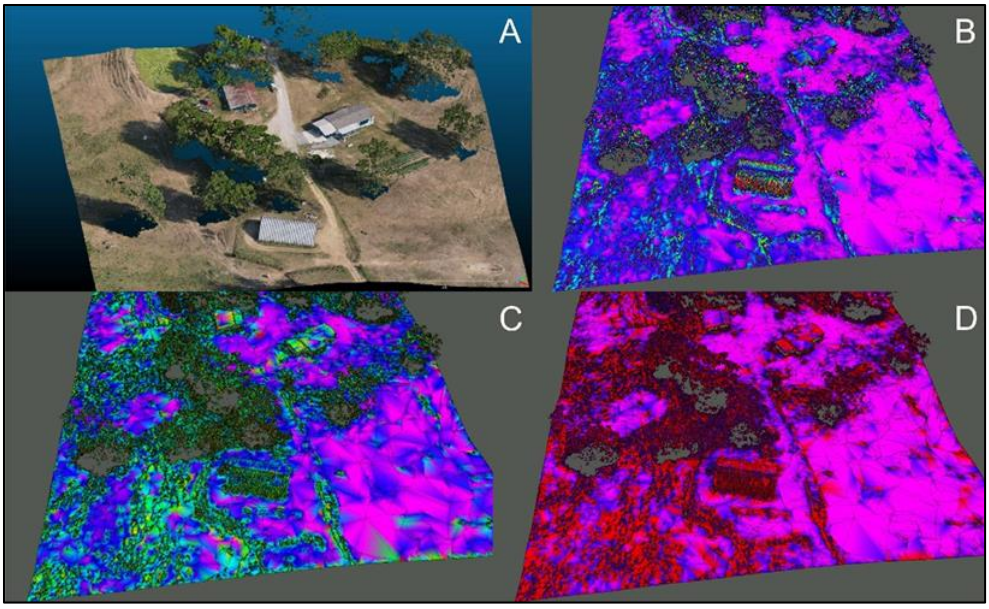
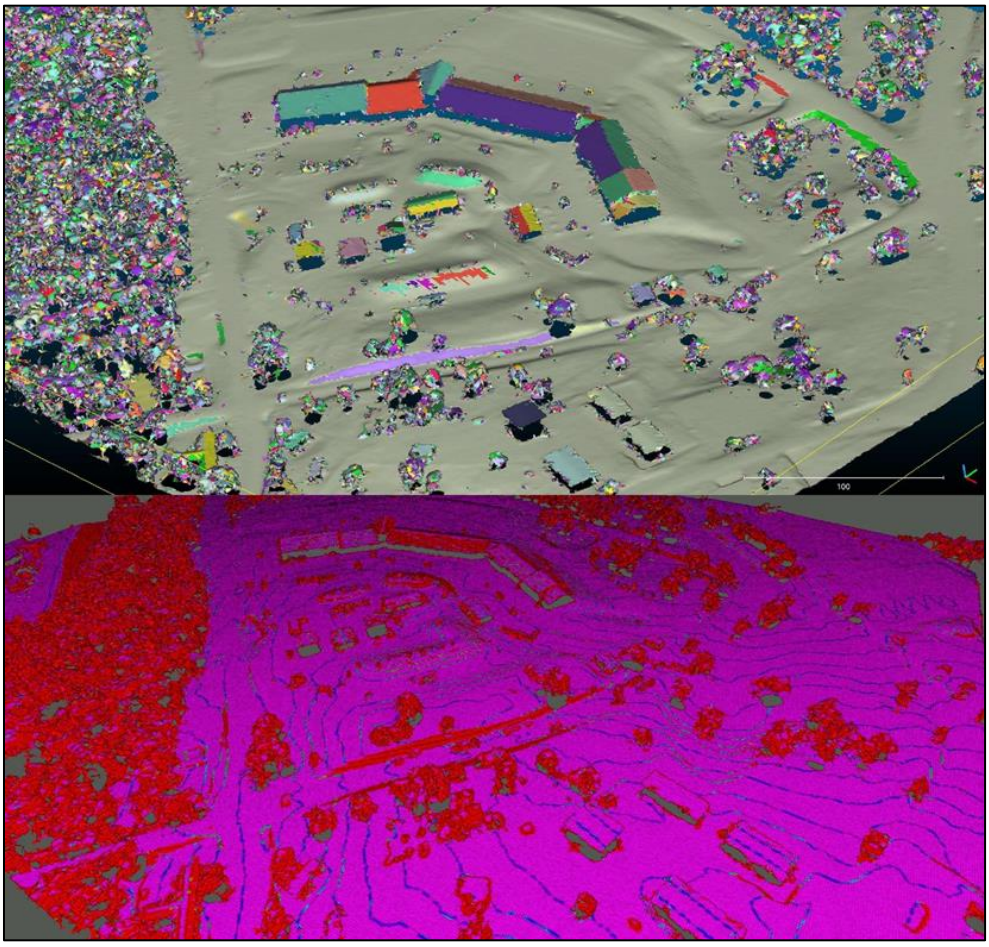


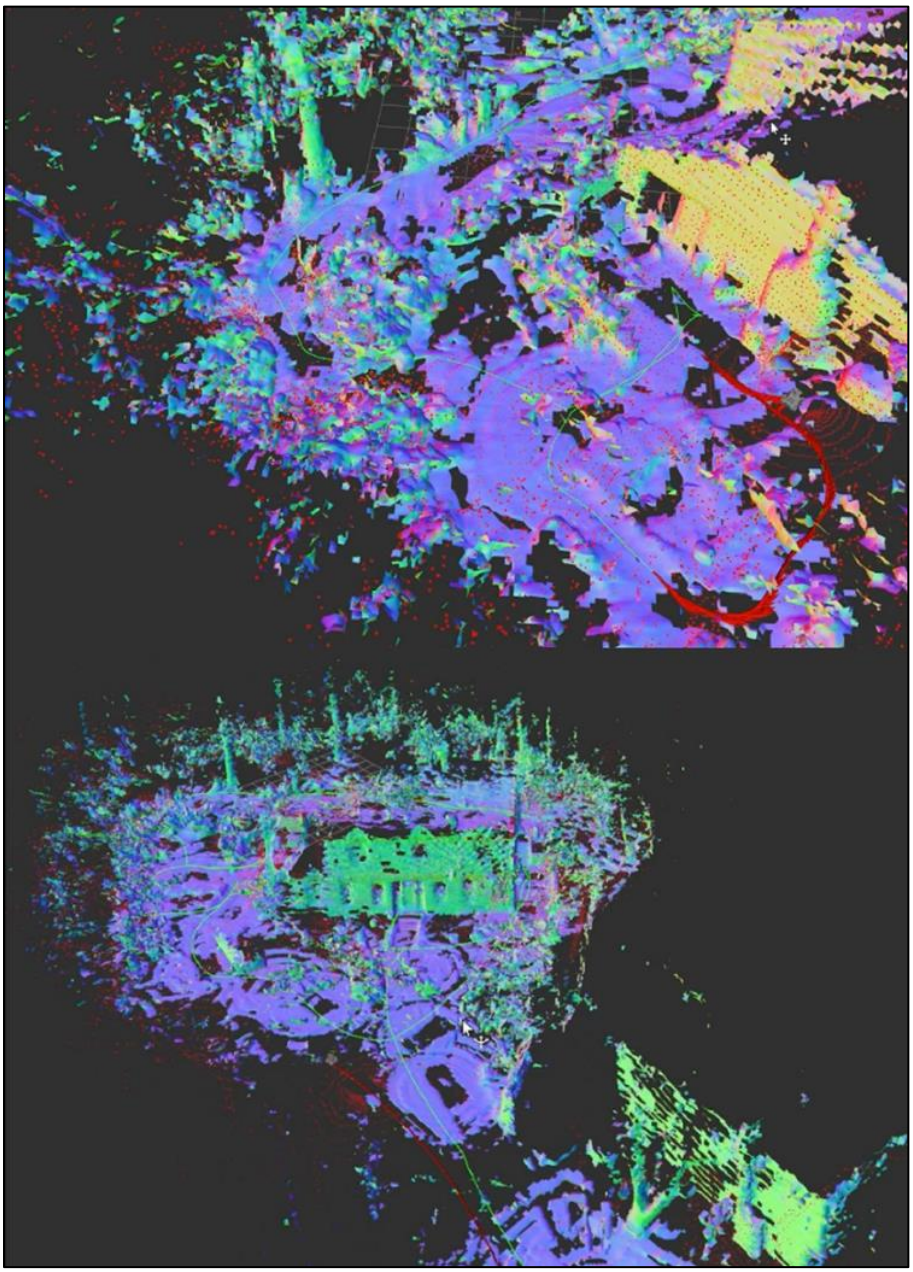
Figure 20. The 3D meshes generated from an aerial lidar, showing the original mesh (*top*) and the inflation layer (*bottom*).



2.2.5 Future Work

In the mesh navigation approach, generating the HEM is a significant bottleneck. This step can take hours to perform if the mesh is quite large or very high resolution. Thus, future work is aimed at generating meshes in near real-time. One approach that we have been investigating, and which will be the focus of a future tech report, is the use of Voxblox (ETHZ ASL 2016). A preview of using Voxblox to generate a near-real-time mesh is shown in Figure 21.

Figure 21. Example of a near-real-time mesh generated by Voxblox.



3 Summary

This report explored using both grid maps and mesh navigation for 2.5D and 3D navigation, respectively. For complex environments, 2.5D navigation is superior to 2D navigation. Specifically, 2.5D navigation allows the global planner to leverage 2.5D information, rather than reacting to it in the local planner. Also, we prefer using traversability, rather than elevation, to generate the 2.5D occupancy grids because doing so allows the UGV to react to the terrain regardless of the robot's current *z* position. Finally, we showed the advantages of using mesh navigation if prior information is available. Mesh navigation excels in identifying negative obstacles and allows for true 3D path planning. We also showed the results of generating traversable meshes from multiple sources, including UAVs and even aerial lidar. However, the time required to generate the traversable meshes presently keeps us from using it in near real-time.

References

ANYbotics. 2021. Elevation Mapping [Source code]. *GitHub*. https://github.com/ANYbotics/elevation_mapping/tree/master/elevation_mapping/config/sensor_processors.

CCNY Robotics Lab. n.d. Scan Tools [Source code]. *GitHub*. https://github.com/CCNYRoboticsLab/scan_tools.

CHAMP. 2019. Chvmp [Source code]. *GitHub*. https://github.com/chvmp/champ.

Christie, B. A., O. Ennasr, and G. P. Glaspell. 2021. *Autonomous Navigation and Mapping in a Simulated Environment*. ERDC/GSL TR-21-5. Alexandria, VA: US Army Engineer Research and Development Center, Geospatial Research Laboratory. https://doi.org/10.21079/11681/42006.

Clearpath Robotics. 2020a. CPR Office Gazebo [Source code]. *GitHub*. https://github.com/clearpathrobotics/cpr_gazebo/tree/noetic-devel/cpr_office_gazebo.

Clearpath Robotics. 2020b. CPR Office Inspection Gazebo [Source code]. *GitHub*. https://github.com/clearpathrobotics/cpr_gazebo/tree/noetic-devel/cpr_inspection_gazebo.

CloudCompare. n.d. "CloudCompare: 3D Point Cloud and Mesh Processing Software." Accessed May 1, 2023. https://www.cloudcompare.org/.

ETHZ ASL. 2016. Voxblox [Source code]. *GitHub*. https://github.com/ethz-asl/voxblox.

Fankhauser, P. 2019. Grid Map [Source code]. *GitHub*. https://github.com/anybotics/grid_map.

Fankhauser, P., M. Bloesch, C. Gehring, M. Hutter, and R. Siegwart. 2014. "Robot-Centric Elevation Mapping with Uncertainty Estimates." In *Proceedings*, *17th International Conference on Climbing and Walking Robots (CLAWAR) and the Support Technologies for Mobile Machines*, 21–23 July, Poznań, Poland.

Fankhauser, P., M. Bloesch, and M. Hutter. 2018. "Probabilistic Terrain Mapping for Mobile Robots with Uncertain Localization." *IEEE Robotics and Automation Letters (RA-L)* 3 (4): 3019–3026. https://doi.org/10.1109/LRA.2018.2849506.

Fankhauser, P., and M. Hutter. 2016. "A Universal Grid Map Library: Implementation and Use Case for Rough Terrain Navigation." In *Robot Operating System (ROS)*: *The Complete Reference (Volume 1),* edited by A. Koubaa, 99–120. New York: Springer. https://doi.org/10.1007/978-3-319-26054-9_5.

Fankhauser, P., and M. Wulf. 2019. Robot-Centric Elevation Mapping [Source code]. *GitHub*. https://github.com/ANYbotics/elevation_mapping.

Flynn, E. 2021. FLOAM Release [Source code]. *GitHub*. https://github.com/flynneva/floam-release.

Gehring, C., C. D. Bellicoso, M. Bloesch, R. Diethelm, P. Fankhauser, P. Furgale, M. Neunert, and H. Sommer. 2019. Kindr—Kinematics and Dynamics for Robotics [Source code]. *GitHub*. https://github.com/anybotics/kindr.

Gehring, C., P. Fankhauser, and R. Diethelm. 2019. Kindr ROS [Source code]. *GitHub*. https://github.com/anybotics/kindr_ros.

Glaspell, G. P., S. R. Lessard, B. A. Christie, K. Jannak-Huang, N. C. Wilde, W. He, O. Ennasr, et al. 2020. *Optimized Low Size, Weight, Power and Cost (SWaP-C) Payload for Mapping Interiors and Subterranean on an Unmanned Ground Vehicle*. ERDC/GRL TR-20-6. Alexandria, VA: US Army Engineer Research and Development Center, Geospatial Research Laboratory. https://doi.org/10.21079/11681/35878.

"An Introduction to Half-Edge Data Structure." 2020. Berkeley, CA: Department of Electrical Engineering and Computer Sciences, University of California. https://cs184.eecs.berkeley.edu/sp20/article/17/an-introduction-to-half-edge-dat.

Jacob, B., and G. Guennebaud. n.d. "Eigen." Last modified April 18, 2023. http://eigen.tuxfamily.org.

Kobbelt, L. n.d. "Open Flipper." Accessed May 1, 2023. https://www.graphics.rwth-aachen.de/software/openflipper/.

Kobbelt, L. n.d. "Open Mesh." Accessed May 1, 2023. https://www.graphics.rwth-aachen.de/software/openmesh/.

LVR2. n.d. "Las Vegas Surface Reconstruction Toolkit 2.0." Accessed May 1, 2023. https://www.las-vegas.uni-osnabrueck.de/.

Magazino. 2018. Move Base Flex [Source code]. *GitHub*. https://github.com/magazino/move_base_flex.

MeshLab. n.d. "MeshLab." Accessed May 1, 2023. https://www.meshlab.net/.

Nature Robots. 2020. Mesh Tools [Source code]. *GitHub*. https://github.com/uos/mesh_tools.

Office of the Deputy Chief of Staff. 2020. *Army Multi-Domain Intelligence: FY21-22 S and T Focus Areas*. AD1114490. Washington, DC: Department of the Army. https://apps.dtic.mil/sti/pdfs/AD1114489.pdf.

Pütz, S. 2019. Mesh Navigation [Source code]. *GitHub*. https://github.com/uos/mesh_navigation.

Pütz, S. 2021. Pluto Robot [Source code]. *GitHub*. https://github.com/uos/pluto_robot.

Pütz, S., J. Santos Simon, and J. Hertzberg. 2018. "Move Base Flex: A Highly Flexible Navigation Framework for Mobile Robots." In *Proceedings, 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 1–5 October, Madrid, Spain, 3416–3421. https://doi.org/10.1109/IROS.2018.8593829.

Pütz, S., T. Wiemann, M. Kleine Piening, and J. Hertzberg. 2021. "Continuous Shortest Path Vector Field Navigation on 3D Triangular Meshes for Mobile Robots." In *Proceedings, 2021 IEEE International Conference on Robotics and Automation (ICRA)*, 30 May to 5 June, Xi'an, China, 2256–2263. https://doi.org/10.1109/ICRA48506.2021.9560981.

Rusu, R. B., and S. Cousins. 2011. "3D is Here: Point Cloud Library." *Point Cloud Library*. http://pointclouds.org/.

University of Osnabrück. 2018. Lvr2. *GitHub*. https://github.com/uos/lvr2.

University of Osnabrück. 2020. Pluto Robot, Physics Campus Westerberg [Source code]. *GitHub*. https://github.com/uos/pluto_robot/tree/master/pluto_navigation/maps/physics_campus_westerberg.

Wiemann, T., H. Annuth, K. Lingemann, and J. Hertzberg. 2015. "An Extended Evaluation of Open Source Surface Reconstruction Software for Robotic Applications." *Journal of Intelligent & Robotic Systems* 77 (1): 149–170. http://dx.doi.org/10.1007/s10846-014-0155-1.

Wiemann, T., S. Pütz, A. Mock, L. Kiesow, L. Kalbertodt, T. Igelbrink, J. M. von Behren, D. Feldschnieders, and A. Löhr. 2022. HalfEdgeMesh [Source code]. *ROS*. https:/docs.ros.org/en/latest-available/api/lvr2/html/classlvr2_1_1HalfEdgeMesh.html.

Appendix A: Configuration Files for 2.5D Navigation

A.1 tb3.yaml

```
# Robot.
map_frame_id:                                                   map
robot_base_frame_id:
  base_footprint
robot_pose_with_covariance_topic:                               /
  base_footprint_pose
robot_pose_cache_size:                                          200
sensor_frame_id:
  base_footprint
point_cloud_topic:                                              /
  velodyne_points
track_point_frame_id:
  base_footprint
track_point_x:                                                  0.0
track_point_y:                                                  0.0
track_point_z:                                                  0.0
min_update_rate:                                                1.0
time_tolerance:                                                 1.0
time_offset_for_point_cloud:                                    0.0
sensor_processor/ignore_points_above:                           0.9
robot_motion_map_update/covariance_scale:                       0.01

# Map.
length_in_x:                                                    50.0
length_in_y:                                                    50.0
position_x:                                                     0.0
position_y:                                                     0.0
resolution:                                                     0.1
min_variance:                                                   0.0001
max_variance:                                                   0.05
mahalanobis_distance_threshold:                                 2.5
multi_height_noise:                                             0.001
surface_normal_positive_axis:                                   z
fused_map_publishing_rate:                                      1.0
enable_visibility_cleanup:                                      false
visibility_cleanup_rate:                                        1.0
scanning_duration:                                              1.0

# Init submap
initialize_elevation_map:                                       false
initialization_method:                                          0
length_in_x_init_submap:                                        1.0
length_in_y_init_submap:                                        1.0
margin_init_submap:                                             0.3
init_submap_height_offset:                                      0.01
target_frame_init_submap:
  base_footprint
```

A.2 velodyne_HDL-32E.yaml

```
# Velodyne_HDL-32E
sensor_processor:
   type: laser
   min_radius: 0.018
   beam_angle: 0.0006
   beam_constant: 0.0015
```

A.3 postprocessor_pipeline.yaml

```
postprocessor_pipeline: # set by
   postprocessor_pipeline_name
  # Fill holes in the map with inpainting.
  - name: inpaint
    type: gridMapCv/InpaintFilter
    params:
      input_layer: elevation
      output_layer: elevation_inpainted
      radius: 0.05

  # Compute Surface normals
  - name: surface normals
    type: gridMapFilters/NormalVectorsFilter
    params:
      input_layer: elevation_inpainted
      output_layers_prefix: normal_vectors_
      radius: 0.1
      normal_vector_positive_axis: z
```

A.4 costmap.yaml

```
grid_map_topic: /elevation_mapping/elevation_map
grid_map_visualizations:
  - name: elevation grid
    type: occupancy_grid
    params:
      layer: elevation
      data_min: -0.5
      data_max: 0.5
```

A.5 filters_demo_filter_chain.yaml

```
grid_map_filters:
  - name: first
    type: gridMapFilters/MockFilter
    params:
      processing_time: 100
      print_name: true
  - name: buffer_normalizer
    type: gridMapFilters/BufferNormalizerFilter

  # Delete color layer.
  - name:delete original layers
```

```
    type: gridMapFilters/DeletionFilter
    params:
      layers: [color] # List of layers.

# Fill holes in the map with inpainting.
- name: inpaint
    type: gridMapCv/InpaintFilter
    params:
      input_layer: elevation
      output_layer: elevation_inpainted
      radius: 0.05

# Boxblur as an alternative to the inpaint and radial
  blurring filter above.
- name: boxblur
    type: gridMapFilters/
 SlidingWindowMathExpressionFilter
    params:
      input_layer: elevation
      output_layer: elevation_smooth
      expression: meanOfFinites(elevation)
      compute_empty_cells: true
      edge_handling: crop # options: inside, crop,
empty, mean
      window_size: 5 # optional

# Compute surface normals.
- name: surface normals
    type: gridMapFilters/NormalVectorsFilter
    params:
      input_layer: elevation_inpainted
      output_layers_prefix: normal_vectors_
      radius: 0.2
      normal_vector_positive_axis: z

# Add a color layer for visualization based on the surface nor-
 mal.
- name: normal_color_map
    type: gridMapFilters/NormalColorMapFilter
    params:
      input_layers_prefix: normal_vectors_
      output_layer: normal_color

# Compute roughness as absolute difference from map to smooth-
 ened map.
-  name: roughness
    type: gridMapFilters/MathExpressionFilter
    params:
      output_layer: roughness
      expression: abs(elevation_inpainted - elevation_smooth)

##Edge detection on elevation layer with convolution filter as
 alternative to filter above.
-  name: edge_detection
    type: gridMapFilters/
 SlidingWindowMathExpressionFilter
    params:
```

```
    input_layer: elevation_inpainted
    output_layer: edges
    # expression: 'sumOfFinites
 ([0,1,0;1,-4,1;0,1,0] .*elevation_inpainted)' # Edge detection.
    expression: 'sumOfFinites
 ([0,-1,0;-1,5,-1,0, -1,0].*elevation_inpainted)' #
 Sharpen.
    compute_empty_cells: false
    edge_handling: mean # options: inside, crop, empty, mean
    window_size: 3 # Make sure to make this
compatible with the kernel matrix.

# Compute traversability as normalized weighted sum of slope and
 roughness.
- name: traversability
  type: gridMapFilters/MathExpressionFilter
  params:
    output_layer: traversability
    # expression: 0.5 * (1.0 - (slope/ 0.6)) + 0.5 *
  (1.0 - (roughness / 0.1))
    expression: ((roughness + edges) / 2)

# Set lower threshold on traversability.
- name: traversability_lower_threshold
  type: gridMapFilters/ThresholdFilter
  params:
    condition_layer: traversability
    output_layer: traversability
    lower_threshold: 0.0
    set_to: 0.0

# Set upper threshold on traversability.
- name: traversability_upper_threshold
  type: gridMapFilters/ThresholdFilter
  params:
    condition_layer: traversability
    output_layer: traversability
    upper_threshold: 1.0
    set_to: 1.0
```

Appendix B: Configuration Files for 3D Navigation

B.1 mesh map.h

```
/*
 *   Copyright 2020, Sebastian Pütz

 *   Redistribution and use in source and binary forms, with or
      without modification, are permitted provided that the fol-
      lowing conditions are met:

      1. Redistributions of source code must retain the above
         copyright notice, this list of conditions and the fol-
         lowing disclaimer.

      2. Redistributions in binary form must reproduce the above
         copyright notice, this list of conditions and the fol-
         lowing disclaimer in the documentation and/or other ma-
         terials provided with the distribution.

      3. Neither the name of the copyright holder nor the names
         of its contributors may be used to endorse or promote
         products derived from this software without specific
         prior written permission.

 *   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CON-
     TRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
     INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
     MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
     DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CON-
     TRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,INCIDENTAL,
     SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,BUT
     NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SER-
     VICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUP-
     TION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
     WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
     NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
     OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
     SUCH DAMAGE.

 *   authors: Sebastian Putz <spuetz@uni-osnabrueck.de>
```

```
#ifndef MESH_MAP_MESH_MAP_H
#define MESH_MAP_MESH_MAP_H

#include <atomic>
#include <dynamic_reconfigure/server.h>
#include <geometry_msgs/Point.h>
#include <lvr2/geometry/BaseVector.hpp>
#include <lvr2/io/HDF5IO.hpp>
#include <mesh_map/MeshMapConfig.h>
#include <mesh_map/abstract_layer.h>
#include <mesh_msgs/MeshVertexCosts.h>
#include <mesh_msgs/MeshVertexColors.h>
#include <mutex>
#include <pluginlib/class_loader.h>
#include <std_msgs/ColorRGBA.h>
#include <tf2_ros/buffer.h>
#include <tuple>
#include "nanoflann.hpp"
#include "nanoflann_mesh_adaptor.h"

namespace mesh_map
{
class MeshMap
{
public:
  typedef boost::shared_ptr<MeshMap> Ptr;

  MeshMap(tf2_ros::Buffer& tf);

  /**
   * @brief Reads in the mesh geometry, normals and cost values
   and publishes all as mesh_msgs
   * @return true if the mesh and its attributes have been
   loaded successfully.
   */
  bool readMap();

  /**
   * @brief Loads all configures layer plugins
   * @return true if the layer plugins have been loaded suc-
     cessfully.
   * /
  bool loadLayerPlugins();

  /**
   * @brief Initialized all loaded layer plugins
   * @return true if the loaded layer plugins have been ini-
     tialized successfully.
   * /
  bool initLayerPlugins();

  /**
   * @brief Return and optional vertex handle to the closest
     vertex for the given position
   * @param pos the search position
```

```
     * @return
     * /
    lvr2::OptionalVertexHandle getNearestVertexHandle(const
     mesh_map: :Vector& pos);

    /**
     * @brief return true if the given position lies inside the
       triangle with respect to the given maximum distance.
     * @param pos The query position
     * @param face The triangle to check for
     * @param dist The maximum distance to the triangle
     * @return true if the query position is inside the triangle
       within the maximum distance
     * /
    bool inTriangle(const Vector& pos, const lvr2:: FaceHandle&
    face, const float& dist);

    /**
     * @brief Searches for a triangle which contains the given po-
       sition with respect to the maximum distance
     * @param position The query position
     * @param max_dist The maximum distance to the triangle
     * @return Optional face handle, the optional is valid if a
       corresponding triangle has been found.
     * /
    lvr2::OptionalFaceHandle getContainingFace(Vector& position,
     const float& max_dist);

    /**
     * @brief Searches for a triangle which contains the given
       position with respect to the maximum distance
     * @param position The query position
     * @param max_dist The maximum distance to the triangle
     * @return optional tuple of the corresponding triangle, the
       triangle's vertices, and barycentric coordi-
       nates, if a corresponding triangle has been found.
    */
    boost::optional<std::tuple<lvr2: :FaceHandle, std:: ar-
    ray<mesh_map::Vector, 3>,
        std::array<float, 3>>> searchContainingFace(Vector& posi-
    tion, const float& max_dist);

    /**
     * @brief reconfigure callback func-
    tion which is called if a dynamic reconfiguration were trig-
    gered.
    */
    void reconfigureCallback(mesh_map::MeshMapConfig& config,
    uint32_t level);

    /**
     * @brief A method which com-
    bines all layer costs with the respective weightings
    */
```

```
  void combineVertexCosts();
  /**
   * @brief Computes contours
   * @param contours the vector to bo filled with contours
   * @param min_contour_size The minimum contour size, i.e. the
     number of vertices per contour.
   */
  void findContours(std::vector<std::vector<lvr2:: VertexHan-
     dle>>& contours, int min_contour_size);

  /**
   * @brief Publishes the given vertex map as mesh msgs/Vertex-
     Costs, e.g. to visualize these.
   * @param costs The cost map to publish
   * @param name The name of the cost map
   */
  void publishVertexCosts(const lvr2::VertexMap<float>& costs,
     const std::string& name);

  /**
   * @briefP Publishes the vertex colors if these exist.
   */
  void publishVertexColors();

  /**
   * @brief Publishes all layer costs as mesh msgs/VertexCosts
   */
  void publishCostLayers();

  /**
   * @brief Computes the projected barycentric coordinates,
     it implements Heidrich's method
     * See https://www.researchgate.net/publication/220494112
       _Computing_the_Barycentric_Coordinates_of_a_Pro-
       jected_Point

     * @param p the query position
     * @param triangle the corresponding triangle
     * @param barycentric_coords The array will be filled with
       the barycentric coordinates
     * @param dist The distance if the query position to the tri-
       angle
     * @return true if the query position lies inside the trian-
       gle
     * /
    bool projectedBarycentricCoords(const Vector& p,const
       lvr2::FaceHandle& triangle,
                                    std::array<float,
         3>& barycentric_coords, float& dist);

    /**
     * Computes barycentric coordinates of the given
```

```
    query position and the corresponding triangle
    * @param p The query position
    * @param triangle The corresponding triangle
    * @param u The barycentric coordinate if the first vertex
    * @param v The barycentric coordinate if the second vertex
    * @param w The barycentric coordinate if the third vertex
    * @return true if the query position lies inside the trian-
      gle
    */
  bool barycentricCoords(const Vector& p, const lvr2:: FaceHan-
    dle& triangle, float& u, float& v, float& w);

    /**
    * @brief Callback function which is called from inside a
    layer plugin if cost values change
    * @param layer_name the name of the layer.
    * /
   void layerChanged(const std::string& layer_name);

    /**
    * @brief Compute all contours and returns the corresponding
      vertices to use these as lethal vertices.
    * @param min_contour_size The minimum contour size, i.e. the
      number of vertices per contour.
    * @param min_contour_size
    * @param lethals the vector which is filled with contour
      vertices
    */
   void findLethalByContours(const int& min_contour_size ,
    std::set<lvr2::VertexHandle>& lethals);


    /**
    * @brief Returns the global frame / coordinate system id
    string
    */
   const std::string getGlobalFrameID();

    /**
    * @brief Checks if the given vertex handle corresponds to a
      lethal / not passable vertex
    */
   inline bool isLethal(const lvr2::VertexHandle& vH);

    /**
    * @brief Converts a vector to a ROS geometry_msgs/Point mes-
    sage
    */
   inline const geometry_msgs::Point toPoint(constVector& vec);

    /**
    * Computes the direction vector for the given tri-
      angle's vertices and barycentric coordinates while us-
      ing the given vector map
___ * @param vector_map The vector map to use
___ * @param vertices The triangles vertices
```

```
    *  @param barycentric_coords The barycentric coordi-
       nates of the query position.
    *  @return An optional vector of the computed direction.
       It is valid if a vector has been computed successfully.
    */
  boost::optional<Vector> directionAtPosition(const lvr2::Ver-
       texMap<lvr2::BaseVector<float>>& vector_map
      '
    ..................................................const std::ar-
       ray<lvr2::VertexHandle, 3>& vertices,
    ..................................................const std::ar-
       ray<float, 3>& barycentric_coords);

  /**
    *  Computes the cost value for the given triangle's vertices
       and barycentric coordinates while using the given cost map
     *  @param costs The cost map to use
     *  @param vertices The triangles vertices
     *  @param barycentric_coords The barycentric coordinates of
        the query position.
     *  @return A cost value for the given barycentric coordi-
        nates.
     */
    float costAtPosition(const lvr2::DenseVertexMap<float>& costs,
     const std::array<lvr2: :VertexHandle, 3>& vertices,
                                    const std::array<float, 3>&
    barycentric_coords);

    /**
     *  Computes the cost value for the given triangle's verti-
     ces and barycentric coordinates while using the com-
     bined cost map
    *  @param vertices The triangles vertices
    *  @param barycentric_coords The barycentric coordi-
      nates of the query position.
    *  @return A cost value for the given barycentric coordi-
      nates.
    */
  float costAtPosition(const std::array<lvr2:: VertexHandle,
  3>& vertices,
    ..........................const std::array<float, 3>& barycen-
       tric_coords);

  /**
    *  Computes the barycentric coordinates of the ray intersec-
  tion point for the given ray
    *  @param orig The ray origin
    *  @param dir The ray direction
    *  @param v0 The triangle's first vertex
     *  @param v1 The triangle's second vertex
     *  @param v2 The triangle's third vertex
     *  @param t The signed distance to the triangle
```

```
     * @param u The first barycentric coordinate
     * @param v The second barycentric coordinate
     * @param p The third barycentric coordinate
     * @return True if the intersection point lies inside the
       triangle
     */
    bool rayTriangleintersect(const Vector& orig, const Vector&
     dir, const Vector& vO, const Vector& vl, const Vector& v2,
                               float& t, float& u, float&
     v, Vector& p);

    /**
     * @brief Resets all layers.
     * @todo implement
     * @return true if successfully reset
     */
    bool resetLayers();

    /**
     * @brief Returns the stored vector map
     */
    const lvr2::DenseVertexMap<mesh_map::Vector>& getVectorMap ()
    {
      return vector_map;
    };

    /**
     * @brief Returns the stored mesh
     */
    const lvr2::HalfEdgeMesh<Vector>& mesh()
    {
      return *mesh_ptr;
    }

    /**
     * @brief Returns the stored combined costs
     */
    const lvr2::DenseVertexMap<float>& vertexCosts()
    {
      return vertex_costs;
    }

    /**
     * @brief Returns the map frame / coordinate system id
     */
    const std::string& mapFrame()
    {
      return global_frame;
    }

    /**
     * @brief Returns the mesh's triangle normals
    ___*/
    __const_lvr2::DenseFaceMap<Normal>&_faceNormals()
    __{
    ____return_face_normals;
```

```
    }

    /**
     * @brief Returns the mesh's vertex normals
     */
    const lvr2::DenseVertexMap<Normal>& vertexNormals()
    {
      return vertex_normals;
    }

    /**
     * @brief Returns the mesh's edge weights
     */
    const lvr2::DenseEdgeMap<float>& edgeWeights()
    {
      return edge_weights;
    }

    /**
     * @brief Returns the mesh's vertex distances
     */
    Const lvr2::DenseEdgeMap<float>& edgeDistances ()
    {
      return edge_distances;
    }

    /**
     * Searches in the surrounding triangles for the triangle in
       which the given
     * position lies.
     * @param pose_vec      Vector of the position which searched
       for
     * Param face           Face handle from which search begins
     * @param max_radius    The radius in which the controller
       searches for a consecutive face
     * @param max_dist      The maximum distance to the given
       vertices
     * @return              Optional of (face handle, vertices,
       and barycentric coord) tuple
     */

    boost::optional<std::tuple<lvr2::FaceHandle, std::array<Vec-
     tor, 3>, std::array<float, 3>>>
    searchNeighbourFaces(const Vector& pos, const lvr2::FaceHan-
     dle& face,
                         const float& max_radius, const
float& max_dist);

    /**
     * Finds the next position given a position vector and its
       corresponding face
     * handle by following the direction For: look ahead when us-
       ing mesh gradient
     * @param vec    direction vector from which the next step
```

```
     vector is calculated
    * @param face   face of the direction vector
    * @param step_width The step length to go ahead on the mesh
     surface
    * @return       new vector (also updates the ahead_face han-
     dle to correspond
    * to the new vector)
    */
  bool meshAhead(Vector& vec, lvr2::FaceHandle& face,const
   float& step_width);

  /**
   * @brief Stores the given vector map
    */
  void setVectorMap(lvr2::DenseVertexMap<mesh_map::Vector>& vec-
   tor_map);

  /**
   * @brief Publishes a position as marker. Used for debug pur-
    poses.
   * @param pos The position to publish as marker
   * @param color The marker's color
   * @param name The marker's name
   */
  void publishDebugPoint(const Vector pos, const std_msgs::Col-
   orRGBA& color, const std::string& name);

  /**
   * @brief Publishes a triangle as marker. Used for debug pur-
    poses.
   * @param face_handle The face handle for the triangle
   * @param color The marker's color
   * @param name The marker's name
   */
  void publishDebugFace(const lvr2::FaceHandle& face_handle,
   const std_msgs::ColorRGBA& color, const std::string& name);

  /**
   * @brief Publishes a vector field as visualisa-
    tion_msgs/Marker
   * @param name The marker's name
   * @param vector_map The vector_field_to_publish
   * @param publish_face_vectors Enables to publish an addi-
  tional vertex for the triangle's center.
   */
  void publishVectorField(const std::string& name, const
   lvr2::DenseVertexMap<lvr2::BaseVector<float>>& vector_map,
                      const bool publish_face_vectors = false);

  /**
   * @brief Publishes a vector field as visualisa-
    tion_msgs/Marker
   * @param name The marker's name
   * @param vector_map The vector_field_to_publish
   * @param values The vertex_cost_values
```

```
___*_@param_cost_function_A_cost_function_to_compute_costs_in-
 side_a_triangle
___*_@param_publish_face_vectors_Enables_to_publish_an_addi-
 tional_vertex_for_the_triangle's center.
  */
 void publishVectorField(const std::string& name, const
  lvr2::DenseVertexMap<lvr2::BaseVector<float>>& vector_map,
                          const lvr2::DenseVertexMap<float>&
 values,
                          const std::function<float(float)>&
 cost_function = {},
                          const bool publish_face_vectors =
 false);

 /**
  * @brief Publishes the vector field as visualisa-
   tion_msgs/Marker
  */
 void publishCombinedVectorField();

 /**
  * @brief returns a shared pointer to the specified layer
  */
 mesh_map::AbstractLayer::Ptr layer(const std::string&
  layer_name);

 std::shared_ptr<lvr2::AttributeMeshIOBase> mesh_io_ptr;
 std::shared_ptr<lvr2::HalfEdgeMesh<Vector>> mesh_ptr;

 lvr2::DenseVertexMap<bool> invalid;

 private:
 //! plugin class loader for the layer plugins
 pluginlib::ClassLoader<mesh_map::AbstractLayer> layer_loader;

 //! mapping from name to layer instance
 std::map<std::string, mesh_map::AbstractLayer::Ptr>
  layer_names;

 //! vector of name and layer instances
 std::vector<std::pair<std::string, mesh_map::Abstract-
  Layer::Ptr>> layers;

 //! each layer maps to a set of impassable indices
 std::map<std::string, std::set<lvr2::VertexHandle>> lethal_in-
  dices;

 //! all impassable vertices
 std::set<lvr2::VertexHandle> lethals;

 //! global frame / coordinate system id
 std::string global_frame;

 //! mesh file type to read
 bool ply_file;

 //! server url
```

```
std::string srv_url;

//! login username to connect to the server
std::string srv_username;

//! login password to connect to the server
std::string srv_password;

std::string mesh_layer;

float min_roughness;
float max_roughness;
float min_height_diff;
float max_height_diff;
float bb_min_x;
float bb_min_y;
float bb_min_z;
float bb_max_x;
float bb_max_y;
float bb_max_z;

std::string mesh_file;
std::string mesh_part;

//! combined layer costs
lvr2::DenseVertexMap<float> vertex_costs;

//! stored vector map to share between planner and controller
lvr2::DenseVertexMap<mesh_map::Vector> vector_map;

//! vertex distance for each edge
lvr2::DenseEdgeMap<float> edge_distances;

//! edge weights
lvr2::DenseEdgeMap<float> edge_weights;

//! triangle normals
lvr2::DenseFaceMap<Normal> face_normals;

//! vertex normals
lvr2::DenseVertexMap<Normal> vertex_normals;

//! publisher for vertex costs
ros::Publisher vertex_costs_pub;

//! publisher for vertex colors
ros::Publisher vertex_colors_pub;

//! publisher for the mesh geometry
ros::Publisher mesh_geometry_pub;

//! publisher for the debug markers
ros::Publisher marker_pub;

//! publisher for the stored vector field
ros::Publisher vector_field_pub;
```

```
  //! shared pointer to dynamic reconfigure server
  boost::shared_ptr<dynamic_reconfig-
   ure::Server<mesh_map::MeshMapConfig>> reconfigure_server_ptr;

  //! dynamic reconfigure callback function binding
  dynamic_reconfigure::Server<mesh_map::MeshMapCon-
   fig>::CallbackType config_callback;

  //! first reconfigure call
  bool first_config;

  //! indicates whether the map has been loaded
  bool map_loaded;

  //! current mesh map configuration
  MeshMapConfig config;

  //! private node handle within the mesh map namespace
  ros::NodeHandle private_nh;

  //! transformation buffer
  tf2_ros::Buffer& tf_buffer;

  //! uuid for the load mesh map
  std::string uuid_str;

  //! layer mutex to handle simultaneous layer changes
  std::mutex layer_mtx;

  //! k-d tree type for 3D with a custom mesh adaptor
  typedef nanoflann::KDTreeSingleIndexAdaptor<
      nanoflann::L2_Simple_Adaptor<float, NanoFlannMeshAdaptor>,
      NanoFlannMeshAdaptor, 3> KDTree;

  //! k-d tree nano flann mesh adaptor to access mesh data
  std::unique_ptr<NanoFlannMeshAdaptor> adaptor_ptr;

  //! k-d tree to query mesh vertices in logarithmic time
  std::unique_ptr<KDTree> kd_tree_ptr;
  };

} /* namespace mesh_map */

#endif  // MESH_NAVIGATION__MESH_MAP_H
```

B.2 mesh_map.cpp

```
/*

 *   Copyright 2020, Sebastian Putz
 *
 *   Redistribution and use in source and binary forms, with or
 without modification, are permitted provided that the follow-
 ing conditions are met:
 *
```

```
 *   1. Redistributions of source code must retain the above
        copyright notice, this list of conditions and the fol-
        lowing disclaimer.
 *
 *   2. Redistributions in binary form must reproduce the above
        copyright notice, this list of conditions and the fol-
        lowing disclaimer in the documentation and/or other ma-
        terials provided with the distribution.
 *
 *   3. Neither the name of the copyright holder nor the names
        of its contributors may be used to endorse or promote
        products derived from this software without specific
        prior written permission.
 *
 *   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
    CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRAN-
    TIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRAN-
    TIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
    PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
    HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
    INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
    (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE
    GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSI-
    NESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LI-
    ABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
    (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
    OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSI-
    BILITY OF SUCH DAMAGE.
 *
 *   authors:
 *           Sebastian Putz <spuetz@uni-osnabrueck.de>
 *
 */


#include <XmlRpcException.h>
#include <algorithm>
#include <boost/uuid/random_generator.hpp>
#include <boost/uuid/uuid.hpp>
#include <boost/uuid/uuid_io.hpp>
#include <functional>
#include <geometry_msgs/PointStamped.h>
#include <geometry_msgs/Vector3.h>
#include <visualization_msgs/MarkerArray.h>
#include <mesh_client/mesh_client.h>

#include <lvr2/geometry/Normal.hpp>
#include <lvr2/algorithm/GeometryAlgorithms.hpp>
#include <lvr2/algorithm/Normal1Algorithms.hpp>
```

```
#include <lvr2/io/hdf5/MeshIO.hpp>
#include <mesh_map/mesh_map.h>
#include <mesh_map/util.h>
#include <mesh_msgs/MeshGeometryStamped.h>
#include <mesh_msgs_conversions/conversions.h>
#include <mutex>
#include <ros/ros.h>
#include <visualization_msgs/Marker.h>

namespace mesh_map
{
using HDF5MeshIO = lvr2::Hdf5IO<lvr2::hdf5features::ArrayIO,
lvr2::hdf5features::ChannelIO,
                                 lvr2::hdf5features::VariantChan-
nelIO, lvr2::hdf5features::MeshIO>;

MeshMap::MeshMap(tf2_ros::Buffer& tf_listener)
  : tf_buffer(tf_buffer)
  , private_nh("~/mesh_map/")
  , first_config(true)
  , map_loaded(false)
  , layer_loader("mesh_map", "mesh_map::AbstractLayer")
  , mesh_ptr(new lvr2::HalfEdgeMesh<Vector>())
{
  private_nh.param<std::string>("server_url", srv_url, "");
  private_nh.param<std::string>("server_username", srv_username,
   "");
  private_nh.param<std::string>("server_password", srv_password,
   "");
  private_nh.param<std::string>("mesh_layer", mesh_layer,
   "mesh0");
  private_nh.param<float>("min_roughness", min_roughness, 0);
  private_nh.param<float>("max_roughness", max_roughness, 0);
  private_nh.param<float>("min_height_diff", min_height_diff,
   0);
  private_nh.param<float>("max_height_diff", max_height_diff,
   0);
  private_nh.param<float>("bb_min_x", bb_min_x, 0);
  private_nh.param<float>("bb_min_y", bb_min_y, 0);
  private_nh.param<float>("bb_min_z", bb_min_z, 0);
  private_nh.param<float>("bb_max_x", bb_max_x, 0);
  private_nh.param<float>("bb_max_y", bb_max_y, 0);
  private_nh.param<float>("bb_max_z", bb_max_z, 0);

  private_nh.param<std::string>("mesh_file", mesh_file, "");
  private_nh.param<std::string>("mesh_part", mesh_part, "");
  private_nh.param<std::string>("global_frame", global_frame,
   "map");
  private_nh.param<bool>("ply_file", ply_file, false);

  ROS_INFO_STREAM("mesh file is set to: " << mesh_file);

  marker_pub = private_nh.advertise<visualiza-
   tion_msgs::Marker>("marker", 100, true);
  mesh_geometry_pub = private_nh.advertise<mesh_msgs::MeshGeome-
   tryStamped>("mesh", 1, true);
```

```
  vertex_costs_pub = private_nh.advertise<mesh_msgs::MeshVertex-
   CostsStamped>("vertex_costs", 1, false);
  vertex_colors_pub = private_nh.advertise<mesh_msgs::MeshVer-
   texColorsStamped>("vertex_colors", 1, true);
  vector_field_pub = private_nh.advertise<visualiza-
   tion_msgs::Marker>("vector_field", 1, true);
  reconfigure_server_ptr = boost::shared_ptr<dynamic_reconfig-
   ure::Server<mesh_map::MeshMapConfig>>(
      new dynamic_reconfigure::Server<mesh_map::MeshMapCon-
  fig>(private_nh));

  config_callback = boost::bind(&MeshMap::reconfigureCallback,
   this, _1, _2);
  reconfigure_server_ptr->setCallback(config_callback);
}

bool MeshMap::readMap()
{
  ROS_INFO_STREAM("server url: " << srv_url);
  bool server = false;

if (!srv_url.empty())
{
  server = true;

  mesh_io_ptr = std::shared_ptr<lvr2::AttributeMeshIOBase>(
     new mesh_client::MeshClient(srv_url,
  srv_username, srv_password, mesh_layer));
      auto mesh_client_ptr = std::static_pointer_cast<mesh_cli-
  ent::MeshClient>(mesh_io_ptr);

  mesh_client_ptr->setBoundingBox(bb_min_x, bb_min_y, bb_min_z,
  bb_max_x, bb_max_y, bb_max_z);
  mesh_client_ptr->addFilter("roughness", min_roughness,
  max_roughness);
  mesh_client_ptr->addFilter("height_diff", min_height_diff,
  max_height_diff);
}
else if (!mesh_file.empty() && !mesh_part.empty())
{
  if(ply_file)
  {
   ROS_INFO_STREAM("Loading ply file at " << mesh_file);

   // generate mesh buffer pointer from file
   lvr2::MeshBufferPtr meshBfrPtr = std::make_shared
  <lvr2::MeshBuffer>();
  mesh_msgs_conversions::readMeshBuffer(meshBfrPtr, mesh_file);

  // create halfedge mesh from mesh_buffer
  ROS_INFO_STREAM("Creating HEM");
  sleep(3);
  lvr2::HalfEdgeMesh<Vector> hem = lvr2::
HalfEdgeMesh<Vector>(meshBfrPtr);
  ROS_INFO_STREAM( "numVertices = " << hem.
numVertices() << ", numFaces = " << hem.numFaces());
```

```
    // ROS_INFO_STREAM("Checking hem integrity");
    // sleep(3);
    // mesh_ptr->debugCheckMeshIntegrity();

    ROS_INFO_STREAM("Adding mesh to IO");
    HDF5MeshIO* hdf_5_mesh_io = new HDF5MeshIO();
    mesh_io_ptr = std::shared_ptr<lvr2::
  AttributeMeshIOBase>(hdf_5_mesh_io);
    hdf_5_mesh_io->open("/home/grl/Desktop/temp.h5");
    hdf_5_mesh_io->setMeshName(mesh_part);
    bool addedMesh = hdf_5_mesh_io->addMesh(hem);

   }
else
{
   ROS_INFO_STREAM("Load \"" << mesh_part << "\" from file \"" <<
mesh_file << "\"...");
   HDF5MeshIO* hdf_5_mesh_io = new HDF5MeshIO();
   hdf_5_mesh_io->open(mesh_file);
   hdf_5_mesh_io->setMeshName(mesh_part);
   mesh_io_ptr = std::shared_ptr<lvr2::AttributeMeshI-
OBase>(hdf_5_mesh_io);
   }
}
else
{
   ROS_ERROR_STREAM("Could not open file or server connection!");
   return false;
}

if (server)
{
   ROS_INFO_STREAM("Start reading the mesh from the server '" <<
srv_url);
}
else if (ply_file)
{
   ROS_INFO_STREAM("Loading a raw ply mesh from '" << mesh_file);
}
else
{
   ROS_INFO_STREAM("Start reading the mesh part '" << mesh_part <<
"' from the map file '" << mesh_file << "'...");
}

auto mesh_opt = mesh_io_ptr->getMesh();

if (mesh_opt)
{
   *mesh_ptr = mesh_opt.get();
   ROS_INFO_STREAM("The mesh has been loaded successfully with "
<< mesh_ptr->numVertices() << " vertices and "

          << mesh_ptr->numFaces() << " faces and "
          << mesh_ptr->numEdges() << " edges.");
```

```
   adaptor_ptr = std::make_unique<NanoFlannMeshAdap-
tor>(*mesh_ptr);
   kd_tree_ptr = std::make_unique<KDTree>(3,*
adaptor_ptr, nanoflann::KDTreeSingleIndexAdaptorParams(10));
   kd_tree_ptr->buildIndex();
   ROS_INFO_STREAM("The k-d tree has been build successfully!");
}
else
{
   ROS_ERROR_STREAM("Could not load the mesh '" << mesh_part <<
"' from the map file '" << mesh_file << "' ");
   return false;
}

vertex_costs = lvr2::DenseVertexMap<float>(mesh_ptr->
   nextVertexIndex(), 0);
edge_weights = lvr2::DenseEdgeMap<float>(mesh_ptr->
   nextEdgeIndex(), 0);
invalid = lvr2::DenseVertexMap<bool>(mesh_ptr->
   nextVertexIndex(), false);

// TODO read and write uuid
boost::uuids::random_generator gen;
boost::uuids::uuid uuid = gen();
uuid_str = boost::uuids::to_string(uuid);

auto face_normals_opt = mesh_io_ptr->
   getDenseAttributeMap<lvr2::DenseFaceMap<Normal>>("face_nor-
   mals");
if (face_normals_opt)
{
   face_normals = face_normals_opt.get();
   ROS_INFO_STREAM("Found " << face_normals.numValues
  () << " face_normals_in_map_file.");
}
else
{
ROS_INFO_STREAM("No face_normals_found_in_the_given_map_file,
computing_them...");
   face_normals = lvr2::calcFaceNormals(*mesh_ptr);
   ROS_INFO_STREAM("Computed " << face_normals.
numValues() << " face_normals.");
   if (mesh_io_ptr->addDenseAttributeMap(face_normals, "face_nor-
   mals"))
   {
    ROS_INFO_STREAM("Saved face_normals_to_map_file.");
   }
   else
   {
    ROS_ERROR_STREAM("Could not save face_normals_to_map_file!");
   }
}

auto vertex_normals_opt = mesh_io_ptr->getDenseAttribute-
   Map<lvr2::DenseVertexMap<Normal>>("vertex_normals");
```

```
if (vertex_normals_opt)
{
  vertex_normals = vertex_normals_opt.get();
  ROS_INFO_STREAM("Found " << vertex_normals.
 numValues() << " vertex normals in map file!");
}
else
{
  ROS_INFO_STREAM("No vertex normals found in the given map file,
 computing them...");
  vertex_normals = lvr2::calcVertexNormals(*mesh_ptr, face_nor-
  mals);
  if (mesh_io_ptr->addDenseAttributeMap(vertex_normals, "ver-
tex_normals"))
  {
   ROS_INFO_STREAM("Saved vertex normals to map file.");
  }
  else
  {
  ROS_ERROR_STREAM("Could not save vertex normals to map file!");
  }
}

mesh_geometry_pub.publish(mesh_msgs_conversions::toMeshGeome-
tryStamped<float>(*mesh_ptr, global_frame, uuid_str, vertex_nor-
mals));

ROS_INFO_STREAM("Try to read edge distances from map file...");
auto edge_distances_opt = mesh_io_ptr->getAttribute-
  Map<lvr2::DenseEdgeMap<float>>("edge_distances");

if (edge_distances_opt)
{
  ROS_INFO_STREAM("Vertex distances have been read success-
fully.");
  edge_distances = edge_distances_opt.get();
}
else
{
  ROS_INFO_STREAM("Computing edge distances...");
  edge_distances = lvr2::calcVertexDistances(*mesh_ptr);
  ROS_INFO_STREAM("Saving " << edge_distances.numValues() <<
"_edge_distances_to_map_file...");

  if (mesh_io_ptr->addAttributeMap(edge_distances, "edge_dis-
tances"))
  {
  ROS_INFO_STREAM("Saved edge distances to map file.");
  }
  else
  {
    ROS_ERROR_STREAM("Could not save edge_dis-
 tances_to_map_file!");
 }
 }
```

```
 ROS_INFO_STREAM("Load layer plugins...");
 if (!loadLayerPlugins())
 {
   ROS_FATAL_STREAM("Could not load any layer plugin!");
 return false;
 }

 ROS_INFO_STREAM("Initialize layer plugins...");
 if (!initLayerPlugins())
 {
  ROS_FATAL_STREAM("Could not initialize plugins!");
  return false;
 }

 sleep(1);

 combineVertexCosts();
 publishCostLayers();
 publishVertexColors();

 map_loaded = true;
 return true;
}

bool MeshMap::loadLayerPlugins()
{
  XmlRpc::XmlRpcValue plugin_param_list;
  if (!private_nh.getParam("layers", plugin_param_list)
  )
  {
   ROS_WARN_STREAM("No layer plugins configured! -
   Use the param \"layers\" " "in the namespace \""
.....<< private_nh.getNamespace()
.....<< "\". \"lay-
   ers\" must be must be a list of "
         "tuples with a name and a type.");
  return false;
 }

 try
 {
   for (int i = 0; i < plugin_param_list.size(); i++)
   {
    XmlRpc::XmlRpcValue elem = plugin_param_list[i];

    std::string name = elem["name"];
    std::string type = elem["type"];

    typename AbstractLayer::Ptr plugin_ptr;

    if (layer_names.find(name) != layer_names.end())
    {
    ROS_ERROR_STREAM("The plugin \"" << name << "\" has al-
    ready been loaded! Names must be unique!");
     return false;
   }
```

```
    try
    {
     plugin_ptr = layer_loader.createInstance(type);
    }
    catch (pluginlib::LibraryLoadException& e)
    {
     ROS_ERROR_STREAM(e.what());
    }

    if (plugin_ptr)
    {
     std::pair<std::string, typename mesh_map::AbstractLayer::Ptr>
elem(name, plugin_ptr);

    layers.push_back(elem);
    layer_names.insert(elem);

    ROS_INFO_STREAM("The␣layer␣plugin␣with␣the␣type␣\""
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣<<␣type␣<<␣"\"␣has␣been␣loaded␣suc-
  cessfully␣under␣the␣name␣\""␣<<␣name␣<<␣"\".");
    }
    else
    {
        ROS_EROR_STREAM("Could␣not␣load␣the␣layer␣plugin␣with␣the␣
  name␣\""␣<<␣name␣<<␣"\"␣and␣the␣type␣\""␣<<␣type
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣<< "\"!");
    }
   }
  }
  catch (XmlRpc::XmlRpcException& e)
  {
    ROS_ERROR_STREAM("Invalid␣parameter␣structure.␣The␣\"lay-
 ers\"␣parameter␣"
          "has␣to␣be␣a␣list␣of␣structs␣"
          << "with␣fields␣\"name\"␣and␣\"type\"!");
   ROS_ERROR_STREAM(e.getMessage());
   return false;
  }
  // is there any layer plugin loaded for the map?
  return !layers.empty();
}

void MeshMap::layerChanged(const std::string& layer_name)
{
  std::lock_guard<std::mutex> lock(layer_mtx);

  ROS_INFO_STREAM("Layer␣\""␣<<␣layer_name␣<<␣"\"␣changed.");

  lethals.clear();

  ROS_INFO_STREAM("Combine␣underlining␣lethal␣sets...");

  // TODO pre-compute combined lethals upto a layer level
```

```
  auto layer_iter = layers.begin();
  for (; layer_iter != layers.end(); layer_iter++)
  {
   // TODO add lethal and removae lethal sets
   lethals.insert(layer_iter->second->lethals().begin(),
   layer_iter->second->lethals().end());
   // TODO merge with std::set_merge
   if (layer_iter->first == layer_name)
   break;
  }

   vertex_costs_pub.publish(mesh_msgs_conversions::toVertex-
   CostsStamped(layer_iter->second->costs(), mesh_ptr->numVerti-
   ces(),

   layer_iter->second->defaultValue(), layer_iter->first,

   global_frame, uuid_str));

  if (layer_iter != layers.end())layer_iter++;

  ROS_INFO_STREAM("Combine lethal sets...");

  for (; layer_iter != layers.end(); layer_iter++)
  {
   // TODO add lethal and remove lethal sets as param
   layer_iter->second->updateLethal(lethals, lethals);

   lethals.insert(layer_iter->second->lethals().begin(),
   layer_iter->second->lethals().end());

     vertex_costs_pub.publish(mesh_msgs_conversions::toVertex-
  CostsStamped(layer_iter->second->costs(), mesh_ptr->numVerti-
  ces(),

   layer_iter->second->defaultValue(), layer_iter->first,

   global_frame, uuid_str));
  }

  ROS_INFO_STREAM("Found " << lethals.size() << " lethal verti-
   ces");
  ROS_INFO_STREAM("Combine layer costs...");

  combineVertexCosts();
  // TODO new lethals old lethals -> renew potential field!
   around this areas
}

bool MeshMap::initLayerPlugins()
{
  lethals.clear();
  lethal_indices.clear();

  std::shared_ptr<mesh_map::MeshMap> map(this);

  for (auto& layer : layers)
```

```
    {
      auto& layer_plugin = layer.second;
      const auto& layer_name = layer.first;

      auto callback = [this](const std::string& layer_name) { layer-
    Changed(layer_name); };

      if (!layer_plugin->initialize(layer_name, callback, map,
      mesh_ptr, mesh_io_ptr))
      {
        ROS_ERROR_STREAM("Could not initial-
        ize the layer plugin with the name \"" << layer_name << "\"!")
        ;
        return false;
      }

      std::set<lvr2::VertexHandle> empty;
      layer_plugin->updateLethal(lethals, empty);
      if (!layer_plugin->readLayer())
      {
        layer_plugin->computeLayer();
      }

      lethal_indices[layer_name].insert(layer_plugin->le-
      thals().begin(), layer_plugin->lethals().end());
      lethals.insert(layer_plugin->lethals().begin(), layer_plugin-
      >lethals().end());
    }
    return true;
}

void MeshMap::combineVertexCosts()
{
    ROS_INFO_STREAM("Combining costs...");

    float combined_min = std::numeric_limits<float>::max();
    float combined_max = std::numeric_limits<float>::min();

    vertex_costs = lvr2::DenseVertexMap<float>(mesh_ptr->nextVer-
      texIndex(), 0);

    bool hasNaN = false;
    for (auto layer : layers)
    {
      const auto& costs = layer.second->costs();
      float min, max;
      mesh_map::getMinMax(costs, min, max);
      const float norm = max - min;
      const float factor = private_nh.param<float>(layer.first +
    "/factor", 1.0);
      const float norm_factor = factor / norm;
      ROS_INFO_STREAM("Layer \"" << layer.first << "\" max value: "
    << max << " min value: " << min << " norm: " << norm
              << " factor: " << factor
      << " norm_factor: " << norm_factor);
```

```
    const float default_value = layer.second->
 defaultValue();
    hasNaN = false;
    for (auto vH : mesh_ptr->vertices())
    {
     const float cost = costs.containsKey(vH) ? costs[vH] : de-
fault_value;
     if (std::isnan(cost))
      hasNaN = true;
     vertex_costs[vH] += factor * cost;
     if (std::isfinite(cost))
     {
      combined_max = std::max(combined_max, vertex_costs[vH]);
      combined_min = std::min(combined_min, vertex_costs[vH]);
     }
   }
   if (hasNaN)
       ROS_ERROR_STREAM("Layer \"" << layer.first << "\" con-
 tains NaN values!");
}

const float combined_norm = combined_max - combined_min;

for (auto vH : lethals)
{
vertex_costs[vH] = std::numeric_limits<float>::infinity();
}

vertex_costs_pub.publish(mesh_msgs_conversions::toVertex-
 CostsStamped(vertex_costs, "Combined Costs", global_frame,
 uuid_str));

hasNaN = false;

ROS_INFO_STREAM("Layer weighting factor is: " << con-
 fig.layer_factor);
for (auto eH : mesh_ptr->edges())
{
   // Get both Vertices of the current Edge
    std::array<lvr2::VertexHandle, 2> eH_vHs = mesh_ptr->getVerti-
 cesOfEdge(eH);
   const lvr2::VertexHandle& vH1 = eH_vHs[0];
   const lvr2::VertexHandle& vH2 = eH_vHs[1];
   // Get the Riskiness for the current Edge (the maximum value
from both
   // Vertices)
   if (config.layer_factor != 0)
   {
    if (std::isinf(vertex_costs[vH1]) || std::isinf(ver-
tex_costs[vH2]))
    {
     edge_weights[eH] = edge_distances[eH];
     // edge_weights[eH] = std::numeric_limits<float
>::infinity();
    }
    else
    {
```

```
     float cost_diff = std::fabs(vertex_costs[vH1] - ver-
tex_costs[vH2]);

     float vertex_factor = config.layer_factor * cost_diff;
     if (std::isnan(vertex_factor))
      ROS_INFO_STREAM("NaN: v1:" << vertex_costs[vH1] << " v2:"
<< vertex_costs[vH2]
                                                << " vertex_factor
:" << vertex_factor << " cost_diff:" << cost_diff);
     edge_weights[eH] = edge_distances[eH] * (1 + vertex_factor);
    }
   }
   else
   {
    edge_weights[eH] = edge_distances[eH];
   }
  }

 ROS_INFO("Successfully combined costs!");
}

void MeshMap::findLethalByContours(const int&
   min_contour_size, std::set<lvr2::VertexHandle>&
   lethals)
{
  int size = lethals.size();
  std::vector<std::vector<lvr2::VertexHandle>> contours;
  findContours(contours, min_contour_size);
  for (auto contour : contours)
  {
    lethals.insert(contour.begin(), contour.end());
  }
  ROS_INFO_STREAM("Found " << lethals.size() - size <<
    " lethal vertices as contour vertices");
}

void MeshMap::findContours(std::vector<std::vector<lvr2
  ::VertexHandle>>& contours, int min_contour_size)
{
  ROS_INFO_STREAM("Find contours...");

  std::vector<std::vector<lvr2::VertexHandle>>
   tmp_contours;

  array<lvr2::OptionalFaceHandle, 2> facepair;
  lvr2::SparseEdgeMap<bool> usedEdges(false);
  for (auto eHStart : mesh_ptr->edges())
  {
    lvr2::SparseVertexMap<bool> usedVertices(false);
    lvr2::SparseEdgeMap<bool> local_usedEdges(false);
    int count = 0;

    // Look for border Edges
    facepair = mesh_ptr->getFacesOfEdge(eHStart);

    // If border Edge found
```

```
if ((!facepair[0] || !facepair[1]) && !usedEdges[eHStart])
{
std:
 vector<lvr2::VertexHandle> contour;
 // Set vector which links to the following Edge
 array<lvr2::VertexHandle, 2> vertexPair =
mesh_ptr->getVerticesOfEdge(eHStart);
 lvr2::VertexHandle vH = vertexPair[1];
 vector<lvr2::EdgeHandle> curEdges;
 lvr2::EdgeHandle eHTemp = eHStart;
 bool moving = true;
 bool vertex_flag = false;

 // While the contour did not come full circle
 while (moving)
 {
  moving = false;
  usedEdges.insert(eHTemp, true);
  local_usedEdges.insert(eHTemp, true);
  // Set vector which links to the following Edge
  vertexPair = mesh_ptr->getVerticesOfEdge(eHTemp);
  // Eliminate the possibility to choose the previous Vertex
  if (vH != vertexPair[0])
  {
    vH = vertexPair[0];
  }
  else if (vH != vertexPair[1])
  {
    vH = vertexPair[1];
  }

  // Add the current Vertex to the contour
  usedVertices.insert(vH, true);
  count++;
  contour.push_back(vH);
  mesh_ptr->getEdgesOfVertex(vH, curEdges);

  // Look for other edge of vertex that is a border Edge
  for (auto eHT : curEdges)
  {
   if (!usedEdges[eHT] && !local_usedEdges[eHT])
   {
    facepair = mesh_ptr->getFacesOfEdge(eHT);
    if (!facepair[0] || !facepair[1])
    {
     eHTemp = eHT;
     moving = true;
     continue;
    }
   }
  }
 }
 // Add contour to list of contours
 if (contour.size() > min_contour_size)
 {
  contours.push_back(contour);
 }
```

```
    }
   }

   ROS_INFO_STREAM("Found " << contours.size() << " contours.");
}

void MeshMap::setVectorMap(lvr2::DenseVertexMap<
  mesh_map::Vector>& vector_map)
{
   this->vector_map = vector_map;
}

boost::optional<Vector> MeshMap::directionAtPosition(
   const lvr2::VertexMap<lvr2::BaseVector<float>>&
    vector_map,
   const std::array<lvr2::VertexHandle, 3>& vertices,
   const std::array<float, 3>& barycentric_coords)
{
  const auto& a = vector_map.get(vertices[0]);
  const auto& b = vector_map.get(vertices[1]);
  const auto& c = vector_map.get(vertices[2]);

  if (a || b || c)
  {
    lvr2::BaseVector<float> vec;
    if (a) vec += a.get() * barycentric_coords[0];
    if (b) vec += b.get() * barycentric_coords[1];
    if (c) vec += c.get() * barycentric_coords[2];
    if (std::isfinite(vec.x) && std::isfinite(vec.y) &&
    std::isfinite(vec.z))
      return vec;
    else
      ROS_ERROR_THROTTLE(0.3, "vector_map_contains_
   invalid_vectors!");
   }
   else
   {
     ROS_ERROR_THROTTLE(0.3, "vector_map_does_not_con-
   tain_any_of_the_corresponding_vectors");
   }
   return boost::none;
}

   float MeshMap::costAtPosition(const std::array<lvr2::
     VertexHandle, 3>& vertices,
                                 const std::array<float,
    3>& barycentric_coords)
  {
   return costAtPosition(vertex_costs, vertices,
    barycentric_coords);
  }

   float MeshMap::costAtPosition(const lvr2::
     DenseVertexMap<float>& costs,
                                 const std::array<lvr2::
    VertexHandle, 3>& vertices,
```

```
                                const std::array<float,
    3>& barycentric_coords)
{
  const auto& a = costs.get(vertices[0]);
  const auto& b = costs.get(vertices[1]);
  const auto& c = costs.get(vertices[2]);

  if (a && b && c)
  {
    std::array<float, 3> costs = { a.get(), b.get(), c.get() };
    return mesh_map::linearCombineBarycentricCoords(
   costs, barycentric_coords);
  }
  return std::numeric_limits<float>::quiet_NaN();
  }

  void MeshMap::publishDebugPoint(const Vector pos, const
     std_msgs::ColorRGBA& color, const std::string& name)
  {
    visualization_msgs::Marker marker;
    marker.header.frame_id = mapFrame();
    marker.header.stamp = ros::Time();
    marker.ns = name;
    marker.id = 0;
    marker.type = visualization_msgs::Marker::SPHERE;
    marker.action = visualization_msgs::Marker::ADD;
    geometry_msgs::Vector3 scale;
    scale.x = 0.05;
    scale.y = 0.05;
    scale.z = 0.05;
    marker.scale = scale;

    geometry_msgs::Pose p;
    p.position.x = pos.x;
    p.position.y = pos.y;
    p.position.z = pos.z;
    marker.pose = p;
    marker.color = color;
    marker_pub.publish(marker);
  }

  void MeshMap::publishDebugFace(const lvr2::FaceHandle&
      face_handle, const std_msgs::ColorRGBA& color,
                                  const std::string& name)
  {
   const auto& vertices = mesh_ptr->getVerticesOfFace(face_han-
    dle);
   visualization_msgs::Marker marker;
   marker.header.frame_id = mapFrame();
   marker.header.stamp = ros::Time();
   marker.ns = name;
   marker.id = 0;
   marker.type = visualization_msgs::Marker::TRIANGLE_LIST;
   marker.action = visualization_msgs::Marker::ADD;
   geometry_msgs::Vector3 scale;
   scale.x = 1.0;
   scale.y = 1.0;
```

```
  scale.z = 1.0;
  marker.scale = scale;

  for (auto vertex : vertices)
  {
   auto& pos = mesh_ptr->getVertexPosition(vertex);
   geometry_msgs::Point p;
   p.x = pos.x;
   p.y = pos.y;
   p.z = pos.z;
   marker.points.push_back(p);
   marker.colors.push_back(color);
  }
 marker_pub.publish(marker);
}

void MeshMap::publishVectorField(const std::string&
    name,
                                 const lvr2::
    DenseVertexMap<lvr2::BaseVector<float>>& vector_map,
                                 const bool
    publish_face_vectors)
{
  publishVectorField(name, vector_map, vertex_costs,
    {}, publish_face_vectors);
}

void MeshMap::publishCombinedVectorField()
{
  lvr2::DenseVertexMap<Vector> vertex_vectors;
  lvr2::DenseFaceMap<Vector> face_vectors;

  vertex_vectors.reserve(mesh_ptr->nextVertexIndex());
  face_vectors.reserve(mesh_ptr->nextFaceIndex());

  for (auto layer_iter : layer_names)
  {
    lvr2::DenseFaceMap<uint8_t> vector_field_faces(
  mesh_ptr->nextFaceIndex(), 0);
  AbstractLayer::Ptr layer = layer_iter.second;
  auto opt_vec_map = layer->vectorMap();
  if (!opt_vec_map)
   continue;

  const auto& vecs = opt_vec_map.get();
  for (auto vH : vecs)
  {
   auto opt_val = vertex_vectors.get(vH);
   vertex_vectors.insert(vH, opt_val ? opt_val.get()
  + vecs[vH] : vecs[vH]);
  for (auto fH : mesh_ptr->getFacesOfVertex(vH))
   vector_field_faces[fH]++;
  }

  for (auto fH : vector_field_faces)
  {
   if (vector_field_faces[fH] != 3)
```

```
      continue;

    const auto& vertices = mesh_ptr->getVertexPositionsOfFace(fH);
    const auto& vertex_handles = mesh_ptr->getVerticesOfFace(fH);
    mesh_map::Vector center = (vertices[0] + vertices
 [1] + vertices[2]) / 3;
    std::array<float, 3> barycentric_coords;
    float dist;
    if (mesh_map::projectedBarycentricCoords(center,
 vertices, barycentric_coords, dist))
    {
     auto opt_val = face_vectors.get(fH);
     auto vec_at = layer->vectorAt(vertex_handles,
 barycentric_coords);
     if (vec_at != Vector())
     {
     face_vectors.insert(fH, opt_val ? opt_val.get
 () + vec_at : vec_at);
      }
     }
    }
  }
}

void MeshMap::publishVectorField(const std::string& name,
                                 const lvr2::Dense-
   VertexMap<lvr2::BaseVector<float>>& vector_map,
                                 const lvr2::
   DenseVertexMap<float>& values,
                                 const std::function<
   float(float)>& cost_function, const bool
   publish_face_vectors)
{
  const auto& mesh = this->mesh();
  const auto& vertex_costs = vertexCosts();
  const auto& face_normals = faceNormals();

  visualization_msgs::Marker vector_field;

  geometry_msgs::Pose pose;
  pose.position.x = pose.position.y = pose.position.z = 0;
  pose.orientation.x = pose.orientation.y = pose.orientation.z =
  0;
  pose.orientation.w = 1;
  vector_field.pose = pose;

  vector_field.type = visualization_msgs::Marker::LINE_LIST;
  vector_field.header.frame_id = mapFrame();
  vector_field.header.stamp = ros::Time::now();
  vector_field.ns = name;
  vector_field.scale.x = 0.01;
  vector_field.color.a = 1;
  vector_field.id = 0;

  vector_field.colors.reserve(2 * vector_map.numValues());
  vector_field.points.reserve(2 * vector_map.numValues());
```

```
  unsigned int cnt = 0;
  unsigned int faces = 0;

  lvr2::DenseFaceMap<uint8_t> vector_field_faces(mesh.
   numFaces(), 0);
  std::set<lvr2::FaceHandle> complete_faces;

  for (auto vH : vector_map)
  {
    const auto& dir_vec = vector_map[vH];
    const float len2 = dir_vec.length2();
    if (len2 == 0 || !std::isfinite(len2))
    {
     ROS_DEBUG_STREAM_THROTTLE(0.3, "Found␣invalid␣direction␣vec-
 tor␣in␣vector_field␣\""␣<<␣name␣<<␣"\".␣Ignoring␣it!");
     continue;
   }

   auto u = mesh.getVertexPosition(vH);
   auto v = u + dir_vec * 0.1;

   u.z = u.z + 0.01;
   v.z = v.z + 0.01;

   if (!std::isfinite(u.x) || !std::isfinite(u.y) || !
std::isfinite(u.z) || !std::isfinite(v.x) ||
      !std::isfinite(v.y) || !std::isfinite(v.z))
   {
    continue;
   }
   vector_field.points.push_back(toPoint(u));
   vector_field.points.push_back(toPoint(v));

   const float value = cost_function ? cost_function(
values[vH]) : values[vH];
   std_msgs::ColorRGBA color = getRainbowColor(value);
   vector_field.colors.push_back(color);
   vector_field.colors.push_back(color);

  cnt++;
  // vector.header.seq = cnt;
  // vector.id = cnt++;
  // vector_field.markers.push_back(vector);
  try
  {
    for (auto fH : mesh.getFacesOfVertex(vH))
    {
      if (++vector_field_faces[fH] == 3)
      {
        faces++;
        complete_faces.insert(fH);
      }
    }
  }
  catch (lvr2::PanicException exception)
  {
    invalid.insert(vH, true);
```

```
    }
  }

  size_t invalid_cnt = 0;
  for (auto vH : invalid)
  {
    if (invalid[vH])
      invalid_cnt++;
  }

  if (invalid_cnt > 0)
  {
    ROS_WARN_STREAM("Found " << invalid_cnt << " non mani-
fold vertices!");
  }
  ROS_INFO_STREAM("Found " << faces << " complete vec-
tor faces!");

  if (publish_face_vectors)
  {
    vector_field.points.reserve(faces + cnt);

    for (auto fH : complete_faces)
    {
      const auto& vertices = mesh.getVertexPositionsOfFace(fH);
      const auto& vertex_handles = mesh.getVerticesOfFace(fH);
      mesh_map::Vector center = (vertices[0] + vertices[1] + ver-
tices[2]) / 3;
      std::array<float, 3> barycentric_coords;
      float dist;
      if (mesh_map::projectedBarycentricCoords(center, vertices,
barycentric_coords, dist))
      {
        boost::optional<mesh_map::Vector> dir_opt = directionAt-
Position(vector_map, vertex_handles, barycentric_coords);
        if (dir_opt)
        {
          const float& cost = costAtPosition(values, vertex_han-
dles, barycentric_coords);
          const float& value = cost_function ? cost_func-
tion(cost) : cost;

          // vector.color = getRainbowColor(value);
          // vector.pose = mesh_map::calculatePoseFromDirection(
          //    center, dir_opt.get(), face_normals[fH]);

          auto u = center;
          auto v = u + dir_opt.get() * 0.1;

          if (!std::isfinite(u.x) || !std::isfinite(u.y) ||
!std::isfinite(u.z) || !std::isfinite(v.x) ||
              !std::isfinite(v.y) || !std::isfinite(v.z))
          {
            continue;
          }

          // vector_field.header.seq = cnt;
```

```
          // vector_field.id = cnt++;
          vector_field.points.push_back(toPoint(u));
          vector_field.points.push_back(toPoint(v));

          std_msgs::ColorRGBA color = getRainbowColor(value);
          vector_field.colors.push_back(color);
          vector_field.colors.push_back(color);
        }
        else
        {
          ROS_ERROR_STREAM_THROTTLE(0.3, "Could not compute the
direction!");
        }
      }
      else
      {
        ROS_ERROR_STREAM_THROTTLE(0.3, "Could not compute the
barycentric coords!");
      }
    }
  }
  vector_field_pub.publish(vector_field);
  ROS_INFO_STREAM("Published vec-
tor field \"" << name << "\" with " << cnt << " elements.");
}

bool MeshMap::inTriangle(const Vector& pos, const lvr2
  ::FaceHandle& face, const float& dist)
{
  const auto& vertices = mesh_ptr->getVerticesOfFace(face);
  return mesh_map::inTriangle(pos, mesh_ptr->
    getVertexPosition(vertices[0]), mesh_ptr->
    getVertexPosition(vertices[1]),
                                  mesh_ptr->
    getVertexPosition(vertices[2]), dist, 0.0001);
}

boost::optional<std::tuple<lvr2::FaceHandle, std::array
    <Vector, 3>, std::array<float, 3>>>
  MeshMap::searchNeighbourFaces(
    const Vector& pos, const lvr2::FaceHandle& face,
    const float& max_radius, const float& max_dist)
{
  std::list<lvr2::FaceHandle> possible_faces;
  possible_faces.push_back(face);
  std::list<lvr2::FaceHandle>::iterator face_iter =
    possible_faces.begin();

  Vector center(0, 0, 0);
  const auto& start_vertices = mesh_ptr->
    getVertexPositionsOfFace(face);
  for (auto vertex : start_vertices)
  {
    center += vertex;
  }
  center /= 3;
```

```
  float vertex_center_max = 0;
  for (auto vertex : start_vertices)
  {
    vertex_center_max = std::max(vertex_center_max, vertex.dis-
tance(center));
  }

  float ext_radius = max_radius + vertex_center_max;
  float max_radius_sq = ext_radius * ext_radius;

  lvr2::SparseFaceMap<bool> in_list_map;
  in_list_map.insert(face, true);

  std::array<float, 3> bary_coords;

  while (possible_faces.end() != face_iter)
  {
    const auto& vertices = mesh_ptr->
getVertexPositionsOfFace(*face_iter);
    float dist;
    if (mesh_map::projectedBarycentricCoords(pos, vertices,
bary_coords, dist) && std::fabs(dist) < max_dist)
    {
      return std::make_tuple(*face_iter, vertices, bary_coords);
    }
    else
    {
      const auto& vertices = mesh_ptr->getVertice-
sOfFace(*face_iter);
      for (auto vertex : vertices)
      {
        if (center.distance2(mesh_ptr->getVertexPosition(vertex))
< max_radius_sq)
        {
          try
          {
            const auto& nn_faces = mesh_ptr->getFacesOfVer-
tex(vertex);
            for (auto nn_face : nn_faces)
            {
              if (!in_list_map.containsKey(nn_face))
              {
                possible_faces.push_back(nn_face);
                in_list_map.insert(nn_face, true);
              }
            }
          }
          catch (lvr2::PanicException exception)
          {
            // TODO handle case properly
          }
        }
      }
      ++face_iter;
    }
  }
```

```
  return boost::none;
}

bool MeshMap::meshAhead(mesh_map::Vector& pos, lvr2::FaceHandle&
    face, const float& step_size)
{
  std::array<float, 3> bary_coords;
  float dist;
  // get barycentric coordinates of the current face or the next
    neighbour face
  if (mesh_map::projectedBarycentricCoords(pos, mesh_ptr->getVer-
    texPositionsOfFace(face), bary_coords, dist))
  {

  }
  else if (auto search_res_opt = searchNeighbourFaces(pos, face,
    step_size, 0.4))
  {
    auto search_res = *search_res_opt;
    face = std::get<0>(search_res);
    std::array<Vector, 3> vertices = std::get<1>(search_res);
    bary_coords = std::get<2>(search_res);

    // project position onto surface
    pos = mesh_map::linearCombineBarycentricCoords(vertices,
bary_coords);
  }
  else
  {
    return false;
  }
  const auto& opt_dir = directionAtPosition(vector_map, mesh_ptr-
    >getVerticesOfFace(face), bary_coords);
  if (opt_dir)
  {
    Vector dir = opt_dir.get().normalized();
    std::array<lvr2::VertexHandle, 3> handels = mesh_ptr->getVer-
ticesOfFace(face);
    // iter over all layer vector fields
    for (auto layer : layers)
    {
      dir += layer.second->vectorAt(handels,
bary_coords);
    }
    dir.normalize();
    pos += dir * step_size;
    return true;
  }
  return false;
}

lvr2::OptionalFaceHandle MeshMap::getContainingFace(Vector& posi-
    tion, const float& max_dist)
{
  auto search_result = searchContainingFace(position, max_dist);
    if(search_result)
    return std::get<0>(*search_result);
```

```
  return lvr2::OptionalFaceHandle();
}

boost::optional<std::tuple<lvr2::FaceHandle, std::ar-
    ray<mesh_map::Vector , 3>,
    std::array<float, 3>>> MeshMap::searchContainingFace(
    Vector& position, const float& max_dist)
{
  if(auto vH_opt = getNearestVertexHandle(position))
  {
    auto vH = vH_opt.unwrap();
    float min_triangle_position_distance = std::numeric_lim-
its<float>::max();
    std::array<Vector, 3> vertices;
    std::array<float, 3> bary_coords;
    lvr2::OptionalFaceHandle opt_fH;
    for(auto fH : mesh_ptr->getFacesOfVertex(vH))
    {
      const auto& tmp_vertices = mesh_ptr->getVertexPosi-
tionsOfFace(fH);
      float dist = 0;
      std::array<float, 3> tmp_bary_coords;
      if (mesh_map::projectedBarycentricCoords(position, verti-
ces, tmp_bary_coords, dist)
          && std::fabs(dist) < max_dist)
      {
        return std::make_tuple(fH, tmp_vertices,
tmp_bary_coords);
      }

      float triangle_dist = 0;
      triangle_dist += (vertices[0] - position).length2();
      triangle_dist += (vertices[1] - position).length2();
      triangle_dist += (vertices[2] - position).length2();
      if(triangle_dist < min_triangle_position_distance)
      {
        min_triangle_position_distance = triangle_dist;
        opt_fH = fH;
        vertices = tmp_vertices;
        bary_coords = tmp_bary_coords;
      }
    }
    if(opt_fH)
    {
      return std::make_tuple(opt_fH.unwrap(), vertices,
bary_coords);
    }
    ROS_ERROR_STREAM("No containing face found!");
    return boost::none;
  }
  ROS_FATAL_STREAM("Could not find the nearest vertex");
  return boost::none;
}

lvr2::OptionalVertexHandle MeshMap::getNearestVertexHandle(const
    Vector& pos)
{
```

```
  float querry_point[3] = {pos.x, pos.y, pos.z};
  size_t ret_index;
  float out_dist_sqr;
  size_t num_results = kd_tree_ptr->knnSearch(&querry_point[0],
    1, &ret_index, &out_dist_sqr);
  return num_results == 0 ? lvr2::OptionalVertexHandle() :
    lvr2::VertexHandle(ret_index);
}

inline const geometry_msgs::Point MeshMap::toPoint(const Vector&
    vec)
{
  geometry_msgs::Point p;
  p.x = vec.x;
  p.y = vec.y;
  p.z = vec.z;
  return p;
}

constexpr float kEpsilon = 1e-8;

bool MeshMap::projectedBarycentricCoords(const Vector& p, const
    lvr2::FaceHandle& triangle,
                                                 std::array<float, 3>&
    barycentric_coords, float& dist)
{
  const auto& face = mesh_ptr->getVertexPositionsOfFace(trian-
    gle);
  return mesh_map::projectedBarycentricCoords(p, face, barycen-
    tric_coords, dist);
}

mesh_map::AbstractLayer::Ptr MeshMap::layer(const std::string&
    layer_name)
{
  return layer_names[layer_name];
}

bool MeshMap::barycentricCoords(const Vector& p, const
    lvr2::FaceHandle& triangle, float& u, float& v, float& w)
{
  const auto& face = mesh_ptr->getVertexPositionsOfFace
    (triangle);
  return mesh_map::barycentricCoords(p, face[0], face[1],
    face[2], u, v, w);
}

bool MeshMap::rayTriangleIntersect(const Vector& orig, const Vec-
    tor& dir, const Vector& v0, const Vector& v1,
                                       const Vector& v2, float& t,
    float& u, float& v, Vector& p)
{
  // compute plane's normal
  Vector v0v1 = v1 - v0;
  Vector v0v2 = v2 - v0;

  // no need to normalize
```

```
__Vector_N_=_v0v1.cross(v0v2);__//_N
__float_denom_=_N.dot(N);

__//_Step_1:_finding_P

__//_check_if_ray_and_plane_are_parallel_?
__float_NdotRayDirection_=_N.dot(dir);
__if_(fabs(NdotRayDirection)_<_kEpsilon)__//_almost_0
____return_false;_____________________________//_they_are_paral-
lel_so_they_don't_intersect_!

  // compute d parameter using equation 2
  float d = N.dot(v0);

  // compute t (equation 3)
  t = (N.dot(orig) + d) / NdotRayDirection;

  // check if the triangle is in behind the ray
  // if (t < 0) return false; // the triangle is behind

  // compute the intersection point using equation 1
  p = orig + dir * t;

  // Step 2: inside-outside test
  Vector C;  // vector perpendicular to triangle's plane

__//_edge_0
__Vector_edge0_=_v1_-_v0;
__Vector_vp0_=_p_-_v0;
__C_=_edge0.cross(vp0);
__if_(N.dot(C)_<_0)
____return_false;__//_P_is_on_the_right_side

__//_edge_1
__Vector_edge1_=_v2_-_v1;
__Vector_vp1_=_p_-_v1;
__C_=_edge1.cross(vp1);
__if_((u_=_N.dot(C))_<_0)
____return_false;__//_P_is_on_the_right_side

__//_edge_2
__Vector_edge2_=_v0_-_v2;
__Vector_vp2_=_p_-_v2;
__C_=_edge2.cross(vp2);
__if_((v_=_N.dot(C)_)<_0)
____return_false;__//_P_is_on_the_right_side;

__u_/=_denom;
_v_/=_denom;

__return_true;__//_this_ray_hits_the_triangle
}

bool_MeshMap::resetLayers()
```

```
{
  return true; // TODO implement
}

void MeshMap::publishCostLayers()
{
  for (auto& layer : layers)
  {
    vertex_costs_pub.publish(mesh_msgs_conversions::toVertex-
      CostsStamped(layer.second->costs(), mesh_ptr->numVertices(),
      layer.second->defaultValue(), layer.first, global_frame,
      uuid_str));
  }
  vertex_costs_pub.publish(mesh_msgs_conversions::toVertex-
    CostsStamped(vertex_costs, "Combined_Costs", global_frame,
    uuid_str));
}

void MeshMap::publishVertexCosts(const lvr2::VertexMap<float>&
    costs, const std::string& name)
{
  vertex_costs_pub.publish(
      mesh_msgs_conversions::toVertexCostsStamped(costs,
    mesh_ptr->numVertices(), 0, name, global_frame, uuid_str));
}

void MeshMap::publishVertexColors()
{
  using VertexColorMapOpt = lvr2::DenseVertexMapOptional<std::ar-
    ray<uint8_t, 3>>;
  using VertexColorMap = lvr2::DenseVertexMap<std::array<uint8_t,
    3>>;
  VertexColorMapOpt vertex_colors_opt = this->mesh_io_ptr-
    >getDenseAttributeMap<VertexColorMap>("vertex_colors");
  if (vertex_colors_opt)
  {
    const VertexColorMap colors = vertex_colors_opt.get();
    mesh_msgs::MeshVertexColorsStamped msg;
    msg.header.frame_id = mapFrame();
    msg.header.stamp = ros::Time::now();
    msg.uuid = uuid_str;
    msg.mesh_vertex_colors.vertex_colors.reserve(col-
      ors.numValues());
    for (auto vH : colors)
    {
      std_msgs::ColorRGBA color_rgba;
      const auto& color_array = colors[vH];
      color_rgba.a = 1;
      color_rgba.r = color_array[0] / 255.0;
      color_rgba.g = color_array[1] / 255.0;
      color_rgba.b = color_array[2] / 255.0;
```

```
      msg.mesh_vertex_colors.vertex_colors.push_back(color_rgba);
    }
    this->vertex_colors_pub.publish(msg);
  }
}

void MeshMap::reconfigureCallback(mesh_map::MeshMapConfig& cfg,
    uint32_t level)
{
  ROS_INFO_STREAM("Dynamic reconfigure callback...");
  if (first_config)
  {
    config = cfg;
    first_config = false;
    return;
  }

  if (!first_config && map_loaded)
  {
    if  (cfg.cost_limit != config.cost_limit)
    {
      combineVertexCosts();
    }

    config = cfg;
  }
}

const std::string MeshMap::getGlobalFrameID()
{
  return global_frame;
}

} /* namespace mesh_map */
```

Abbreviations

AI	Artificial intelligence
FLOAM	Fast lidar odometry and mapping
HEM	Half-edge mesh
LVR2	Las Vegas Reconstruction Toolkit 2.0
ML	Machine learning
PCL	Point Cloud Library
PGM	Portable gray map
RAS	Robotics and autonomous systems
RGB	Red-green-blue
RGB-D	Red-green-blue and depth
ROS	Robot Operating System
SDF	Signed Distance Field
SLAM	Simultaneous localization and mapping
UGV	Unmanned ground vehicle

REPORT DOCUMENTATION PAGE

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED	
		START DATE	END DATE
August 2023	Final	FY21	FY23

4. TITLE AND SUBTITLE

Unmanned Ground Vehicle (UGV) Path Planning in 2.5D and 3D

5a. CONTRACT NUMBER	5b. GRANT NUMBER	5c. PROGRAM ELEMENT
5d. PROJECT NUMBER	**5e. TASK NUMBER**	**5f. WORK UNIT NUMBER**

6. AUTHOR(S)

Osama Ennasr, Charles Ellison, Anton Netchaev, Ahmet Soylemezoglu, and Garry Glaspell

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

US Army Engineer Research and Development Center (ERDC)
Geospatial Research Laboratory (GRL)
7701 Telegraph Road
Alexandria, VA 22315-3864

See reverse

8. PERFORMING ORGANIZATION REPORT NUMBER

ERDC TR-23-12

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)	11. Sponsor/Monitor's Report Number
US Army Engineer Research and Development Center (ERDC) 3909 Halls Ferry Road Vicksburg, MS 39180-6199	ERDC	

12. DISTRIBUTION/AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

13. SUPPLEMENTARY NOTES

Funding provided by FLEX-4.

14. ABSTRACT

Herein, we explored path planning in 2.5D and 3D for unmanned ground vehicle (UGV) applications. For real-time 2.5D navigation, we investigated generating 2.5D occupancy grids using either elevation or traversability to determine path costs. Compared to elevation, traversability, which used a layered approach generated from surface normals, was more robust for the tested environments. A layered approached was also used for 3D path planning. While it was possible to use the 3D approach in real time, the time required to generate 3D meshes meant that the only way to effectively path plan was to use a preexisting point cloud environment. As a result, we explored generating 3D meshes from a variety of sources, including handheld sensors, UGVs, UAVs, and aerial lidar.

15. SUBJECT TERMS

Autonomous vehicles; Navigation; Path planning; Three-dimensional modeling; Vehicles, Military

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
a. REPORT	b. ABSTRACT	C. THIS PAGE		
Unclassified	Unclassified	Unclassified	SAR	85

19a. NAME OF RESPONSIBLE PERSON	19b. TELEPHONE NUMBER (include area code)

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) (concluded)

US Army Engineer Research and Development Center (ERDC)
Information Technology Laboratory (ITL)
3909 Halls Ferry Road
Vicksburg, MS 39180-6199

US Army Engineer Research and Development Center (ERDC)
Construction Engineering Research Laboratory (CERL)
2902 Newmark Drive
Champaign, IL 61822