A Universal Grid Map Library: Implementation and Use Case for Rough Terrain Navigation

Chapter · January 2016
DOI: 10.1007/978-3-319-26054-9_5

2 authors:

Péter Fankhauser
ETH Zurich
39 PUBLICATIONS **308** CITATIONS
SEE PROFILE

Marco Hutter
ETH Zurich
71 PUBLICATIONS **912** CITATIONS
SEE PROFILE

A Universal Grid Map Library: Implementation and Use Case for Rough Terrain Navigation

Péter Fankhauser* and Marco Hutter

Robotic Systems Lab, ETH Zurich, LEE J 201,
Leonhardstrasse 21, 8092 Zurich, Switzerland
{pfankhauser,mahutter}@ethz.ch
http://www.rsl.ethz.ch
*corresponding author

Abstract. In this research chapter, we present our work on a universal grid map library for use as mapping framework for mobile robotics. It is designed for a wide range of applications such as online surface reconstruction and terrain interpretation for rough terrain navigation. Our software features multi-layered maps, computationally efficient repositioning of the map boundaries, and compatibility with existing ROS map message types. Data storage is based on the linear algebra library Eigen, offering a wide range of data processing algorithms. This chapter outlines how to integrate the grid map library into the reader's own applications. We explain the concepts and provide code samples to discuss various features of the software. As a use case, we present an application of the library for online elevation mapping with a legged robot. The grid map library and the robot-centric elevation mapping framework are available open-source at http://github.com/ethz-asl/grid_map and http://github.com/ethz-asl/elevation_mapping.

Keywords: ROS, Grid map, Elevation mapping

1 Introduction

Mobile ground robots are traditionally designed to move on flat terrain and their mapping, planning, and control algorithms are typically developed for a two-dimensional abstraction of the environment. When it comes to navigation in rough terrain (e.g. with tracked vehicles or legged robots), the algorithms must be extended to take into account all three dimensions of the surrounding. The most popular approach is to build an elevation map of the environment, where each coordinate on the horizontal plane is associated with an elevation/height value. For simplicity, elevation maps are often stored and handled as grid maps, which can be thought of as a 2.5-dimensional representation, where each cell in the grid holds a height value.

In our recent work, we have developed a universal grid map library for use as a generic mapping framework for mobile robotics with the Robotic Operating System (ROS). The application is universal in the sense that our implementation is not restricted to any special type of input data or processing step. The library supports multiple data layers and is for example applicable to elevation, variance, color, surface normal, occupancy etc. The underlying data

storage is implemented as two-dimensional circular buffer. The circular buffer implementation allows for non-destructive and computationally efficient shifting of the map position. This is for example important in applications where the map is constantly repositioned as the robot moves through the environment (e.g. robot-centric mapping [1]). Our software facilitates the handling of map data by providing several helper functions. For example, iterator functions for rectangular, circular, and polygonal regions enable convenient and memory-safe access to sub-regions of the map. All grid map data is stored as datatypes from Eigen [2], a popular C++ linear algebra library. The user can apply the available Eigen algorithms directly to the map data, which provides versatile and efficient tools for data manipulation.

A popular ROS package that also works with a grid-based map representation is the *costmap_2d* [3, 4] package. It is part of the *2D navigation stack* [5] and used for two-dimensional robot navigation. Its function is to process range measurements and to build an occupancy grid of the environment. The occupancy is typically expressed as a status such as *Occupied*, *Free*, and *Unknown*. Internally, costmaps are stored as arrays of `unsigned char` with an integer value range of 0–255. While sufficient for processing costmaps, this data format can be limiting in more general applications. To overcome these deficiencies, the grid map library presented in this work stores maps as matrices of type `float`. This allows for more precision and flexibility when working with physical types such as height, variance, surface normal vectors etc. To ensure compatibility with the existing ROS ecosystem, the grid map library provides converters to transform grid maps to *OccupancyGrid* (used by the *2D navigation stack*), *GridCells*, and *PointCloud2* message types. Converting and publishing the map data as different message types also allows to make use of existing RViz visualization plugins. Another related package is the *OctoMap* library [6] and its associated ROS interface [7]. *OctoMap* represents a map as three-dimensional structure with occupied and free voxels. Structured as octree, *OctoMap* maps can be dynamically expanded and can contain regions with different resolutions. In comparison to a 2.5-dimensional grid representation, the data structure of *OctoMap* is well suited to represent full three-dimensional structures. This is often useful when working with extended maps with multiple floors and overhanging structures or robot arm motion planning tasks. However, accessing data in the octree entails additional computational cost since a search over the nodes of the tree has to be performed [6]. Instead, the representation of grid maps allows for direct value access and simplified data management in post-processing and data interpretation steps.

The grid map library has served as underlying framework for several applications. In [1], elevation maps are built to plan the motion of a legged robot through rough terrain (see Fig. 1a). Range measurements acquired from an onboard Kinect depth sensor and robot pose estimates are fused in a probabilistic representation of the environment. The mapping procedure is formulated from a robot-centric perspective to explicitly account for drift of the pose as the robot

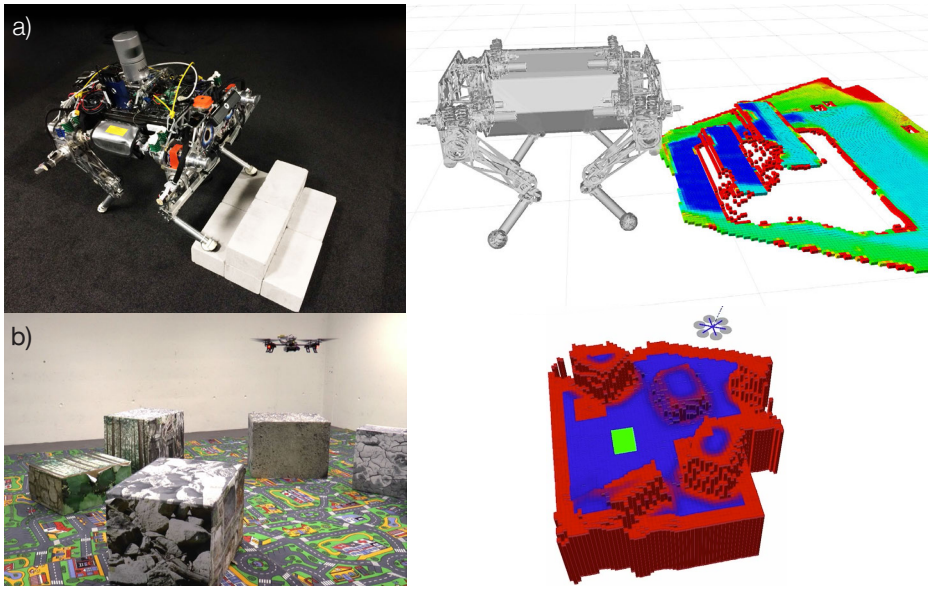

Fig. 1. The grid map library has been applied to various mapping tasks. a) An elevation map created from an onboard Kinect depth sensor allows the quadrupedal robot StarlETH [9] to navigate in rough terrain [1]. Colors represent the uncertainty of the height estimates, *red* corresponds to less certain and *blue* to more certain estimates. b) Autonomous landing of a multicopter is achieved by finding a safe landing spot [8]. The map is created from the depth estimation from an onboard IMU and monocular camera. Unsafe landing areas are marked in *r*ed, safe areas in *blue*, and the chosen landing spot in *green*.

moves.[1] In [8], the grid map library has been used in work for autonomous landing of a Micro Aerial Vehicle (MAV) (see Fig. 1b). An elevation map is generated from estimated depth data from an onboard Inertial Measurement Unit (IMU) and a monocular camera. The map is then used to automatically find a safe landing spot for the vehicle.[2]

In this chapter, we discuss the steps required to implement our software for various applications. In the remainder of this chapter, we cover the following topics:

- First, we demonstrate how to download the grid map library and give an overview of its components. Based on a simple example node, we present the basic steps required to integrate the library.
- Second, we describe the main functionalities of the library. We highlight the main concepts and demonstrate their usage with code samples from a tutorial node.
- Third, we discuss a use case of the software with an elevation mapping application presented in [1].

[1] A video demonstration is available at `http://youtu.be/I9eP8GrMyNQ`
[2] A video demonstration is available at `http://youtu.be/phaBKFwfcJ4`

2 Overview

2.1 Prerequisites & Installation

In the following, we assume a functioning installation of Ubuntu 14.04 LTS (Trusty Tahr) and ROS Indigo. Installation instructions for ROS Indigo on Ubuntu are given in [10]. Although we will present the procedures for these versions, the Grid Map packages have been also tested for ROS Jade and should work with future version with no or minor adaptations. Furthermore, we presume a catkin workspace has been setup as described in [11].

Except for ROS packages that are part of the standard installation (*cmake_modules*, *roscpp*, *sensor_msgs*, *nav_msgs* etc.), the grid map library depends only on the linear algebra library Eigen [2]. If not already available on your system, install Eigen with

```
$ sudo apt-get install libeigen3-dev
```

To use the grid map library, clone the associated packages into the `/src` folder of your catkin workspace with

```
$ git clone git@github.com:ethz-asl/grid_map.git
```

Finish the installation by building your catkin workspace with

```
$ catkin_make
```

To maximize performance, make sure to build in *Release* mode. You can specify the build type by setting

```
$ catkin_make -DCMAKE_BUILD_TYPE=Release
```

If desired, you can build and run the associated unit tests with

```
$ catkin_make run_tests_grid_map_core run_tests_grid_map
```

Note that our library makes use of C++11 features such as list initializations and range-based `for`-loops. Therefore, the CMake flag `-std=c++11` is added in the `CMakeLists.txt` files.

2.2 Software Components

The grid map library consists of several components:

grid_map_core implements the core algorithms of the grid map library. It provides the `GridMap` class and several helper classes. This package is implemented without ROS dependencies.

grid_map is the main package for ROS dependent projects using the grid map library. It provides the interfaces to convert grid map objects to several ROS message types.

grid_map_msgs holds the ROS message and service definitions for the *GridMap* message type.

grid_map_visualization contains a node written to visualize *GridMap* messages in RViz by converting them to standard ROS message types. The visualization types and parameters are fully user-configurable through ROS parameters.

grid_map_demos contains several nodes for demonstration purposes. The *simple_demo* node is a short example on how to use the grid map library. An extended demonstration of the library's functionalities is given in the *tutorial_demo* node. Finally, the *iterators_demo* and *image_to_gridmap_demo* nodes showcase the usage of the grid map iterators and the conversion of images to grid maps, respectively.

grid_map_filters builds on the ROS filters library [12] to implement a range of filters for grid map data. The filters provide a standardized API to define a chain of filters based on runtime parameters. This allows for great flexibility when writing software to process grid maps as a sequence of configurable filters.

2.3 A Simple Example

In the following, we describe a simple example on how to use the grid map library. Use this code to verify your installation of the grid map packages and to get you started with your own usage of the library. Locate the file `grid_map_demos/src/simple_demo_node.cpp` with the following content:

```
#include <ros/ros.h>
#include <grid_map/grid_map.hpp>
#include <grid_map_msgs/GridMap.h>
#include <cmath>

using namespace grid_map;

int main(int argc, char** argv)
{
  // Initialize node and publisher.
  ros::init(argc, argv, "grid_map_simple_demo");
  ros::NodeHandle nh("~");
  ros::Publisher publisher =
      nh.advertise<grid_map_msgs::GridMap>("grid_map", 1, true);

  // Create grid map.
  GridMap map({"elevation"});
  map.setFrameId("map");
  map.setGeometry(Length(1.2, 2.0), 0.03);
  ROS_INFO("Created map with size %f x %f m (%i x %i cells).",
    map.getLength().x(), map.getLength().y(),
    map.getSize()(0), map.getSize()(1));

  // Work with grid map in a loop.
  ros::Rate rate(30.0);
```

```
25   while (nh.ok()) {
26
27     // Add data to grid map.
28     ros::Time time = ros::Time::now();
29     for (GridMapIterator it(map); !it.isPastEnd(); ++it) {
30       Position position;
31       map.getPosition(*it, position);
32       map.at("elevation", *it) = -0.04 + 0.2 * std::sin(3.0 *
             time.toSec() + 5.0 * position.y()) * position.x();
33     }
34
35     // Publish grid map.
36     map.setTimestamp(time.toNSec());
37     grid_map_msgs::GridMap message;
38     GridMapRosConverter::toMessage(map, message);
39     publisher.publish(message);
40     ROS_INFO_THROTTLE(1.0, "Grid map (timestamp %f) published.",
           message.info.header.stamp.toSec());
41
42     // Wait for next cycle.
43     rate.sleep();
44   }
45   return 0;
46 }
```

In this program, we initialize a ROS node which creates a grid map, adds data, and publishes it. The code consists of several code blocks which we explain part by part.

```
10   // Initialize node and publisher.
11   ros::init(argc, argv, "grid_map_simple_demo");
12   ros::NodeHandle nh("~");
13   ros::Publisher publisher =
         nh.advertise<grid_map_msgs::GridMap>("grid_map", 1, true);
```

This part initializes a node with name *grid_map_simple_demo* (Line 11) and creates a private node handle (Line 12). A publisher of type `grid_map_msgs::GridMap` is created which advertises on the topic `grid_map` (Line 13).

```
15   // Create grid map.
16   GridMap map({"elevation"});
17   map.setFrameId("map");
18   map.setGeometry(Length(1.2, 2.0), 0.03);
19   ROS_INFO("Created map with size %f x %f m (%i x %i cells).",
20     map.getLength().x(), map.getLength().y(),
21     map.getSize()(0), map.getSize()(1));
```

We create a variable `map` of type `grid_map::GridMap` (we are setting `using namespace grid_map` on Line 6). A grid map can contain multiple map layers.

Here the grid map is constructed with one layer named *elevation*.[3] The frame id is specified and the size of the map is set to 1.2×2.0 m (side length along the x and y-axis of the `map` frame) with a resolution of 0.03 m/cell. Optionally, the position (of the center) of the map could be set with the third argument of the `setGeometry(...)` method. The print out shows information about the generated map:

```
[INFO ][..]: Created map with size 1.200000 x 2.010000 m (40 x 67 cells).
```

Notice that the requested map side length of 2.0 m has been changed to 2.01 m. This is done automatically in order to ensure consistency such that the map length is a multiple of the resolution (0.03 m/cell).

```
23  // Work with grid map in a loop.
24  ros::Rate rate(30.0);
25  while (nh.ok()) {
```

After having setup the node and the grid map, we add data to the map and publish it in a loop of 30 Hz.

```
27    // Add data to grid map.
28    ros::Time time = ros::Time::now();
29    for (GridMapIterator it(map); !it.isPastEnd(); ++it) {
30      Position position;
31      map.getPosition(*it, position);
32      map.at("elevation", *it) = -0.04 + 0.2 * std::sin(3.0 *
            time.toSec() + 5.0 * position.y()) * position.x();
33    }
```

Our goal here is to add data to the *elevation* layer of the map where the elevation for each cell is a function of the cell's position and the current time. The `GridMapIterator` allows to iterate through all cells of the grid map (Line 29). Using the `*`-operator on the iterator, the current cell index is retrieved. This is used to determine the position for each cell with help of the `getPosition(...)` method of the grid map (Line 31). The current time in seconds is stored in the variable `time`. Applying the temporary variables `position` and `time`, the elevation is computed and stored in the current cell of the *elevation* layer (Line 32).

```
35    // Publish grid map.
36    map.setTimestamp(time.toNSec());
37    grid_map_msgs::GridMap message;
38    GridMapRosConverter::toMessage(map, message);
39    publisher.publish(message);
40    ROS_INFO_THROTTLE(1.0, "Grid map (timestamp %f) published.",
          message.info.header.stamp.toSec());
```

We update the timestamp of the map and then use the `GridMapRosConverter` class to convert the grid map object (of type `grid_map::GridMap`) to a ROS grid

[3] For simplicity, we use the list initialization feature of C++11 on Line 16.

map message (of type `grid_map_msgs::GridMap`, Line 38). Finally, the message is broadcasted to ROS with help of the previously defined publisher (Line 39).

After building with `catkin_make`, make sure a *roscore* is active, and then run this example with

```
$ rosrun grid_map_demos simple_demo
```

This will run the node *simple_demo* and publish the generated grid maps under the topic `grid_map_simple_demo/grid_map`.

In the next step, we show the steps to visualize the grid map data. The *grid_map_visualization* package provides a simple tool to visualize a ROS grid map message in RViz in various forms. It makes use of existing RViz plugins by converting the grid map message to message formats such as *PointCloud2*, *OccupancyGrid*, *Marker* etc. We create a launch-file under `grid_map_demos/launch/simple_demo.launch` with the following content

```
1  <launch>
2    <!-- Launch the grid map simple demo node -->
3    <node pkg="grid_map_demos" type="simple_demo"
         name="grid_map_simple_demo" output="screen" />
4    <!-- Launch the grid map visualizer -->
5    <node pkg="grid_map_visualization" type="grid_map_visualization"
         name="grid_map_visualization" output="screen">
6      <rosparam command="load" file="$(find
           grid_map_demos)/config/simple_demo.yaml" />
7    </node>
8    <!-- Launch RViz with the demo configuration -->
9    <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
         grid_map_demos)/rviz/grid_map_demo.rviz" />
10 </launch>
```

This launches *simple_demo* (Line 3), *grid_map_visualization* (Line 5), and *RViz* (Line 9). The *grid_map_visualization* node is loaded with the parameters from `grid_map_demos/config/simple_demo.yaml` with following configuration

```
1  grid_map_topic: /grid_map_simple_demo/grid_map
2  grid_map_visualizations:
3    - name: elevation_points
4      type: point_cloud
5      params:
6       layer: elevation
7    - name: elevation_grid
8      type: occupancy_grid
9      params:
10      layer: elevation
11      data_min: 0.1
12      data_max: -0.18
```

This connects the *grid_map_visualization* with the *simple_demo* node via the `grid_map_topic` parameter (Line 1) and adds a *PointCloud2* (Line 3) and *Oc-*

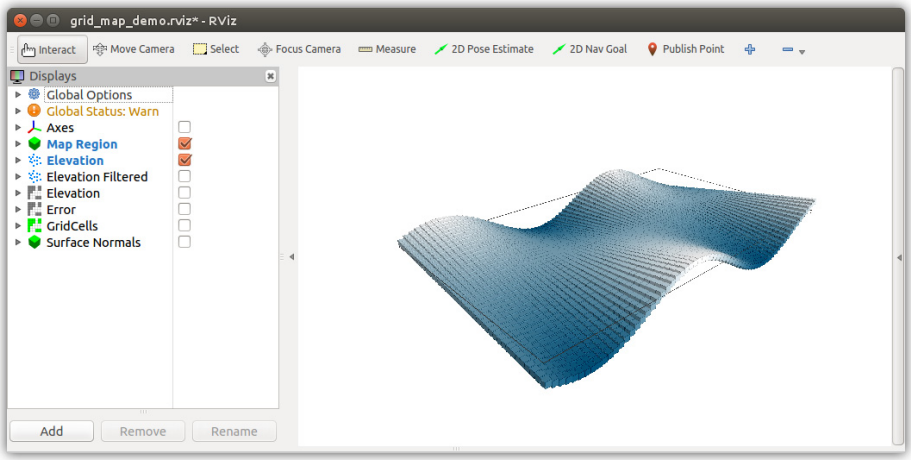

Fig. 2. Launching the `simple_demo.launch` file will run the *simple_demo* and *grid_map_visualization* node and open RViz. The generated grid map is visualized as point cloud.

cupancyGrid visualization (Line 7) for the *elevation* layer. Run the launch file with

```
$ roslaunch grid_map_demos simple_demo.launch
```

If everything is setup correctly, you should see the generated grid map in RViz as shown in Fig. 2.

3 Package Description

This section gives an overview of the functions of the grid map library. We describe the API and demonstrate its usage with example code from a tutorial file at `grid_map_demos/src/tutorial_demo_node.cpp`. This *tutorial_demo* extends the *simple_demo* by deteriorating the *elevation* layer with noise and outliers (holes in the map). An average filtering step is applied to reconstruct the original data and the remaining error is analyzed. You can run the *tutorial_demo* including visualization with

```
$ roslaunch grid_map_demos tutorial_demo.launch
```

3.1 Adding, Accessing, and Removing Layers

When working with mobile robotic mapping, we often compute additional information to interpret the data and to guide the navigation algorithms. The grid map library uses *layers* to store information for different types of data. Figure 3

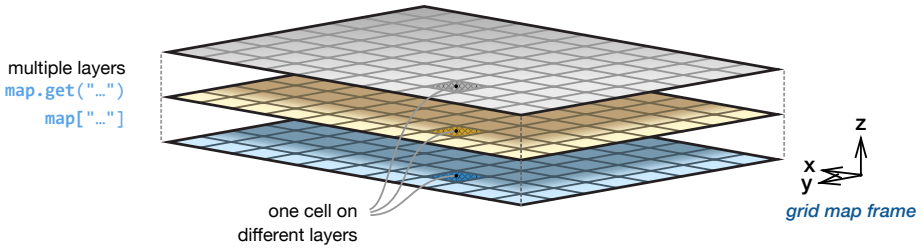

Fig. 3. The grid map library uses multilayered grid maps to store data for different types of information.

illustrates the multilayered grid map concept, where for each cell data is stored on the congruent layers.

The grid map class can be initialized empty or directly with a list of layers such as

```
18 GridMap map({"elevation", "normal_x", "normal_y", "normal_z"});
```

A new layer can be added to an existing map with

```
void add(const std::string& layer, const float value) ,
```

where the argument `value` determines the value with which the new layer is populated. Alternatively, data for a new layer can be added with

```
45 map.add("noise", Matrix::Random(map.getSize()(0), map.getSize()(1)));
```

In case the added layer already exists in the map, it is replaced with the new data. The availability of a layer in the map can be checked with the method `exists(...)`. Access to the layer data is given by the `get(...)` methods and its short form operator `[...]`:

```
46 map.add("elevation_noisy", map.get("elevation") + map["noise"]);
```

More details on the use of the addition operator + with grid maps is given in Section 3.7. A layer from the map can be removed with `erase(...)`.

3.2 Setting the Geometry and Position

For consistent representation of the data, we define the geometrical properties of a grid map as illustrated in Fig. 4. The map is specified in a *grid map frame*, to which the map is aligned to. The map's position is defined as the center of the rectangular map in the specified frame.

The basic geometry of the map such as side lengths, resolution, and position (see Fig. 4) is set through the `setGeometry(...)` method. When working with different coordinate frames, it is important to specify the grid map's frame:

```
19 map.setFrameId("map");
20 map.setGeometry(Length(1.2, 2.0), 0.03, Position(0.0, -0.1));
```

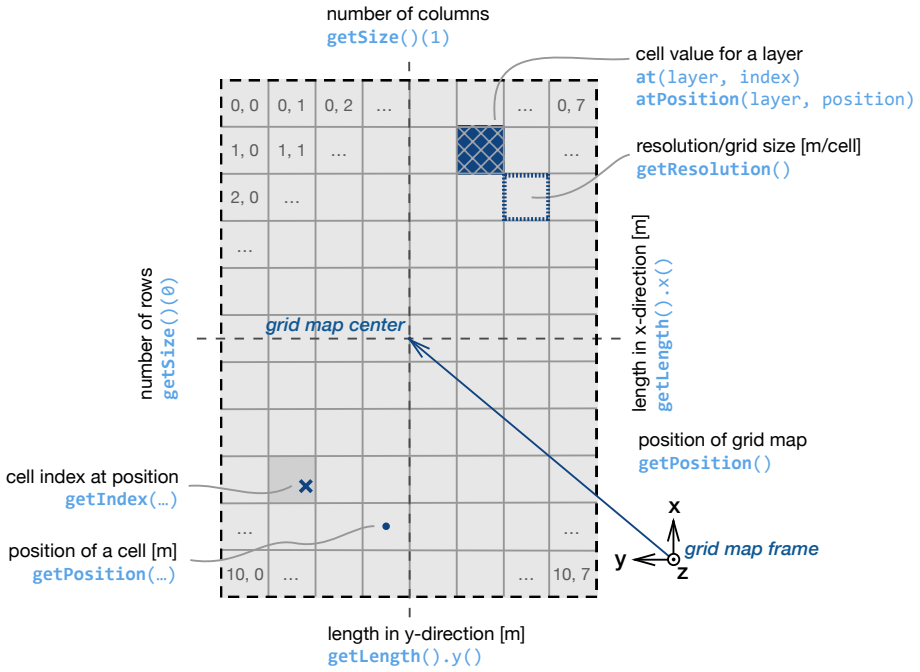

Fig. 4. The grid map is defined to be aligned with its *grid map frame*. The position of the map is specified as the center of the map relative the its frame.

It is important to set the geometry before adding any data as all cell data is cleared when the grid map is resized.

3.3 Accessing Cells

As shown in Fig. 4, individual cells of grid map layer can be accessed in two ways, either by direct index access with

```
float& at(const std::string& layer, const grid_map::Index& index)
```

or by referring to a position of the underlying cell with

```
float& atPosition(const std::string& layer,
                  const grid_map::Position& position) .
```

Conversions between a position and the corresponding cell index and vice versa are given with the `getIndex(...)` and `getPosition(...)` methods. These conversions handle the underlying algorithmics involved with the circular buffer storage and are the recommended way of handling with cell indices/positions. An example of usage is:

```
51 if (map.isInside(randomPosition))
52   map.atPosition("elevation_noisy", randomPosition) = ...;
```

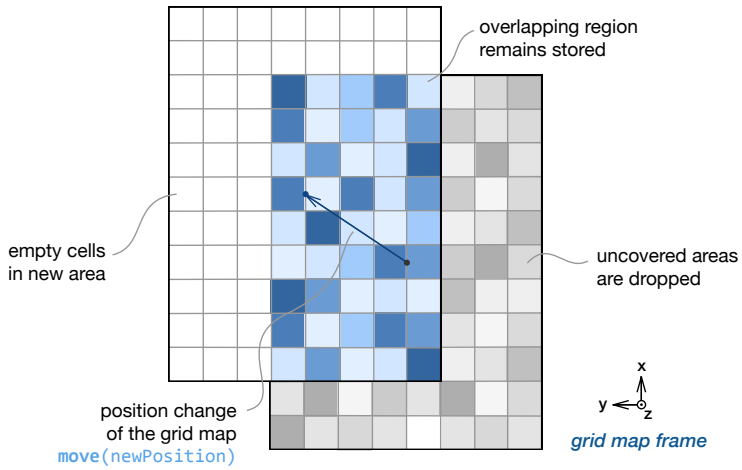

Fig. 5. The grid map library offers with the `move(...)` method a computationally efficient and non-destructive method to change the position of grid map. The underlying implementation of the grid map as two-dimensional circular buffer allows to move the map without allocating new memory.

Here, the helper method `isInside(...)` is used to ensure that the requested position is within the bounds of the map.

3.4 Moving the Map

The grid map library offers a computationally efficient and non-destructive method to change the position of grid map with

```
void move(const grid_map::Position& position) .
```

The argument `position` specifies the new absolute position of the grid map w.r.t. the grid map frame (see Section 3.2). This method relocates the grid map such that the grid aligns between the previous and new position (Fig. 5). Data in the overlapping region before and after the position change remains stored. Data that falls outside of the map at its new position is discarded. Cells that cover previously unknown regions are emptied (set to `nan`). The data storage is implemented as two-dimensional circular buffer, meaning that data in the overlapping region of the map is not moved in memory. This minimizes the amount of data that needs to be changed when moving the map.

3.5 Basic Layers

For certain applications it is useful to define a set of *basic layers* with `setBasicLayers(...)`. These layers are considered when checking the validity of a grid map cell. If the cell has a valid (finite) value in *all* basic layers, the cell is declared valid by the method

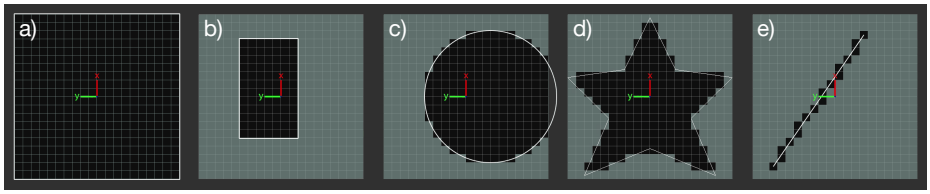

Fig. 6. Iterators are a convenient way to work with cells belonging to certain geometrical regions. a) The `GridMapIterator` is used to iterate through all cells of the grid map. b) The `SubmapIterator` accesses cells belonging to a rectangular region of the map. c) The `CircleIterator` iterates through a circular region specified by a center and radius. d) The `PolygonIterator` is defined by n vertices and covers all inlying cells. e) The `LineIterator` uses Bresenham's line algorithm [13] to iterate over a line defined by a start and end point. Example code on how to use the grid map iterators is given in the *iterators_demo* node of the *grid_map_demos* package.

```
bool isValid(const grid_map::Index& index) const .
```

Similarly, the method `clearBasic()` will clear only the basic layers and is therefore more efficient than `clearAll()` which clears all layers. By default, the list of basic layers is empty.

3.6 Iterating Over Cells

Often, one wants to iterate through a number of cells within a geometric shape within the grid map, for example to check the cells which are covered by the footprint of a robot. The grid map library offers iterators for various regions such as rectangular submaps, circular and polygonal regions, and lines (see Fig. 6). Iteration can be achieved with a `for`-loop of the form

```
32 for (GridMapIterator it(map); !isPastEnd(); ++it) { ... }
```

The dereference operator `*` is used to retrieve the current cell index of the iterator such as in following examples:

```
35 map.at("elevation", *it) = ...;
```

```
67 Position currentPosition;
68 map.getPosition(*it, currentPosition);
```

```
76 if (!map.isValid(*circleIt, "elevation_noisy")) continue;
```

These iterators can be used at the border of the maps without concern as they internally ensure to access only cells within the map boundary.

3.7 Using Eigen Functions

Because each layer of the grid map is stored internally as an Eigen matrix, all functions provided by the Eigen library [2] can directly be applied. This is illustrated with following examples:

```
46 map.add("elevation_noisy", map.get("elevation") + map["noise"]);
```

Here, the cell values of the two layers *elevation* and *noise* are summed up arithmetically and the result is stored in the new layer *elevation_noisy*. Alternatively, the noise could have been added to the *elevation* layer directly with the `+=` operator:

```
map.get("elevation") += 0.015 * Matrix::Random(map.getSize()(0),
    map.getSize()(1));
```

A more advanced example demonstrates the simplicity that is provided by making use of the various Eigen functions:

```
91 map.add("error", (map.get("elevation_filtered") -
       map.get("elevation")).cwiseAbs());
92 unsigned int nCells = map.getSize().prod();
93 double rmse = sqrt(map["error"].array().pow(2).sum() / nCells);
```

Here, the absolute error between two layers is computed (Line 91) and used to calculate the Root Mean Squared Error (RMSE) (Line 93) as

$$e_{\text{RMS}} = \sqrt{\frac{\sum_{i=1}^{n} (f_i - c_i)^2}{n}} \quad , \tag{1}$$

where c_i denotes the value of cell i of the original *elevation* layer, f_i the cell values in the filtered *elevation_filtered* layer, and n the number of cells of the grid map.

3.8 Creating Submaps

A copy of a part from the grid map can be generated with the method

```
GridMap getSubmap(const grid_map::Position& position,
                  const grid_map::Length& length, bool& isSuccess) .
```

The return value is of type `GridMap` and all described methods apply also to the retrieved submap. When retrieving a submap, a deep copy of the data is made and changing data in the original or the submap will not influence its counterpart. Working with submaps is often useful to minimize computational load and data transfer if only parts of the data is of interest for further processing. To access or manipulate the original data in a submap region without copying, refer to the `SubmapIterator` described in Section 3.6.

3.9 Converting from and to ROS Data Types

The grid map class can be converted from and to a ROS *GridMap* message with the methods `fromMessage(...)` and `toMessage(...)` of the `GridMapRosConverter` class:

```
97 grid_map_msgs::GridMap message;
98 GridMapRosConverter::toMessage(map, message);
```

Additionally, a grid map can be saved in and loaded from a ROS bag file with the `saveToBag(...)` and `loadFromBag(...)` methods. For compatibility and visualization purposes, a grid map can be converted also to *PointCloud2*, *OccupancyGrid*, and *GridCells* message types. For example, using the method

```
static void toPointCloud(const grid_map::GridMap& gridMap,
    const std::vector<std::string>& layers,
    const std::string& pointLayer,
    sensor_msgs::PointCloud2& pointCloud) ,
```

a grid map is converted to a *PointCloud2* message. Here, the second argument `layers` determines which layers are converted to the point cloud. The layer in the third argument `pointLayer` specifies which of these layers is used as z-value of the points (x and y are given by the cell positions). All other layers are added as additional fields in the point cloud message. These additional fields can be used in RViz to determine the color of the points. This is a convenient way to visualize characteristic of the map such as uncertainty, terrain quality, material etc.

Figure 7 shows the different ROS messages types that have been generated with the *grid_map_visualization* node for the *tutorial_demo*. The setup of the *grid_map_visualization* for this example is stored in the `grid_map_demos/config/tutorial_demo.yaml` parameters file.

3.10 Adding Data from Images

The grid map library allows to load data from images. In a first step, the geometry of the grid map can be initialized to the size of the image with the method `GridMapRosConverter::initializeFromImage(...)`. Two different methods are provided to add data from an image as layer in a grid map. To add data as scalar values (typically from a grayscale image), the method `addLayerFromImage()` can be used. This requires to specify the lower and upper values for the corresponding black and white pixels of the image. Figure 8a shows an example, where an image editing software was used to draw a terrain. To add data from an image as color information, the `addColorLayerFromImage()` is available. This can be used for example to add color information from a camera to the grid map as shown in Fig. 8b.

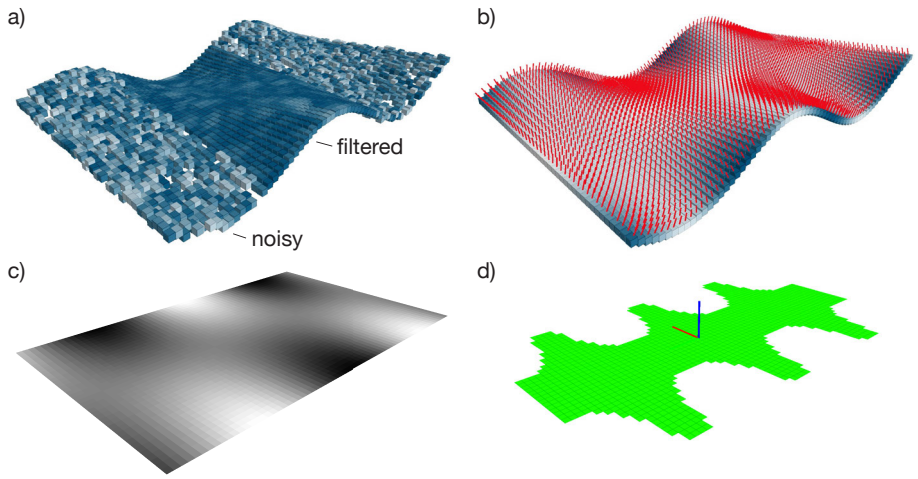

Fig. 7. Grid maps can be converted to several ROS message types for visualization in RViz. a) Visualization as *PointCloud2* message type showing the results of the *tutorial_demo* filtering step. Colors represent the absolute error between the filtered result and the original data before noise corruption. b) Vectors (here surface normals) can be visualized with help of the *Marker* message type. c) Representation of a single layer (here the *elevation* layer) of the grid map as *OccupancyGrid*. d) The same layer is shown as *GridCells* type where cells within a threshold are visualized.

4 Use Case: Elevation Mapping

Based on the grid map library, we have developed a ROS package for local terrain mapping with an autonomous robot. This section introduces the technical background of the mapping process, discusses the implementation and usage of the package, and presents results from real world experiments.

4.1 Background

Enabling robots to navigate in previously unseen, rough terrain, requires to reconstruct the terrain as the robot moves through the environment. In this work, we assume an existing robot pose estimation and an onboard range measurement sensor. For many autonomous robots, the pose estimation is prone to drift. In case of a drifting pose estimate, stitching fresh scans with previous data leads to inconsistent maps. Addressing this issue, we formulate a probabilistic elevation mapping process from a robot-centric perspective. In this approach, the generated map is an estimate of the shape of the terrain from a local perspective. We illustrate the robot-centric elevation mapping procedure in Fig. 9. Our mapping method consists of three main steps:

1. **Measurement Update:** New measurements from the range sensor are fused with existing data in the map. By modeling the three-dimensional noise distribution of the range sensor, we can propagate the depth measurement

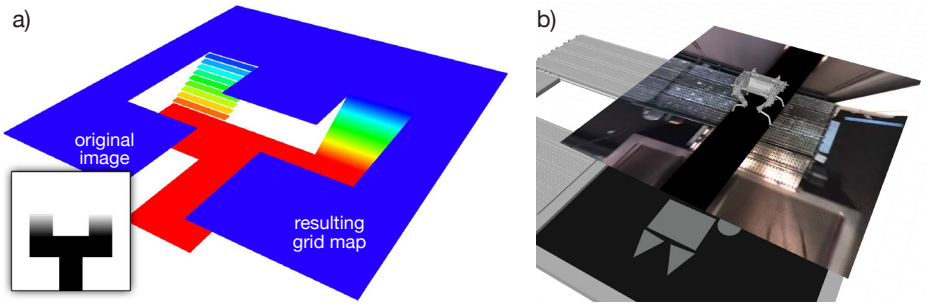

Fig. 8. Image data can be used to populate a grid map. a) A terrain is drawn in an image editing software and used to generate a height field grid map layer. b) A color layer in the grid map is updated by the projected video stream of two wide-angle cameras in the front and back of the robot. Example code on how to add data from images to a grid map is given in the *image_to_gridmap_demo* node of the *grid_map_demos* package.

errors to the corresponding elevation uncertainty. By means of a Kalman filter, new measurements are fused with existing data in the map.

2. **Map Update:** For a robot-centric formulation, the elevation map needs to be updated when the robot moves. We propagate changes of pose covariance matrix to the spatial uncertainty of the cells in the grid map. This reflects errors of the pose estimate (such as drift) in the elevation map. At this step, the uncertainty for each cell is treated separately to minimize computational load.
3. **Map Fusion:** When map data is required for further processing in the planning step, an estimation of the cell heights is computed. This requires to infer the elevation and variance for each cell from its surrounding cells.

Applying our mapping method, the terrain is reconstructed under consideration of the range sensor errors and the robots pose uncertainty. Each cell in the grid map holds information about the height estimate and a corresponding variance. The region ahead of the robot has typically the highest precision as it is constantly updated with new measurements from the forward-looking distance sensor. Regions which are out of the sensor's field of view (below or behind the robot) have decreased certainty due to drift of the robot's relative pose estimation. Applying a probabilistic approach, motion/trajectory planning algorithms can take advantage of the available certainty estimate of the map data.

4.2 Implementation

We have implemented the robot-centric elevation mapping process as a ROS package.[4] The node subscribes to *PointCloud2* depth measurements and a robot pose estimate of type *PoseWithCovarianceStamped*. For the depth measurements, we provide *sensor processors* which generate the measurement variances

[4] Available at: `http://github.com/ethz-asl/elevation_mapping`

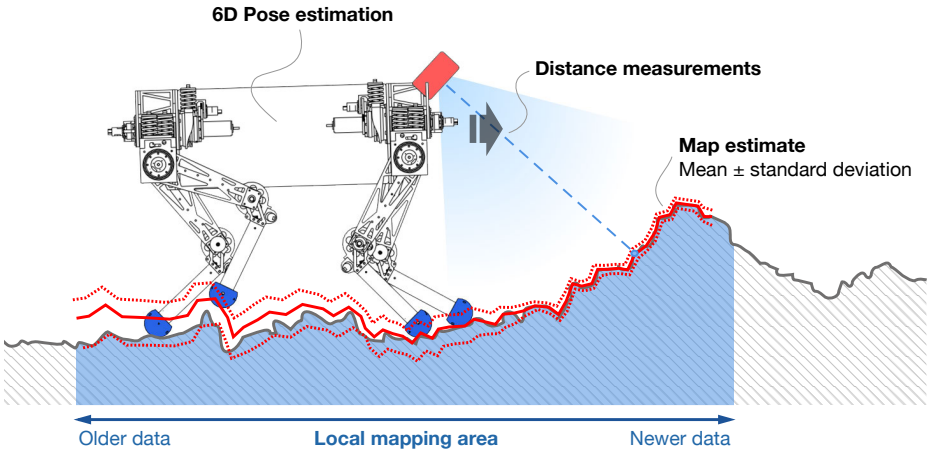

Fig. 9. The elevation mapping mapping procedure is formulated from a robot-centric perspective.

based on a noise model of the device. We support several range measurement devices such as Kinect-type sensors [14], laser range sensors, and stereo cameras.

Building a local map around the robot, the map's position is updated continuously to cover the area around the robot. This can be achieved with minimal computational overhead by using the `move(...)` method provided by the grid map library (see Section 3.4).

The elevation mapping node publishes the generated map as *GridMap* message (containing layers for elevation, variance etc.) on two topics. The unfused elevation map (considering only processing steps 1 and 2 from Section 4.1) is continuously published under the topic `elevation_map_raw` and can be used to monitor the mapping process. The fused elevation map (the result of processing step 3 Section 4.1) is computed and published under `elevation_map` if the ROS service `trigger_fusion` is called. Partial (fused) maps can be requested over the `get_submap` service call. The node implements the map update and fusion steps as multi-threaded processes to enable continuous measurement updates.

Using the *grid_map_visualization* node, the elevation maps can be visualized in RViz in several ways. It is convenient to show the map as point cloud and color the points with data from different layers. For example, in Fig. 1 the points are colored with the variance data of the map and in Fig. 10 with colors from an RGB camera. Using a separate node for visualization is a good way to split the computational load to different systems and limit the data transfer from the robot to an operator computer.

4.3 Results

We have implemented the elevation mapping software on the quadrupedal robot StarlETH [9] (see Fig. 10). A downward-facing PrimeSense Carmine 1.09 structured light sensor is attached in the front of the robot as distance sensor. The

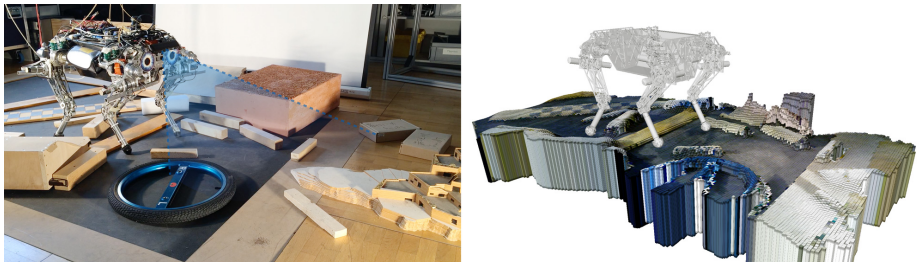

Fig. 10. The quadrupedal robot StarlETH [9] is walking over obstacles by using the *elevation mapping* node to map the environment around the its current position. A Kinect-type range sensor is used in combination with the robot pose estimation to generate a probabilistic reconstruction of the terrain.

state estimation is based on stochastic fusion of kinematic and inertial measurements [15], where the position and the yaw-angle are in general unobservable and are therefore subject to drift. We generate an elevation map with a size of 2.5×2.5 m with a resolution of 1 cm/cell and update it with range measurements and pose estimates at 20 Hz on an Intel Core i3, 2.60 GHz onboard computer. The mapping software can process the data at sufficient speed such that the map can be used for real-time navigation. As a result of the high resolution and update rate of the range sensor, the map holds dense information and contains only very few cells with no height information. This is an important prerequisite for collision checking and foothold selection algorithms. Thanks to the probabilistic fusion of the data, the real terrain structure is accurately captured and outliers and spurious data is suppress effectively.

In another setup, we have also used the elevation map data to find a traversable path through the environment. Figure 11 depicts the robot planning a collision free path in the map of a corridor. A rotating Hokuyo laser range sensor in the front of the robot is used the generate the elevation map with a size of 6×6 m with a resolution of 3 cm/cell. In this work, the elevation information is processed to estimate the traversability of the terrain. Multiple custom grid map filters (see Section 2.2) are used to interpret the data based on the slope, step height, and roughness of the terrain. An RRT-based [16] planner finds a traversable path to the goal pose for the horizontal translation x and y and yaw-rotation ψ. At each expansion of the search tree, the traversability of all the cells contained in the footprint of the robot is checked with help of the `PolygonIterator` (see Section 3.6).

5 Summary & Conclusion

In this chapter, we introduced a grid map library developed for mapping applications with mobile robots. Based on a simple example, the first steps to integrate the library in existing or new frameworks were presented. We highlighted some of the features of the software and illustrated their usage with code samples from

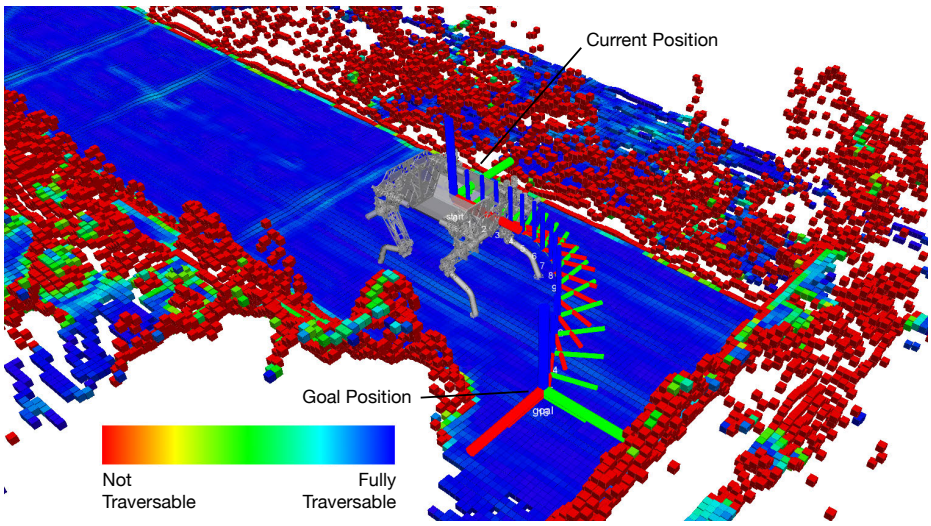

Fig. 11. The traversability of the terrain is judged based on the acquired elevation map. The traversability estimation takes factors such as slope, step size, and roughness of the terrain into consideration. A collision free path for the pose of the robot is found using an RRT-based [16] planner.

a tutorial example. Finally, an application of the grid map library for elevation mapping was shown, whereby the background, usage, and results were discussed.

In our current work, we use the grid map library (in combination with the elevation mapping package) in applications such as multi-robot mapping, foothold quality evaluation, collision checking for motion planning, and terrain property prediction through color information. We hope our software will be useful to other researcher working on rough terrain mapping and navigation with mobile robots.

Authors' Biographies

Péter Fankhauser is a Ph.D. student at the Robotic Systems Lab (RSL) at ETH Zurich. His current work focuses on mapping and motion planning for legged robots in rough terrain. Péter received his B.Sc. degree in mechanical engineering and his M.Sc. degree in Robotics, Systems and Control from ETH Zurich in 2010 and 2012.

Marco Hutter is assistant professor for Robotic Systems at ETH Zurich and Branco Weiss Fellow. He received his M.Sc. in mechanical engineering (2009) and his doctoral degree in robotics (2013) from ETH Zurich. In his work, Marco focuses on design, actuation, and control of legged robotic systems that can work in challenging environments.

Acknowledgments. This work was supported in part by the Swiss National Science Foundation (SNF) through project 200021_149427 / 1 and the National Centre of Competence in Research Robotics.

Bibliography

[1] P. Fankhauser, M. Bloesch, C. Gehring, M. Hutter, and R. Siegwart. Robot-Centric Elevation Mapping with Uncertainty Estimates. In *International Conference on Climbing and Walking Robots (CLAWAR)*, 2014.

[2] G. Guennebaud and B. Jacob. Eigen 3. `http://eigen.tuxfamily.org`, August, 2015.

[3] E. Marder-Eppstein, D. V. Lu, and D. Hershberger. costmap_2d. `http://wiki.ros.org/costmap_2d`, August, 2015.

[4] D. V. Lu, D. Hershberger, and W. D. Smart. Layered Costmaps for Context-Sensitive Navigation. In *IEEE International Conference on Intelligent Robots and Systems (IROS)*, 2014.

[5] D. V. Lu and M. Ferguson. navigation. `http://wiki.ros.org/navigation`, August, 2015.

[6] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard. OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees. *Autonomous Robots*, 34(3):189–206, 2013.

[7] K. M. Wurm and A. Hornung. octomap. `http://wiki.ros.org/octomap`, August, 2015.

[8] C. Forster, M. Faessler, F. Fontana, M. Werlbeger, and D. Scaramuzza. Continuous On-Board Monocular-Vision–based Elevation Mapping Applied to Autonomous Landing of Micro Aerial Vehicles. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2015.

[9] M. Hutter, C. Gehring, M. Hoepflinger, M. Bloesch, and R. Siegwart. Towards combining Speed, Efficiency, Versatility and Robustness in an Autonomous Quadruped. *IEEE Transactions on Robotics*, 30(6):1427–1440, 2014.

[10] ROS.org. Ubuntu install of ROS Indigo. `http://wiki.ros.org/indigo/Installation/Ubuntu`, August, 2015.

[11] ROS.org. Installing and Configuring Your ROS Environment. `http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment`, August, 2015.

[12] ROS.org. filters. `http://wiki.ros.org/filters`, August, 2015.

[13] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.

[14] P. Fankhauser, M. Bloesch, D. Rodriguez, R. Kaestner, M. Hutter, and R. Siegwart. Kinect v2 for Mobile Robot Navigation: Evaluation and Modeling. In *International Conference on Advanced Robotics (ICAR)*, 2015.

[15] M. Bloesch, C. Gehring, P. Fankhauser, M. Hutter, M. A. Hoepflinger, and R. Siegwart. State Estimation for Legged Robots on Unstable and Slippery Terrain. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2013.

[16] LaValle S. M. Rapidly-Exploring Random Trees: A New Tool for Path Planning. 1998.