2018

APPLICATIONS OF ROBOT OPERATING SYSTEM (ROS) TO MOBILE MICROGRID FORMATION OUTDOORS

John Naglak
Michigan Technological University, jenaglak@mtu.edu

APPLICATIONS OF ROBOT OPERATING SYSTEM (ROS) TO MOBILE MICROGRID FORMATION OUTDOORS

By

John E. Naglak

A THESIS

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

In Mechanical Engineering

MICHIGAN TECHNOLOGICAL UNIVERSITY

2018

This thesis has been approved in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE in Mechanical Engineering.

Department of Mechanical Engineering - Engineering Mechanics

Thesis Advisor: *Dr. Nina Mahmoudian*

Committee Member: *Dr. Mo Rastgaar*

Committee Member: *Dr. Wayne Weaver*

Department Chair: *Dr. William W. Predebon*

Dedication

He who finds a wife finds a good thing and obtains favor from the Lord.

Proverbs 18:22

Contents

List of Figures

List of Abbreviations

AUV	Autonomous Underwater Vehicle
COTS	Commercial-Off-The-Shelf
DoD	Department of Defense
FEMA	Federal Emergency Management Agency
FOB	Forward Operating Base
FOV	Field of View
GPS	Global Positioning System
GPU	Graphics Processing Unit
IMU	Inertial Measurement Unit
LiDAR	Light Detection And Ranging
NASLab	Non-Linear and Autonomous Systems Laboratory
PCI	Peripheral Component Interconnect
PCL	Point Cloud Library
ROS	Robot Operating System
SDK	Software Development Kit
TIB	Timed Elastic Band
UAV	Un-manned Arial Vehicle
UGV	Unmanned Ground Vehicle

USB	Universal Serial Bus
USV	Un-manned Surface Vehicle (boat)
UTM	Universal Transverse Mercator

Abstract

Application of mobile robots to microgrid formation has value for disaster response and service of forward operating bases. This thesis describes the development, testing and demonstration of broad effort across multiple disciplines to enable outdoor positioning and connection of mobile microgrids for the first time. This work includes an outdoor waypoint controller for a UGV agent, specifically the Clearpath Husky. It details sensor fusion of 2D LiDAR and stereo vision, and fusion of odometry sources using an Extended Kalman Filter. Development of these software tools entails integration of many of the packages available through the Robot Operating System (ROS), with control code for this application primarily written in Python. Hardware improvements were necessary to support these developments, including the integration of a GPU on the Husky UGV, and design and installation of a active electrical cable delivery system. Results begin with representation of the capabilities of each of these contributions. Finally, outdoor demonstration results are presented and a code appendix is included.

Chapter 1

Introduction

In this section we introduce the motivation for this work, namely mobile microgrids, and explore prior work related to applications of UGV's in outdoor and unstructured environments. We also present a background and rationale for depending on ROS for many of the algorithms and software dependencies for autonomous UGV navigation and planning. In context of the motivation and prior efforts by NASLab, the contribution of the work in this thesis is a a waypoint navigation controller with multiple perception inputs suitable for outdoor UGV navigation.

1.1 Background on Mobile Microgrids

As power grid infrastructure ages and deteriorates, and the challenges of integrating periodic electrical supply from renewables increases in proportion to their adoption, research into the implementation and control of microgrids has become imperative. Microgrids are localized networks of power sources, loads, grid storage, and control. They have applications for critical infrastructure during disasters, to serve small or rural populations out of reach of the power grid, as a "leap frog" technology in developing countries, the in the military. They will provide a more efficient and robust energy future as legacy power grid infrastructure is replaced and upgraded.

Military applications are especially dependent on microgrid research. Supply routes for diesel fuel are a major hazard for troops operating in hostile regions, Figure 1.1. FOB are distributed throughout a large hostile area, decentralized from protected supply infrastructure. They also can be established and moved at a more frequent interval as engagement changes. Microgrids that are robust to periodicity in supply inherent in renewable power sources are an ideal solution which reduces the current dependence on diesel supply lines. Consider that an autonomous mobile microgrid can establish power grid infrastructure before a large cohort of vulnerable personnel arrive at the location of the FOB. Early personnel on location can focus on security and mission specific tasks, rather than working to install the power grid infrastructure.

Figure 1.1: Fuel supply chains are a liability for the diesel generators which are an important power source on FOB's. DoD: https://www.defense.gov/observe/photo-gallery/igphoto/2001125374/

Be it the effects of natural disasters, such as the failure of the Fukashima nuclear power plant in 2011 [1], effects from cyber-attacks, portended by the Ukraine hack in 2015 [2], or failure caused by growth in demand outpacing growth in supply, causing power loss to over 600 million people in India in 2012 [3], the message is clear: development of emergency power supply and distribution is urgent. These infrastructure must be able to be deployed quickly to the affected area, reconfigured to address diverse applications, and robust to the aftermath of disasters.

A particularly heinous disaster that occurred recently was the destruction of Puerto Rico by Hurricane Maria in 2017. Although Puerto Rico is a U.S. territory, response and recovery was extremely slow, especially to the power and communications infrastructure on the mountainous island. This is an example where implementation

Figure 1.2: Devestation in Puerto Rico from Hurricane Maria in 2017 included most of the communication infrastructure on the island. FEMA/Richard Cardona https://www.fema.gov/media-library/assets/images/151854#details

of autonomous mobile microgrids would have been well suited, as personnel were in short supply and humanitarian resources were already overtaxed.

1.2 Relevant and Prior Work

One platform that lends itself to disaster recovery is the rugged-ized Unmanned Ground Vehicle (UGV). The agents have enhanced modular payload capacity and are often marketed as a modular base unit which can support sensing, communications hardware, power generation, and storage capacity. Current work in this field

includes autonomous survey and communication by UGV of areas affected by disasters [4], multi-agent human-denied hazard response [5], and modular, transportable, microgrid development [6], but little work which combines autonomous multi-agent robotics with modular microgrid infrastructure. The elusiveness of a general solution to the autonomous manipulation and coupling problem is substantiated by the large body of current work on its different aspects, including UAV-mounted manipulators [7], [8], varietal-specific agricultural harvest robots [9], and in multi-agent human-denied hazard response [5]. Compounding the inherent challenges of the coupling problem is localization uncertainty in outdoor and unstructured environments.

Consider a simple mobile microgrid, consisting of 4 UGV's comprising a renewable source, a control bus, and cabling agents. Certain constraints exist on their configuration, for example, orientation of a PV Array to the incidence angle of the sun, the location of fixed loads which must be serviced, or the terrain or hazards in the region. Figure 1.3 depicts this mission, where the UGV agents are delivered in some initial configuration and must navigate to waypoint goals within the operating region before performing their mission-specific tasks.

Hardware which satisfies these requirements has been developed through efforts at Michigan Tech. Power grids support power transmission over distance, power distribution to different loads from multiple sources, and voltage transformation and inversion/rectification to support different types of load devices [10]. To support

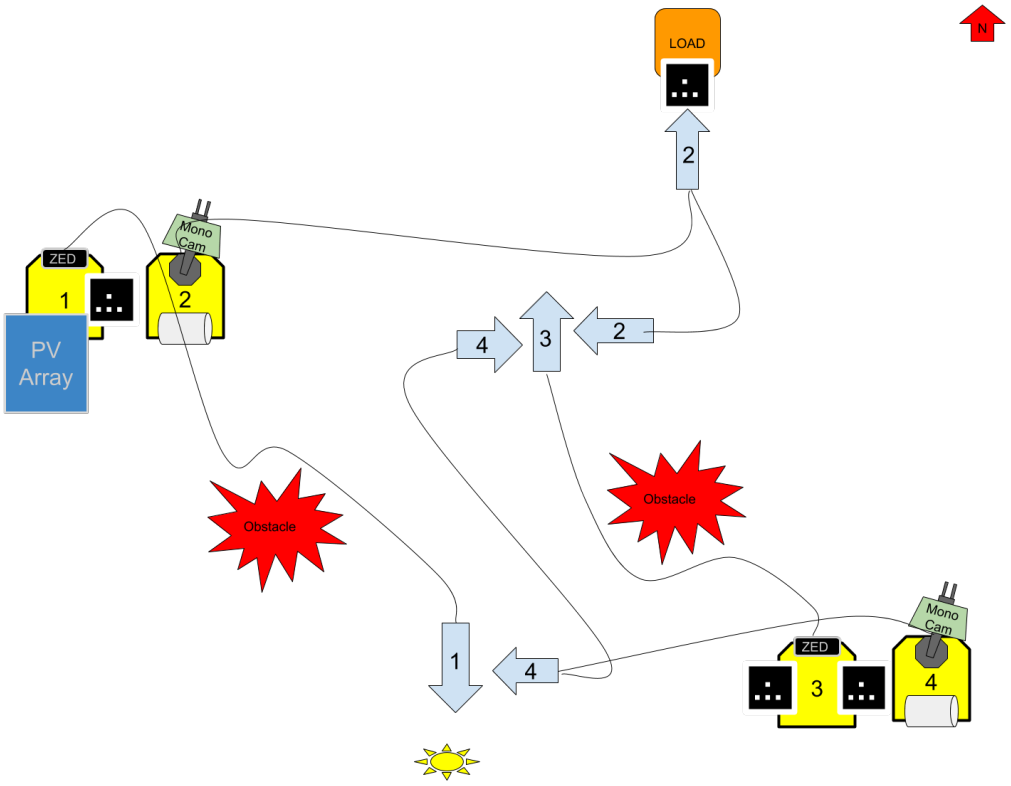

Figure 1.3: Autonomous UGV agents can form mobile microgrids. This example includes 4 agents with different roles.

the development of this hardware, a 1kW bus system was demonstrated on lab-scale robots [11]. Further development of the cabling and electrical connection mechanisms was implemented in [12], as well as a hierarchy for source, bus, and cabling agent co-operative roles. In [13], a framework for motivating the agent configuration based on power grid optimization rules was proposed. Recently, these efforts were extended to a microgrid consisting of 4 rugged-ized UGV agents, including a source robot, bus robot, and two cabling agents for loads [14], with improvements to the cabling systems, the electrical connectors, and docking controller. Robotic demonstrations developed by the NASLab at Michigan Tech have formerly been performed indoors

using a motion capture camera system for localization and ground truth.

1.3 Statement of Contribution

The work presented in this thesis is the integration of robotic systems and control design for outdoor operation of mobile microgrids. This work is applied at the intersection of many domains. Control software for outdoor waypoint navigation using the ROS Navigation stack allows microgrid formation outside the lab environment. Additionally, enhanced perception sensors have been added to the platform with filter and segmentation software which identifies unstructured obstacles common in the outdoor environment. Ongoing hardware improvements to the UGV platform are necessary as new algorithms unlock additional capability. In this work a new powered feed system for the electrical cable is presented and its controller is described. Outdoor navigation results are provided, as well as an appendix of ROS configuration scripts and control code written in python.

Chapter 2

Software Implementation

In this chapter, the details of that software implementation will be fully explored, starting with the ROS Navigation package and then going deeper into about packages for perception, hardware drivers, and how we implement Kalman filtering for position estimate. Figure 2.1, outlines the overall control architecture of a microgrid agent. This thesis focuses on the inputs and behavior of the waypoint navigation controller. While much of the "code" implemented in this waypoint controller is available from the ROS repositories, its particular configuration and tuning for the particular UGV, as well as the mission behavior, is not deterministic and requires experience on the part of the developer. An additional component that will be addressed at length is the integration of the ZED camera, a relatively recent entrant to the perception market at the time of the work, and especially our processing stack, which creates

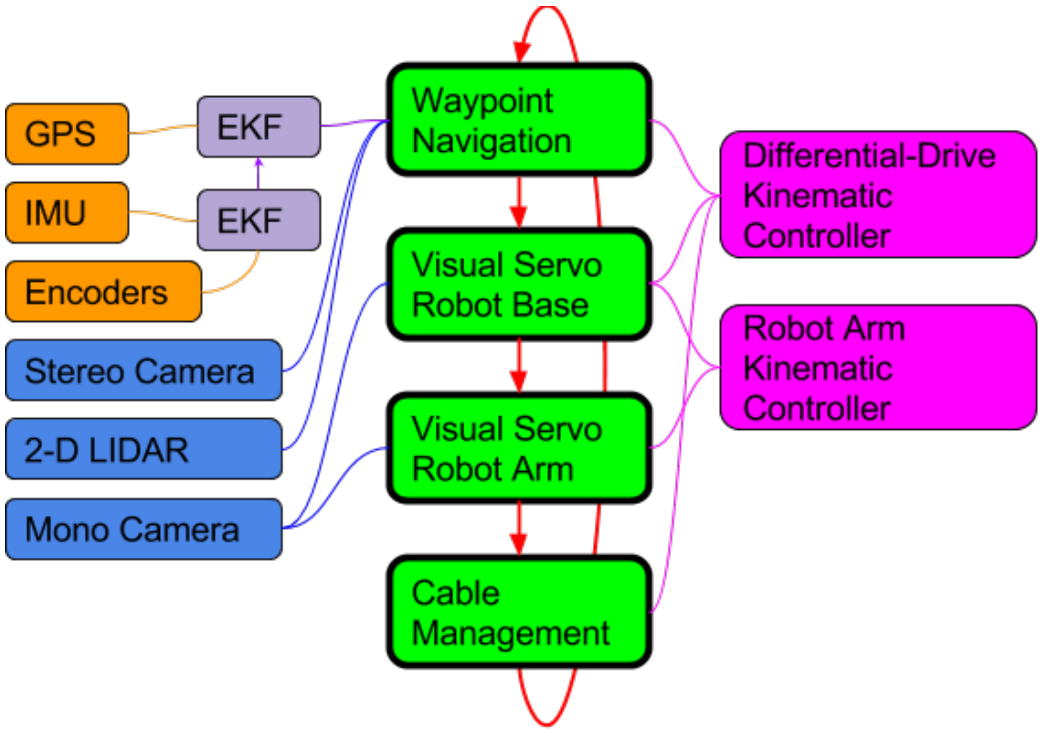

Figure 2.1: Microgrid formation depends on a multi-layered controller. In this thesis, the Waypoint and Departure controllers are highlighted, as well as sensor fusion of proprioceptive and exteroceptive sensors.

meaningful contribution to robot behavior from the raw sensor data.

2.1 Modular and Transferable Development

A force multiplier in robotics development has been the growth of the open source software community. Established tools and algorithms are implemented in packages that are maintained and distributed by individuals from academia and industry. One popular "ecosystem" is ROS. The early code development which became ROS originated at Stanford and Willow Garage in the 2000's, with early application on the PR2

Figure 2.2: ROS includes bindings and integration with many other important open-source software tools.

hardware platform. The ROS license allows for branching of existing code for private development and use, or contribution of new code additions back to the community.

One advantage of using ROS as the software "backbone" of a robot is that ROS is closely integrated with other open source software, some of which are indicated in Figure 2.2. ROS includes many sub-components, summarized in Figure 2.3. Packages are developed in C++ or Python. ROS is supported and integrated into many COTS robots. These include AUV's, UAV's, USV's, and UGV's. Although the Clearpath Husky UGV's are available from the manufacturer with a basic ROS install, we add drivers and packages as needed, both from the open source community and those we develop in-house.

It is important to point out benefits of using ROS for this microgrid work. First,

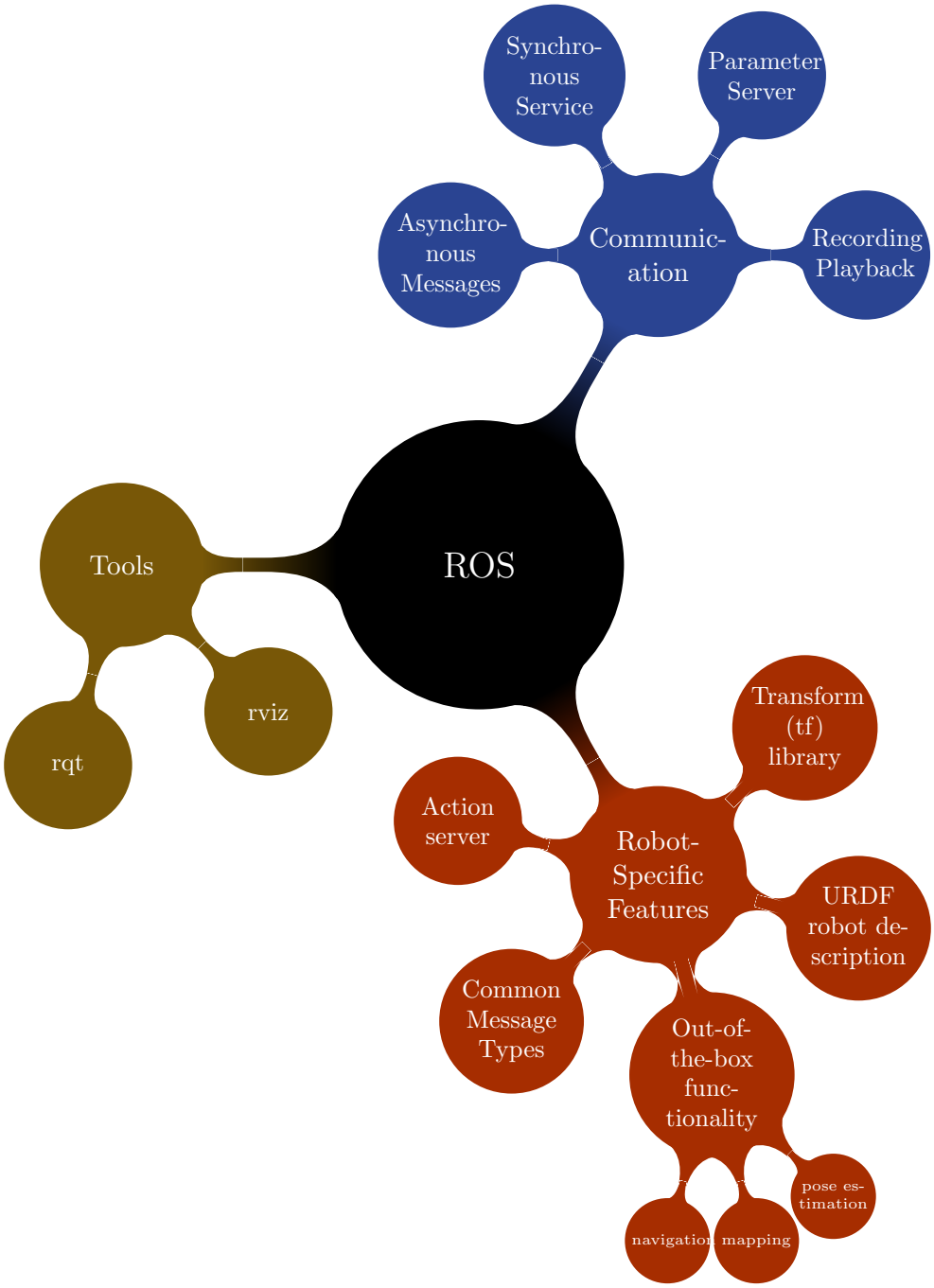

Figure 2.3: ROS is composed of many sub-architectures which provide important algorithmic support for robotics tasks.

the code base that now exists will look familiar to any future roboticist who also works in ROS. It conforms to established standards and is similar to examples they have already encountered, greatly aiding transfer of knowledge. Second, the individual components which constitute the waypoint controller and perception processing stack are inherently modular. If an upgrade is desired in the future, say to the local kinematic planner, it must only conform to the standards for the class and will serve as a drop-in replacement.

Although open source tools are available, hardware limitations constitute the difficulty of application. For instance, when two types of robots, say robotic arm and UGV, are integrated together, the constraints on each of them compound the difficulty in implementation. The controls engineer must work from first principles to ensure the hardware-software integration will succeed, especially for applications which need to perform in unknown, unstructured, environments.

2.2 ROS Navigation Stack

The ROS Navigation stack was developed as a 2D indoor sensor fusion, planning, and control tool for the PR2 robot [15]. It is inherently modular and has evolved over ensuing years with additional tools and for application on a multiplicity of platforms. As Figure 2.4 explains, the navigation stack has a number of important components:

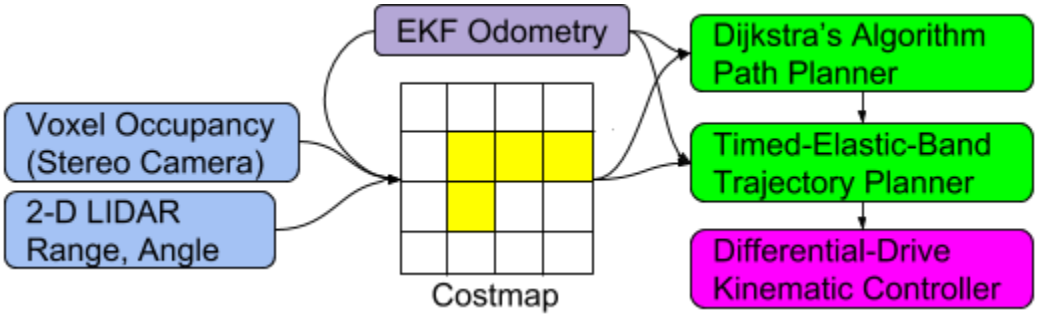

Figure 2.4: The ROS Navigation stack is comprised of a number of distinct elements.

the sensor inputs to a grid-based costmap, a position estimate, long range (global) and close range (local) path planners, and the robot kinematic controller. These components, of course, rely on the base ROS utilities, like the tf engine and the URDF robot description. This section will discuss the costmap and planners. In sections 2.3 and 2.4 the sensor inputs and localization are explored.

The concept of a grid based map from which a planner calculates a preferred route through obstacles, i.e. minimizes a cost function from start to end point, is well established in the field. In ROS this is called a costmap and is built and updated by a highly configurable [16] package called costmap_2d. Its basic requirements are the transforms between the orientation of the robot chassis relative to a fixed reference frame, the transforms between the perception sensor data and the robots chassis frame, and definition of the sensor data input layers. Each input layer is configurable with filters for range and precision of the sensor data. The overall occupancy map with all layers fused is also configurable, especially for grid resolution and for how the cost gradient relaxes from the grid squares marked as occupied.

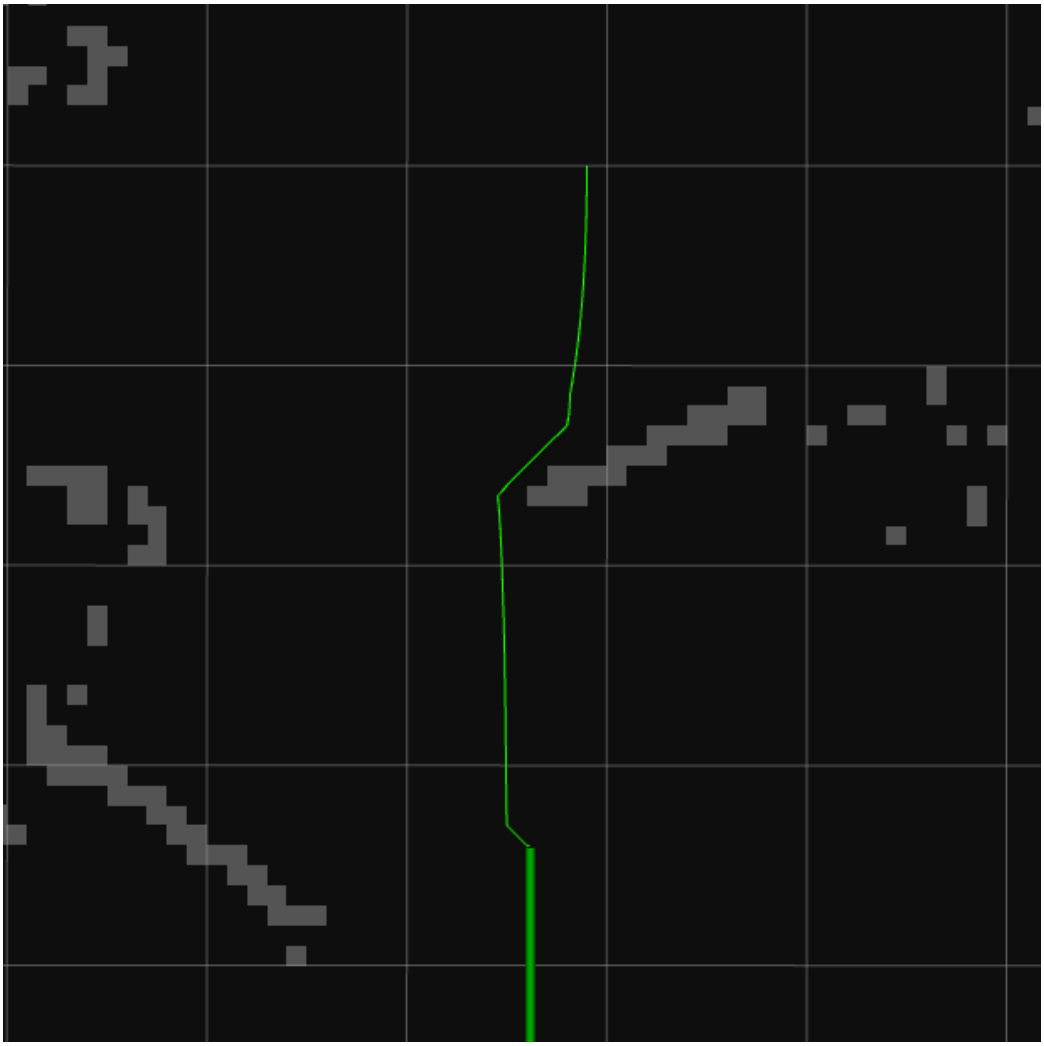

Figure 2.5: This example of a global planner output shows the low cost route in green. The regions marked as obstacles are indicated by light grey squares.

Every component of the Navigation package is designed to be modular. This includes the path planners. There are a number of established grid-based path planning algorithms provided to the user, including A* and Dijkstra's method. In the parlance of ROS, the kinematic agnostic planner is called the global planner. This grid based planner should calculate a low cost path to a waypoint using the fused costmap described above, with the important constraint of the computational cost of calculating

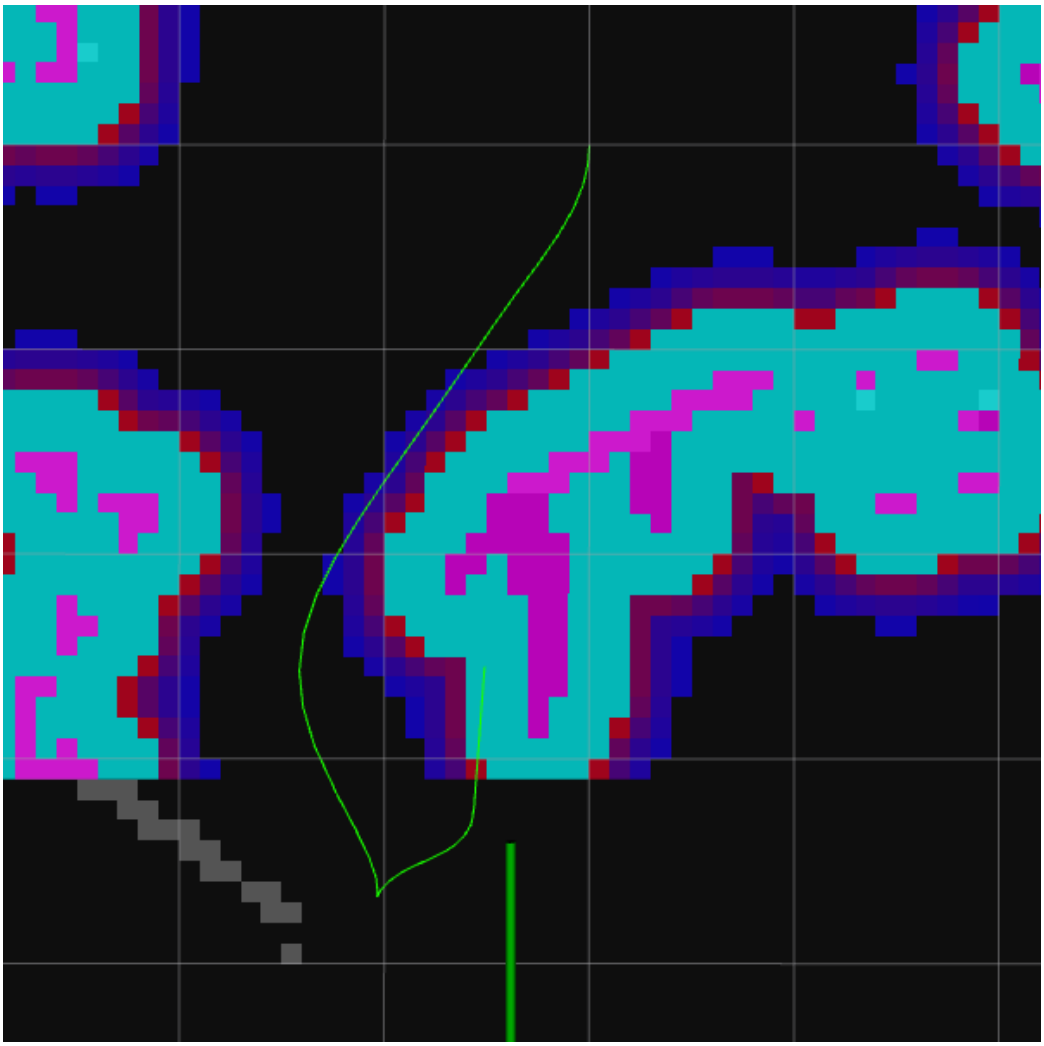

Figure 2.6: A TEB motion planner generates x and θ velocity commands to the UGV chassis, based on a costmap which takes into account the robot geometry, and with regard for the motion constraints of the platform.

and updating the best path as the UGV identifies new obstacles. Often, the global planner is configured to run at a lower resolution than the kinematic planner. Because there is not an apriori heuristic available for navigation in unknown environments, the default configuration of the global planner to Dijkstra's method was retained, Figure 2.5.

In many regards, the performance of the UGV is more dependent on the characteristics of its kinematic planner, called the local planner. This planner must take into account the physical constrains of the hardware, including the number of DOF it can maneuver, and limits on velocity, acceleration, and jerk. The goal of the local planner is to follow the global route within some limited segment, while planning smooth motions with a higher resolution map of the obstacles. Another element that can be introduced into the local planner is confidence about proximity to obstacles, which is often used to modulate the velocity of the UGV. There is substantially more room for innovation in development of kinematic planners. For this application, the TEB local planner [17] was implemented, with good success. Figure 2.6 shows an example of a local plan. This planner allows for more complicated motions, such as smooth reversing while turning. It is relatively computationally heavy, and as such the local costmap size has to be small and resolution somewhat suffers. In the outdoor environment, these trade-offs allow quick navigation through the environment, with no penalty for reduced resolution.

There are additional elements of the ROS Navigation package which are important to mention. The package accepts ROS actions published on a pose_goal topic. Like all ROS topics, any node can be written to publish to the topic. RVIZ can be used to efficiently generate wapoints, by clicking and dragging goal points in the costmap which are then publishee. For autonomous operation, a script is written which publishes and monitors actions for each waypoint. It is very easy to include

this waypoint action class in the multi-layered microgrid mission controller. There are a number of recovery behaviors included in the navigation packages. These behaviors are engaged subservient to the waypoint action, in situations where a valid path cannot be calculated or the UGV is too close to an obstacle.

2.3 Perception Sensor Fusion

UGV perception in unstructured environments is an open problem. Recent developments include augmenting SLAM methods with filtering to increase the robustness of keypoints in unstructured environments [18], application of machine learning to improve controller confidence in visual path tracking through varied terrain [19], and new image processing semantics to aid registration of live robot data to an associated satellite image [20]. One goal of the ongoing research in traversability classification from vision data is to offload the computational cost of the classifier to a powerful GPU on-board the robot, then deliver a reduced costmap result to the CPU which performs pathplanning [21].

To prepare for implementing advanced obstacle and terrain classification methods on the microgrid agents, the sensor package was upgraded from a 2D LiDAR to include a stereo camera. At this time, a basic obstacle classification and sensor fusion method is employed.

2.3.1 2D LiDAR

The Husky UGV ships with a SICK LMS1xx series 2D LiDAR. This is a cost effective way to obtain basic, but robust, exteroceptive data. This sensor returns an array of return distances through 270 degrees in the plane of the sensor. Although the strength of the return signal is dependent on the incident material, the range of the sensor is up to 20m, with a 50 hz update rate. On microgrid agents, the 2D LiDAR serves a practical role as a mid-range sensor which especially produces inputs to the global obstacle map, and hence the global path planner running Dijkstra's algorithm. It is also fused with the stereo vision inputs for the close range obstacle avoidance and motion planner, the TEB planner.

2.3.2 ZED Stereo Camera

Stereo vision is not new in the field of mobile robotics. What is new is the ease and speed at which the feature registration on the stereo images can be performed using GPU architecture. ZED stereo camera is a COTS product that pairs competitive sensor hardware with a software solution which depends on a CUDA equipped graphics card. An additional software dependency is OpenCV. The SDK outputs both colorized point clouds and grey-scale depth images at up to 1080p and 30 fps,

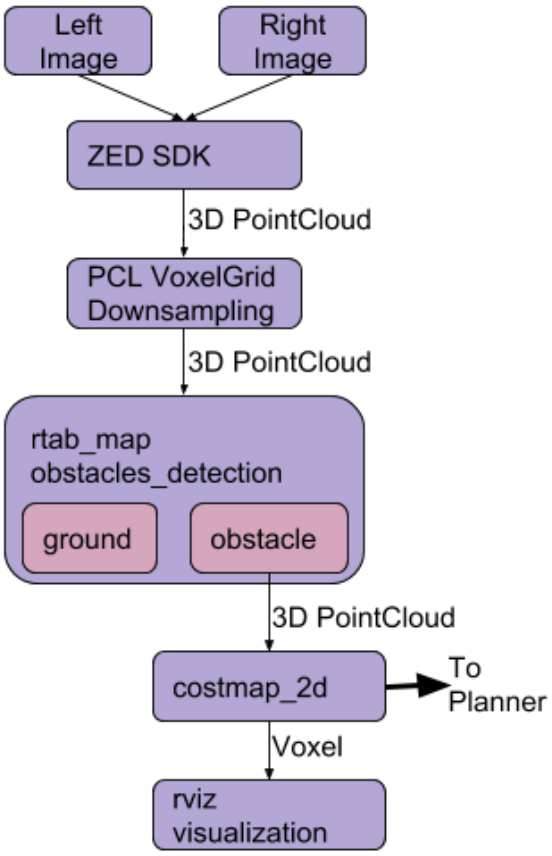

Figure 2.7: Processing of stereo images depends on a number of software packages, some from ROS and some from other open source development.

although the rate is dependent on supporting graphics and computer hardware. An additional feature of the SDK as a VO estimation at 100 hz. Unfortunately at the time of development, the VO output did not include its covariance, so it was difficult to fuse it with other localization data sources.

After implementing different methods to include the data from ZED into the costmap, and thus the path planning, architecture on the UGV's, the processing method shown in Figure 2.7 proved most advantageous. This method results in suitable resolution of obstacles, while minimizing resource consumption. Ultimate update rate is around 5

hz, as the graphics cards installed in the UGV's are constrained by space and power.

The 3D pointcloud output by the ZED SDK is passed to a PCL downsampling algorithm, VoxelGrid, which calculates the centroid of points within a cubic volume, or voxel. The size of the voxels determines the course-ness, and memory reduction, of the downsample. VoxelGrid outputs a reduced resolution 3D pointcloud.

Next, the reduced pointcloud is input to rtab_map obstacles_detection nodelet. This package groups regions of the pointcloud into surfaces and calculates the normal vector of that region. Any region with a surface normal pointing within a user-defined cone about the z-axis (up) is grouped into a "ground" label pointcloud. All other regions of the input pointcloud are labeled "obstacle". This is a rudimentary but effective method of segmenting the input pointcloud.

The obstacle pointcloud is passed to the Navigation Stack package costmap_2d as a VoxelCostmapPlugin Obstacle Layer. The costmap is configured with some grid resolution suitable for the operational environment. This plugin marks any grid square which contains a point from the obstacle cloud in a column normal to the costmap as an occupied region. There are user parameters for how much of the vertical volume above the grid square to consider for collision avoidance. For instance, the UGV can surmount obstacles of some height from the ground, so any obstacle below that height does not have to be labeled as such. Similarly, the UGV has a finite height. It is computationally advantageous to filter out data above this height.

This obstacle occpancy is superimposed with the 2D LiDAR occupancy as described above, and delivered to the planners.

Finally, there are some tools available to visualize each step of this process in RVIZ. Especially useful is to display the output of the VoxelCostmapPlugin above the costmap itself, as this is fairly computationally lightweight and helps debug the UGV behavior. The alternative, displaying any of the pointclouds generated throughout the process, while possible, taxes network bandwidth and computer resources.

2.4 Position Estimate via Kalman Filter

Localization of an autonomous robot agent is an important task in the overall control software. Localization sensors include wheel encoders, IMU, GPS, and visual sensors. Localization sensors can be separated into two groups. Propreoceptive sensors measure the state of the robot without regard to its interaction with its environment; wheel odometers measure rotation of the drive shaft and IMU reports orientation changes with respect to an initial state. Exteroceptive sensors measure the movement of a robot relative to its surroundings; GPS receives a pose update relative to a fixed world coordinate system and visual odometry calculates displacement relative to environmental visual indicators.

There are a number of challenges that arise when working with position sensors. First, errors in Propreoceptive sensors tend to compound over time and displacement. Without some method of correcting this "drift" in the sensor relative to a fixed, or world, coordinate frame, the error in position estimate will eventually grow so large that the robot cannot perform its task. Exteroceptive sensors, on the other hand, are subject to disturbance. Interactions with the environment that are outside the engineers control can interfere with electromagnetic signals like GPS. Visual methods for localization are often foiled by difficult lighting, rain, snow, and fog.

ROS includes hardware drivers for most of the sensors that a user would like to install on their robot. It should be mentioned, that while the libraries exist to extract the data from the sensor, it is not trivial to ensure that the integration of the sensor with the robot is performed well. For instance, an IMU sensor had a particular coordinate frame associated with its data. This coordinate frame may be different than the convention used in the control algorithm and must be transformed. Another example is GPS position data. Latitude and longitude in degrees is not copacetic to other odometry sources measured in meters and must be converted, usually by calculating the UTM coordinate.

Often, the various localization inputs on a UGV are fused using a Kalman Filter. robot_localization package is a configurable tool which provides a generalized implementation of the Kalman Filter to ROS. It accepts sensor data of the first three orders

of location, that is pose, velocity, and acceleration. This data must have a coordinate frame associated with it that can be resolved to the output frame through the transforms defined in the ROS tf engine. It is very important that the sensor covariance matrix is updated with accurate values, or the filter behavior will be erratic. The particular configuration for the Extended Kalman Filter used on the UGV can be found in Appendix A.

Chapter 3

Hardware Development

To be able to form a microgrid outdoors robustly and reliably, some upgrades needed to be performed to the UGV agents. The base UGV is a Clearpath Husky. The upgrades can be grouped into those supporting perception and those supporting electrical cable deployment.

Husky UGV's ship with a Mini-ITX computer, running an Intel i5 processor at 2.9 GHz and Ubuntu 14.04. Sensors include LORD MicroStrain 3DM-GX3-25 9-DOF IMU and NovAtel SMART6-L GPS receiver, as well as the SICK LMS-111 2D LiDAR. The computer is located in a small payload bay, with no provisions for thermal dissipation and only 5 amps of power available to a 12v rail. A 5-DOF robot are is installed on the UGV, with its own power source. There are also electral connectors

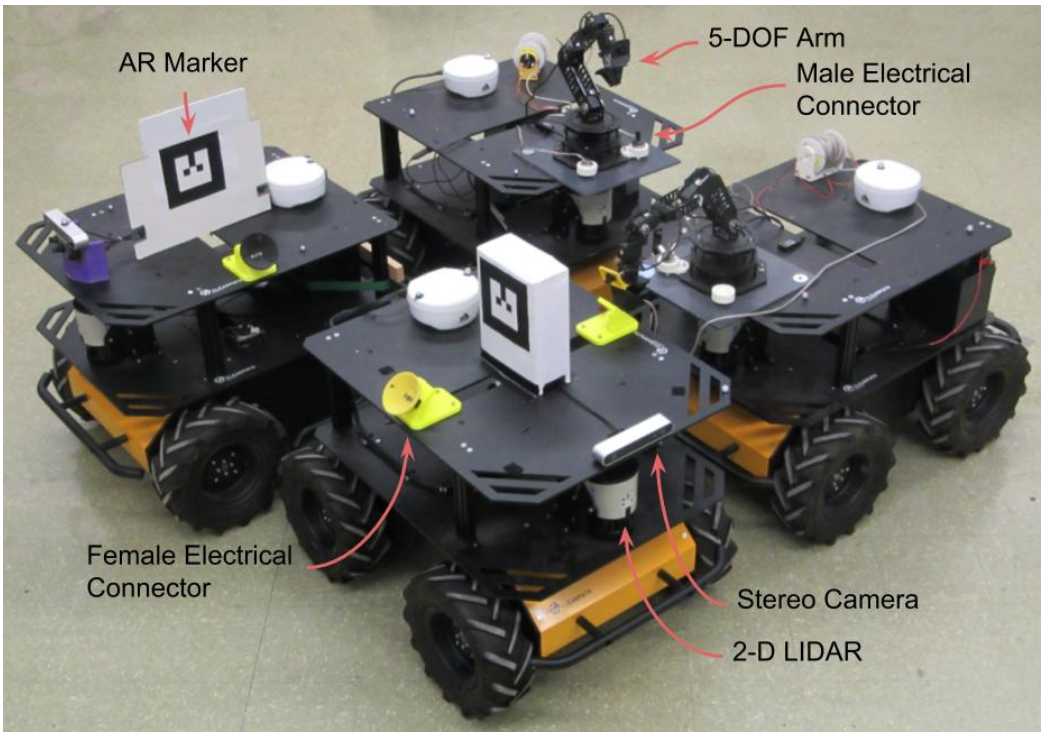

Figure 3.1: The base UGV is augmented with sensors and actuators to fulfill different microgrid configuration tasks.

and augmented reality markers mounted on the agents. An overview of microgrid agents is provided in Figure 3.1.

3.1 ZED Stereo Camera for Enhanced Perception

Current market trends in 3D sensing technology present exciting trade-offs in perception capability with dramatically decreasing cost. One goal of this communication is to present improvements to the microgrid agent platform which enable implementation and development of state-of-the-art terrain and obstacle classification methods.

Figure 3.2: ZED Camera is a COTS stereo vision solution. https://cdn.stereolabs.com/img/product/ZED_product_main.jpg

While 3D LiDAR provides a very robust measurement of the contours of obstacles and terrain, they do not provide the types of visual features available from a video stream. As machine learning techniques are improved for obstacle and terrain classification from images, the value of vision compared to LiDAR increases. A stereo vision system provides some of the depth measurement capacity of a 3D LiDAR, while allowing development or implementation of new classification methods from visual features. Additionally, 3D LiDAR is still very expensive compared to a stereo vision solution.

ZED Stereo camera is a COTS product which provides stereo hardware in a small and simple physical package. Power and data are transmitted through a single USB 3.0 plug. The SDK provided by ZED developers depends on a powerful GPU, running CUDA hardware. This SDK simplifies the development process for the controls engineer. On the microgrid agents, the ZED camera is mounted above the 2D Lidar at the front of the robot (Silver bar in lower center of Figure 3.1). A powered USB hub is necessary to provide for the substantial current draw of the camera, as many built-in USB controllers do not supply sufficient current.

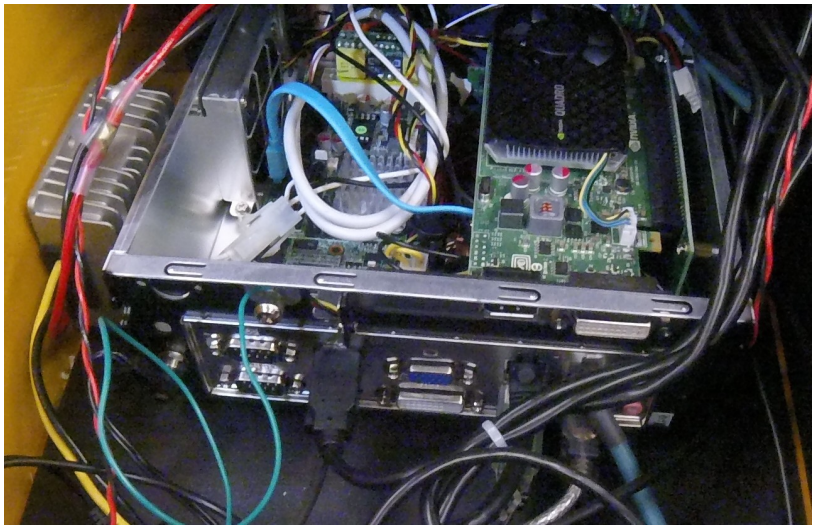

Figure 3.3: The Husky payload bay squeezes in a Mini-ITX computer with CUDA equipped GPU, with upgraded power supply.

3.2 GPU Integration for Stereo Vision Processing

As stated in the previous section, ZED stereo camera depends on a CUDA equipped GPU. Provisions for the GPU are non-trivial on a small mobile robot. Although a popular GPU option is the NVIDIA Jedsen platform, we chose to install PCI cards in the Mini-ITX on the Husky. The Mini-ITX computer has one PCI slot. The physical size of the computer, not to mention the payload bay, is very small. There are a few families of suitable GPU's which are compatible with these constraints.

The biggest factor which should be considered when specifying a processing platform is power consumption. Although the Husky UGV is advertised as supporting the required GPU, in actuality there is no provision for supplying the upwards of 20

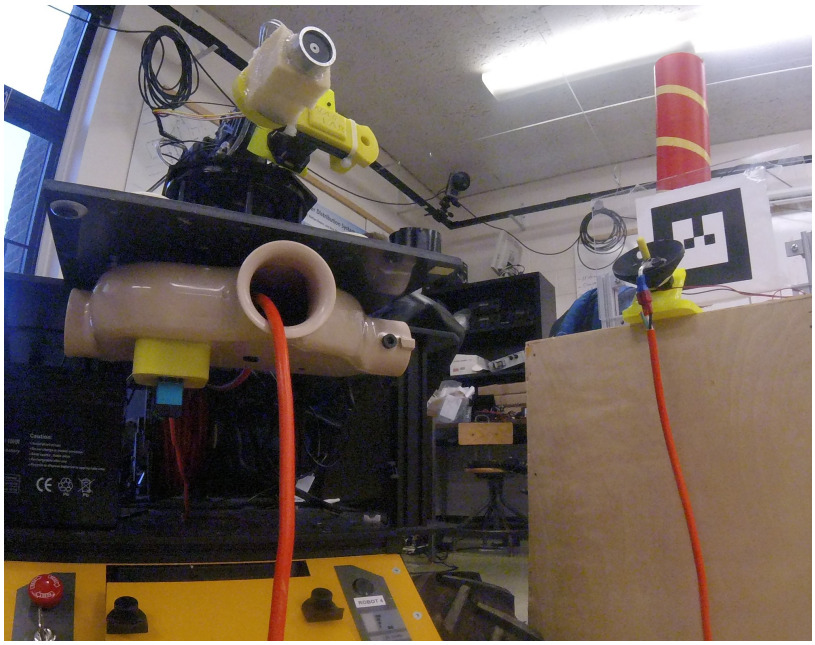

Figure 3.4: Custom hardware was designed to support powered cable deployment.

amps required by any suitable product at peak load. We were forced to hack an additional power supply to the battery connectors, visible on the left in Figure 3.3. The increased power consumption at idle proves to severely reduce the battery life of the robot. Indeed, traditional lead acid batteries should not be considered as a power source in this type of application.

3.3 Powered Electrical Cable Deployment System

One challenge throughout the generations of prior microgrid development has been management of the electrical cable while the agents are navigating between waypoints. The cable should deploy behind the UGV at the rate of travel of the agent such that

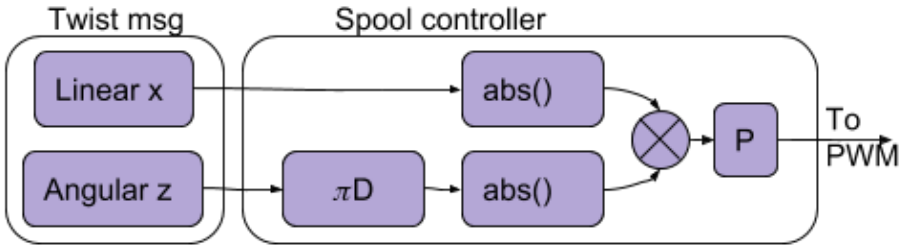

Figure 3.5: Control of the spool recieves a ROS twist message as input and calculates an open-loop gain to drive the feed drum.

the there is no undue stress or tension on the electrical connection as the agent maneuvers. Prior experience with the platform has suggested that a passive system has too much inherent friction at the cable drum or through a fairlead. One approach to mitigating this problem is a powered feed system which actively feeds the cable behind the agent as it drives. The system utilized on the microgrid UGV integrates COTS components in a feed drum, shown in Figure 3.4. The components of the feed drum include a cable drum driven by a hobby servo. A DC to DC switch mode power supply supplies 7.4V at high amperage to the servo. A PWM control signal is generated by an Arduino UNO, with serial communication via USB to the UGV control computer. The hobby servo is a product which integrates a DC motor with an encoder and simple closed loop feedback control. The open loop control function which calculates servo PWM commands from the UGV linear and angular velocity, x and z respectively, is presented in Figure 3.5. The gain is dependent on the diameter D from the zero-point turn center of the chassis to the exit point of the cable. The proportional gain P is determined empirically.

Chapter 4

Results and Discussion

Results from this work focus on three areas. The first area is the outcome of the perception sensor fusion, namely 2D LiDAR with stereo vision. Second, fusion of the odometry sources and waypoint navigation capability with obstacles is represented. Third, navigation for microgrid formation is reported, including outdoor ground truth from multiple agents. Finally, the work is wrapped up in a conclusion and opportunities for future development are explored.

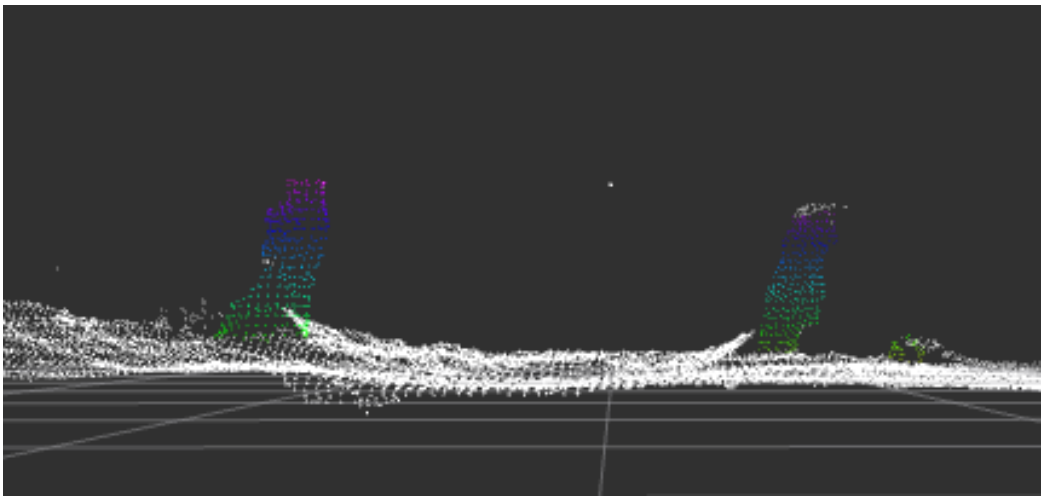

Figure 4.1: Segmentation of ground surface normals reveals obstacles. The colored points have been labeled as obstacles, in this case two tree trunks. White points are labeled as traversable.

4.1 Sensor Fusion of Stereo Vision and LiDAR

While the output from 2D LiDAR can be used for costmap population and decision making with very little filtering, the 3D pointcloud from ZED requires, at a minimum, ground plane segmentation before it can be used to determine the location of obstacles. In Figure 4.1 we see the result of the filtering and segmentation stack discussed in section 2.3.2. The pointcloud colored white is the subset of this frame that is labeled as ground. The pointcloud which is colored by height are the elements that have been labeled obstacle. This data was obtained outdoors in a grassy area with trees. The settings for the segmentation have been tuned such that small irregularities in the ground plane, terrain that is surmountable by the UGV, are not labeled as obstacle. This data is visualized using RVIZ.

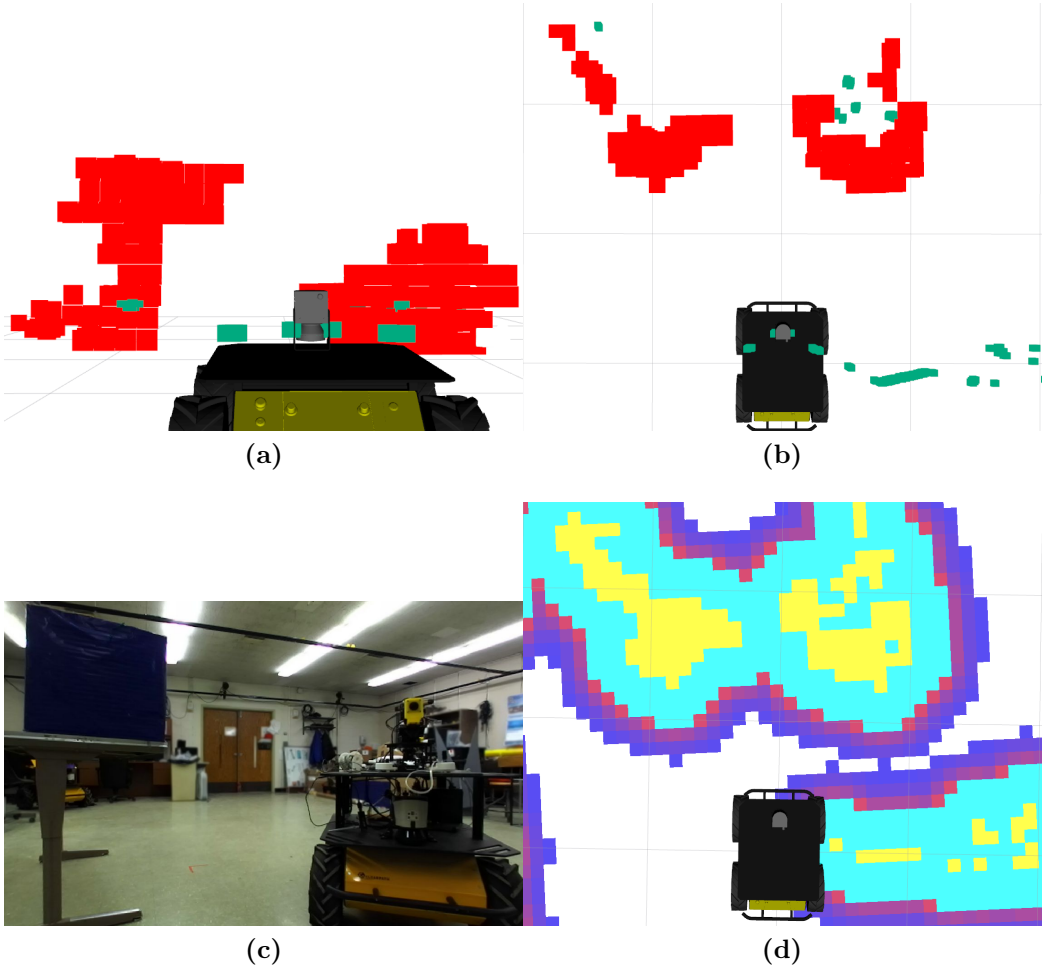

Figure 4.2: Costmap from fusion of ZED stereo camera and 2D LIDAR. A table and another UGV are obstacles. (a) shows these obstacles. Red areas are regions marked as occupied using the stereo disparity data. Green areas are points marked occupied based on LiDAR reflectance. (b) is the same data viewed from above. (c) is the view from the ZED camera on the active robot. (d) Lower-right image is the fused costmap. Yellow indicates collision regions, with cost descending to blue.

Now that ground segmentation has been achieved, the obstacle cloud can be passed to the costmap and fused with 2D LiDAR data. In Figure 4.2 this fusion is depicted using complicated, and overhanging obstacles. Immediately in front of the UGV are two obstacles, a desk with box on top, and another UGV. The 2D LiDAR data

is indicated by green dots. It is clear that using 2D LiDAR alone, the data for obstacle avoidance would be very sparse. Most distressing, the UGV would not be able to navigate with a reasonable expectation of safety in proximity to these types of obstacles as collision hazards are present which are not represented in the 2D data. By adding occupancy volumes identified using ZED camera and the software stack presented here, the red regions are also available to the costmap. These red regions reveal important features of the environment, such as the overhanging desk top and the wheels and front bumper of the adjacent UGV. One feature of the 2D LiDAR that complements ZED is the LiDAR's 270 degree FOV. Out of the frame of the camera to the right is the border of the test area, which the LiDAR identifies. When these layers are fused into a costmap, it shows that the safe navigation region does not pass between the overhanging desk and the adjacent UGV, but rather traverses around to the right along the edge of the test area.

4.2 Waypoint Navigation with EKF Localization

Odometry fusion is a well established result in ground robotics. Of interest to this work is the effect of GPS disturbance to localization and thus waypoint navigation. Figure 4.3 shows a working data from our UGV which replicates a common representation of the EKF fusion of wheel encoder, IMU, and GPS localization sources. The

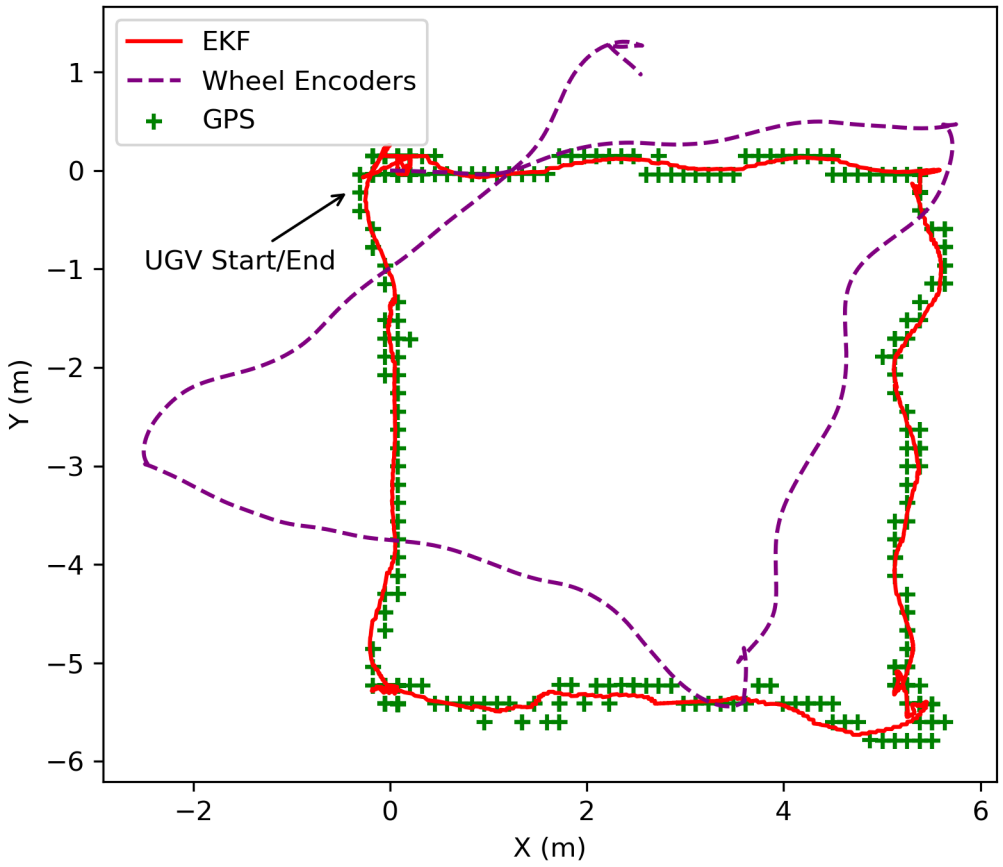

Figure 4.3: Fusion of multiple sources of odometry results in an improved estimate of the UGV position.

UGV is instructed to navigate a sequence of poses which form a square. The maneuvers are performed on loose gravel outdoors, a non-linear surface which is hard to compensate for in wheel encoder data and which causes many of the disturbances seen in the vehicle trajectory. The final pose of the UGV is identical to it's starting pose. It is clear in the figure that the estimate from wheel encoder data would be unsuitable

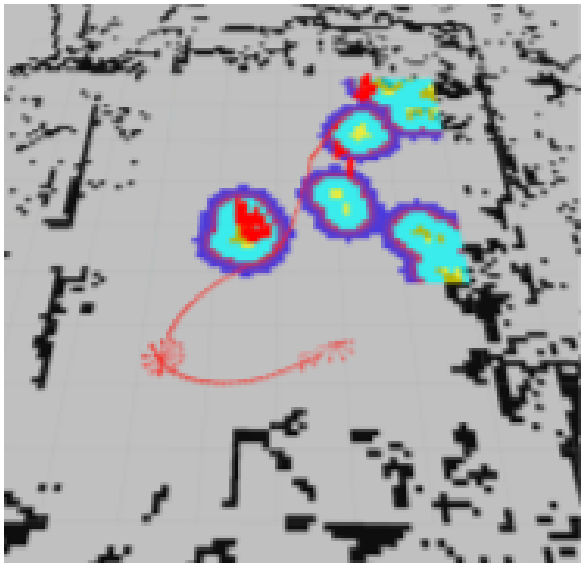

Figure 4.4: With all the elements of the navigation stack in place, the UGV can achive goal poses in a cluttered environment. In this example, ocupied volumes are marked with red boxes, and the costmap gradient is in yellow and blue. The fused odometry estimate as the UGV avoids obstacles and traverses the room is shown with red arrows.

for any navigation task. The final pose estimate is displaced around 3 meters from the true pose of the agent, and none of the intermediate waypoints have been achieved. What is interesting to note is that the un-filtered GPS shows substantial noise, especially during turns. This proves to be highly destabilizing to kinematic planners, and would not allow for the localization precision on the order of centemeters required for the microgrid application. By introducing an IMU source, not depicted in the figure, and wrapping the odometry sources in an Extended Kalman Filter, the fused pose estimate is suitable for navigation.

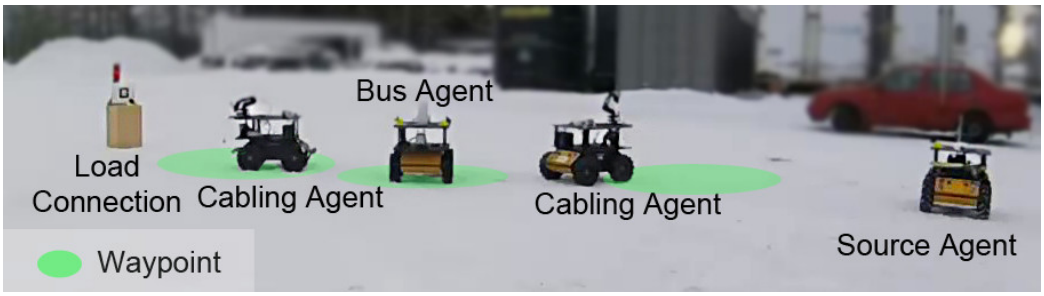

Figure 4.5: Outdoor test scenario in snow, with a power source, a load, bus agent, and two cabling agents.

These components allow for waypoint naviagtion and mission accomplishment in cluttered and outdoor environments. Figure 4.4 is an RVIZ representation of the progression of an agent in a cluttered map with obstacles. An initial waypoint is located in the lower left region of the test area. The UGV is then given a waypoint to achieve in the upper right. The UGV avoids obstacles throughout the area, with the global and local planner working together to form smooth and efficient path. The EKF odometry estimate is displayed as a series of small red arrows.

4.3 Multiple Agent Microgrid Formation

Mobile microgrids can be applied in many scenarios and environments. One example of a field test using this method is formation of a microgrid with three active UGV's. There is a load which must be serviced, a source, and electrical connections that must be formed between them, with a bus agent in the middle. Figure 4.5 depicts this test arrangement. The test region was snow covered. Waypoints were assigned

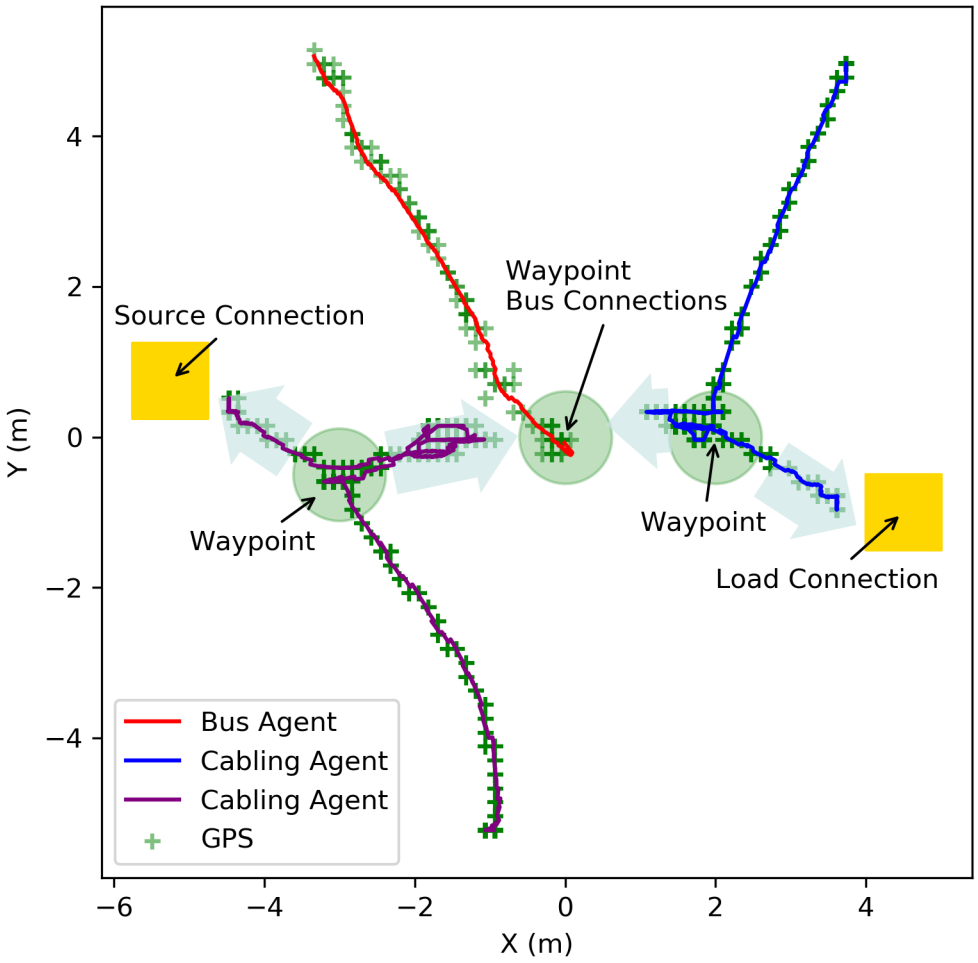

Figure 4.6: Odometry of three agents in an outdoor environment. The agents attain all their pose goals for microgrid formation.

adjacent to the connection hardware. The connection method is outside the scope of this discussion.

The agents are able to navigate to the waypoints and achieve the goal poses which

prepare for the remaining control tasks outlined in Figure 2.1. Representative odometry results from this field work are shown in Figure 4.6. EKF estimates are solid lines, while raw GPS are small crosses. Each agent attains it's waypoint, then continues to perform microgrid tasks. Disturbance in the raw GPS pose is again visable. The solid lines show the improved EKF pose estimate.

4.4 Summary and Conclusion

We conclude this work by summarizing the interconnected tools which have been developed and optimized to support positioning and connection of UGV agents in an outdoor environment. The ROS Navigation stack was adopted to provide a framework for localization, sensor fusion, a grid-based costmap, and path and motion planning. A stereo vision system was selected and implemented, including supporting hardware upgrades. A downsampling and obstacle segmentation scheme was implemented to prepare 3D obstacle data for fusion with 2D data already available on the robot. An entirely new cable deployment system was designed and tested which prevents entanglement of the electrical cable with the UGV chassis or obstacles. Although many of these tools are general purpose, in this work they are elevated specifically for the mobile microgrid application.

Because of the use of ROS as a software backbone, the code base from this project

is transferable and upgrade-able. This work is also extendable and implementable in other domains and for other purposes. A similar framework has already been applied by the author to an AUV [22]. The flexibility of these tools supports underwater docking and recharging [23] and has been proposed as a method for exploring other worlds [24].

4.5 Future Work with Opportunities

As the results show, the status of the waypoint controller and the perception hardware fully supports continued development of the docking and coupling tasks outdoors. This implementation is largely platform agnostic. Perhaps a larger, more capable UGV is needed for future demonstration. Simply exchange the kinematic controller and all other elements remain the same.Some refinements could be considered and future development was implicit in the design criteria. There are two refinements which should be considered. The recovery behaviors are not chassis or perception sensor specific. These behaviors are enlisted when a planner is not able to calculate a valid path to a waypoint from the current pose of the UGV. New behaviors specific to the differential drive chassis and the outdoor environment should be developed to obtain fresh obstacle data when the UGV is "stuck". Periodically the stereo image processing generates spurious occupancy regions, this should also be considered when developing recovery behaviors. Another refinement is the addition of close range

proximity sensors at the rear of the UGV. Reversing is an important navigation tool, but often does not require high fidelity remapping of the environment. Simple collision avoidance would be sufficient. Finally, the sensor package was always intended to support future development of novel obstacle and terrain classification methods. Although some analytical methods have been discussed, the availability of machine learning tools in Python provides a promising path forward.

References

[1] E. HOLLNAGEL and Y. FUJITA, "THE FUKUSHIMA DISASTER SYSTEMIC FAILURES AS THE LACK OF RESILIENCE," *Nuclear Engineering and Technology*, vol. 45, no. 1, pp. 13–20, Feb. 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1738573315300024

[2] Z. Chu, J. Zhang, O. Kosut, and L. Sankar, "Evaluating power system vulnerability to false data injection attacks via scalable optimization," in *2016 IEEE International Conference on Smart Grid Communications (SmartGridComm)*, Nov. 2016, pp. 260–265.

[3] J. J. Romero, "Blackouts illuminate India's power problems," *IEEE Spectrum*, vol. 49, no. 10, pp. 11–12, Oct. 2012.

[4] J. Gregory, J. Fink, E. Stump, J. Twigg, J. Rogers, D. Baran, N. Fung, and S. Young, "Application of Multi-Robot Systems to Disaster-Relief Scenarios with

Limited Communication," in *Field and Service Robotics*, ser. Springer Tracts in Advanced Robotics. Springer, Cham, 2016, pp. 639–653.

[5] J. Moore, K. C. Wolfe, M. S. Johannes, K. D. Katyal, M. P. Para, R. J. Murphy, J. Hatch, C. J. Taylor, R. J. Bamberger, and E. Tunstel, "Nested marsupial robotic system for search and sampling in increasingly constrained environments," in *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, Oct. 2016, pp. 002 279–002 286.

[6] S. Janko, S. Atkinson, and N. Johnson, "Design and Fabrication of a Containerized Micro-Grid for Disaster Relief and Off-Grid Applications," *ASME 2016 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, vol. 2A, Aug. 2016.

[7] H. Seo, S. Kim, and H. J. Kim, "Aerial grasping of cylindrical object using visual servoing based on stochastic model predictive control," in *Robotics and Automation (ICRA), 2017 IEEE International Conference on.* IEEE, 2017, pp. 6362–6368. [Online]. Available: http://ieeexplore.ieee.org/abstract/document/7989751/

[8] K. Morton and L. F. G. Toro, "Development of a robust framework for an outdoor mobile manipulation UAV," in *2016 IEEE Aerospace Conference*, Mar. 2016, pp. 1–8.

[9] H. Yaguchi, K. Nagahama, T. Hasegawa, and M. Inaba, "Development of an autonomous tomato harvesting robot with rotational plucking gripper," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Oct. 2016, pp. 652–657.

[10] Weaver, Wayne W., Nina Mahmoudian, and G. Parker, "Autonomous mobile power blocks for prepositioned power conversion and distribution," in *TARDEC Ground Vehicle Systems Engineering and Technology Symposium*, 2012.

[11] B. Moridian, D. Bennett, N. Mahmoudian, W. W. Weaver, and R. Robinnett, "Autonomous Power Distribution System," *IFAC Proceedings Volumes*, vol. 47, no. 3, pp. 7–12, Jan. 2014.

[12] B. Moridian, D. Bennett, N. Mahmoudian, R. Robinett, and W. W. Weaver, "Design of Mobile Microgrids Hierarchy for Power Distribution," *ASME 2015 Dynamic Systems and Control Conference*, vol. 3, Oct. 2015.

[13] B. Moridian, N. Mahmoudian, W. W. Weaver, and R. D. Robinett, "Robotic power distribution system for post-disaster operations," in *2015 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, Oct. 2015, pp. 1–6.

[14] ——, "Postdisaster Electric Power Recovery Using Autonomous Vehicles," *IEEE Transactions on Automation Science and Engineering*, vol. 14, no. 1, pp. 62–72, Jan. 2017.

[15] E. Marder-Eppstein, E. Berger, T. Foote, B. Gerkey, and K. Konolige, "The Office Marathon: Robust navigation in an indoor office environment," in *2010 IEEE International Conference on Robotics and Automation*, May 2010, pp. 300–307.

[16] D. V. Lu, D. Hershberger, and W. D. Smart, "Layered costmaps for context-sensitive navigation," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sep. 2014, pp. 709–715.

[17] C. Rsmann, F. Hoffmann, and T. Bertram, "Planning of multiple robot trajectories in distinctive topologies," in *2015 European Conference on Mobile Robots (ECMR)*, Sep. 2015, pp. 1–6.

[18] C. Brand, M. J. Schuster, H. Hirschmller, and M. Suppa, "Submap matching for stereo-vision based indoor/outdoor SLAM," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Sep. 2015, pp. 5670–5677.

[19] C. J. Ostafew, A. P. Schoellig, T. D. Barfoot, and J. Collier, "Learning-based Nonlinear Model Predictive Control to Improve Vision-based Mobile Robot Path Tracking: Learning-based Nonlinear Model Predictive Control to Improve Vision-based Mobile Robot Path Tracking," *Journal of Field Robotics*, vol. 33, no. 1, pp. 133–152, Jan. 2016.

[20] A. Viswanathan, B. R. Pires, and D. Huber, "Vision-based robot localization

across seasons and in remote locations," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, May 2016, pp. 4815–4821.

[21] T. Suleymanov, L. M. Paz, P. Pinis, G. Hester, and P. Newman, "The path less taken: A fast variational approach for scene segmentation used for closed loop control," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Oct. 2016, pp. 3620–3626.

[22] N. M. J. Naglak, B. Page, "Backseat control of sandshark auv using ros on raspberrypi," in *OCEANS 2018 - CHARLESTON*, October 2018.

[23] B. L. et al., "Integrated mission planning and adaptable docking system for auv persistence," in *IEEE OES Autonomous Underwater Vehicle Symposium PORTO*, November 2018.

[24] C. K. N. M. B. Page, J. Naglak, "Autonomous docking for exploration of extraterrestrial lakes," in *AIAA Guidance, Navigation, and Control Conference, San Diego*, January 2019.

[25] J. N. Twigg, J. M. Gregory, and J. R. Fink, "Towards online characterization of autonomously navigating robots in unstructured environments," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Oct. 2016, pp. 1198–1205.

[26] T. Pire, T. Fischer, J. Civera, P. D. Cristforis, and J. J. Berlles, "Stereo parallel

tracking and mapping for robot localization," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Sep. 2015, pp. 1373–1378.

[27] M. Labbe and F. Michaud, "Online global loop closure detection for large-scale multi-session graph-based slam," in *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*. IEEE, 2014, pp. 2661–2666.

[28] T. Moore and D. Stouch, "A Generalized Extended Kalman Filter Implementation for the Robot Operating System," in *Intelligent Autonomous Systems 13*, ser. Advances in Intelligent Systems and Computing. Springer, Cham, 2016, pp. 335–348.

[29] K. Pentenrieder, P. Meier, G. Klinker, and others, "Analysis of tracking accuracy for single-camera square-marker-based tracking," in *Proc. Dritter Workshop Virtuelle und Erweiterte Realitt der GIFachgruppe VR/AR, Koblenz, Germany*, 2006. [Online]. Available: https://www.researchgate.net/profile/Peter_Meier6/publication/251736130_Analysis_of_Tracking_Accuracy_for_Single-Camera_Square-Marker-Based_Tracking/links/544781000cf2f14fb811f89b.pdf

[30] O. Egbue, D. Naidu, and P. Peterson, "The role of microgrids in enhancing macrogrid resilience," in *2016 International Conference on Smart Grid and Clean Energy Technologies (ICSGCE)*, Oct. 2016, pp. 125–129.

[31] A. Gawel, M. Kamel, T. Novkovic, J. Widauer, D. Schindler, B. P. von Altishofen, R. Siegwart, and J. Nieto, "Aerial Picking and Delivery

of Magnetic Objects with MAVs," *arXiv preprint arXiv:1612.02606*, 2016. [Online]. Available: https://arxiv.org/abs/1612.02606

[32] M. Laranjeira, C. Dune, and V. Hugel, "Catenary-based visual servoing for tethered robots," in *Robotics and Automation (ICRA), 2017 IEEE International Conference on*. IEEE, 2017, pp. 732–738. [Online]. Available: http://ieeexplore.ieee.org/abstract/document/7989090/

[33] N. Yao, E. Anaya, Q. Tao, S. Cho, H. Zheng, and F. Zhang, "Monocular vision-based human following on miniature robotic blimp," in *Robotics and Automation (ICRA), 2017 IEEE International Conference on*. IEEE, 2017, pp. 3244–3249. [Online]. Available: http://ieeexplore.ieee.org/abstract/document/7989369/

[34] D. U. Case, "Analysis of the Cyber Attack on the Ukrainian Power Grid," 2016. [Online]. Available: https://ics.sans.org/media/E-ISAC_SANS_Ukraine_DUC_5.pdf

[35] Y. Li, K.-D. Nguyen, and H. Dankowicz, "A Robust Adaptive Controller for a Seed Refilling System on a Moving Platform**This work was supported by National Institute of Food and Agriculture, U.S. Department of Agriculture, grant number 2014-67021-22109." *IFAC-PapersOnLine*, vol. 49, no. 16, pp. 341–346, Jan. 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2405896316316263

[36] S. Siltanen, *Theory and Applications of Marker-based Augmented Reality*. VTT, Finland: VVT Technical Research Center of Finland, 2012.

[37] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: an efficient probabilistic 3d mapping framework based on octrees," *Autonomous Robots*, vol. 34, no. 3, pp. 189–206, Apr. 2013.

[38] J. G. Rogers, J. R. Fink, and E. A. Stump, "Mapping with a ground robot in GPS denied and degraded environments," in *2014 American Control Conference*, Jun. 2014, pp. 1880–1885.

[39] C. Marnay, H. Aki, K. Hirose, A. Kwasinski, S. Ogura, and T. Shinji, "Japan's Pivot to Resilience: How Two Microgrids Fared After the 2011 Earthquake," *IEEE Power and Energy Magazine*, vol. 13, no. 3, pp. 44–57, May 2015.

[40] "The story of Puerto Ricos power grid is the story of Puerto Rico," *The Economist*, Oct. 2017. [Online]. Available: https://www.economist.com/news/united-states/21730432-even-hurricane-maria-hit-it-was-mess-story-puerto-ricos-power-grid

[41] D. Wagman, "Logistics Complicate Puerto Rico's Electric Grid Recovery," Sep. 2017. [Online]. Available: https://spectrum.ieee.org/energywise/energy/policy/logistics-complicates-puerto-ricos-electric-grid-recovery

[42] "statusPR." [Online]. Available: http://status.pr/Home

[43] L. Alvarez, "A Great Migration From Puerto Rico Is Set to Transform Orlando," *The New York Times*, Nov. 2017. [Online]. Available: https://www.nytimes.com/2017/11/17/us/puerto-ricans-orlando.html

[44] F. Robles, "Contractor for Puerto Rico Power Suspends Work, Citing Unpaid Bills," *The New York Times*, Nov. 2017. [Online]. Available: https://www.nytimes.com/2017/11/21/us/whitefish-puerto-rico-power.html

[45] G. Joos, J. Reilly, W. Bower, and R. Neal, "The Need for Standardization: The Benefits to the Core Functions of the Microgrid Control System," *IEEE Power and Energy Magazine*, vol. 15, no. 4, pp. 32–40, Jul. 2017.

[46] E. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[47] Z. Zhang, "A flexible new technique for camera calibration," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 11, pp. 1330–1334, Nov. 2000.

[48] S. Chitta, E. Marder-Eppstein, W. Meeussen, V. Pradeep, A. Rodrguez Tsouroukdissian, J. Bohren, D. Coleman, B. Magyar, G. Raiola, M. Ldtke, and E. Fernandez Perdomo, "ros_control: A generic and simple control framework for ROS," *The Journal of Open Source Software*, vol. 2, no. 20, p. 456, Dec. 2017.

[49] J. Carius, M. Wermelinger, B. Rajasekaran, K. Holtmann, and M. Hutter, "Autonomous Mission with a Mobile ManipulatorA Solution to the MBZIRC," in *Field and Service Robotics*. Springer, 2018, pp. 559–573.

[50] S. Maharjan, Y. Zhang, S. Gjessing, O. Ulleberg, and F. Eliassen, "Providing Microgrid Resilience during Emergencies Using Distributed Energy Resources," in *2015 IEEE Globecom Workshops (GC Wkshps)*, Dec. 2015, pp. 1–6.

[51] C. Abbey, D. Cornforth, N. Hatziargyriou, K. Hirose, A. Kwasinski, E. Kyriakides, G. Platt, L. Reyes, and S. Suryanarayanan, "Powering Through the Storm: Microgrids Operation for More Efficient Disaster Recovery," *IEEE Power and Energy Magazine*, vol. 12, no. 3, pp. 67–76, May 2014.

[52] A. Kwasinski, "Technological assessment of distributed generation systems operation during extreme events," in *2012 3rd IEEE International Symposium on Power Electronics for Distributed Generation Systems (PEDG)*, Jun. 2012, pp. 534–541.

[53] S. Lei, J. Wang, C. Chen, and Y. Hou, "Mobile Emergency Generator Pre-Positioning and Real-Time Allocation for Resilient Response to Natural Disasters," *IEEE Transactions on Smart Grid*, vol. PP, no. 99, pp. 1–1, 2017.

[54] C. Chen, J. Wang, and D. Ton, "Modernizing Distribution System Restoration to Achieve Grid Resiliency Against Extreme Weather Events: An Integrated Solution," *Proceedings of the IEEE*, vol. 105, no. 7, pp. 1267–1288, Jul. 2017.

Appendix A

Configuration and Control Code

A.1 zed_husky_mission.py

```
#!/usr/bin/env python


import rospy
import actionlib
from actionlib_msgs.msg import *
from geometry_msgs.msg import Pose, Point, Quaternion, ↩
   Twist
from move_base_msgs.msg import MoveBaseAction, ↩
   MoveBaseGoal
from tf.transformations import quaternion_from_euler
from math import radians, pi
```

```
class ZedMission():

    def __init__(self):
        rospy.init_node('ZedMission', anonymous=False)

        rospy.on_shutdown(self.shutdown)

        #publisher to manually control the robot (e.g. ↩
           to stop it)
        self.cmd_vel_pub = rospy.Publisher('/cmd_vel', ↩
           Twist, queue_size=5)

        #subscribing to the move_base action server
        self.move_base = actionlib.SimpleActionClient("↩
           move_base",MoveBaseAction)

        rospy.loginfo("Waiting for move-base action ↩
           server...")

        #allowing 60 seconds for the action server to ↩
           become available
        self.move_base.wait_for_server(rospy.Duration↩
           (60))

        rospy.loginfo("connected to move base server")
        rospy.loginfo("starting navigation")

    def MoveToWaypoint(self, x, y, theta):
        """
        x, y positions are relative to the origin of the↩
            lab qualysis coordinate frame
        theta should be given in radians
        """
```

```
#must convert Euler angle to quaternians
q_angle = quaternion_from_euler(0, 0, theta, ↩
   axes= 'sxyz')
q = Quaternion(*q_angle)

waypoint = Pose(Point(x,y,0.0),q)

#initializing the waypoint goal
goal = MoveBaseGoal()

#frame header to define goal pose
goal.target_pose.header.frame_id = 'odom'

#setting the time stamp to "now"
goal.target_pose.header.stamp = rospy.Time.now()

#setting the waypoint
goal.target_pose.pose = waypoint

#start move_base to go to waypoint
self.move_base.send_goal(goal)

#allowing for one minute to reach goal, ↩
   otherwise aborting mission
finished_within_time = self.move_base.↩
   wait_for_result(rospy.Duration(180))

if not finished_within_time:
    self.move_base.cancel_goal()
    rospy.loginfo("could not achieve goal in ↩
       time")
else:
    state = self.move_base.get_state()
    if state == GoalStatus.SUCCEEDED:
```

```
                rospy.loginfo("Goal achieved")

    def shutdown(self):
        rospy.loginfo("stopping the robot")
        #cancelling any active goals
        self.move_base.cancel_goal()
        rospy.sleep(2)
        #stop the robot
        self.cmd_vel_pub.publish(Twist())

if __name__ == '__main__':
    try:
        bob = ZedMission()
# bob.MoveToWaypoint(2.5,2,-1.6)
        bob.MoveToWaypoint(-1.33, -0.309, -0.6)
#         bob.MoveToWaypoint(10, 10, pi)
#         bob.MoveToWaypoint(0, 10, 3*pi/2)
#         bob.MoveToWaypoint(0, 0, 0)
    except rospy.ROSInterruptException:
        rospy.loginfo("Navigation Complete")
```

A.2 zed_costmap.launch

```
<?xml version="1.0"?>
<!-- This launch file based on examples from Guimaraes ↩
   et al "ROS Navigation: concepts and tutorial" 2016 and↩
    the Clearpath Robotics, Inc. Husky move_base.launch ↩
   file. Also http://official-rtab-map-forum.67519.x6.↩
   nabble.com/Using-obstacles-detection-nodelet-from-↩
   camera-node-point-cloud-td1191.html
```

```
-->
<launch>

<!-- TRANSFORMS -->
<node name="ZED_BL_tf" pkg="tf" type="↩
   static_transform_publisher" args="0.4 0 0.65 0 0 0 ↩
   base_link zed_cloud 10"/>

<!--<node name="QS_BL_tf" pkg="tf" type="↩
   static_transform_publisher" args="0 0 0.62 0 0 0 /↩
   base_link /QSpose 20"/>-->

<!--  <node name="lms1xx_base_laser_tf" pkg="tf" type="↩
   static_transform_publisher" args="0 0 0 0 0 0 ↩
   base_laser lms1xx 20"  />
  <node name="GPS_BL_tf" pkg="tf" type="↩
     static_transform_publisher" args="-0.4 0 0.65 0 0 0 ↩
     base_link gps 50"/>
-->
<!-- SENSOR CONFIGURATION -->

<!--  <node pkg="ZEDwLidarCostmap" name="↩
   qs_communication" type="communication.py" /> -->

  <!-- launch ZED -->
  <include file="$(find zed_wrapper)/launch/zed.launch" ↩
     />

  <!-- launch the proprioceptive odometry filter -->
  <node pkg="robot_localization" type="↩
     ekf_localization_node" name="ekf_localization" >
    <rosparam command="load" file="$(find ↩
       ZEDwLidarCostmap)/config/EncIMU_ekf.yaml" />
  </node>
```

```
  <!-- launch the navsat transform -->
<!--
  <node pkg="robot_localization" type="↩
     navsat_transform_node" name="navsat_transform" ↩
     output="screen">
    <rosparam>
      magnetic_declination_radians: 0.07
      roll_offset: 0
      pitch_offset: 0
      yaw_offset: 0
      zero_altitude: true
      broadcast_utm_transform: false
    </rosparam>
  </node>
-->

<!--try ENU package instead-->
  <node pkg="enu" type="from_fix" name="enu_from_fix">
    <remap from="fix" to="gps/fix"/>
    <param name="output_frame_id" value="odom"/>
<!--params for MEEM parking lot-->
<!--    <param name="datum_latitude" value="47.120105"/>
    <param name="datum_longitude" value="-88.548758"/>
    <param name="datum_altitude" value="200"/>-->

<!--params for APSLab parking lot-->
<!--    <param name="datum_latitude" value="47.169542"/>
    <param name="datum_longitude" value="-88.507616"/>
    <param name="datum_altitude" value="301"/>-->

<!-- params for Brians dirt lot -->
    <param name="datum_latitude" value="47.110041"/>
    <param name="datum_longitude" value="-88.528618"/>
```

```
    <param name="datum_altitude" value="189"/>

  </node>


  <!-- launch the GPS EKF node -->
<!--  <node pkg="robot_localization" type="↩
   ekf_localization_node" name="ekf_GPS" >
    <rosparam command="load" file="$(find ↩
       ZEDwLidarCostmap)/config/GPS_ekf.yaml" />
  </node>
-->
  <!-- Run a VoxelGrid filter to clean NaNs and ↩
     downsample the data -->
  <node pkg="nodelet" type="nodelet" name="pcl_manager" ↩
     args="manager" output="screen" />

 <node pkg="nodelet" type="nodelet" name="voxel_grid" ↩
    args="load pcl/VoxelGrid pcl_manager" output="screen"↩
    >
    <remap from="~input" to="/camera/point_cloud/cloud" ↩
       />
    <rosparam>
      filter_field_name: z
      filter_limit_min: -10
      filter_limit_max: 10
      filter_limit_negative: False
      leaf_size: 0.05
    </rosparam>
  </node>


  <!-- RTABMap obstacles_detection nodlet for ground ↩
     plane segmentation -->
```

```
  <node pkg="nodelet" type="nodelet" name="↩
     obstacles_detection" args="load rtabmap_ros/↩
     obstacles_detection pcl_manager" output="screen">
      <remap from="cloud"      to="/voxel_grid/output"/>
      <remap from="obstacles" to="/obstacles_cloud"/>
      <remap from="ground"    to="/ground_cloud"/>
      <param name="frame_id"             type="string" ↩
        value="base_link"/>
      <param name="map_frame_id"         type="string" ↩
        value="odom"/>
      <param name="wait_for_transform"   type="bool" ↩
        value="true"/>
      <param name="min_cluster_size"     type="int" ↩
        value="20"/>
      <param name="max_obstacles_height" type="double" ↩
        value="1.0"/>
  </node>

<!-- NAVIGATION -->

  <!-- Run the map server -->
  <!-- <arg name="map_file" default="$(find ↩
     ZEDwLidarCostmap)/maps/empty50x50m.yaml"/>
  <node name="map_server" pkg="map_server" type="↩
     map_server" args="$(arg map_file)" />
 -->
  <!-- launch move_base with configurations -->
  <arg name="base_global_planner" default="navfn/↩
     NavfnROS"/>
  <arg name="base_local_planner" value="↩
     teb_local_planner/TebLocalPlannerROS" />

  <node pkg="move_base" type="move_base" respawn="false"↩
      name="move_base" output="screen">
```

```
    <remap from="/odom" to="/odometry/filtered"/>

    <param name="base_global_planner" value="$(arg ↩
      base_global_planner)"/>
    <param name="base_local_planner" value="$(arg ↩
      base_local_planner)"/>
    <rosparam file="$(find ZEDwLidarCostmap)/config/↩
      planner.yaml" command="load"/>

    <!-- observation sources -->
    <rosparam file="$(find ZEDwLidarCostmap)/config/↩
      costmap_common.yaml" command="load" ns="↩
      global_costmap" />
    <rosparam file="$(find ZEDwLidarCostmap)/config/↩
      costmap_common.yaml" command="load" ns="↩
      local_costmap" />

    <!-- local costmap -->
    <rosparam file="$(find ZEDwLidarCostmap)/config/↩
      costmap_local.yaml" command="load" ns="↩
      local_costmap" />

    <!-- global costmap -->
    <rosparam file="$(find ZEDwLidarCostmap)/config/↩
      costmap_global.yaml" command="load" ns="↩
      global_costmap"/>
  </node>

<!--VOXEL VIZUALIZATION-->
 <node name="voxel_grid_2_point_cloud" pkg="costmap_2d" ↩
    type="costmap_2d_cloud">
   <remap from="voxel_grid" to="/move_base/local_costmap↩
     /obstacles_zed/voxel_grid"/>
```

```
    <remap from="marked_cloud" to="/move_base/↩
       local_costmap/obstacles_zed/marked_cloud"/>
    <remap from="unknown_cloud" to="/move_base/↩
       local_costmap/obstacles_zed/unknown_cloud"/>
 </node>

</launch>
```

A.3 costmap_common.yaml

```
footprint: [[-0.5, -0.33], [-0.5, 0.33], [0.5, 0.33], ↩
   [0.5, -0.33]]
footprint_padding: 0.02

robot_base_frame: base_link
# update_frequency: 2
# publish_frequency: 1.0
transform_tolerance: 2.5

resolution: 0.1

plugins:
  # - {name: static,                    type: "↩
     costmap_2d::StaticLayer"}
  - {name: obstacles_laser,          type: "↩
     costmap_2d::ObstacleLayer"}
  - {name: obstacles_zed,            type: "↩
     costmap_2d::VoxelLayer"}
  # - {name: obstacles_zed_clearing,  type: "↩
     costmap_2d::VoxelLayer"}
```

```
  - {name: inflation,                type: "↩
     costmap_2d::InflationLayer"}

max_obstacle_height: 1.1

# static:
#     map_topic: map
#     subscribe_to_updates: true

obstacles_laser:
  obstacle_range: 10
  raytrace_range: 11
  observation_sources: laser
  laser: {data_type: LaserScan, clearing: true, marking:↩
      true, topic: /scan, inf_is_valid: true}

obstacles_zed:

  obstacle_range: 3
  raytrace_range: 3.5
  origin_z: 0
  z_resolution: 0.1
  z_voxels: 12
  unknown_threshold: 0
  mark_threshold: 0
  publish_voxel_map: true
  observation_sources: point_cloud_sensor
  point_cloud_sensor: {sensor_frame: base_link, ↩
     data_type: PointCloud2, topic: /obstacles_cloud, ↩
     marking: true, clearing: true, min_obstacle_height: ↩
     0.1, max_obstacle_height: 99999.0}

# obstacles_zed_clearing:
```

```
#    obstacle_range: 4
#    raytrace_range: 5
#    origin_z: 0
#    z_resolution: 0.1
#    z_voxels: 12
#    unknown_threshold: 11
#    mark_threshold: 1
#    publish_voxel_map: false
#    observation_sources: point_cloud_sensor
#    point_cloud_sensor: {sensor_frame: zed_tracked_frame
   , data_type: PointCloud2, topic: /camera/point_cloud/
   cloud, marking: false, clearing: true}

inflation_layer:
    inflation_radius: 0.3
```

A.4 costmap_global.yaml

```
global_frame: odom
#global_frame: /zed_initial_frame
rolling_window: true
# track_unknown_space: true
static_map: false

update_frequency: 1
publish_frequency: 1

width: 20.0
height: 20.0
```

```
plugins:
  - {name: obstacles_laser,          type: "↩
     costmap_2d::ObstacleLayer"}
```

A.5 costmap_local.yaml

```
global_frame: odom
rolling_window: true

update_frequency: 7
publish_frequency: 5

width: 6.0
height: 6.0

plugins:
  - {name: obstacles_laser,            type: "↩
     costmap_2d::ObstacleLayer"}
  - {name: obstacles_zed,            type: "↩
     costmap_2d::VoxelLayer"}
  # - {name: obstacles_zed_clearing,            type: "↩
     costmap_2d::VoxelLayer"}
  - {name: inflation,                  type: "↩
     costmap_2d::InflationLayer"}
```

A.6 EncIMU_ekf.yaml

```
odom_frame: odom
base_link_frame: base_link
world_frame: odom

two_d_mode: true

frequency: 50

odom0: husky_velocity_controller/odom
odom0_config: [false, false, false,
               false, false, false,
               true, true, false,
               false, false, true,
               false, false, false]
odom0_differential: false
odom0_queue_size: 10

imu0: imu/data
imu0_config: [false, false, false,
              false, false, true,
              false, false, false,
              false, false, true,
              true, false, false]
imu0_differential: true
imu0_queue_size: 10
imu0_remove_gravitational_acceleration: true

odom1: enu
odom1_config: [true,  true,  false,
               false, false, false,
               false, false, false,
               false, false, false,
               false, false, false]
odom1_queue_size: 10
```

```
odom1_differential: false

#pose0: QSpose
#pose0_config: [true, true, true,
#               true, true, true,
#               false, false, false,
#               false, false, false,
#               false, false, false]
#pose0_differential: false
#pose0_queue_size: 10
```

A.7 planner.yaml

```
controller_frequency: 5.0
recovery_behaviour_enabled: true

NavfnROS:
  allow_unknown: true # Specifies whether or not to ↩
     allow navfn to create plans that traverse unknown ↩
     space.
  default_tolerance: 0.2 # A tolerance on the goal point↩
      for the planner.
  visualize_potential: true

TebLocalPlannerROS:

 odom_topic: odom
 map_frame: /odom

 # Trajectory
```

```
teb_autosize: True
dt_ref: 0.3
dt_hysteresis: 0.1
global_plan_overwrite_orientation: True
max_global_plan_lookahead_dist: 3.0
feasibility_check_no_poses: 5

# Robot

max_vel_x: 0.75
max_vel_x_backwards: 0.25
max_vel_theta: 1.0
acc_lim_x: 0.5
acc_lim_theta: 1
min_turning_radius: 0.0
footprint_model: # types: "point", "circular", "↩
   two_circles", "line", "polygon"
  type: "point"
  radius: 0.2 # for type "circular"
  line_start: [-0.3, 0.0] # for type "line"
  line_end: [0.3, 0.0] # for type "line"
  front_offset: 0.2 # for type "two_circles"
  front_radius: 0.2 # for type "two_circles"
  rear_offset: 0.2 # for type "two_circles"
  rear_radius: 0.2 # for type "two_circles"
  vertices: [ [0.25, -0.05], [0.18, -0.05], [0.18, ↩
     -0.18], [-0.19, -0.18], [-0.25, 0], [-0.19, 0.18], ↩
     [0.18, 0.18], [0.18, 0.05], [0.25, 0.05] ] # for ↩
     type "polygon"

# GoalTolerance

xy_goal_tolerance: 0.2
```

```
yaw_goal_tolerance: 0.1
free_goal_vel: False

# Obstacles

min_obstacle_dist: 0.4
include_costmap_obstacles: True
costmap_obstacles_behind_robot_dist: 1.0
obstacle_poses_affected: 10
costmap_converter_plugin: ""
costmap_converter_spin_thread: True
costmap_converter_rate: 5

# Optimization

no_inner_iterations: 5
no_outer_iterations: 4
optimization_activate: True
optimization_verbose: False
penalty_epsilon: 0.1
weight_max_vel_x: 2
weight_max_vel_theta: 1
weight_acc_lim_x: 1
weight_acc_lim_theta: 1
weight_kinematics_nh: 1000
weight_kinematics_forward_drive: 1
weight_kinematics_turning_radius: 1
weight_optimaltime: 1
weight_obstacle: 50
weight_dynamic_obstacle: 10 # not in use yet
#selection_alternative_time_cost: False # not in use ↩
   yet

# Homotopy Class Planner
```

```
enable_homotopy_class_planning: True
enable_multithreading: True
simple_exploration: False
max_number_classes: 4
roadmap_graph_no_samples: 15
roadmap_graph_area_width: 5
h_signature_prescaler: 0.5
h_signature_threshold: 0.1
obstacle_keypoint_offset: 0.1
obstacle_heading_threshold: 0.45
visualize_hc_graph: False
```

A.8 communication.py

```
#!/usr/bin/env python

import socket
import numpy as np
import sys
import struct
from time import strftime
import logging
from os import getcwd
import rospy
from geometry_msgs.msg import Pose, Point, Quaternion, ↩
   PoseWithCovarianceStamped
import tf
```

```
class LocalizationError(Exception):
    pass


def setupLogger(verbose, name=[]):
    logger = logging.getLogger(name)
    logger.setLevel(logging.DEBUG)
    formatter = logging.Formatter(
        '%(asctime)s-%(process)d-%(levelname)s-%(message)s')
    fh = logging.FileHandler(
        filename='CommunicationLogs_' + strftime("%d%b%Y") + '.log')
    fh.setLevel(logging.INFO)
    fh.setFormatter(formatter)
    logger.addHandler(fh)
    ch = logging.StreamHandler()
    if verbose:
        ch.setLevel(logging.DEBUG)
    else:
        ch.setLevel(logging.WARNING)
    ch.setFormatter(formatter)
    logger.addHandler(ch)
    return logger

# logging.basicConfig(level=logging.DEBUG, datefmt='%m/%d/%Y %I:%M:%S %p')


def getIP():
    # Get the IP address of the computer executing this file
    temp_sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

```
        temp_sock.setsockopt(socket.SOL_SOCKET, socket.↩
            SO_REUSEADDR, 1)
        temp_sock.connect(('192.168.0.1', 0))
        ip_address = temp_sock.getsockname()[0]
        temp_sock.close()
        return ip_address


class MessageVerification(object):
    def __init__(self, verbose=False, name='root.↩
        messaging'):
        self.logger = setupLogger(verbose, name)

        self.logger.debug(
            "All the errors of communication module will↩
                be logged in %s", getcwd())
        self.logger.debug(
            "You can disable the verbosity by removing ↩
              the 'verbose=True' from the ↩
              MessageVerification() constructor.")

        self.sock = socket.socket(socket.AF_INET, socket↩
            .SOCK_STREAM)
        self.sock.setsockopt(socket.SOL_SOCKET, socket.↩
            SO_REUSEADDR, 1)
        self.timeout = None
        self.port = 65400
        self.backlog = 1  # maximum number of queued ↩
            connections

    def setTimeout(self, seconds):
        self.timeout = seconds
        self.logger.info("Connection timeout changed to ↩
            %s", seconds)
```

```
def setPort(self, port_number):
    self.logger.info(
        "Connection port has been changed to %s from↩
            %s", self.port_number, self.port)
    self.port = port_number
    self.logger.info(
        "Make sure you also configure both ends of ↩
           communication.")

def setNumberOfConnections(self, number):
    self.backlog = number
    self.logger.info(
        "Maximum number of queued connections ↩
           changed to %s", number)

def connectToServer(self, ip_address):
    self.logger.info("Connecting to the server on %s↩
       ", ip_address)
    self.sock.settimeout(self.timeout)
    server_address = (ip_address, self.port)
    try:
        self.sock.connect(server_address)
        self.logger.info('Connected To %s.', ↩
           server_address)
    except socket.error:
        logging.exception("Error in ↩
           MessageVerification")
        print 'Connection failed. Check server.'
        raise

def connectToClient(self, ip_address=[]):
    if not ip_address:
        ip_address = getIP()
```

```
        self.logger.debug("Waiting for the client to be ↩
            connected.")
        self.sock.settimeout(self.timeout)
        server_address = (ip_address, self.port)
        self.sock.bind(server_address)
        # NOTE: maybe have to be changed in future
        self.sock.listen(self.backlog)
        try:
            self.sock, address = self.sock.accept()
            self.logger.info("Connection accepted from %↩
                s", address)
        except:
            logging.exception("Error in ↩
                MessageVerification")
            print 'Connection failed. Check server.'
            raise

    def sendMessage(self, text):
        self.logger.debug("Sending: %s", text)
        self.sock.sendall(text)
        # sleep(1)
        self.logger.info("Sent: %s", text)

    def verifyMessage(self, text):
        self.logger.info("Waiting for message.")
        received_text = self.sock.recv(4096)
        self.logger.debug("Received: %s", received_text)
        if received_text == text:
            self.logger.info("Successful message ↩
                verification: %s", text)
            return True
        else:
            self.logger.info(
```

```
                "Failed to receive '%s', received '%s' ↩
                    instead.", text, received_text)
            return False

    def close(self):
        self.logger.debug("Closing the connection.")
        self.sock.close()
        self.logger.info("Connection closed.")


class NaslabNetwork(object):

    def __init__(self, ip_address='192.168.0.21'):
        self.sock = socket.socket(socket.AF_INET, socket↩
            .SOCK_STREAM)
        self.sock.setsockopt(socket.SOL_SOCKET, socket.↩
            SO_REUSEADDR, 1)
        self.sock.settimeout(1)
        server_address = (ip_address, 1895)
        print >>sys.stderr, 'Connecting To %s Port %s' %↩
            server_address
        self.sock.connect(server_address)
        self.degree_to_rad = np.pi / 180

    def getStates(self, num):
        # get position
        check = struct.unpack('<B', self.sock.recv(1))↩
            [0]
        if check is not 2:
            print 'Warning: Bad Qualisys Packet'
            pose_msg_x = float('nan')
        recieved_data = self.sock.recv(4096)
        if len(recieved_data) < 12:
            print 'bad 2'
```

```
            pose_msg_x = float('nan')
        num_byte = num * 24
        pose_msg_x = struct.unpack(
            '<f', recieved_data[num_byte:num_byte + 4])↩
               [0]
        pose_msg_y = struct.unpack(
            '<f', recieved_data[num_byte + 4:num_byte + ↩
               8])[0]
        pose_msg_theta = struct.unpack(
            '<f', recieved_data[num_byte + 12:num_byte +↩
                16])[0]
        # print num, pose_msg_theta
        return pose_msg_x, pose_msg_y, pose_msg_theta

    def close(self):
        self.sock.close()


class LabNavigation(object):

    def __init__(self, ip_address='192.168.0.21', port↩
       =1895):
        self.sock = socket.socket(socket.AF_INET, socket↩
           .SOCK_STREAM)
        self.sock.setsockopt(socket.SOL_SOCKET, socket.↩
           SO_REUSEADDR, 1)
        self.sock.settimeout(10)
        server_address = (ip_address, port)
        # print >>sys.stderr, 'Connecting To %s Port %s'↩
            % server_address
        try:
            self.sock.connect(server_address)
        except socket.error:
            print 'Connection failed. Check server.'
```

```
            raise

    def getStates(self, num):
        # get position
        self.sock.send(str(num))
        try:
            packed_buffer = self.sock.recv(25)
        except socket.error as e:
            if e.errno == 4:
                raise KeyboardInterrupt(
                    'There seems to be KeyboardInterrupt↩
                        during socket receiving data.')
            else:
                raise

        agent_id, x, y, z, roll, pitch, yaw = struct.↩
           unpack(
            '<Bffffff', packed_buffer)
        if x != x:
            # raise LocalizationError("Information of ↩
               body is missing")
            print 'Information of body is missing, ↩
               waiting for data'
        return x, y, z, yaw, pitch, roll

    def setTimeout(self, value):
        self.sock.settimeout(value)

    def close(self):
        self.sock.close()


class PublishPose(object):
```

```
    """Create ROS publisher of type geometry_msgs.msg.↩
       Pose and instantiate LabNavigation"""

    def __init__(self, num):

        QSpose = rospy.Publisher('QSpose', Pose, ↩
           queue_size=10)
        rospy.init_node('Qualisys', anonymous=True)
        rate = rospy.Rate(10)
        self.QSraw = LabNavigation()
        while not rospy.is_shutdown():
            x, y, z, yaw, pitch, roll = self.QSraw.↩
               getStates(num)
            msg = Pose()
            msg.position = Point(x, y, z)
            msg.orientation = Quaternion(
                *tf.transformations.↩
                   quaternion_from_euler(roll, pitch, yaw↩
                   , 'sxyz'))
            rospy.loginfo(msg)
            QSpose.publish(msg)
            rate.sleep()
        self.QSraw.close()

class PoseWCov(object):
    """Create ROS publisher of type geometry_msgs.msg.↩
       PoseWithCovarianceStamped using LabNavigation"""

    def __init__(self, num):

        QSpose = rospy.Publisher('/QSpose', ↩
           PoseWithCovarianceStamped, queue_size=10)
        rospy.init_node('Qualisys', anonymous=True)
        rate = rospy.Rate(100)
```

```
        self.QSraw = LabNavigation()
        while not rospy.is_shutdown():
            x, y, z, yaw, pitch, roll = self.QSraw.↩
               getStates(num)
            # print (yaw,pitch,roll)
            msg = PoseWithCovarianceStamped()
            msg.header.stamp = rospy.Time.now()
            msg.header.frame_id = "odom"
            msg.pose.pose.position = Point(x, y, z)
            msg.pose.covariance=[0.000000000025, 0.0, ↩
               0.0, 0.0, 0.0, 0.0, 0.0, 0.000000000025, ↩
               0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ↩
               0.000000000025, 0.0, 0.0, 0.0, 0.0, 0.0, ↩
               0.0, 0.000000000025, 0.0, 0.0, 0.0, 0.0, ↩
               0.0, 0.0, 0.000000000025, 0.0, 0.0, 0.0, ↩
               0.0, 0.0, 0.0, 0.000000000025]
            msg.pose.pose.orientation = Quaternion(
                *tf.transformations.↩
                   quaternion_from_euler(roll, pitch, yaw↩
                   , 'sxyz'))
            rospy.loginfo(msg)
            QSpose.publish(msg)
            rate.sleep()
        self.QSraw.close()



if __name__ == '__main__':
        # nav1 = LabNavigation()
        # print nav1.getStates(0)
        # nav1.ReSetTimeout(50)

    try:
        bob = PoseWCov(5)
```

```
    except rospy.ROSInterruptException:
        pass
```

A.9 gpslogger.py

```
#!/usr/bin/env python

import xlsxwriter
from datetime import datetime
import rospy
from sensor_msgs.msg import NavSatFix



class GPSwb():

  def __init__(self):
    wbname = str(datetime.now()) + ".xlsx"
    print wbname
    self.workbook = xlsxwriter.Workbook(wbname)
    self.worksheet = self.workbook.add_worksheet()
    self.row = 0
    self.col = 0

bob = GPSwb()

rospy.init_node('logGPStoCSV',disable_signals=True)

def callback(data):
# print data.latitude()
```

```
    lat=data.latitude
    lon=data.longitude
    stuff=str(lat) + "," + str(lon)
    bob.worksheet.write(bob.row, bob.col,stuff)
    bob.row += 1
    print bob.row
    print stuff


rospy.Subscriber('gps/fix',NavSatFix,callback)
try:
    rospy.spin()
except Exception:
    pass
print "DONE!!!"
bob.workbook.close()




# def logGPSforMaps():
#     print "hi"
#     bob=GPSwb()
#     rospy.init_node('logGPStoCSV')
#     rospy.Subscriber('gps/fix',NavSatFix,callback)
#     rospy.spin()

#     workbook.close()


# logGPSforMaps()
```

A.10 arduinoRosseriaSpool.ino

```
/*
 * rosserial Servo Control Example
 *
 * This sketch demonstrates the control of hobby R/C ↩
    servos
 * using ROS and the arduiono
 *
 * For the full tutorial write up, visit
 * www.ros.org/wiki/rosserial_arduino_demos
 *
 * For more information on the Arduino Servo Library
 * Checkout :
 * http://www.arduino.cc/en/Reference/Servo
 */

#if (ARDUINO >= 100)
#include <Arduino.h>
#else
#include <WProgram.h>
#endif

#include <ros.h>
#include <geometry_msgs/Twist.h>
#include <avr/io.h>

ros::NodeHandle  nh;

void servo_cb( const geometry_msgs::Twist& cmd_msg){
```

```
  int spoolSpeed = -125*(abs(cmd_msg.linear.x)+1.65*abs(cmd_msg.angular.z))+375;
  OCR1A = constrain(spoolSpeed,249,375);

  digitalWrite(13, HIGH-digitalRead(13));  //toggle led
}


ros::Subscriber<geometry_msgs::Twist> sub("husky/cmd_vel", servo_cb);

void setup(){
  pinMode(13, OUTPUT);
  pinMode(9, OUTPUT);

  nh.initNode();
  nh.subscribe(sub);

  DDRD |= _BV(5) | _BV(6);
  TCCR1A = _BV(COM1A1) | _BV(COM1B1) | _BV(WGM11);
  TCCR1B = _BV(WGM13) | _BV(WGM12) | _BV(CS11) | _BV(CS10);
  ICR1 = 4999;
  OCR1A = 375;
}

void loop(){
  nh.spinOnce();
  delay(1);
}
```