University Rover Challenge: Tutorials and Team Survey

Daniel Snider*, Matthew Mirvish†, Michal Barcis‡, and Vatan Aksoy Tezer§

Abstract. In this tutorial chapter we present a guide to building a robot through 11 tutorials. We prescribe simple software solutions to build a wheeled robot and manipulator arm that can autonomously drive and be remotely controlled. These tutorials are what worked from several teams at the University Rover Challenge 2017 (URC). The tutorials provide a quick start guide to using existing Robot Operating System (ROS) tools, others are new contributions, or explain challenging topics such as wireless communication and robot administration. We also present the results of an original survey of 8 competing teams to gather information about trends in URC's community which consists of hundreds of university students on over 80 teams. Additional topics include satellite mapping of robot location (mapviz), GPS integration (original code) to autonomous navigation (move_base), and more. We hope to promote collaboration and code reuse.

Keywords: Outdoor robot, Arm control, Autonomous navigation, Teleoperation, Panoramas , Image Overlay, Wireless, GPS, Robot administration

1 Introduction

The University Rover Challenge (URC) is an engineering design competition held in the Utah desert that requires large teams of sometimes 50 or more university students. Students spend a year preparing and building from scratch a teleoperated and autonomous rover with an articulated arm. This chapter gives readers an overview of eight rover designs used at the URC, as well as a deep dive into contributions from three design teams: Team R3 (Ryerson University), Team Continnuum (University of Wroclaw), and Team ITU (Istanbul Technical University). We detail how to build a rover by piecing together existing code, lowering the challenges for new Robot Operating System (ROS)[12] users. The contributions of our book chapter include 11 short tutorials, 7 new ROS packages, and an original survey of 8 teams after they participated in the URC 2017 rover competition.

*Ryerson University, Toronto, Canada, danielsnider12@gmail.com
†Bloor Collegiate Institute, Toronto, Canada, matthewmirvish@hotmail.com
‡University of Wroclaw, Poland, mbarcis@mbarcis.net
§Istanbul Technical University, Turkey, vatanaksoytezer@gmail.com

1.1 Motivation

At URC there is a rule [1] limiting the budget that is allowed to be spent by teams on their rover to $15,000 USD. Therefore students typically engineer parts and software themselves rather than buying. This makes ROS's free and open source ecosystem a natural fit for teams to cut costs and avoid re-engineering common robotics software. At URC 2017, Team R3 spoke to several teams who are not using ROS but want to, and others who want to expand their use of it.

Several authors of this chapter joined URC because they are passionate about hands on learning and ROS. After our URC experience we are even more confident that ROS is an incredible tool for building advanced robotics quickly with strong tooling that makes administering ROS robots enjoyable.

Our motivation comes from a passion to share lessons that enable others to build better robots. Software is eating the world and there is a large positive impact that can be made in the application of software interacting with the physical world. Our software is freely shared so it can have the largest unencumbered impact and usefulness.

1.2 Main Objectives

Aim one is to help ROS users quickly learn new capabilities. Therefore, the bulk of our contributions are in the form of tutorials. Tutorials contained in this chapter are more relevant and interesting by giving them in context of the URC competition (throughout the chapter). Readers can better assess the usefulness of the tutorials by comparing those solutions given to the other approaches that our survey of eight other teams have revealed in section 3. The team survey results acts as overview and the 12 tutorials act as a deep drive into the technical implementations of 3 teams: Team R3 from Ryerson University in Toronto, Canada, and Team Continuum from the University of in Wroclaw, Poland, and the ITU Rover Team from Istanbul Technical University in Turkey. Original ROS packages are documented with examples, installation and usage instructions, and implementation details. At the end of the chapter readers should have a better sense of what goes into building a rover and of the challenges of the URC.

1.3 Overview of Chapter

Following the introduction and background sections, a wide angle look at rover systems with a survey and two case-studies is presented. Then specific tutorials delve into new packages and implementations mentioned in the case studies and team survey.

Section 2 "Background" provides an explanation of the URC rover competition and some of its rules.

Section 3 "Survey of URC Competing Teams" presents the results of an original survey of 8 teams who competed at URC 2018. It details each team's

[1] URC 2017 Rules `http://tinyurl.com/urc-rules`

rover computer setup, ROS packages, control software, and avionics hardware for communication, navigation, and monitoring.

Section 4 “Case Study: Continuum Team” gives a case-study of their rover and what lead them to a second place result at the URC 2017 competition.

Section 5 “Case Study: Team R3” gives a case-study of the ROS software architecture used in Team R3’s Rover. It provides the big picture for some of the tutorials in later sections to dive into more detailed explanations.

Section 6 “Tutorial: Autonomous Waypoint Following” details the usage of a new, original ROS package that will queue multiple move_base navigation goals and navigate to them in sequence. This helps URC teams in the autonomous terrain traversal missions.

Section 7 “Tutorial: Image Overlay Scale and Compass” details a new, original ROS package that meets one of the URC requirements to overlay an image of a compass and scale bar on imagery produced by the rover. It is intended to add context of the world around the rover.

Section 8 “Tutorial: A Simple Drive Software Stack” details the usage and technical design of a new, original ROS package that will drive PWM motors given input from a joystick in a fashion known as skid steering. The new package contains Arduino firmware and controls a panning servo so that a teleoperator can look around with a camera while driving.

Section 9 “Tutorial: Velocity Controlled Arm” details the usage and technical design of a new, original ROS package that will velocity control arm joint motors, a gripper, and camera panning. The new package contains Arduino firmware to control PWM motors and a servo for the camera.

Section 10 “Tutorial: Autonomous Recovery after Lost Communications” details the technical design and usage of a new, original ROS package that uses ping to determine if the robot has lost connection to a remote base station. If the connection is lost then motors will be stopped or an autonomous navigation goal will be issued so to reach a configurable location.

Section 11 “Tutorial: Stitch Panoramas with Hugin” details the usage and technical design of a new, original ROS package that will create panoramic images using ROS topics. At the URC competition teams must document locations of interest such as geological sites with panoramas.

Section 12 “Tutorial: GPS Navigation Goal” details the usage and technical design of a new, original ROS package that will convert navigation goals given in latitude and longitude GPS coordinates to ROS frame coordinates.

Section 13 “Tutorial: Wireless Communication” gives a detailed explanation of the primary and backup wireless communication setup used between ITU Rover Team’s rover and base station for up to 1 km in range.

Section 14 “Tutorial: Autonomous Navigation by Team R3” explains the technical architecture of the autonomous system used at URC 2017 by Team R3 from Ryerson University, Toronto. It is based on the ZED stereo camera, the RTAB-Map ROS package for simultaneous localization and mapping (SLAM), and the move_base navigation ROS package.

Section 15 "Tutorial: MapViz Robot Visualization Tool" presents the MapViz ROS package and illustrates how a top-down, 2D visualization tool with support for satellite imagery can be useful for outdoor mobile robotics and URC. Our original Docker container created to easy the use of MapViz with satellite imagery is also documented.

Section 16 "Tutorial: Effective Robot Administration" discusses a helpful pattern for robot administration that makes use of tmux and tmuxinator to roslaunch many ROS components in separate organized terminal windows. This makes debugging and restarting individual ROS components easier.

Section 17 "Conclusion" ends with the main findings of the chapter and with ideas for further collaboration between URC teams and beyond.

2 Background

2.1 About the University Rover Challenge

The University Rover Challenge is an international robotics competition run annually by The Mars Society. Rovers are built for a simulated Mars environment with challenging missions filling three days of competition. It is held in the summer time at the very hot Mars Desert Research Station, in Utah. There were 35 rovers and more than 500 students from seven countries that competed in the 2017 competition[2]. The winning team's rover can be seen in Fig. 1 shows an example

Fig. 1. A URC competition judge watches as the winning team of 2017, Missouri University of Science and Technology, completes the Equipment Servicing Task.

[2] URC 2017 competition score results and standings `http://urc.marssociety.org/home/urc-news/americanroverearnsworldstopmarsrovertitle`

Rovers must be operated remotely by team members who cannot see the rover or communicate with people in the field, such actions are punished by penalty points. Teams must bring and setup their own base station in a provided trailer or shelter and a tall communication mast nearby for wireless communication to the rover. The rover may have to travel up to 1km and even leave direct line of sight to the wireless communication mast.

The idea is that the rover is on Mars (the Utah desert serves as a substitute) performing scientific experiments and maintenance to a Mars base. An assumption is made that the rovers are being operated by astronauts on or orbiting Mars rather than on Earth and therefore there is no major communication delay.

Fig. 2. Group photo of URC 2017 finalists at the Mars Desert Research Station in Utah.

2.2 University Rover Challenge Tasks

The URC rules[3] detail four tasks. The science cache task involves retrieving and testing subsurface soil samples without contamination. For this the rover must have an auger to drill into the hard desert soil. After the science task teams present to a panel of judges about their scientific findings. The evidence collected by the rovers cameras, soil collection, and minimum three 3 sensors (e.g. temperature, humidity, pH) is presented in a way that purports the possibility of water and life on Mars.

The extreme retrieval and delivery task requires teams to search out tools and marked rocks in the desert and then use an arm on the rover to bring them back to the base station. The equipment servicing task has teams perform finer

[3]URC 2017 Rules `http://tinyurl.com/urc-rules`

manipulations with their rover arm to start a fake generator. This consists of pouring a fuel canister, pressing a button, flicking a switch and more manipulation tasks.

In the autonomous task teams must start with their rover within 2 m of the designated start gate and must autonomously navigate to the finish gate, within 3 m. Teams are provided with GPS coordinates for each gate and the gates are marked with a tennis ball elevated 10 cm to 50 cm off the ground and are not typically observable from a long distance. Teams may conduct teleoperated excursions to preview the course but this will use their time. Total time for this task is 75 minutes per team and the total distance of all stages will not exceed 1000 m.

Teams must formally announce to judges when they are entering autonomous mode and not transmit any commands that would be considered teleoperation, although they can monitor video and telemetry information sent from the rover. On-board systems are required to decide when the rover has reached the finish gate.

The newer 2018 rules[4] are very similar, but with more difficult manipulation tasks such as typing on a keyboard and a more demanding autonomous traversal challenge that explicitly calls for obstacle avoidancesomething that Team R3 had last year and is explained in detail in section 14.

2.3 Planetary Rovers beyond the Competition

Although the University Rover Challenge (URC) competition is a simulation of planetary rovers for Mars, there are significant differences between student built URC rovers and realistic planetary rovers. For example URC teams are limited in budget, manpower and engineering knowledge level, Martian environmental conditions such as radiation, low atmospheric pressure, and very low oxygen levels prompt, and a lack of communication and navigation systems found on Earth such as GPS.

The following paragraphs compare the systems of NASA's Curiosity rover[8][7] and URC rovers. When comparing the technology difference between the production year of Curiosity (2011) and now should also be kept in mind.

On Board Computer (OBC) The Curiosity Rove carries redundant 200MHz BAE RAD750 CPUs, which is a special CPU that is designed to work in high radiation environments and has 256MB RAM, 2GB Flash, 256KB EEPROM. The CPU is currently running a real time operating system called VxWorks. Interestingly, most of the URC rovers use on board computers that have more features and power than the planetary rovers due to the improvement of technology over the years. For example, Team R3 uses a Jetson TX1, running Ubuntu 16.04 which has 4GB RAM.

[4]URC 2018 Rules `http://tinyurl.com/urc-rules2018`

Autonomous Navigation Although URC rovers and Curiosity have several common sensors, the lack of GPS leads means that autonomous navigation algorithms work differently. Both of them use their Internal Measurement Units (IMU) and cameras to navigate, combining the odometry from internal sensors, visual odometry and some custom image processing algorithms to reach their target. The difference is that while URC rovers can rely on their GPS to navigate, Curiosity must rely on its position data from internal sensors only. Also, as the Curiosity is navigating on the harsh Mars terrain it decides to navigate through the better terrain using complex image processing algorithms.

Cameras Curiosity has 17 cameras that are used for various objectives, such as obstacle avoidance, navigation and science. Some of these cameras are very high resolution due to their scientific purposes. URC rovers generally has less cameras, such as 2 or 3 and less resolution is used.

Wireless Communication Curiosity can communicate directly with the Earth with its X-band (7-12 GHz) communication modules or it can communicate with satellites orbiting Mars, specifically the Mars Reconnaissance Orbiter (MRO) or Odyssey over a 400MHz UHF link. URC rovers generally prefer a 2.4 GHz UHF link to communicate with their ground stations.

Power Despite most of planetary rovers using solar power, Curiosity carries a 4.8 kg of radioactive plutonium-238 to provide energy to its instruments during its planned mission time of several years. On the other hand, URC rovers are generally powered with Li-Po or Li-Ion batteries that only last for several hours.

2.4 Prerequisite Skills for Tutorials

The tutorials in this chapter expect the following skills at a basic level.

- ROS basics (such as `roslaunch`)
- Command line basics (such as `bash`)
- Ubuntu basics (such as `apt` package manager)

3 Survey of URC Competing Teams

Fig. 3 and 3 shows eight teams that were surveyed for their rover computer setup, ROS packages, control software, and avionics hardware (for communication, navigation, and monitoring). The team survey results have been edited and condensed for publication.

This survey serves the purpose of providing a broad overview of components and rover development styles before diving into a few detailed implementations in subsequent sections provided by the teams who authored this chapter: Team

R3 (Ryerson University), Team Continnuum (University of Wroclaw), and Team ITU (Istanbul Technical University).

Several trends that emerged from the survey are interesting to note. Of the 8 teams surveyed, teams that used Raspberry Pis or STM microcontrollers all placed better than teams that used Arduinos or Teensy microcontrollers. For teleoperator input, Logitech controllers or joysticks were extremely popular and used by all teams. Teams most often expressed difficulty using IMUs or not testing their rover enough. A wide variety of autonomous systems were experimented with: from custom OpenCV implementations, to using existing vision-based obstacle avoidance software (RTAB-Map), to GPS only approaches.

The winning approach of the Mars Rover Design Team from Missouri University of Science and Technology utilized a large number of custom solutions. Their high quality solutions and extremely comprehensive testing is exemplified in just one case by developing a custom UDP communication software. Dubbed "RoveComm", their communication system can reduce latency and increase video quality and was key to successful teleoperation.

4 Case Study: Continuum Team

In the following section a case study is presented of the Continuum team (University of Wroclaw) and their rover, Aleph1. It is particularly interesting one, because they managed to score second place during the URC 2017 and had multiple other successes since they debuted in 2015. Michal Barcis decided to share with us some insights about their rover and his opinions on the competition.

The differences between team Continuum and other participants will be identified in order to find the key strengths that supported their achievements.

4.1 Recipe for success

The teams, especially the ones that managed to place themselves in the first ten places during the competition, do not differ very much. Both software and hardware solutions are similar, many teams also decided to implement programs using ROS. We will try to identify some features that distinguish the Continuum team and let the reader decide which of them, if any, were the most advantageous.

One of the key differences is the size of the team. On average there were only around 12 members working on the rover during the period between 2014 and 2017. This makes the Continuum one of the smallest groups on the URC. Such situation has both positive and negative effects the smaller workforce means each person has more work to do and there is less shared knowledge, but also makes each member more important and increases motivation. Each of the key components in the rover had a person responsible for it.

There is one especially interesting hardware component that the Continuum team decided to do a bit differently than other teams and the team was often asked about. It is the choice of cameras. Aleph1 is equipped with inexpensive Raspberry Pi cameras[2]. Although much better devices in terms of specification

	Mars Rover Design Team	Team Continuum	Cornell Mars Rover	ITU Rover Team	UWRT Robotics	Ryerson Rams Robotics (R3)	SJSU Robotics	Team Anveshak
School Name	Missouri University of Science and Technology	University of Wroclaw, Poland	Cornell University, USA	Istanbul Technical University, Turkey	University of Waterloo, Canada	Ryerson University, Canada	San Jose State University, USA	Indian Institute of Technology, Madras
Final Score (Rank)	403.4 **(1)**	336.3 (2)	264.1 (11)	243.1 (13)	225.7 (15)	190.9 (21)	164.3 (26)	151.4 (29)
Computers on rover	Raspberry Pi, TIVA-C Connected, MSP-432, Launchpad-C2000	A Banana Pi, 3x Raspberry Pi, 1x Jetson (optionally), multiple STM microcontrollers	A Intel NUC N82E16856102053, and 8x PIC32 MX530F128H microcontroller	A Raspberry Pi 3 with 64gb SD card running Ubuntu 16.04, STM32F103 microcontrollers	A FitPC miniature fanless PC	A Jetson TX1 with 32 GB SD card, Ubuntu 16.04, and 2x Arduino Mega microcontrollers	Odroid XU4, and Teensy 3.2 microcontroller	A Thinkpad T460 laptop running Ubuntu 14.04, and Arduino microcontrollers
Joysticks	Xbox Controller, Logitech Extreme 3D Pro	Logitech Gamepads	Logitech Gamepad F310, Thrustmaster VG T16000M FCS Joystick	2x Logitech Extreme 3D Pro, one for driving and one for the arm	2x Logitech joysticks for the arm, and an Xbox controller for driving	Xbox 360 Controller for drive, Logitech Extreme 3D Pro for arm	Logitech Extreme 3D Pro Joystick	2x Logitech F310 Gamepads, one for telemetry control and one for auger/arm
Cameras	Lorex, Sony EFFIO CCD Superhead	Standard Raspberry Pi cameras and two with wide angle lenses	Logitech HD Laptop Webcam C615, x264 video encoding	5 IP cameras used for security and an Xbox 360 Kinect v1 for image processing and fake laser	2x Pointgrey cameras, 1x USB Camera	ZED depth camera, 2x BL170 degree fisheye cameras	CCD 700TVL Composite video cameras (RunCam Swift 2.0)	SJ-CAM, IP-Camera, and a Logitech webcam. Cameras were interfaced using the "motion" Linux package, though it lags and quality was not great
GPS	MTK 3339	Ublox GPS	USGlobalsat BU-353-S4	Radiolink M8N	Microstrain	Linx FM Series GPS Receiver	UBlox GPS 7	ROS All Sensors Android App
IMU	LSM9DS1	Tried multiple units, nothing really worked	SparkFun SEN-13762, chip: MPU-9250	GY-80	Microstrain	MPU-9250 module, couldn't get it working	BNO055	ROS All Sensors Android App on Moto Play G4 phone
Software Packages	Energia, TI motorware, OpenCV	ROS kinetic with joint_state_ controller, rviz, rqt, robot_localization, and more	ROS packages control-toolbox, dwa-local-planner, gazebo-ros-pkgs, gpsd-client, image-transport-plugins, image-rotate, pid, ros-controllers, spacenav-node, usb-cam, rplidar-ros, and gmapping	ROS Kinetic with packages depthimagetolasers can, huksy_control, move_base, actionlib, cv_bridge, image_transport and more	ROS Indigo with packages socket_canbridge, rosbridge_server, teleop_twist_joy, and more	ROS Kinetic with packages rqt_image_view, rtabmap, move_base, mapviz, joy, rtimulib_ros, zed_ros_wrapper, rgbd_odometry, usb_cam, and nmea_navsat_driver	Custom framework RoverCore-S, RoverCore-F, RoverCore-MC, built in house	We used ROS Kinetic and Indigo with packages joy, rosserial, amcl, and robot_localization
Autonomous System	OpenCV, Python	Implemented on our own using GPS and distance to the goal. A control PID with some constraints and logic to back up if necessary to leads us to a given point. Goals are set when previous one was reached.	ROS move_base	ROS move_base and as a backup waypoint navigation using yaw and gps. Also, a C++ OpenCV tennis ball finding algorithm on top of ROS. We could find and navigate to the tennis ball from 8m.	move_base and robot_ localization	ZED depth camera, rtabmap, move_base. We first teleoperate to build a SLAM map and find the tennis ball by human eye, then we go back to the start and set an autonomous goal in the SLAM map.	GPS and drive system, no need for anything else	We had plans of using AMCL and sensor fusion by making use of the existing packages in ROS, but ran out of time.
Arm Control Software	Custom in solution in Energia. interfaced with custom control software RED (Rover Engagement Display) at base station	Tried MoveIt but implemented our own	Some experiments with MoveIt inverse kinematics but used forward kinematics at competition	Wrote our own inverse kinematics and simulation in Unity using C#	Wrote our own PWM library for arm motors	We had plans to use MoveIt but due to lack of testing time used velocity control for each joint mapped to a joystick	We wrote firmware into our framework for our Teensy 3.2 MCUs	Open-loop control with commands sent to an Arduino

Fig. 3. Survey of eight rover teams that competed in URC 2017.

	Mars Rover Design Team	Team Continuum	Cornell Mars Rover	ITU Rover Team	UWRT Robotics	Ryerson Rams Robotics (R3)	SJSU Robotics	Team Anveshak
School Name	Missouri University of Science and Technology	University of Wroclaw, Poland	Cornell University, USA	Istanbul Technical University, Turkey	University of Waterloo, Canada	Ryerson University, Canada	San Jose State University, USA	Indian Institute of Technology, Madras
Final Score (Rank)	403.4 **(1)**	336.3 (2)	264.1 (11)	243.1 (13)	225.7 (15)	190.9 (21)	164.3 (26)	151.4 (29)
Wireless radios and antennas	Ubiquiti 900MHz, Cloverleaf MIMO antenna on rover and dual polarity yagi at base station	Ubiquiti Bullet	Base station antenna was the Ubiquiti AM-2G15-120, rover antenna was the Super Power Supply B00O7ZEK7S, rover and base transceiver was the Ubiquiti Rocket M2	Microhard pDDL2450 could achieve 1km in non-line of sight with 5 dBi omnidirectional antennas. We also backed up comms except the cameras and the TCP link via a RF link with 433 MHz LoRa module.	2.4 GHz and 900 MHz antennas	Ubiquiti M2 Rockets 2.4GHz 802.11n MIMO paired with TP-Link 2408CL omnidirectional antennas	Ubiquiti Rockets M900 and the directional Ubiquiti Loco M900	TP-Link WA 5210 2.4GHz with included directional antenna
Battery System	LGChem18650HE4 Lithium Ion, 80 set up in custom pack, 10 set in parallel with 8 of those sets in series	Custom LiPo modules	1x MaxAmps 7S LIPO Battery	Tattu 6 cell LiPo 22Ah	1x Tattu 6 cell 22Ah Tattu LiPo	Panasonic NCR18650BD 3.7V 3200mAh Li-Ion 4 batteries in series to achieve 14.8V and 6 in parallel to achieve a 19.2Ah	3x Zippy LiPos 7S with a power board we designed	3x 24V LiPo batteries for drive, 2x 12V LiPo batteries for auger/arm
Wired Communication Protocols	I2C, RS232, RoveComm (Custom UDP)	CAN built-into the bananapi with two networks, one for driving wheels, another for the manipulator	CAN bus for interboard, UART for Intel NUC to microcontroller s	I2C for sensors, USB for Raspberry Pi to microcontroller s	CAN for most things, USB for drive motor controller, I2C/SPI for sensors	I2C sensors, USB for cameras, USB serial for Arduinos, UART for GPS, PWM for motor controllers	I2C, UART, Bluetooth (RFCOMM), SPI, PWM, PPM	Serial from the main computer to the various Arduinos
Sensor Fusion	Kalman filtering and custom filtering	robot_ localization	robot_ localization	robot_ localization, custom EKF backup in microcontrollers	robot_ localization	robot_ localization, didn't end up using due to IMU issues	None	None
Team Strengths	Manufacturing capabilities and access to programs that allow us to have many custom components on our rover.	Drive and manipulator controls. Also, I think being just 10 people ups our motivation a lot. Everyone has important work to do	Modularity	Our wireless communication modules. We never lost control or communication to our rover at the competition.	A lot of different experiences from team members because of our coop program.	Tmux for terminal organization, keeping things simple, team dedication, and keeping it fun.	The absolute passion from each and every member of our team as well as our team manage system.	Dedicated team, always ready to learn new things, not shy of challenges. We made great strides in learning ROS in a matter of 3 months.
Improvements for next year	Fix bugs and flaws we found while at URC 2017 and push the boundaries of innovation as we build a new rover.	More field tests of the whole Rover.	Ease of use: easy way to launch and monitor the entire system. Live sensor diagnostics and robust CV.	I really want to add machine learning for finding the tennis ball from further.	Improve our project management.	Clearly labelled wires and pin outs, avoid USB hubs, and a geologist team member.	Secure bigger budget, start earlier, and update our technologies.	More development time, exploit ROS even more, test things more often, more collaboration with other teams.
Source code	github.com/mst-mrdt	Inverse kinematics only gist.github.com/danielsnider/5181ca50cef0ec8fdea5c11279a9fdbc	https://drive.google.com/open?id=0B1r9QYTd8YNrWXNjNmdtcGlwMjQ	github.com/itu-rover	github.com/uwrobotics	github.com/teamr3/URC	github.com/kammce/RoverCore-S	github.com/Team-Anveshak/rover-control

Fig. 4. Survey of eight rover teams that competed in URC 2017. Cont.

are available on the market, those cameras had a big advantage it was possible to easily integrate and customize them using raspberry pi. Therefore, it was easy for the team to experiment with different configurations and find a compromise between good quality and low latency.

The team was also asked what would be one thing they wished to do differently next time and the answer was always the same: more field tests with all components of the rover. This is also the advice that is often given by other teams and it seems very reasonable. By testing the rover with similar tasks as in the competition, it is possible to identify problems sooner and fix them. It also forces the team to complete the work sooner. Unfortunately, to do that properly, a lot of self-control and good organization of the whole group is necessary.

4.2 Rover Manipulator Arm

Fig. 5. Team Continuum's base station setup. A screenshot of their rover control GUI can be seen in 8

In the following section, the robotic arm (or "manipulator") of the Aleph1 rover is described. Team Continuum decided to focus on this particular element, because it is crucial in most of the tasks at URC and at the same time is relatively hard to control.

Before starting the work on the arm controller, the Continuum team decided to conduct a survey of currently available solutions of similar problems in ROS. One of the most promising options was the MoveIt! Motion Planning Framework[11]. Unfortunately, it was not designed with teleoperation in mind and the team was unable to make it perform reasonably well with a goal specified in real time. Therefore, they have decided to implement their own solution, tailored for the specific hardware they were using.

The main component that allows the team to control the arm is the inverse kinematics software (python source code[5]). It utilizes the feedback of four relative encoders placed on the joints of the manipulator to provide the operator two important features: the visualization of the state of the device and the ability to give more intuitive commands to the effector. For example, with this system it is possible to move the gripper up, down, front or back and the speeds of all the motors are automatically adjusted to reach given position.

Another big advantage of the arm system that has proven to be very helpful during the competition is that even when the data connection is not good, it is possible to operate the manipulator. As soon as the instruction reaches the rover, the arm will position itself in the correct way deterministically. This would not be true when using an alternative way of controlling robots where the device is performing some action for as long as a button is being pressed and the loss of packets from the control station might change the result of the operation. Therefore, without such system the operator needs to depend on feedback from cameras which might be delayed or not even available.

In figures 6, 8 and 7 the graphical user interface used to control and visualize the state of the manipulator and the whole rover is presented. It is mainly used to support the operator in collision detection and pose estimation, because the visual feedback from the cameras often was not sufficient. The manipulator could be controlled using a mouse, but usually a Logitech game pad was used.

Team Continuum also implemented a semi-automatic system for picking objects up and for flicking manually operated switches. The system was developed for the European Rover Competition (ERC) 2016 because such functionality provided bonus points. Even though it was not deployed during the URC 2017, we have decided to present it in this section, because it is an interesting example of relatively simple extension to the already described system, enabling much more complex tasks.

To get additional points during ERC 2016 the team must have positioned the effector at least 20cm from the object it wanted to pick up or from the two-state switch. Then, the operator should announce he is starting the autonomous mode and put down the controller. Next, the rover should pick up the object or actuate the switch and move back at least 20cm.

The team decided to implement a simple idea: they wanted to place the effector exactly 20cm from the object and directly in front of it. Then, using the inverse kinematics system they were able to execute pre-recorded movements in order to complete the task.

The first challenge they have faced was how to measure exact distance from objects. They decided to mount two laser pointers on the effector, directed in the direction of each other. They can be seen in multiple scenes of the recently

[5]Continuum Inverse kinematics python source: `https://gist.github.com/danielsnider/5181ca50cef0ec8fdea5c11279a9fdbc`

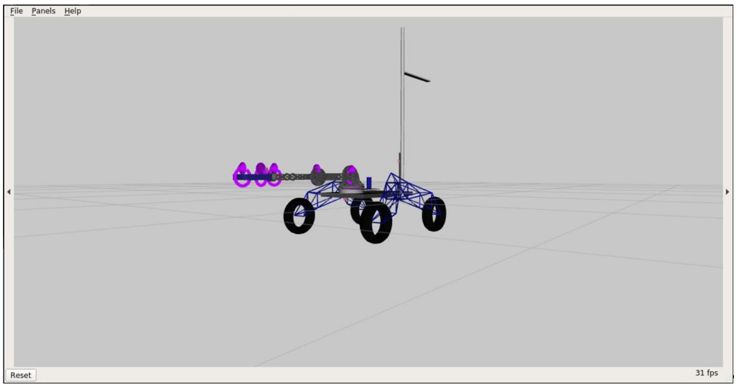

Fig. 6. 3D visualization of the Aleph1 rover used by the team during teleoperation.

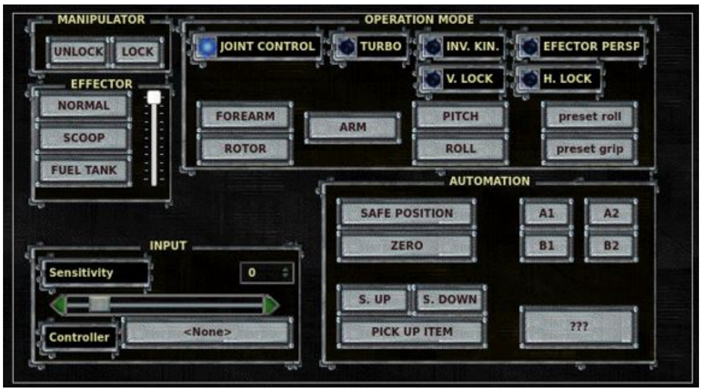

Fig. 7. Team Continuum's GUI used to control the manipulator of the rover.

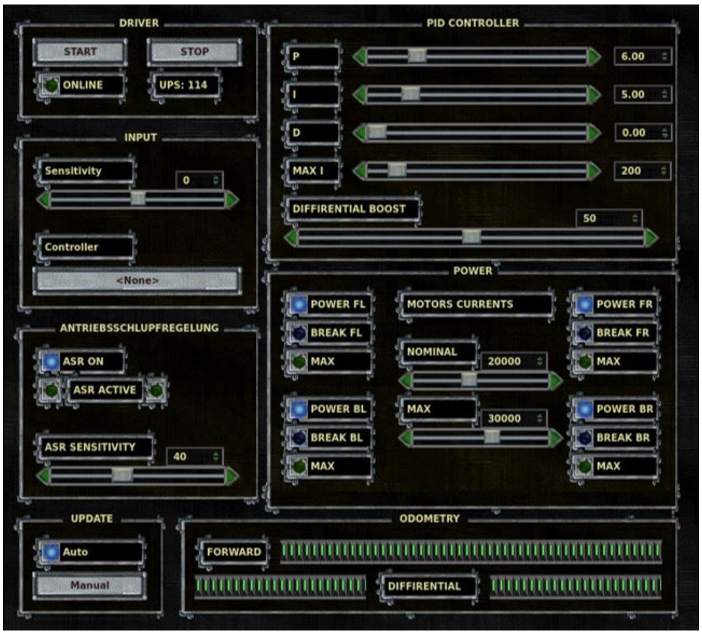

Fig. 8. Team Continuum's GUI used to control the movements of the rover.

released video[6] from the University of Wroclaw. The two observable dots meet exactly at 20cm.

The two preprogrammed arm movements were developed as follows:

For flicking switches, the effector went forward for 20cm and 2.5cm down simultaneously, then up for 5cm and back to the initial position (20cm back and 2.5cm down). Due to the mechanical construction and the softness of the end of the arm, Aleph1 was able to switch most of the actuators using this technique, both very small and big ones.

For picking objects up the effector goes down 20cm, then the grip motor engages until the force measurement on this motor crosses a predetermined threshold, then it goes back up 20cm.

The arm of the Aleph1 rover is far from the state of the art manipulators. It is not as fast or precise as it could be and the control sometimes is tricky. However, even though the described solutions might seem simple it was still one of the most advanced robotic arms on URC 2017. It is hard to construct a robust and fail-proof manipulator that is mountable on a movable platform and meets the strict mass and costs limits of URC. The Continuum team has proven that this is possible and the capabilities of such simple platform can be exceptional.

[6] A recent video of from the University of Wroclaw of their rover has incredible cinematography: `https://www.youtube.com/watch?v=MF8DkKDBXtg`

5 Case Study: Team R3

5.1 ROS Environment

At Team R3 (Ryerson University), our Kinetic ROS software built on Ubuntu 16.04 had five main systems: the drive system, the autonomous system, the global positioning system, the visual feedback system, and the odometry system. The full diagram, as seen in Fig. 9, has been made available in Microsoft Visio format[7].

To learn more about each software component, links to the most relevant documentation are provided.

Autonomous System Team R3's autonomous system consists of the ZED depth camera and the RTAB-Map ROS package for SLAM mapping[5]. The authors have also contributed gps_goal and follow_waypoints ROS packages for added goal setting convenience. For further details about the autonomous system see the tutorial in section 14.

Listing 1.1. Autonomous system software used in Team R3's rover (numbers refer to Fig. 9).

```
1. zed-ros-wrapper (http://wiki.ros.org/zed-ros-wrapper)
2. follow_waypoints.py (http://wiki.ros.org/follow_waypoints)
3. rgbd_odometry (http://wiki.ros.org/rtabmap_ros#rgbd_odometry)
4. rtabmap (http://wiki.ros.org/rtabmap_ros)
21. gps_goal.py (http://wiki.ros.org/gps_goal)
```

Odometry System At the URC competition teams are surrounded by sandy desert terrain in Utah. As a result of wheel slippage on sand, Team R3 did not use wheel odometry. Instead we focused on fusing IMU and visual odometry into a more reliable position and orientation. However, because our IMU was not working well enough to produce a good fused result, at the competition we did not actually use the ekf_localization[10] ROS nodes of the odometry system. Instead we relied on odemetry from the rgbd_odometry ROS node only and this worked well for our autonomous system based on the RTAB-Map package.

Listing 1.2. Odometry software components used in Team R3's rover (numbers refer to Fig. 9).

```
5. ekf_localization (http://docs.ros.org/kinetic/api/robot_localization/html
   /)
6. rtimulib_ros (https://github.com/romainreignier/rtimulib_ros)
7. navsat_transform (http://docs.ros.org/kinetic/api/robot_localization/html
   /)
8. nmea_navsat_driver (http://wiki.ros.org/nmea_navsat_driver)
```

[7]Team R3's rover software architecture diagram in Microsoft Visio format `https://github.com/danielsnider/ros-rover/blob/master/diagrams/Rover_Diagram.vsdx?raw=true`

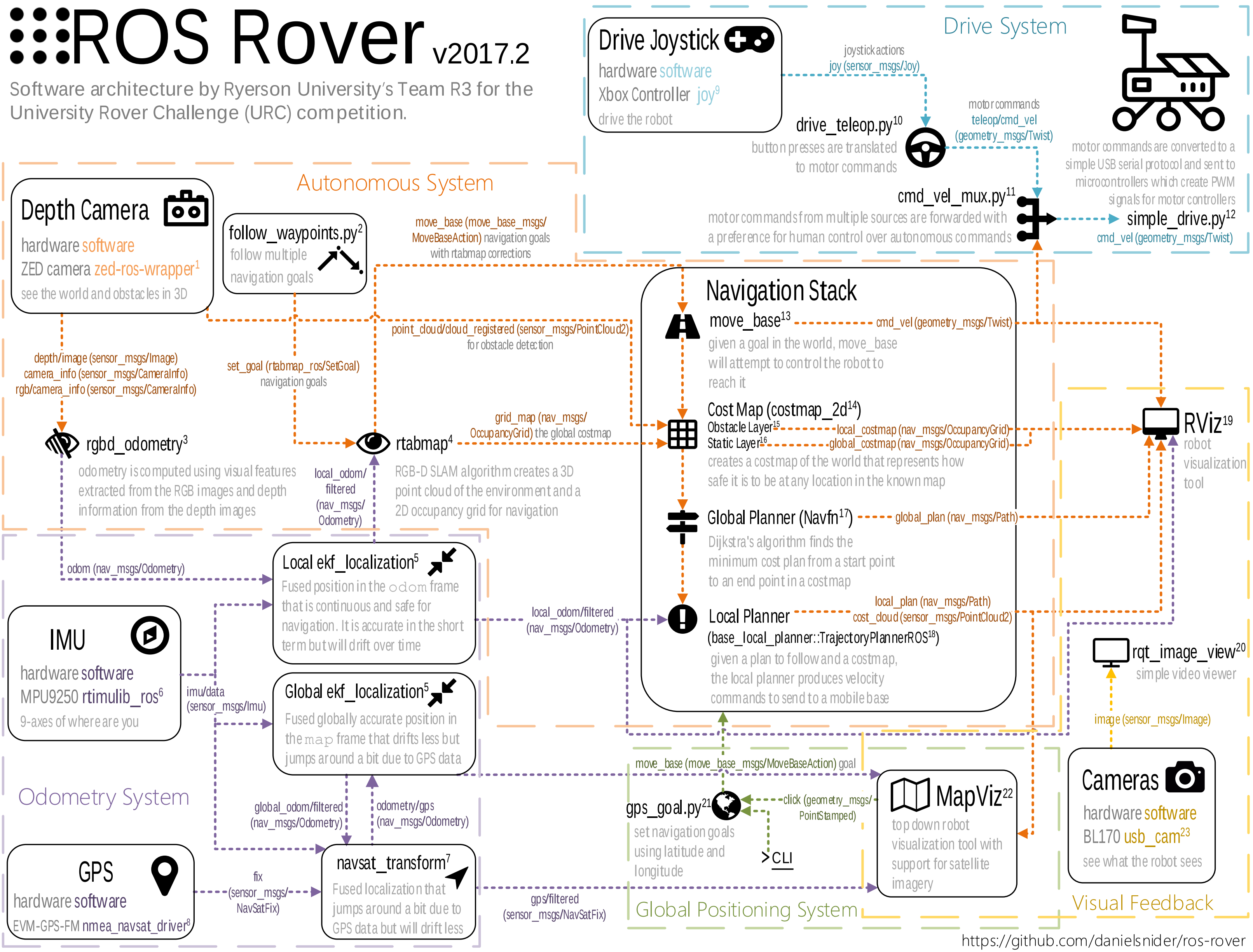

Fig. 9. ROS software architecture of Team R3's rover.

Drive System The drive software created by Team R3 is called simple_drive because it does not produce wheel odometry or transforms. That is left to the autonomous system via SLAM and is more robust in slippery desert environments. The simple_drive package controls 6 motors with PWM pulses using an Arduino dedicated to driving. For further details of the drive system see the tutorial in section 8.

Listing 1.3. Drive system software used in Team R3's rover (numbers refer to Fig. 9).

```
9. joy (http://wiki.ros.org/joy)
10. drive_teleop.py (http://wiki.ros.org/simple_drive#drive_teleop)
11. cmd_vel_mux.py (http://wiki.ros.org/simple_drive#cmd_vel_mux)
12. simple_drive.py (http://wiki.ros.org/simple_drive#simple_drive-1)
```

Navigation Stack The navigation stack used by Team R3 follows the commonly used development patterns of the ROS navigation stack[8]. Our setting choices were inspired by RTAB-Map's tutorial[9] and we share our tips in section 14.

Listing 1.4. Navigation software stack used in Team R3's rover (numbers refer to Fig. 9).

```
13. move_base (http://wiki.ros.org/move_base)
14. Cost Map costmap_2d (http://wiki.ros.org/costmap_2d)
15. Cost Map Obstacle Layer (http://wiki.ros.org/costmap_2d/hydro/obstacles)
16. Cost Map Static Layer (http://wiki.ros.org/costmap_2d/hydro/staticmap)
17. Global Planner Navfn (http://wiki.ros.org/navfn)
18. Local Planner base_local_planner (http://wiki.ros.org/base_local_planner)
```

Visual Feedback To assist in teleoperation of the robot, sensors and navigation plans were visualized in rviz for local information such as point clouds and in mapviz for a global context that includes satellite imagery. The visual feedback software and joystick software was executed remotely on a laptop in the control station. The rest of software was executed on a Jetson TX1 inside the rover.

Listing 1.5. Visual feedback software used by Team R3 (numbers refer to Fig. 9).

```
19. RViz (http://wiki.ros.org/rviz)
20. rqt_image_view (http://wiki.ros.org/rqt_image_view)
22. MapViz (http://wiki.ros.org/mapviz)
23. usb_cam (http://wiki.ros.org/usb_cam)
```

[8] ROS Nagivation stack `http://wiki.ros.org/navigation`

[9] RTAB-Map tutorial for the ROS navigation stack `http://wiki.ros.org/rtabmap_ros/Tutorials/StereoOutdoorNavigation`

6 Tutorial: Autonomous Waypoint Following

The following tutorial documents an original ROS package follow_waypoints[10,11] that will buffer move_base goals until instructed to navigate to them in sequence. If you can autonomously navigate from A to B, then you can combine multiple steps of A to B to form more complicated paths and use cases. For example, do you want your rover to take the scenic route? Are you trying to reach your goal and come back? Do you need groceries on the way home from Mars?

Team R3 (Ryeson Univerisity) has developed the follow_waypoints ROS package to use actionlib to send the goals to move_base in a robust way. The code structure of follow_waypoints.py is a barebones state machine. For this reason it is easy to add complex behavior controlled by state transitions. For modifying the script to be an easy task, you should learn about the Python state machine library in ROS called SMACH[12,13]. The state transitions in the script occur in the order `GET_PATH` (buffers goals into a path list), `FOLLOW_PATH`, and `PATH_COMPLETE` and then they repeat.

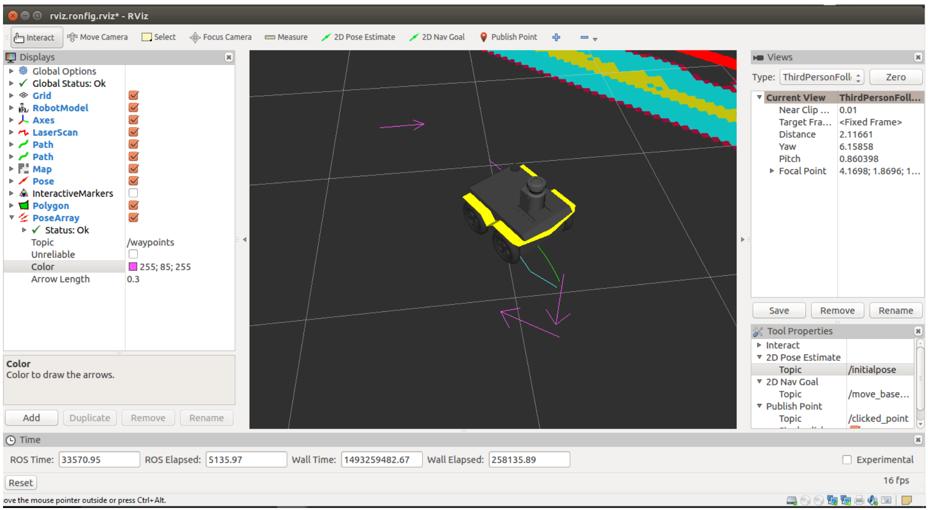

Fig. 10. A simulated Clearpath Jackal robot[15]navigating to one of several waypoints displayed as pink arrows.

[10]Source code for follow_waypoints ROS package `https://github.com/danielsnider/follow_waypoints`

[11]Wiki page for follow_waypoints ROS package `http://wiki.ros.org/follow_waypoints`

[12]SMACH state machine library for python `http://wiki.ros.org/smach`

[13]One alternative to SMACH is py-trees, a behavior tree library `http://py-trees.readthedocs.io/en/devel/background.html`

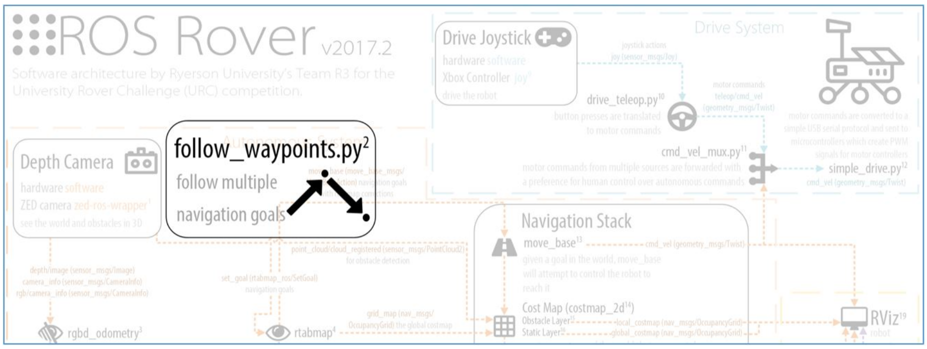

Fig. 11. ROS node follow_waypoints as seen in the larger architecture diagram. See Fig. 9 for the full diagram.

6.1 Usage in the University Rover Challenge (URC)

A big advantage of waypoint following is that the rover can go to points beyond reachable by Wi-Fi. In the autonomous traversal task, Team ITU's rover at one point lost connection but then got it back again when it reached the waypoint.

Other possible uses of waypoint following: To navigate to multiple goals in the autonomous task with a single command (use in combination with GPS goals, see Section 12). To search a variety of locations, ideally faster than by teleoperation. To allow for human assisted obstacle avoidance where an obstacle is known to fail detection.

6.2 Usage Instructions

1. Install the ROS package:

```
$ sudo apt-get install ros-kinetic-follow-waypoints
```

2. Launch the ROS node:

```
$ roslaunch follow_waypoints follow_waypoints.launch
```

3. To set waypoints you can either publish a ROS Pose message to the /initialpose topic directly or use RViz's tool "2D Pose Estimate" to click anywhere. Fig. 10 shows the pink arrows representing the current waypoints in RViz. To visualize the waypoints in this way, use the topic /current_waypoints, published by follow_waypoints.py as a PoseAarray type.

4. To initiate waypoint following send a "path ready" message.

[15]Clearpath Jackal tutorials for Ubuntu 14.04 `http://docs.ros.org/indigo/api/jackal_tutorials/html/simulation.html`

```
$ rostopic pub /path_ready std_msgs/Empty -1
```

To cancel the goals do the following. This is the normal move_base command to cancel all goals.

```
$ rostopic pub -1 /move_base/cancel actionlib_msgs/GoalID --
    {}
```

6.3 Normal Output

When you launch and use the follow_waypoints ROS node you will see the following console output.

```
$ roslaunch follow_waypoints follow_waypoints.py

[INFO] : State machine starting in initial state 'GET_PATH' with userdata: ['
    waypoints']
[INFO] : Waiting to receive waypoints via Pose msg on topic /initialpose
[INFO] : To start following waypoints: 'rostopic pub /path_ready std_msgs/
    Empty -1'
[INFO] : To cancel the goal: 'rostopic pub -1 /move_base/cancel
    actionlib_msgs/GoalID -- {}'
[INFO] : Received new waypoint
[INFO] : Received new waypoint
[INFO] : Received path ready message
[INFO] : State machine transitioning 'GET_PATH':'success'-->'FOLLOW_PATH'
[INFO] : Executing move_base goal to position (x,y): 0.0123248100281,
    -0.0620594024658
[INFO] : Executing move_base goal to position (x,y): -0.0924506187439,
    -0.0527720451355
[INFO] : State machine transitioning 'FOLLOW_PATH':'success'-->'PATH_COMPLETE
    '
[INFO] : ###############################
[INFO] : ##### REACHED FINISH GATE #####
[INFO] : ###############################
[INFO] : State machine transitioning 'PATH_COMPLETE':'success'-->'GET_PATH'
[INFO] : Waiting to receive waypoints via Pose msg on topic /initialpose
```

7 Tutorial: Image Overlay Scale and Compass

In an effort to add context about of the world to imagery recorded, Team R3 has developed a ROS package image_overlay_compass_and_scale[16,17] that can add an indication of scale and compass to images and video streams. A compass graphic will be embedded into imagery in a way that makes north direction apparent. A scale bar is also added so that the size of objects in images is more easily interpreted.

Compass and scale values must be provided using standard ROS Float32 messages. Alternatively, a command interface can be used without ROS.

[16] Source code for the image_overlay_compass_and_scale ROS package `https://github.com/danielsnider/image_overlay_scale_and_compass`

[17] Wiki page for the image_overlay_compass_and_scale ROS package `http://wiki.ros.org/image_overlay_scale_and_compass`

This tool meets one of the requirements of URC 2017 (in an automated way) and is applied to images of soil sampling sites and scenic panoramas for scientific and geological purposes.

This package uses the OpenCV[1] python library to overlay a compass graphic, the scale bar and dynamic text which is set using a ROS topic.

The implementation of overlaying the compass graphic on the input image follows these steps: 1) resize the compass graphic to be 60% the size of the input image's smaller side (whichever is smaller, x or y resolution). 2) Rotate the compass to the degrees specified on the /heading input topic. 3) Warp the compass to make it appear that the arrow is pointing into the image. This assumes that the input image has a view forward facing with the sky in the upper region of the image.

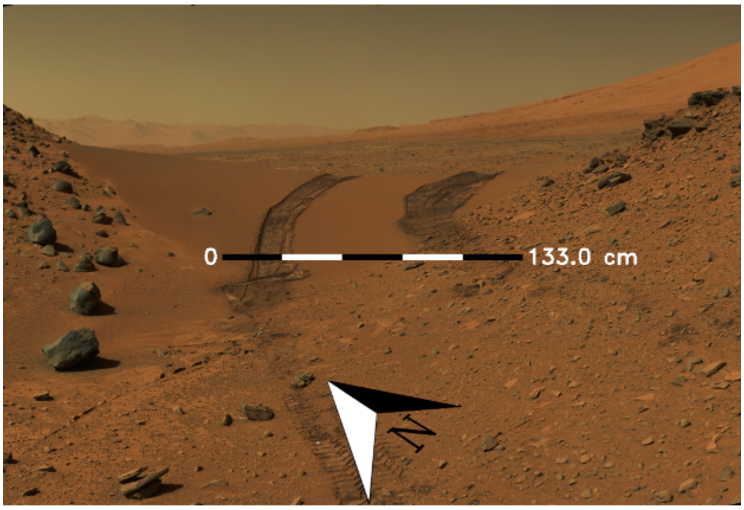

Fig. 12. Example of the image_overlay_compass_and_scale ROS package.

7.1 Usage Instructions

1. Install the ROS package:

```
$ sudo apt-get install ros-kinetic-image-overlay-compass-and-
  scale
```

2. Launch the ROS node:

```
$ roslaunch image_overlay_compass_and_scale overlay.launch
```

3. Publish heading and scale values

```
$ rostopic pub /heading std_msgs/Float32 45 # unit is degrees
$ rostopic pub /scale std_msgs/Float32 133 # unit is
    centimeters
```

4. View resulting image

```
$ rqt_image_view /overlay/compressed
```

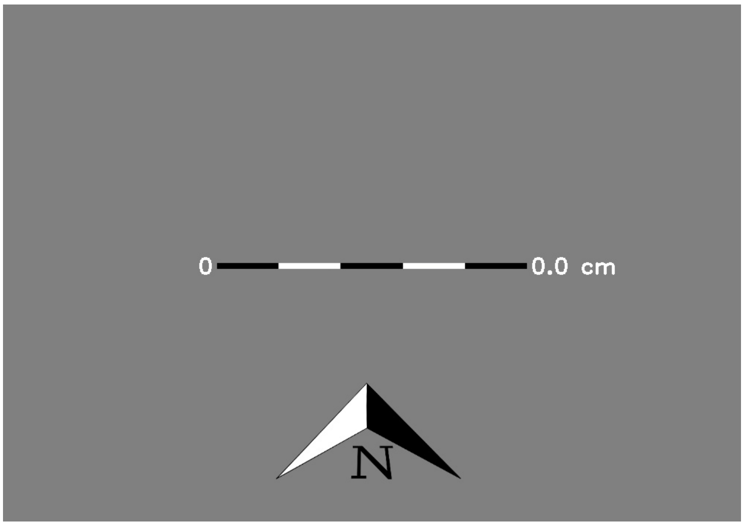

Fig. 13. This is what to expect when nothing is received by the node. This is the default published image.

7.2 Command Line Interface (CLI)

The image_overlay_compass_and_scale ROS package includes a command line interface to invoke the program once and output and image with the overlayed graphics.

Listing 1.6. Example usage of the image_overlay_compass_and_scale CLI script to save the image overlay to disk instead of publishing to ROS.

```
$ roscd image_overlay_compass_and_scale
$ ./src/image_overlay_compass_and_scale/image_overlay.py --
    input-image ~/mars.png --heading 45 --scale-text 133 --
    output-file output.png
```

8 Tutorial: A Simple Drive Software Stack

The authors have published a new ROS package, simple_drive[18,19], used at the University Rover Challenge (URC) 2017 on Team R3's rover. It proved simple and effective in desert conditions. The package is simple in the sense that it does not publish TF odometry from wheel encoders because on sand wheels slip very substantially.

The package implements skid steering joystick teleoperation with three drive speeds, dedicated left and right thumbsticks control left and right wheel speeds, control of a single axis panning servo to look around the robot, a cmd_vel multiplexer to support a coexisting autonomous drive system, and Arduino firmware to talk to PWM motors and a servo. For the sake of simplicity, this package does not do the following: TF publishing of transforms, wheel odometry publishing, PID control loop, no URDF, or integration with ros_control. Though all of these simplifications are normal best practices in sophisticated robots. The simple_drive package gives ROS users the ability advance their robot more quickly and hopefully to find more time to implement best practices.

This package is divided into four parts: drive_teleop ROS node, cmd_vel_mux ROS node, simple_drive ROS node, drive_firmware Arduino code. In the following sections we will explain the main features and implementation details but for the full ROS API documentation please see the simple_drive online ROS wiki.

8.1 Usage Instructions

1. Install the ROS package:

```
$ sudo apt-get install ros-kinetic-simple-drive
```

2. Install the drive_firmware onto a microcontroller connected to your motors and wheels by PWM. See section 8.5 for detailed instructions. The microcontroller must also be connected to the computer running the simple_drive ROS node by a serial connection (e.g. USB).
3. Launch the three simple_drive ROS nodes separately or together using the included `drive.launch` file:

```
$ roslaunch simple_drive drive_teleop.launch joy_dev:=/dev/
    input/js0
$ roslaunch simple_drive cmd_vel_mux.launch
$ roslaunch simple_drive simple_drive.launch serial_dev:=/dev
    /ttyACM0
# OR all-in-one launch
$ roslaunch simple_drive drive.launch
```

4. Your robot should now ready to be driven.

[18] Source code for simple_drive ROS package `https://github.com/danielsnider/simple_drive`

[19] Wiki page for simple_drive ROS package `http://wiki.ros.org/simple_drive`

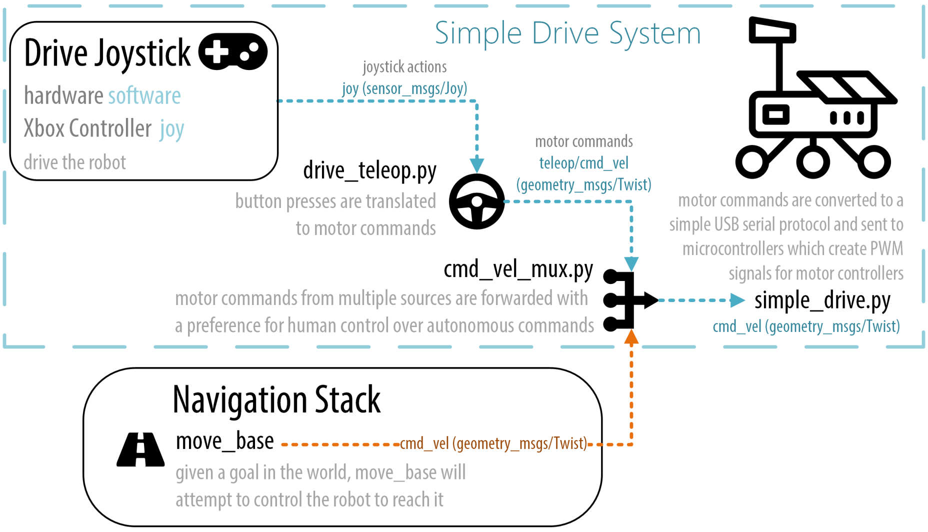

Fig. 14. Architecture of the drive ROS software used by Team R3. See fig. 9 for a larger diagram. Note that the drive_firmware microcontroller software component is not illustrated but it useful to note that it communicates with simple_drive.py.

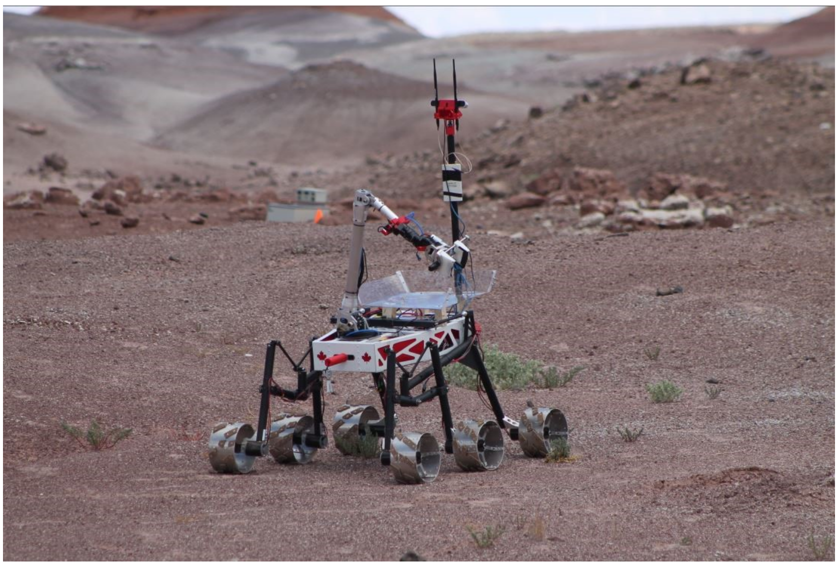

Fig. 15. Team R3's rover being teleoperated by the simple_drive ROS package.

8.2 drive_teleop ROS Node

The drive_teleop node handles joystick input commands and outputs desired drive speeds to the cmd_vel_mux ROS Node. This node handles joystick inputs in a in skid steering style, also known as diff drive or tank drive where the left joystick thumbstick controls the left wheels and the right thumbstick controls the right wheels. We refer to this layout as tank drive through this section.

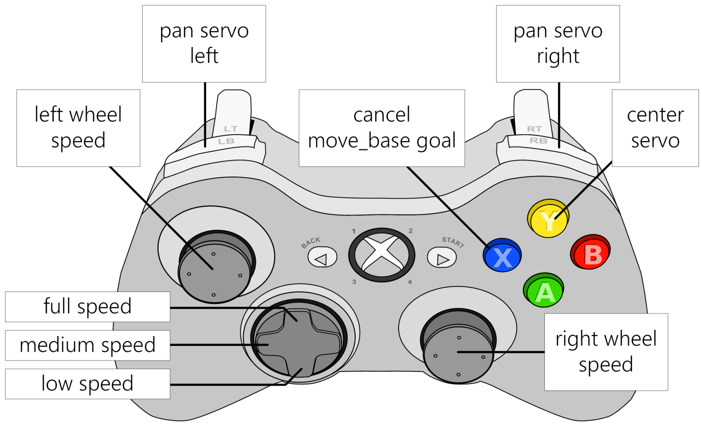

Fig. 16. The Xbox 360 joystick button layout for the simple_drive package (diagram is available in Visio format[21]). Image credit: Microsoft Corporation

More specifically, this node converts sensor_msgs/Joy messages from the joy ROS node into geometry_msgs/Twist messages which represent the desired drive speed. Listing 16 shows that there are programmed buttons to set the drive speed to low, medium, or high speed, look around with a single axis servo, and cancel move_base goals at any moment.

Typically the servo is used to move a camera so that the teleoperator pan around the surroundings of the robot. The servo's rotation speed (in degrees per button press) can be set using the `servo_pan_speed` ROS parameter. The minimum and maximum angle of servo rotation in degrees can be set using the `servo_pan_min` and `servo_pan_max` ROS parameters respectively.

The button mapping was tested on an Xbox 360 controller and should require little or no modification for similar controllers, if they support a DirectInput mode.

[21] Xbox 360 joystick button layout diagram in Visio format `https://github.com/danielsnider/ros-rover/blob/master/diagrams/simple_drive_Xbox_Controller.vsdx?raw=true`

8.3 cmd_vel_mux ROS Node

The cmd_vel_mux node receives movement commands on two sensor_msgs/Twist topics, one for teleoperation and one for autonomous control, typically move_base. Movement commands are multiplexed (i.e. forwarded) to a final topic for robot consumption with a preference for human control over autonomous commands. If any teleoperation movement command is received the cmd_vel_mux node will block autonomous movement commands for a set time defined by the `block_duration` ROS parameter.

8.4 simple_drive ROS Node

The simple_drive node sends commands to motors by communicating with a microcontroller over serial using the protocol defined by the drive_firmware. The simple_drive node listens to geometry_msgs/Twist for motor commands and std_msgs/Float32 for the servo position. The serial device that simple_drive communicates with is set with the `serial_dev` and `baudrate` ROS parameters.

This node is very simple and could be eliminated if your microcontroller supports ROS. For example Arduinos can use rosserial_arduino. However, this node is written in Python so you could more easily add complex functionality in Python and then in your microcontroller do the minimum amount of work necessary allowing for the use of smaller and more lightweight microcontrollers.

8.5 drive_firmware Ardiuno Software

The drive_firmware is software for an Arduino microcontroller and it does not run as a ROS node. The Arduino is assumed to be dedicated to use of the simple_drive package. It does the minimum amount of work possible to receive motor commands from the simple_drive node over a USB serial connection and output voltages to digital PWM output to be received by motor controllers.

We tested on an Arduino Mega 2560 and Arduino Due, however many other boards should work with the same code and setup steps thanks to PlatformIO. You may need to change to change the pin numbers. Note that this software does not stop moving the robot if no messages are received, or if communications are lost.

Serial Communication Protocol The drive_firmware uses a serial protocol that is designed for simplicity rather than integrity of messages. As such it should not be perceived as especially robust. However, it has worked consistently in our experience.

Fig. 17 shows how data is encoded over the serial connection. A header command byte is transmitted followed by one or two (depending on the command) IEEE 754 standard binary float values. Linear and angular velocity are expected to be between -1 and 1, which are linearly scaled to motor duty-cycles by the drive_firmware.

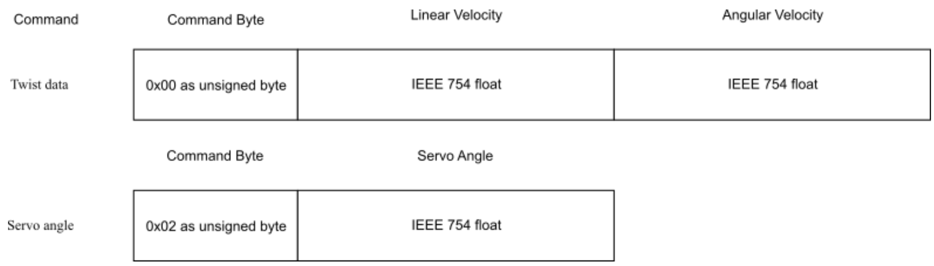

Fig. 17. Diagram of the serial format used by drive_firmware to communicate between microcontroller and an on board computer.

An example packet following this format could be encoded as bytes: 0x00 0x3f 0x80 0x00 0x00 0x00 0x00 0x00 0x00. This would be decoded as a twist data (leading 0x00) with 1.0 for linear velocity (0x3f 0x80 0x00 0x00) and 0.0 for angular velocity (0x00 0x00 0x00 0x00).

Tank to Twist Calculation When left and right joystick inputs are received by the drive_teleop node, representing left and right wheel linear velocities[22] (i.e. skid steering or differential drive), a conversion calculation to a geometry_msgs/Twist with linear and rotational velocities is performed as seen in equations 1 and 2. The parameter r is the half of the distance between your rover's wheels in meters.

$$Linear\ Velocity\ (\frac{m}{s}) = \frac{left\_velocity + right\_velocity}{2.0} \tag{1}$$

$$Angular\ Velocity\ (\frac{rad}{s}) = \frac{left\_velocity - right\_velocity}{2.0 \times r} \tag{2}$$

Twist to Tank Calculation When a geometry_msgs/Twist containing linear and rotational velocities is received by the drive_firmware, corresponding linear velocities are calculated for the left and right sides of wheel banks on the vehicle as seen in equation 3 and 4.

$$Left\ Side\ Linear\ Velocity\ (\frac{m}{s}) = linear\_velocity + angular\_velocity \times r \tag{3}$$

$$Right\ Side\ Linear\ Velocity\ (\frac{m}{s}) = linear\_velocity - angular\_velocity \times r \tag{4}$$

[22] Wheel linear velocity is meant to be the speed at which distance is travelled and not rpm.

PlatformIO We deploy the drive_firmware to an Arduino microcontroller using PlatformIO[23] because it allows for a single source code to be deployed to multiple platforms. PlatformIO supports approximately 200 embedded boards and all major development platforms such as Atmel, ARM, STM32 and more.

8.6 Install and configure drive_firmware

The following steps demonstrate how to install and configure drive_firmware component of the simple_drive package. These steps were tested on Ubuntu 16.04.
1. Install PlatformIO[24]:

```
$ sudo python -c "$(curl -fsSL https://raw.githubusercontent.
    com/platformio/platformio/master/scripts/get-platformio.
    py)"
# Enable Access to Serial Ports (USB/UART)
$ sudo usermod -a -G dialout <your username here>
$ curl https://raw.githubusercontent.com/platformio/
    platformio/develop/scripts/99-platformio-udev.rules  > /
    etc/udev/rules.d/99-platformio-udev.rules
# After this file is installed, physically unplug and
    reconnect your board.
$ sudo service udev restart
```

2. Create a PlatformIO project[25]:

```
$ roscd simple_drive
$ cd ./drive_firmware/
# Find the microcontroller that you have in the list of
    PlatformIO boards
$ pio boards | grep mega2560
# Use the name of your board to initialize your project
$ pio init --board megaatmega2560
```

3. Modify the microcontroller pin layout to match wirings to motor controller hardware. First open the file containing the pin settings then change the pin numbers as needed:

```
$ vim src/main.cpp +4

   // Pins to Left Wheels
   #define pinL1 13
   #define pinL2 12
   #define pinL3 11
```

[23] PlatformIO is an open source ecosystem for IoT development `http://platformio.org/`

[24] More information on how to install PlatformIO is here `http://docs.platformio.org/en/latest/installation.html#super-quick-mac-linux`

[25] More documentation about PlatformIO: `http://docs.platformio.org/en/latest/quickstart.html`

```
// Pins to Right Wheels
#define pinR1 9
#define pinR2 8
#define pinR3 7
// Pin to the Servo
#define pinServo 5
```

4. Depending on the specs of your motor controllers, modify the PWM settings as needed (values are duty-cycles in microseconds):

```
$ vim src/main.cpp +17

// PWM specs of the Spark motor controller. Spark manual:
//      http://www.revrobotics.com/content/docs/LK-ATFF-SXAO-UM.pdf
#define sparkMax 1000 // Default full-reverse input pulse
#define sparkMin 2000 // Default full-forward input pulse
```

5. Deploy the drive_firmware to the microcontroller:

```
$ pio run --target upload
```

6. Your robot is now ready to be driven.

9 Tutorial: Velocity Controlled Arm

In this tutorial the original simple_arm[26,27] package is presented. The simple_arm package is teleoperation software and firmware for a simple, velocity controlled arm with 6 degrees of freedom. Forces input by the operator's joystick motions are converted to individual motor velocities. By using velocity control instead of positional control it is more difficult to control an arm but it is simpler because it does not needing hardware to sense joint positions.

At the University Rover Competition (URC) the rover's manipulator arm is probably the most important component because it is needed for a large portion of point awarding tasks. It is also very complex. Team R3 ran out of development time to integrate position encoders with MoveIt!, so the arm software in simple_arm that went to URC 2017 was simple yet effective.

In the science URC mission, the arm was used to drill soil and collect samples. In the equipment servicing tasks the arm was used for unscrewing a cap, pouring a container of liquid, and more. In the extreme delivery and retrieval mission the arm was used to pick up and carry hand tools and small rocks.

[26] Source code for simple_arm ROS package `https://github.com/danielsnider/simple_arm`

[27] Wiki page for simple_arm ROS package `http://wiki.ros.org/simple_arm`

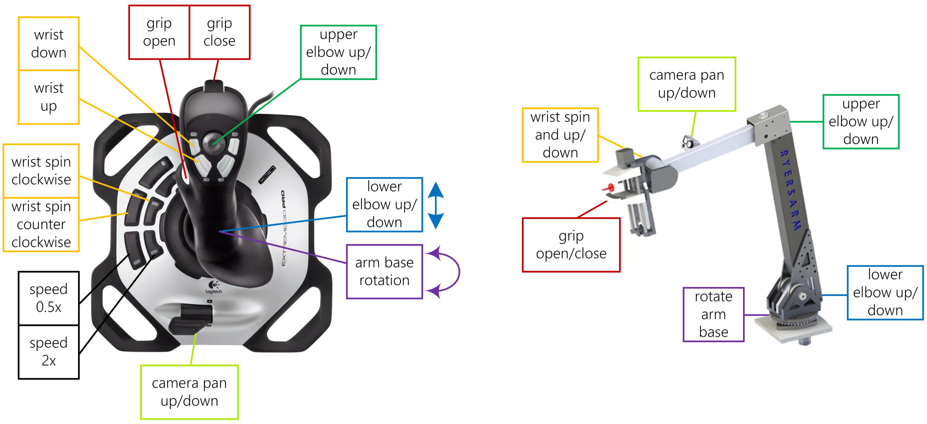

Fig. 18. The Logitech Extreme 3D Pro joystick button layout for the simple_arm package (diagram is also available in Visio format[29]). Image credit: Logitech International

Features of the simple_arm package include: velocity control of individual arm joint motors, fast and slow motor speed modifier buttons, buttons to open and close a gripper, control of a camera servo (one axis only), simple Arduino firmware to talk to PWM motors and a servo. For the sake of simplicity, this package does not implement the following best practices: no tf publishing, no URDF, no PID control loop, no integration with ros_control or MoveIt!. The simple_arm package gives ROS users the ability advance their robot more quickly and hopefully to find more time to implement best practices.

9.1 Usage Instructions

1. Install the ROS package:

```
$ sudo apt-get install ros-kinetic-simple-arm
```

2. Launch the ROS node and specify as arguments the joystick device path for controlling the arm and the Arduino to control the arm motors:

```
$ roslaunch simple_arm simple_arm.launch joystick_serial_dev
    :=/dev/input/js0 microcontroller_serial_dev:=/dev/ttyACM0
```

In most cases however, the joystick is connected to another computer, such as a teleoperation station. To do this, run the `joy` ROS node separately over the ROS network.

3. Install the arm_firmware onto a microcontroller as described in section 8.5. The microcontroller must be connected to the arm's motors and to the on board computer running the simple_arm ROS node by a serial connection (ex. USB).

[29] Logitech Extreme 3D Pro joystick button layout diagram in Visio format `https://github.com/danielsnider/ros-rover/blob/master/diagrams/simple_arm_joystick_diagram.vsdx?raw=true`

4. Your robot arm is ready to be moved.

9.2 simple_arm ROS Node

The simple_arm node is written in python and simply converts ROS sensor_msgs/Joy messages from the common `joy` joystick ROS node into serial messages. The serial messages are sent to the arm_firmware on the microcontroller to drive the robot arm. The serial messages follow a simple protocol. Each command is a list of 7 floats, one velocity command for each of the 6 joints and one target angle to control the single axis camera.

The button mapping implemented by the simple_arm node can be seen in Fig. 18 and was tested on a Logitech Extreme 3D Pro joystick. The button layout should only need small modifications if any to work for similar controllers that support a DirectInput mode.

9.3 arm_firmware Arduino Software

The arm_firmware microcontroller code does the minimum amount of work possible to receive motor commands from a USB serial connection and output voltages to digital PWM to be received by motor controllers. It simply receives and fires commands to the lower hardware level with no feedback.

We connected an Arduino by USB serial to our robot's on-board computer and dedicated its use to the simple_arm package. We tested on an Arduino Mega 2560, however many other boards should work with the same code and setup steps thanks to PlatformIO. You may need to change to change the pin numbers. For details on how to install and use PlatformIO with the simple_arm package see the ROS wiki page or section 8.5 which contains very similar instructions.

Please note that this software does not stop moving the robot if no messages are received for certain period of time.

10 Tutorial: Autonomous Recovery after Lost Communications

At the URC competition, Team R3 (Ryerson University) was worried about travelling into a communication deadzone and losing wireless control of our rover from a distant base station. This is one of the challenges put forth by URC competition and is often found in the real world.

We have published a new package, called lost_comms_recovery[30,31] that will trigger when the robot loses connection to the base station and it will navigate to a configurable home or stop all motors. The base station connection check uses

[30] Source code for lost_comms_recovery ROS package `https://github.com/danielsnider/lost_comms_recovery`

[31] Wiki page for lost_comms_recovery ROS package `http://wiki.ros.org/lost_comms_recovery`

ping to a configurable list of IPs. The monitoring loop waits 3 seconds between checks and by default failure is triggered after 2 consecutive failed pings. Each ping will wait up to one second to receive a response.

Please note that this node is not designed to be *realtime*[32] safe. What is safer than relying on this package is using motor control software that sets zero velocity after a certain amount of time not receiving any new commands.

10.1 Usage Instructions

1. Install:

```
$ sudo apt-get install ros-kinetic-lost-comms-recovery
```

2. Launch:

```
$ roslaunch lost_comms_recovery lost_comms_recovery.launch
    ips_to_monitor:=192.168.1.2
```

3. Then the following behavior will take place:

If move_base is running, an autonomous recovery navigation will take place. The default position of the recovery goal is the origin (0,0) of the frame given it the goal_frame_id ROS parameter and the orientation is all 0s by default. This default pose can be overridden if a messaged is published on the recovery_pose topic. If move_base is already navigating to a goal it will not be interrupted and recovery navigation will happen when move_base is idle.

If move_base is not running when communication failure occurs then motors and joysticks are set to zero by publishing a zero geometry_msgs/Twist message and a zero sensor_msgs/Joy message to simulate a joystick returning to a neutral, non-active position.

10.2 Normal Output

When you launch and use the lost_comms_recovery ROS node you will see the following console output.

```
$ roslaunch lost_comms_recovery lost_comms_recovery.launch
    ips_to_monitor:=192.168.190.136

[INFO] Monitoring base station on IP(s): 192.168.190.136.
[INFO] Connected to base station.
[INFO] Connected to base station.
...
[ERROR] No connection to base station.
```

[32] More information about real-time computing `https://en.wikipedia.org/wiki/Real-time_computing`

```
[INFO] Executing move_base goal to position (x,y) 0.0, 0.0.
[INFO] Initial goal status: PENDING
[INFO] This goal has been accepted by the simple action
   server
[INFO] Final goal status: SUCCEEDED
[INFO] Goal reached.
```

11 Tutorial: Stitch Panoramas with Hugin

Hugin[9] is professional software popularly used to create panoramic images by compositing and rectifying multiple still images. We have created a package called hugin_panorama[33,34] to wrap one high level function of Hugin, the creation of panoramas.

In the science task of the URC competition, teams are awarded points if they document locations of scientific interest such as geological and soil sampling sites with panoramas.

Our package uses the Hugin command line tools[35] to compose panoramas in 8 steps according to a well-documented workflow[36]. To summarize, it consists of creating a Hugin project file, finding matching feature control points between images, pruning control points with large error distances, find vertical lines across images to be straighten, doing the straightening and other photometric optimization, optimal cropping, and saving to tiff and compressed png image formats. The compressed panoramic image is published on at output ROS topic. An example panorama can be seen in Fig. 19.

The hugin_panorama launch implementation[37] makes use of the image_saver[38] node provided by the image_view package. The image_saver node will save all images from a sensor_msgs/Image topic as jpg/png files. The saved images are used as the source image parts when creating the panoramas.

11.1 Usage Instructions

1. Install the ROS package:

```
$ sudo apt-get install ros-kinetic-hugin-panorama hugin-tools
    enblend
```

[33] Source code for hugin_panorama ROS package `https://github.com/danielsnider/hugin_panorama`

[34] Wiki page for hugin_panorama ROS package `http://wiki.ros.org/hugin_panorama`

[35] The Hugin image processing library `http://hugin.sourceforge.net/`

[36] Panorama scripting with Hugin `http://wiki.panotools.org/Panorama_scripting_in_a_nutshell`

[37] Main launch file of the hugin_panorama package `https://github.com/danielsnider/hugin_panorama/blob/master/launch/hugin_panorama.launch`

[38] Documentation for the image_saver ROS node `http://wiki.ros.org/image_view#image_view.2BAC8-diamondback.image_saver`

Fig. 19. Panorama example made with Hugin. Image credit: Senza Senso

2. Launch the ROS node:

```
$ roslaunch hugin_panorama hugin_panorama.launch image:=/
    image_topic
```

3. Save individual images for input to the panorama: (order doesn't matter)

```
$ rosservice call /hugin_panorama/image_saver/save
# change angle of camera
$ rosservice call /hugin_panorama/image_saver/save
# repeat as many times as you like...
```

4. Stitch the panorama:

```
$ rosservice call /hugin_panorama/stitch
```

5. View resulting panorama:

```
$ rqt_image_view /hugin_panorama/panorama/compressed
# or open the panorama file
$ roscd hugin_panorama; eog ./images/output.png
```

6. Start again:

```
$ rosservice call /hugin_panorama/reset
```

This command will clear the images waiting to be stitched so you can start collecting images for an entirely new panorama.

11.2 Live Panorama Mode

If you have more than one camera on your robot and you want to stitch them together repetitively in a loop, then use stitch_loop.launch. However, expect a slow frame rate of less than 1 Hz because this package is not optimized for speed.
1. Launch the stitch_loop node:

```
$ roslaunch hugin_panorama stitch_loop.launch image1:=/
    image_topic2 image2:=/image_topic2
```

2. View resulting live panorama:

```
$ rqt_image_view /hugin_panorama/panorama/compressed
```

If you have more than two cameras then the quick fix is you'll to edit the simple python script (`rosed hugin_panorama stitch_loop.py`) and the launch file (`rosed hugin_panorama stitch_loop.launch`) to duplicate some parts.

12 Tutorial: GPS Navigation Goal

In the autonomous task of the URC competition, a series of goal locations are given to teams as approximate GPS coordinates. Rovers are expected to autonomously drive to the GPS location and then find and stop near a tennis ball marker. To achieve the GPS navigation requirements of this task, Team R3 has created the gps_goal[39,40] package. We believe this is the first packaged for ROS solution to convert navigation goals in given in GPS coordinates to ROS frame coordinates. The package uses one known GPS location in the ROS frame to facilitate converting between coordinate systems. Fig. 20 shows how this package fits into Team R3's larger ROS software architecture.

The new gps_goal package uses the WGS84 ellipsoid[41] and geographiclib[42] python library to calculate the surface distance between GPS points. WGS84 is the standard coordinate system for GPS and thus the packages configures GreographicLib to use it because it is important for calculating the correct distance between GPS points.

The GPS goal can be set using a geometry_msgs/PoseStamped or sensor_msgs/NavSatFix message. The robot's desired yaw, pitch, and roll can be set in a PoseStamped message but when using a NavSatFix they will always be set to 0 degrees.

The goal is calculated in a ROS coordinate frame by comparing the goal GPS location to a known GPS location at the origin (0,0) of a ROS frame given by the local_xy_frame ROS parameter which is typically set to 'world' but can be any ROS frame. This initial origin GPS location is best published using a helper initialize_origin node (see section 12.3 below for more details).

12.1 Usage Instructions

1. Install the ROS package:

[39] Source code for gps_goal ROS package `https://github.com/danielsnider/gps_goal`

[40] Wiki page for gps_goal ROS package `http://wiki.ros.org/gps_goal`

[41] The World Geodetic System (WGS) 84 is the reference coordinate system used by the Global Positioning System (GPS). WGS84 uses degrees. It consists a latitudinal axis from -90 to 90 degrees and a longitudinal axis from -180 to 180 degrees. As it is the standard coordinate system for GPS it is also commonly used in robotics. `https://en.wikipedia.org/wiki/World_Geodetic_System#A_new_World_Geodetic_System:_WGS_84`

[42] The GreographicLib software library `https://geographiclib.sourceforge.io/`

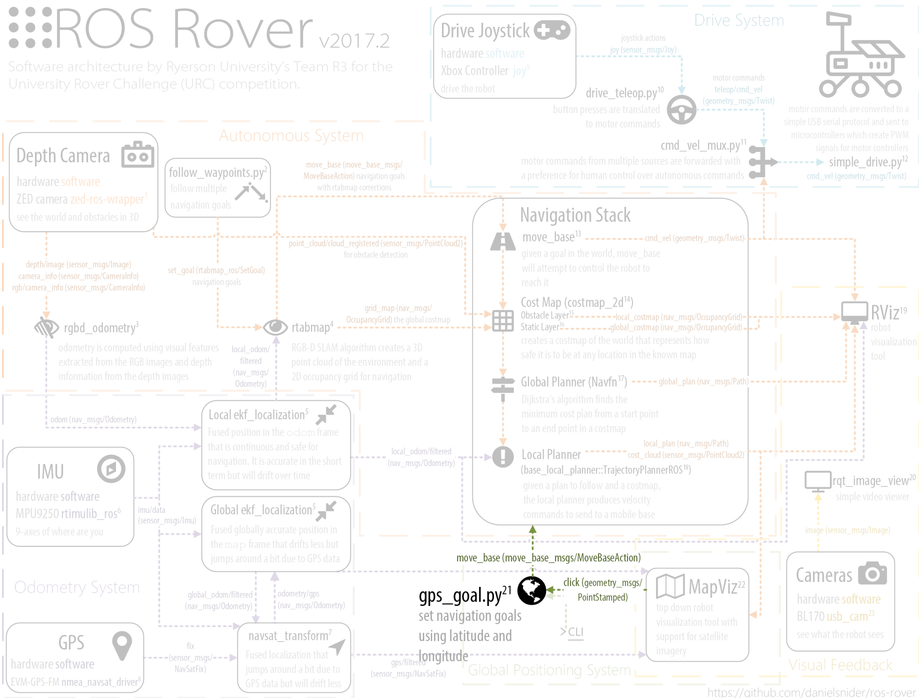

Fig. 20. The gps_goal ROS node seen within Team R3's rover system. See fig. 9 for the full diagram.

```
$ sudo apt-get install ros-kinetic-gps-goal
```

2. Launch the ROS node:

```
$ roslaunch gps_goal gps_goal.launch
```

3. Set a known GPS location using one of the following approaches (a) or (b). The given GPS location will be attached to the origin (0,0) of the ROS frame given by the local_xy_frame ROS parameter. This is used to calculate the distance to the goal.

3a. Use the next GPS coordinate published on a ROS topic (requires package ros-kinetic-swri-transform-util):

```
$ roslaunch gps_goal initialize_origin.launch origin:=auto
```

3b. Or set the initial origin manually using a rostopic publish command:

```
$ rostopic pub /local_xy_origin geometry_msgs/PoseStamped '{
    header: { frame_id: "/map" }, pose: { position: { x:
    43.658, y: -79.379 } } }' -1
```

4. Set a navigation goal using GPS coordinates set with either a Pose or NavSatFix GPS message.

```
$ rostopic pub /gps_goal_fix sensor_msgs/NavSatFix "{latitude
    : 38.42, longitude: -110.79}" -1
OR
$ rostopic pub /gps_goal_pose geometry_msgs/PoseStamped '{
    header: { frame_id: "/map" }, pose: { position: { x:
    43.658, y: -79.379 } } }' -1
```

12.2 Command Line Interface (CLI)

Alternatively, a Command Line Interface (CLI) is available to set GPS navigation goals. When using the CLI interface you can use one of two coordinate formats: either degree, minute, and seconds (DMS) or decimal GPS format. Using command line arguments, users can also set the desired roll, pitch, and yaw final position. You can invoke the gps_goal script once using the Command Line Interface (CLI) with any of the following options.

Listing 1.7. Example usages of the gps_goal CLI script to set a navigation goal.

```
$ roscd gps_goal
$ ./src/gps_goal/gps_goal.py --lat 43.658 --long -79.379 #
    decimal format
OR
$ ./src/gps_goal/gps_goal.py --lat 43,39,31 --long -79,22,45
    # DMS format
```

12.3 initialize_origin Helper ROS Node

The initialize_origin node will continuously publish (actually in a latched[43] manner) a geometry_msgs/PoseStamped on the local_xy_origin topic and this is the recommended approach over manually publishing the origin GPS location with rostopic pub. This location is the origin (0,0) of the frame (typically world) given by the local_xy_frame parameter to the initialize_origin node. This location is used to calculate distances for goals. One message on this topic is consumed when the node starts only.

This node is provided by the swri_transform_util package (apt-get install ros-kinetic-swri-transform-util) and it is often launched as a helper node for MapViz, a top-down robot and world visualization tool that is detailed in section 15. There are two modes for initialize_origin: static or auto.

Static Mode You can hard code a GPS location (useful for testing) for the origin (0,0). In the following example the coordinates for the Mars Desert Research Station (MDRS) are hard coded in initialize_origin.launch and selected on the command line with the option "origin:=MDRS".

```
$ roslaunch gps_goal initialize_origin.launch origin:=MDRS
```

Auto Mode When using the "auto" mode, the origin will be to the first GPS fix that it receives on the topic configured in the initialize_origin.launch file.

```
$ roslaunch gps_goal initialize_origin.launch origin:=auto
```

Launch example Starting the initialize_origin ROS node can be done in the following way.

Listing 1.8. An example launch config to start the initialize_origin ROS node.

```
<node pkg="swri_transform_util" type="initialize_origin.py"
   name="initialize_origin" output="screen">
  <param name="local_xy_frame" value="/world"/>
  <param name="local_xy_origin" value="MDRS"/> <!-- setting "
     auto" here will set the origin to the first GPS fix
     that it receives -->
  <remap from="gps" to="gps"/>
  <rosparam param="local_xy_origins">
    [{ name: MDRS,
       latitude: 38.40630,
       longitude: -110.79201,
       altitude: 0.0,
       heading: 0.0}]
```

[43] When a connection is latched, the last message published is saved and automatically sent to any future subscribers of that connection.

```
  </rosparam>
</node>
```

12.4 Normal Output

When you launch and use the gps_goal ROS node or CLI interface you will see the following console output.

```
$ roscd gps_goal
$ ./src/gps_goal/gps_goal.py --lat 43.658 --long -79.379

[INFO]: Connecting to move_base...
[INFO]: Connected.
[INFO]: Waiting for a message to initialize the origin GPS
    location...
[INFO]: Received origin: lat 43.642, long -79.380.
[INFO]: Given GPS goal: lat 43.658, long -79.379.
[INFO]: The distance from the origin to the goal is 97.3 m.
[INFO]: The azimuth from the origin to the goal is 169.513
    degrees.
[INFO]: The translation from the origin to the goal is (x,y)
    91.3, 13.6 m.
[INFO]: Executing move_base goal to position (x,y) 91.3,
    13.6, with 138.14 degrees yaw.
[INFO]: To cancel the goal: 'rostopic pub -1 /move_base/
    cancel actionlib_msgs/GoalID -- {}'
[INFO]: Inital goal status: PENDING
[INFO]: Final goal status: COMPLETE
```

13 Tutorial: Wireless Communication

At Team ITU (Istanbul Technical University), communication from our base station to our mobile outdoor rover was of utmost importance and received a good amount of development time. Non-line of sight (NLOS) communication is an important issue and often a cause of unsuccessful runs in URC. Although there are various commercial products available claiming to solve the issues of long range and high throughput communication, many of them don't solve the problems as advertised. So a combination of systems and products were tested and used in the competition[44,45].

The primary system must be capable of delivering a video feed to the ground station for operators to use while piloting the vehicle remotely. This system is

[44] Team ITU's low level communication source code https://github.com/itu-rover/2016-2017-Sensor-GPS-STM-Codes-/blob/master/STM32/project/mpu_test/Src/main.c

[45] Team ITU's high level communication source code https://github.com/itu-rover/communication

generally preferred to be a Wi-Fi network that can multiplex high-res video and data traffic at high speeds. After performing a series of unsuccessful non-line of sight (NLOS) tests at long distances (400-500 meters) with the Ubiquiti Bullet M2, a popular 2.4 GHz product, a less popular Microhard pDDL2450 module was chosen based on our sponsor's advice. Thankfully, the sponsor had a few spares and donated them to Team ITU. In tests, this 2.4 GHz module was generally successful in sending useful data from the vehicle's instruments and 720p video feed compressed with MJPEG from five cameras at the same time over a 1 km range in NLOS course with 8dBi omnidirectional antennas. This module is physically connected to the high level on-board computer (OBC), a Raspberry Pi 3 with running Ubuntu 16.04, with a standard CAT5 Ethernet cable.

Secondly, our radio frequency (RF) backup link was able to pass over the natural obstacles such as large rocks and hills. This link operates on lower UHF frequencies and lower baud rates to increase performance needed to send crucial information about the vehicle's condition to ensure health and function of the rover at extreme distances (5km). We consider our rover's heartbeat, GPS position, attitude and current speed to be important data that should be sent through the RF link. Tests were conducted using 433Mhz LoRa modules (see Fig. 21) on both sides with 3dBi omnidirectional antennas, in 9600 and 115200 baud rates. The tests showed that these modules have no problem sending the data over a 5 km range in NLOS conditions. The LoRa modules were wired to the standard RX-TX wires on our STM32F103 microprocessor and uses UART communication. The microprocessor also controls the driving system, communicates with the sensors, the Raspberry Pi 3 on-board computer (OBC).

Fig. 21. UHF LoRa radio module for long range communication. Image credit: Semtech Corporation

To make the LoRa RF link active in the C# language see Listing 1.9.

Listing 1.9. Start communication with the LoRa RF module in the C# language.

```
SerialPort _serialPort = new SerialPort("COM1", 115200,
    Parity.None, 8, StopBits.One);
_serialPort.Open();
```

In normal, connected conditions, only the Wi-Fi network system is active and the RF system is inactive. In these conditions all the processing is done in the Raspberry Pi 3 on-board computer (OBC). Commands from the human pilot reach the OBC first and then are distributed to the low level STM microprocessor via a RS232 link. The video feed is active and the pilot can easily drive the rover using the video images from the cameras. The low band RF system was remarkably reliable although interruptions were encountered at times with the Wi-Fi network.

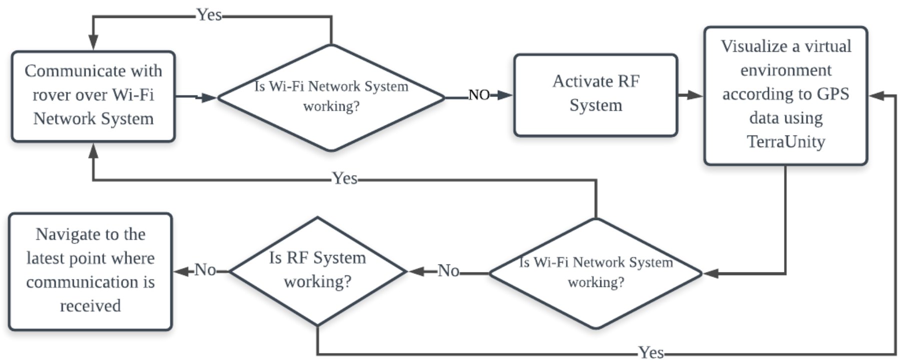

Fig. 22. Flowchart of decision making algorithm used on ITU's rover.

In the case of losing the Wi-Fi network system or OBC, the video feed will be lost and piloting the vehicle without a video feed is almost impossible. So an innovative solution was implemented to overcome this problem and continue the mission. In such condition, first the RF link is activated and crucial information and the pilot commands are redirected to this link. Therefore, the pilot directly communicates with low level processor. To help the pilot visualize the environment a virtual environment around the GPS coordinates is simulated with computer graphics. This visual environment is created using the TerraUnity software. The software creates a one-to-one, colorized, topographical landscape with natural objects loaded from a 3D map database of the location[46]. This way the driver could look at the ground station monitor and see the terrain around the vehicle in the Unity graphics simulation, which was pretty precise in our tests. The software was found to be very successful in creating a realistic environment and it gives a clue to the driver about where the vehicle is and what natural obstacles are around it. Knowing the locations of natural obstacles is essential as the rover has physical limits that prevents it from navigating some

[46]TerraUnity computer graphics software used to the visualize the rover at its GPS location in a topographical simulation of earth `http://terraunity.com/`

terrain. Although the TerraUnity solution is an imperfect representation of the world, it provides a useful avenue to continue the mission in a catastrophic failure scenario.

Finally, in the case where both Wi-Fi and RF communication is lost to the rover, a third backup system is initialized where the rover navigates autonomously to the last GPS point that it communicated successfully with the ground station.

14 Tutorial: Autonomous Navigation by Team R3

In this section we present Team R3's (Ryerson University) autonomous software architecture that was designed for outdoor autonomous driving at the University Rover Competition (URC) 2017. The design uses a stereo camera and SLAM to navigate to a goal autonomously and avoid static obstacles. For a description of the requirements of the autonomous task for URC 2017, please refer back to Section 2.2 of this chapter.

In the following subsections we will elaborate on the ZED stereo camera, rbgd_odometry, RTAB-Map SLAM, and move_base. Fig. 23 depicts a diagram of Team R3's autonomous software design. To see this diagram within a larger diagram with more of Team R3's rover software components see Section 5.

14.1 ZED Depth Camera

The ZED stereo camera[49,50,51] and ROS wrapper software perform excellently for the price of $450. With the ZED camera Team R3 was able to avoid obstacles such as rocks and steep cliffs. However, the rover could not move quickly, no faster than slow-moderate human walking speed because the performance of our on board computer, a Nvidia Jetson TX1[52], was fully utilized. It is important to know that a restriction of the ZED camera is that it requires an Nvidia GPU, a dual-core processor, and 4GB of RAM. All of which the Nvidia Jetson TX1 has.

The ZED camera combined with RTAB-Map for SLAM localization and mapping worked reasonably robustly even in Utah's desert where the ground's feature complexity is low and even with a significant amount of shaking on the pole which our ZED camera was attached to.

[48] Team R3's autonomous software architecture diagram in Microsoft Visio format `https://github.com/danielsnider/ros-rover/blob/master/diagrams/team_r3_AUTO_Diagram.vsdx?raw=true`

[49] ZED stereo camera technical specs `https://www.stereolabs.com/zed/`

[50] More ZED camera documentation `https://www.stereolabs.com/documentation/guides/using-zed-with-ros/ZED_node.html`

[51] Team R3's ZED launch file `https://github.com/teamr3/URC/blob/master/rosws/src/rover/launch/zed_up.launch`

[52] Nvidia Jetson TX1 technical specs `https://developer.nvidia.com/embedded/buy/jetson-tx1-devkit`

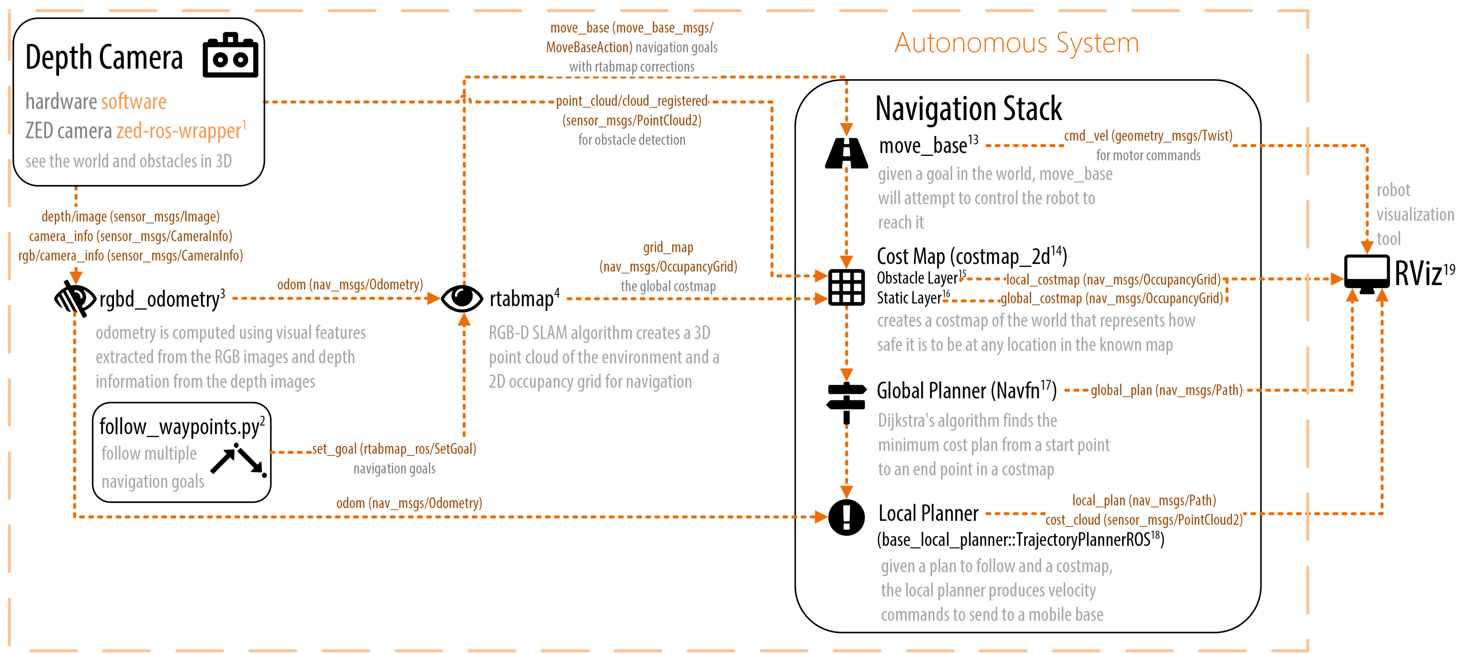

Fig. 23. Team R3's autonomous navigation system used at URC 2017 rover competition. This diagram is also available in Visio format[48].

Fig. 24. The ZED depth camera. Image credit: Stereolabs

Here is a tip when using the ZED camera, launch the node with arguments so you can more easily find the right balance between performance and resolution. At URC we wanted the lowest latency so we default to VGA resolution, at 10 FPS, and low depth map quality. Also, note that the ZED camera is designed for outdoor textured surfaces. Indoor floors that are featureless which will make testing more difficult. Also when testing indoors, you may can a blinder on top of the camera so that it doesn't see the ceiling as an obstacle.

```
$ roslaunch rover zed_up frame_rate:=30 resolution:=2
   depth_quality:=3
```

```
1  <launch>
2      <arg name="frame_rate" default="10"/>
3      <arg name="resolution" default="3"/>
4      <arg name="depth_quality" default="1"/>
5      <node output="screen" pkg="zed_wrapper" name="zed_node"
           type="zed_wrapper_node">
6          <param name="frame_rate" value="$(arg frame_rate)"/>
7          <!-- Image resolution options: -->
8          <!--  0 ': HD2K,  1 ': HD1080,  2 ': HD720,  3 ':
               VGA -->
9          <param name="resolution" value="$(arg resolution)"/>
10         <!-- Depth map quality options: -->
11         <!--  0 ': NONE,  1 ': PERFORMANCE,  2 ': MEDIUM,
                3 ': QUALITY -->
12         <param name="quality" value="$(arg depth_quality)"/>
13     </node>
14     <node pkg="image_transport" type="republish" name="
           zed_camera_feed" args="raw in:=rgb/image_rect_color
           out:=rgb_republished"/>
15 </launch>
```

Reduce Bandwidth Used by Video Streams To lower the amount of data on our wireless link, on line 14 we publish the ZED camera as JPEG compressed stills and Theora video streaming using the republish[53] node of the image_transport ROS package. republish listens on one uncompressed (raw) image topic and republishes JPEG compressed stills and Theora video on different topics.

To lower bandwidth even further you can convert images to greyscale, cutting data usage by 3. Team R3 has a small ROS node for this[54].

Additionally, you should use the republish node when more than one ROS node is subscribing to a depth or image stream over a wireless connection. Instead you should have one republish node subscribe at the base station, then multiple ROS nodes at the base station can subscribe to the republish node without consuming a lot of wireless bandwidth. This is also referred to as a ROS relay.

Another easy way to reduce bandwidth used by the ZED camera is to downsample its pointcloud using the VoxelGrid nodelet in the pcl_ros[55] ROS package.

14.2 Visual Odometry with rgbd_odometry

The ZED camera does not have a gyroscope or accelerometer in it. It uses visual information for odometry and it is quite good. We found that the rgbd_odometry[56,57] node provided by the RTAB-Map package produces better visual odometry than the standard ZED camera odometry algorithm. Visual odometry was very robust to jitter and shaking as the rover moved over rough terrain, even with our camera on a tall pole which made the shaking extreme. Optimizations to rgbd_odometry used by Team R3 at the URC 2017 rover competition are shown in Listing 1.10

```
<launch>
    <node output="screen" type="rgbd_odometry" name="zed_odom" pkg="rtabmap_ros">
        <!-- 2D SLAM makes the position drift less over time -->
        <param name="Reg/Force3DoF" type="string" value="true"/>
        <!-- Change if camera is tilted downwards or any non-level pose -->
        <param name="initial_pose" value="0 0 0 0 0 0"/>

        <!-- Options to Reduce Resource Usage -->
        <!-- 0=Frame-to-Map (F2M) 1=Frame-to-Frame (F2F) -->
```

[53]republish ROS node documentation `http://wiki.ros.org/image_transport#republish`

[54]Python ROS script to reduce bandwidth usage of video streams `https://github.com/teamr3/URC/blob/master/rosws/src/rover/src/low_res_stream.py`

[55]pcl_ros ROS documentation `http://wiki.ros.org/pcl_ros`

[56]rgbd_odometry ROS node documentation `http://wiki.ros.org/rtabmap_ros#rgbd_odometry`

[57]Team R3's rgbd_odometry launch file `https://github.com/teamr3/URC/blob/master/rosws/src/rover_navigation/launch/rgbd_odometry.launch`

```
        <param name="Odom/Strategy" value="1"/>
        <!-- Correspondences: 0=Features Matching, 1=Optical Flow -->
        <param name="Vis/CorType" value="1"/>
        <!-- maximum features map size, default 2000 -->
        <param name="OdomF2M/MaxSize" type="string" value="1000"/>
        <!-- maximum features extracted by image, default 1000 -->
        <param name="Vis/MaxFeatures" type="string" value="600"/>
    </node>
</launch>
```

Listing 1.10. Important settings to optimize the rgbd_odometry ROS node.

14.3 3D Mapping in ROS with RTAB-Map

Using depth camera data, RTAB-Map[58,59] creates a continuously growing point cloud of the world using simultaneous localization and mapping (SLAM)[4]. Inherent to the SLAM algorithm is pin pointing your own location in the map that you are building as you move. Using this map, RTAB-Map then creates an occupancy grid map[3], which represents free and occupied space, needed to avoid obstacles in the rover's way. RTAB-Map's algorithm has real-time constraints so that when mapping large-scale environments time limits are respected and performance does not degrade[6].

In the launch file seen in Listing 1.11 on lines 3-5 show configurations to reduce noisy detection of obstacles. If you set `MaxGroundAngle` to 180 degrees, this effectively disables obstacle detection, which can be both useful and dangerous.

RTAB-Map also performs loop closures. Loop closure is the problem of recognizing a previously-visited location and updates the beliefs accordingly[62]. When an image is matched to a previously-visited location, a loop closure is said to have occurred. At this point RTAP-Map will adjust the map to compensate for drift that occurred since the last time the location was visited. Lines 8-10 of listing 1.11 increase the likelihood of loop closures being detected.

```
1 <launch>
2     <node pkg="rtabmap_ros" name="rtabmap" type="rtabmap" output="screen">
3         <!-- Improve obstacle detection -->
```

[58] RTAB-Map documentation `http://wiki.ros.org/rtabmap_ros`

[59] Team R3's launch file for RTAB-Map `https://github.com/teamr3/URC/blob/master/rosws/src/rover_navigation/launch/rtabmap.launch`

[61] Video of an autonomous navigation by Team R3 with RTAB-Map and the ZED stereo camera `https://www.youtube.com/watch?v=p_1nkSQS8HE`

[62] More information about loop closures `https://en.wikipedia.org/wiki/Simultaneous_localization_and_mapping#Loop_closure`

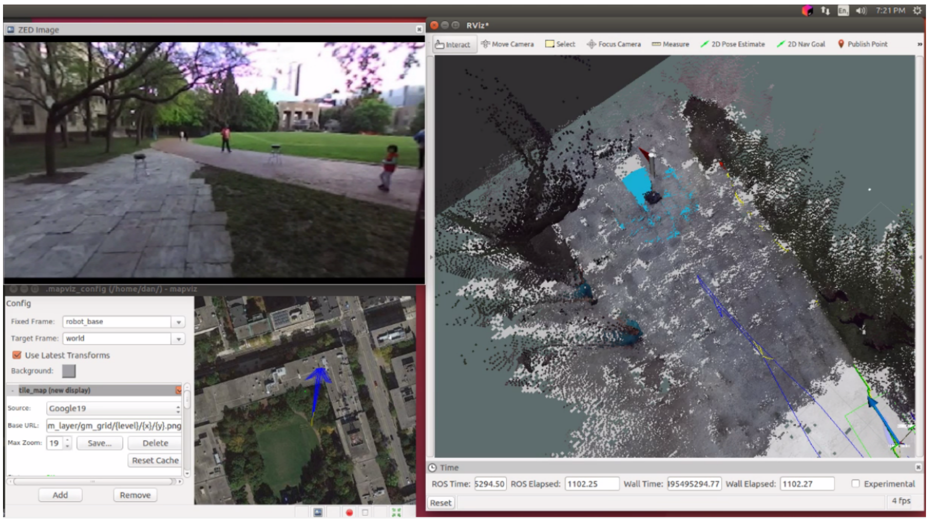

Fig. 25. Screenshot of R3's autonomous system tests with RTAB-Map (video available on YouTube[61])

```
        <param name="Grid/MaxGroundAngle" value="110"/> <!--
            Maximum angle between point's normal to ground's
            normal to label it as ground. Points with higher
            angle difference are considered as obstacles.
            Default: 45 -->
        <param name="grid_eroded" value="true"/> <!-- remove
            obstacles which touch 3 or more empty cells -->

        <!-- Improve loop closure chances -->
        <param name="RGBD/LoopClosureReextractFeatures" type=
            "string" value="true"/> <!-- Extract features
            even if there are some already in the nodes, more
             loop closures will be accepted. Default: false
            -->
        <param name="Vis/MinInliers" type="string" value="10"
            /> <!-- Minimum feature correspondences to
            compute/accept the transformation. Default: 20 --
            >
        <param name="Vis/InlierDistance" type="string" value=
            "0.15"/> <!-- Maximum distance for feature
            correspondences. Used by 3D->3D estimation
            approach (the default approach). Default: 0.1 -->
    </node>
</launch>
```

Listing 1.11. Important settings for tuning the RTAB-Map 3D mapping ROS package.

Team R3's main strategy for the autonomous task of URC 2017 was to:

1. Build a SLAM map by teleoperating from the start gate all the way to the tennis ball objective (actually this could be an autonomous navigation attempt by using the GPS location of the tennis ball as the goal), 2. then we would then put a flag[63] in RViz to mark where we observed the tennis ball, 3. and then we would teleoperate back to the start gate, 4. and complete a loop closure to correct for drift, 5. and then use RViz to set an autonomous goal for where we saw the tennis ball.

14.4 move_base Path Planning

The ROS navigation stack[64], also known as move_base[65], is a collection of components/plugins that are selected and configured by YAML configuration files as seen in Listing 1.12. For global path planning, the NavFn plugin is used which implements Dijkstra's shortest path algorithm.

```xml
<launch>
    <node pkg="move_base" type="move_base" name="move_base" output="screen" clear_params="true">
        <rosparam file="$(find rover)/costmap_common_params.yaml" command="load" ns="global_costmap"/>
        <rosparam file="$(find rover)/costmap_common_params.yaml" command="load" ns="local_costmap"/>
        <rosparam file="$(find rover)/local_costmap_params.yaml" command="load"/>
        <rosparam file="$(find rover)/global_costmap_params.yaml" command="load"/>
        <rosparam file="$(find rover)/base_local_planner_params.yaml" command="load"/>
    </node>
</launch>
```

Listing 1.12. move_base is configured by independent YAML files that configure the subcomponents of move_base.

The most interesting configuration file for move_base is the base_local_planner_params.yaml[66]. Given a path for the robot to follow and a costmap, the base_local_planner[67] produces velocity commands to send to a mobile base. This configuration is where you set minimum and maximum velocities and accelerations for your robot, as

[63] RViz flag tool http://docs.ros.org/jade/api/rviz_plugin_tutorials/html/tool_plugin_tutorial.html

[64] ROS Navigation Stack http://wiki.ros.org/navigation

[65] Team R3's move_base configuration file https://github.com/teamr3/URC/blob/master/rosws/src/rover_navigation/launch/move_base.launch

[66] Team R3's config file for base_local_planner_params.yaml configuration of move_base https://github.com/teamr3/URC/blob/master/rosws/src/rover_navigation/config/base_local_planner_params.yaml

[67] Documentation for base_local_planner http://wiki.ros.org/base_local_planner

well as goal tolerance. Make sure that the minimum velocity multiplied by the sim_period is less than twice the tolerance on a goal. Otherwise, the robot will prefer to rotate in place just outside of range of its target position rather than moving towards the goal.

```
TrajectoryPlannerROS:
  acc_lim_x:  0.5
  acc_lim_y:  0.5
  acc_lim_theta: 1.00

  max_vel_x:  0.27
  min_vel_x:  0.20
  max_rotational_vel: 0.4
  min_in_place_vel_theta: 0.27
  max_vel_theta: 0.1
  min_vel_theta: -0.1
  escape_vel: -0.19

  xy_goal_tolerance: 1
  yaw_goal_tolerance: 1.39626 # 80 degrees

  holonomic_robot: false
  sim_time: 1.7 # set between 1 and 2. The higher he value, the smoother the path (though more samples would be required)
```

Listing 1.13. Important velocity settings in the base_local_planner_params.yaml configuration file of move_base.

15 Tutorial: MapViz Robot Visualization Tool

At the URC competition, Team R3 (Ryerson University) improved their situational awareness by using MapViz[68]. Mapviz is a ROS-based visualization tool with a plug-in system similar to rviz but focused only on a top down, 2D view of data. Any 3D data is flatening into the 2D view of MapViz. Created by the Southwest Research Institute in Florida for their outdoor autonomous robotics research, it is still under active open source development at the time of writing in December 2017. Using a plugin called tile-map, Google Maps satellite view can be viewed in MapViz Tile Map plugin.

The authors have contributed a Docker container[69] to make displaying Google Maps in MapViz as easy as possible. This container run software called MapProxy and which converts from the format of Google Maps API to a standard format called Web Map Tile Service (WMTS) which MapViz Tile Map plugin

[68] Documentation and source code for MapViz `https://github.com/swri-robotics/mapviz`

[69] Docker container for proxying Google Maps to MapViz`https://github.com/danielsnider/MapViz-Tile-Map-Google-Maps-Satellite`

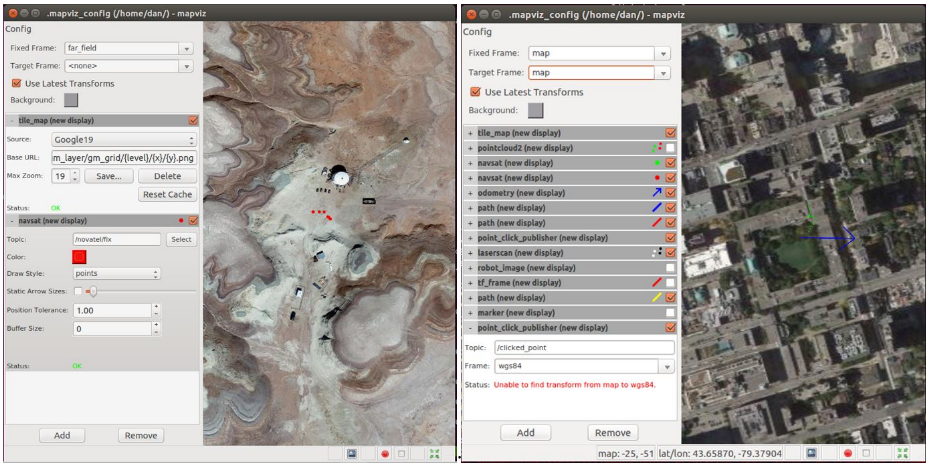

Fig. 26. Screenshots of MapViz ROS visualization tool. Red dots indicate gps coordinates. Many more visualization layers are possible.

can display. The authors have set MapProxy's configuration[70] to cache any maps that you load to `~/mapproxy/cache_data/` so that they are available offline.

15.1 Usage Instructions

The goal of this tutorial is to install and configure MapViz to display Google Maps.

1. Install mapviz, the plugin extension software, and the plugin for supporting tile maps which is needed to display Google Maps:

```
$ sudo apt-get install ros-kinetic-mapviz ros-kinetic-mapviz-
    plugins ros-kinetic-tile-map
```

2. Launch the MapViz GUI application:

```
$ roslaunch mapviz mapviz.launch
```

3. Using Docker to setup a proxy of the Google Maps API so that it can be cached and received by MapViz in WMTS format. To make this as simple as possible, run the Docker container created by the authors:

```
$ sudo docker run -p 8080:8080 -d -t -v ~/mapproxy:/mapproxy
    danielsnider/mapproxy
```

The `-v ~/mapproxy:/mapproxy` option is a shared volume, a folder that it synced between the Docker container and the host computer. The `~/mapproxy`

[70] MapProxy config file `https://github.com/danielsnider/docker-mapproxy-googlemaps/blob/master/mapproxy.yaml`

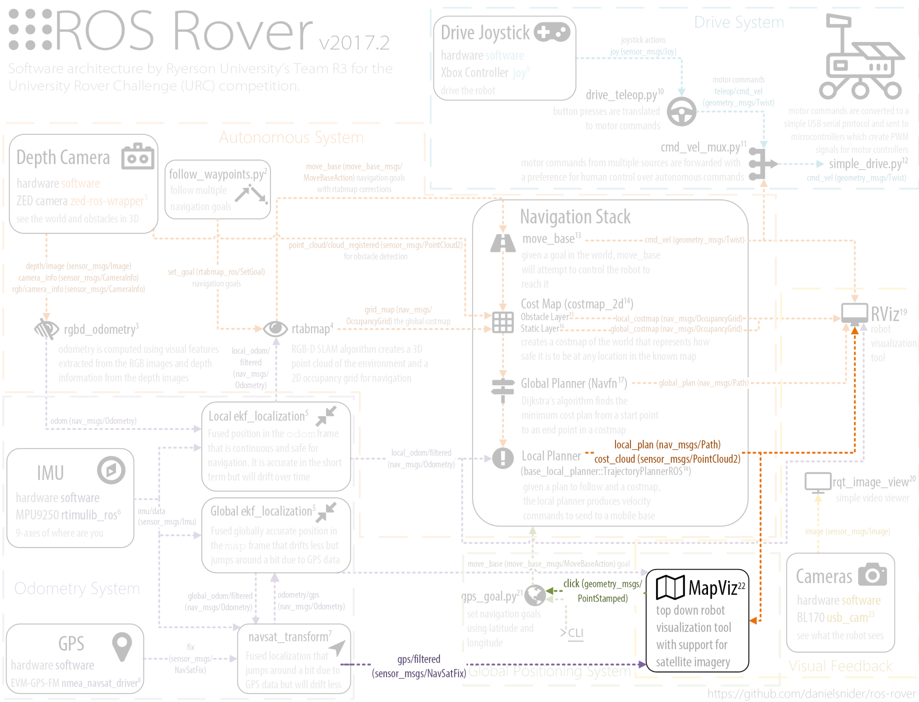

Fig. 27. MapViz ROS node seen within Team R3's rover system. See fig. 9 for the full diagram.

folder needs to be created, though it could be another location. The `-t` option allocates a pseudo-tty which gives the program a terminal environment to run in. It is needed for most programs. The `-p` option sets the Docker port mapping between host and container.
4. Confirm MapProxy is working by browsing to `http://127.0.0.1:8080/demo/`. The MapProxy logo will be displayed and you can click on "Image-format png" to get an interactive map. Also, test that the first map tile is working by browsing to `http://localhost:8080/wmts/gm_layer/gm_grid/0/0/0.png`.
5. In the MapViz GUI, click the "Add" button and add a new map_tile display component.
6. In the "Source" dropdown select "Custom WMTS Source...".
7. In the "Base URL:" field enter the following: `http://localhost:8080/wmts/gm_layer/gm_grid/{level}/{x}/{y}.png`
8. In the "Max Zoom:" field enter 19 and Click "Save...". This will permit MapViz to zoom in on the map 19 times.

Google Maps will now display in MapViz. To set a default location in the world to display at program start up time, you can edit `~/.mapviz_config`.

```
$ vim ~/.mapviz_config
# edit the following lines
offset_x: 1181506
offset_y: -992564.2
```

Listing 1.14. MapViz setting for default viewing location (within a ROS frame) when GUI opens.

16 Tutorial: Effective Robot Administration

In this tutorial, Team R3 (Ryerson University) shares two favorite preferences for making command line administration of ROS robots easier both for the URC competition and any other use case.

16.1 tmux Terminal Multiplexer

Tmux[71] is a popular linux command line program that can take over one terminal window and organize many terminals into grouped layouts and will continue running when you close the parent window or lose an SSH connection. Multiple people can join a tmux session to share an identical terminal view of a Linux system. Many technology professionals (especially linux and IT professionals) see tmux as essential to their workflow[72].

Tmux works harmoniously with ROS's modular design. Separate tmux windows can display different ROS components. Almost any ROS component can

[71] Homepage for Tmux the terminal multiplexer `https://github.com/tmux/tmux/wiki`

[72] A crash course to learn tmux `https://robots.thoughtbot.com/a-tmux-crash-course`

be launched, controlled, and debugged using ROS's command line tools. Using tmuxinator[73], you can codify the launching and debugging commands that you most often use into a repeatable layout.

At URC 2017, Team R3 used tmuxinator to launch all our robot's software systems. There was a section that contained the terminals running the drive software, another section with terminals running the arm software, another for the IMU, for the GPS, for the cameras, etc. All of it in an organized way. Just about every software can be started on the command line using tmuxinator after the rover's computer boots.

The tmuxinator configuration used by Team R3 was split into two sides: the robot config[74], and the base station config[75]. The robot configuration launches all of the rover's software. The base station configuration launches all of the software needed to visualize and control the robot remotely by the teleoperator.

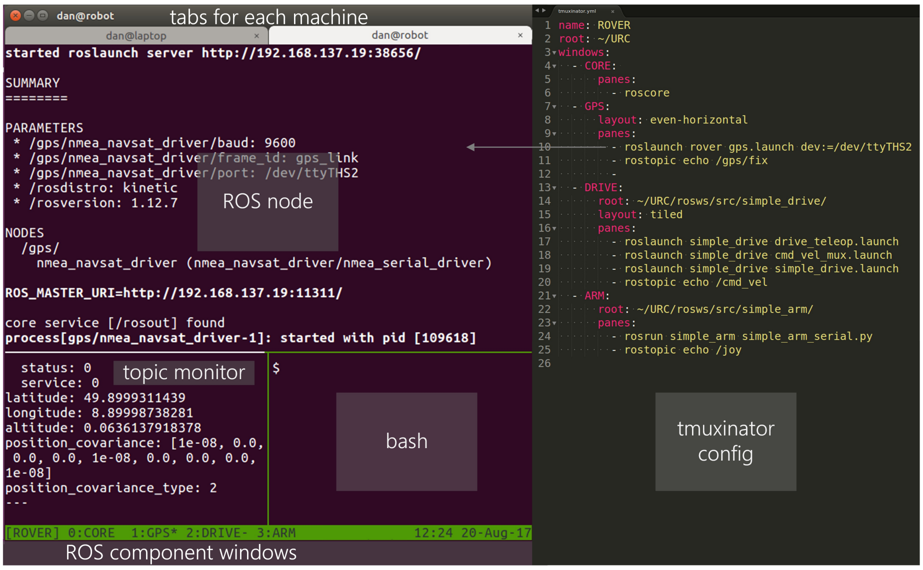

Fig. 28. An annotated example of tmuxinator's usefulness for ROS.

Using Tmuxinator can be thought of as is a quick way to create a very simple user interface to help administer a robot (but it is not a replacement for Rviz

[73]Tmuxinator is an tool for tmux that lets to write tmux layout configuration files for repeatable layouts `https://github.com/tmuxinator/tmuxinator`

[74]The tmuxinator config used by Team R3 to start all the rover software components `https://github.com/teamr3/URC/blob/master/.tmuxinator.yml`

[75]The tmuxinator config used by Team R3 to start all the base station software components `https://github.com/teamr3/URC/blob/master/devstuff/dan/.tmuxinator.yml`

and other existing tools). Building a robot GUI as a web interface or desktop application can be useful for some applications and novices who are unwilling to learn common command line tools, but such a GUI will require a lot more “plumbing” and “glue code” to create.

An implementation imperfection is that tmuxinator starts all of its panes (i.e. terminals) at the same time, running multiple roslaunch instances may try and fail to create multiple masters. The solution we implemented was to run a roscore separately, which has the added benefit of being able to stop and start roslaunches without worrying about which one is running the master. This still has the problem of roslaunches starting before the roscore though, so to solve this naively we have used a small wait time, for example “sleep 3; roslaunch...” in our tmuxinator config.

16.2 ROS Master Helper Script

Team R3 has developed a script[76] to make it a little easier to connect your computer to a remote master that is not on your machine. The script when run will automatically set bash environment variables needed for ROS networking to work in a convenient way. The script will detect if your robot is online using ping (using a static IP for your robot) and set your ROS_MASTER_URI environment variable to point to your robot. If your robot is not online your own computer's IP will be used for your ROS_MASTER_URI, assuming you will do local or simulation development since you are away from your robot. To use this script run `source set_robot_as_ROS_master.sh` or add it to your `~/.bashrc`. Also, the script sets your own machine's ROS_IP environment variable because it is needed in any case for ROS networking.

17 Conclusion

This chapter presented an overview of rover systems through the lenses of the University Rover Competition. Design summaries of 8 URC teams were surveyed and implementation details from 3 URC teams were discussed in a series of tutorials. Several new ROS packages were documented with examples, installation and usage instructions along with implementation details.

To summarize the main findings in this chapter: Rovers can be built by integrating existing software as thanks to the ROS ecosystem. The URC competition is very challenging and students learned a lot by participating. A variety of creative rover designs exist and the best rover teams were the most prepared and practiced.

We hope this chapter spurs greater collaboration between teams. Ideally, the teams of URC will look past the competitive nature of the event and view collaborating and building better robots as the more important goal. Building

[76] Team R3's ROS master helper script `https://gist.github.com/danielsnider/13aa8c21e4fb12621b7d8ba59a762e75`

on a common core frees up time to focus on the hardest parts. Here are a few ways to further collaboration: 1) Contribute to a ROS package or the ROS core. 2) Open an issue, feature request, or pull request. 3) Discuss URC on the URC Hub forum. 4) Discuss ROS on their forum. 5) Contribute to a book like this.

Acknowledgments We thank our advisor Professor Michael R. M. Jenkin P.Eng., Professor of Electrical Engineering and Computer Science, York University, NSERC Canadian Field Robotics Network. We also thank and appreciate the contributions of our survey respondents: Khalil Estell from San Jose State University, SJSU Robotics, and Jacob Glueck from Cornell University, Cornell Mars Rover, and Hunter D. Goldstein from Cornell University, Cornell Mars Rover, and Akshit Kumar from Indian Institute of Technology, Madras, Team Anveshak, and Jerry Li from University of Waterloo, UWRT, and Gabe Casciano from Ryerson University, Team R3, and Jonathan Boyson from Missouri University of Science and Technology (Missouri S&T), Mars Rover Design Team.

References

[1] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000).

[2] *Camera Module - Raspberry Pi Documentation.* URL: `https://www.raspberrypi.org/documentation/hardware/camera/` (visited on 12/23/2017).

[3] Péter Fankhauser and Marco Hutter. "A Universal Grid Map Library: Implementation and Use Case for Rough Terrain Navigation". In: *Robot Operating System (ROS) - The Complete Reference (Volume 1).* Ed. by Anis Koubaa. Springer, 2016. Chap. 5. ISBN: 978-3-319-26052-5. URL: `http://www.springer.com/de/book/9783319260525`.

[4] Mathieu Labbe and Francois Michaud. "Appearance-based loop closure detection for online large-scale and long-term operation". In: *IEEE Transactions on Robotics* 29.3 (2013), pp. 734–745.

[5] Mathieu Labbe and François 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.

[6] Mathieu Labbé and François Michaud. "Long-term online multi-session graph-based SPLAM with memory management". In: *Autonomous Robots* (2017), pp. 1–18.

[7] *Mars Science Laboratory.* NASA. URL: `https://mars.nasa.gov/msl/` (visited on 12/28/2017).

[8] *Mars Science Laboratory - Curiosity.* NASA. URL: `https://www.nasa.gov/mission_pages/msl/index.html` (visited on 12/28/2017).

[9] Sebastian Montabone. "Beginning Digital Image Processing: Using Free Tools for Photographers". In: Apress, 2010. ISBN: 978-1-430-22841-7.

[10] T. Moore and D. Stouch. "A Generalized Extended Kalman Filter Implementation for the Robot Operating System". In: *Proceedings of the 13th International Conference on Intelligent Autonomous Systems (IAS-13).* Springer, July 2014.

[11] *MoveIt! Motion Planning Framework.* URL: `http://moveit.ros.org/` (visited on 12/24/2017).

[12] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: *ICRA Workshop on Open Source Software.* 2009.

Authors Biographies

Daniel Snider received a bachelor of information technology (BIT) from the University of Ontario Institute of Technology in Ontario, Canada (2013). He is currently a member of the R3 robotics team at Ryerson University and works as a computer vision software developer at SickKids Research Institute, associated with the University of Toronto. His current research is on the design of modular polyglot frameworks (such as ROS) and scientific workflow systems.

Matthew Mirvish is currently a student at Bloor Collegiate Institute. He is also currently a member of the R3 robotics team at Ryerson University and helps write the code for their rover for URC. He mainly specializes in the autonomous task, but usually ends up helping with anything he can get his hands on. In his spare time he dabbles with microcontrollers as well as creating small computer games.

Michal Barcis received a bachelor's and master's degree in computer science from University of Wroclaw, Poland, in 2015 and 2016, respectively. He is currently pursuing the Ph.D. degree as a researcher with the Alpen-Adria-Universitat Klagenfurt, Austria. Between 2015 and 2017 he was a member of the Continuum Student Research Group at the University of Wroclaw. His research interests include robotics, computer networks and machine learning.

Vatan Aksoy Tezer is currently studying Aerospace Engineering in Istanbul Technical University, Turkey. He was the software sub-team leader of ITU Rover Team during 2016-2017 semester and worked on autonomous navigation, communication, computer vision and high level control algorithms. He previously worked on underwater robots, rovers and UAVs. He is currently conducting research on navigation in GPS denied environments using ROS.