University Rover Challenge: Tutorials and Control System Survey

Daniel Snider, Matthew Mirvish, Michał Barciś, Vatan Aksoy Tezer

Abstract In this tutorial chapter we present a guide to building a robot through 12 tutorials. We prescribe simple software solutions to build a 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 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 7 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

D. Snider (✉)
Ryerson University, Team R3
350 Victoria St, Toronto, Canada
e-mail: danielsnider12@gmail.com

M. Mirvish
Bloor Collegiate Institute, Team R3
1141 Bloor St W, Toronto, Canada
e-mail: matthewmirvish@hotmail.com

M. Barciś
University of Wrocław, Team Continuum
plac Uniwersytecki 1, 50-137 Wrocław, Poland
e-mail: mbarcis@mbarcis.net

V. A. Tezer
Istanbul Technical University, ITU Rover Team
Maslak, 34467 Sarıyer/İstanbul, Turkey
e-mail: vatanaksoytezer@gmail.com

Contents

1 Introduction

Our mission is to clearly explain how to build a robot by piecing together existing code, lowering the challenges for new ROS users. At the University Rover Challenge (URC), Team R3 spoke to several teams who are not using ROS but want to, and others who want to expand their use of it. The contributions of our book chapter include 12 tutorials, 7 new ROS packages, and an original survey of 7 teams after they participated in the URC 2017 rover competition.

2 Background

2.1 About the University Rover Challenge

The University Rover Challenge is an international robotics competition run annually by The Mars Society. 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[1].

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

The competition takes place over three days and consists of four tasks. The science cache task (retrieve and test subsurface soil sample without contamination), the extreme retrieval and delivery task (find and move hand tools, eg. hammer), the equipment servicing task (manipulate equipment, eg. pour a fuel canister, press a button, flick a switch), and the autonomous traversal task which is explained in detail below. See the competition 2017 rules[2] for more information. The newer 2018 rules[3] are very similar, but with more difficult manipulation tasks such as typing on a keyboard and more a demanding autonomous traversal challenge that explicitly calls for obstacle avoidance– something that Team R3 had last year and is explained in detail in section 5.9.

[1] http://urc.marssociety.org/home/urc-news/americanroverearnsworldstopmarsrovertitle
[2] http://tinyurl.com/urc-rules
[3] http://tinyurl.com/urc-rules2018

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

2.1.1 Autonomous Task Rules

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 – 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.

2.2 Prerequisite Skills for Tutorials

The tutorials in this chapter expect the following skills at a basic level, though they are useful to continue learning about.

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

3 Survey of URC Competing Teams

The following teams were surveyed to ask about 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.

	Mars Rover Design Team	Team Continuum	Cornell Mars Rover	ITU Rover Team	UWRT Robotics Team	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 microcontrollers	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 depthimagetolaserscan, 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. Link to source[4].	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

[4] ITU Rover Autonomous source code: https://github.com/itu-rover/ros/tree/master/catkin_ws/src

	Mars Rover Design Team	Team Continuum	Cornell Mars Rover	ITU Rover Team	UWRT Robotics Team	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. Link to source code[5].	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 microcontrollers	I2C for sensors, USB for Raspberry Pi to microcontrollers	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. 3 Control system survey of URC 2017.

[5] ITU Rover communication modules: https://github.com/itu-rover/communication

3.1 Highlighted Rover Team Experiences

What do you think is your team's strength?

Michał Barciś from the Continuum team:

To be honest, it's hard to say. Last year I believed it was the software part – back then only a few teams used ROS or even an OS on the rover, but we did and it allowed us to prototype quickly, get whatever sensor we wanted on the rover and really understand what's going on.

This year much more teams had ROS on board, so this was clearly not our advantage. The mechanical construction is pretty good, but surely not the best out there. We do have pretty good control on the rover and often people appreciate our manipulator controls. The size of the team also distinguishes us – we are fairly small (~10 people) and I think it affects our motivation a lot. Everyone has something important to do and he knows the whole competition depends on it. Such motivation is probably also possible in a bigger, well-managed team, but for us it was pretty natural, we are *not* well-managed :).

What should your team improve for the next URC competition?

Michał Barciś from the Continuum team:

More field tests of the whole Rover. We knew we needed to test a lot, but still we were never satisfied with the amount of tests we've done and we have had some problems this year due to the lack of testing.

This is just my opinion, though.

Did your team really only use Raspberry Pi cameras?

Michał Barciś from the Continuum team:

Yes, we have just been using the standard Raspberry Pi cameras, like this one: https://www.amazon.de/Raspberry-100003-Pi-Camera-Module/dp/B00E1GGE40. We have two with wide angle lenses.

They are not the best, but Raspberry Pi's can process the images quickly and we've been able to tweak their parameters. Using different cameras we were not able to get low latency image in a good resolution, so those modules were working rather OK. In **fig. 4** you will find a photo of us working in the base station.

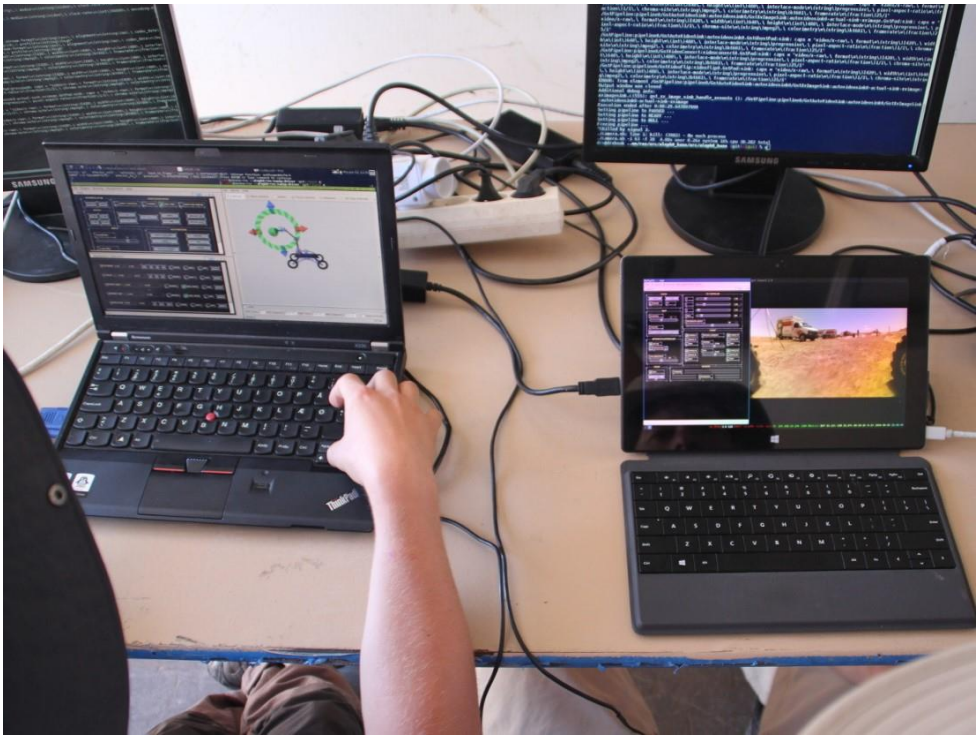

Fig. 4 Team Continuum's base station setup. A screenshot of their rover control GUI can be seen in Fig. 5.

Can you explain your arm functionality?

Michał Barciś from the Continuum team:

Team Continuum implemented "semi-automatic button pressing and picking up things" for the ERC 2016, because there were additional points for that, but it was not deployed at URC 2017 because manual operation is usually quicker and we are able to correct mistakes. Our inverse kinematics (python source code[6]) determines desired arm joint positions, but we lack feedback from the end effector's roll position and we only have force feedback from the gripping motor.

We mounted two laser pointers on the effector. You can see them in some frames of the recently released video[7] from the University of Wrocław. The two dots meet exactly at 20cm (this was the minimum distance from the picked object specified for ERC 2016).

We preprogrammed two arm moves:

- For switching switches the effector went forward and a little bit down for 20cm, then up for 5cm and back 20cm. We were able to switch most switches this way, both very small and bigger ones.
- For picking things up the effector goes down 20cm, then the grip motor engages until the force measurement rises, then back up 20cm.

[6] Continuum Inverse kinematics python source:
https://gist.github.com/danielsnider/5181ca50cef0ec8fdea5c11279a9fdbc
[7] A recent video of from the University of Wrocław of their rover has incredible cinematography:
https://www.youtube.com/watch?v=MF8DkKDBXtg

And this is basically it. As you can see, the solutions are pretty simple, so we call it "semi-automatic" instead of autonomous, etc. We did not experiment in implementing anything more robust – we usually just do as simple a solution as possible to get the most points.

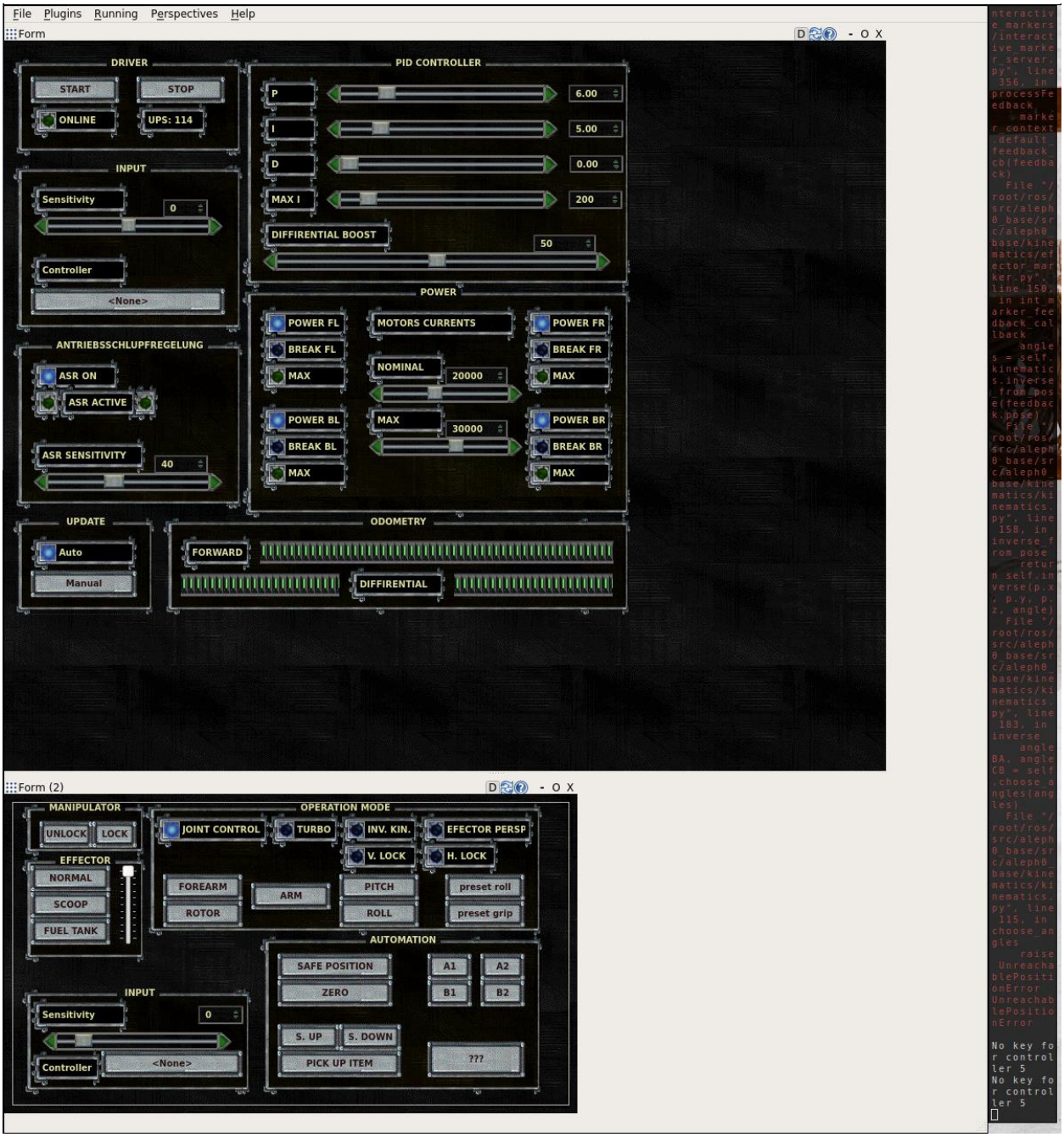

Fig. 5 Team Continuum's rover control GUI. On top there is the drivetrain controller and then manipulator controller.

3.1.1 Other URC Tips

A URC team sponsorship package (PDF format) is available from the University of Toronto: https://drive.google.com/file/d/0B6qINUvvDwDfV09xQ1Q5eUhsd1U/view?usp=sharing

A useful ROS Android app is available that can publish sensors into ROS (GPS, IMU, magnetic field, camera, illuminance sensor, and more): http://wiki.ros.org/android_all_sensors_driver

4 ROS Environment

4.1 Overall System Diagram by Team R3

At team R3, our ROS software was interconnected as you see in the diagram below. There were five main systems: the drive system, the autonomous system, the global positioning system, the visual feedback system, and the odometry system. At the competition we did not actually use the `ekf_localization` ROS nodes of the odometry system because our IMU was not working well enough to produce a good fused result. Instead we relied on odemetry from the `rgbd_odometry` ROS node.

The diagram is available in Microsoft Visio format here: https://github.com/danielsnider/ros-rover/blob/master/diagrams/Rover_Diagram.vsdx?raw=true

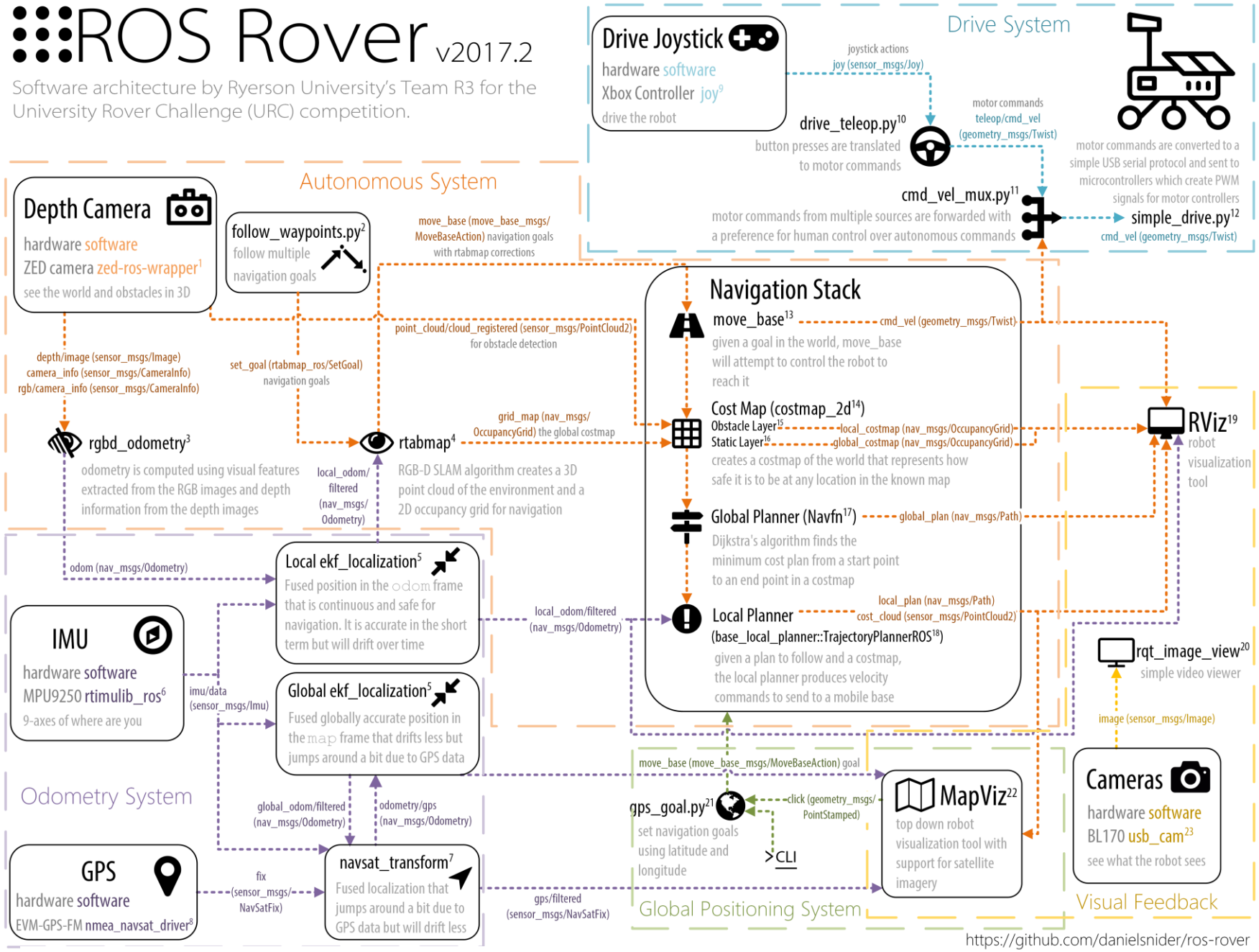

Fig. 6 Software architecture of a best practices rover.

4.2 Diagram Documentation

Documentation links for each software component:

1. zed-ros-wrapper[1] – http://wiki.ros.org/zed-ros-wrapper
2. follow_waypoints.py[2] – http://wiki.ros.org/follow_waypoints
3. rgbd_odometry[3] – http://wiki.ros.org/rtabmap_ros#rgbd_odometry
4. rtabmap[4] – http://wiki.ros.org/rtabmap_ros
5. ekf_localization[5] – https://github.com/danielsnider/simple_drive/blob/master/launch/cmd_vel_mux.launch
6. rtimulib_ros[6] – https://github.com/romainreignier/rtimulib_ros
7. navsat_transform[7] – http://docs.ros.org/lunar/api/robot_localization/html/navsat_transform_node.html
8. nmea_navsat_driver[8] – http://wiki.ros.org/nmea_navsat_driver
9. joy[9] – http://wiki.ros.org/joy
10. drive_teleop.py[10] – http://wiki.ros.org/simple_drive#drive_teleop
11. cmd_vel_mux.py[11] – http://wiki.ros.org/simple_drive#cmd_vel_mux
12. simple_drive.py[12] – http://wiki.ros.org/simple_drive#simple_drive-1
13. move_base[13] – http://wiki.ros.org/move_base
14. Cost Map (costmap_2d[14]) – http://wiki.ros.org/costmap_2d
15. Cost Map (Obstacle Layer[15]) – http://wiki.ros.org/costmap_2d/hydro/obstacles
16. Cost Map (Static Layer[16]) – http://wiki.ros.org/costmap_2d/hydro/staticmap
17. Global Planner (Navfn[17]) – http://wiki.ros.org/navfn
18. Local Planner (base_local_planner::TrajectoryPlannerROS[18]) – http://wiki.ros.org/base_local_planner
19. RViz[19] – http://wiki.ros.org/rviz
20. rqt_image_view[20] – http://wiki.ros.org/rqt_image_view
21. gps_goal.py[21] – http://wiki.ros.org/gps_goal
22. MapViz[22] – http://wiki.ros.org/mapviz
23. usb_cam[23] – http://wiki.ros.org/usb_cam

5 Tutorials

5.1 Tutorial: Autonomous Waypoint Following

Source code: https://github.com/danielsnider/follow_waypoints
Wiki page: http://wiki.ros.org/follow_waypoints

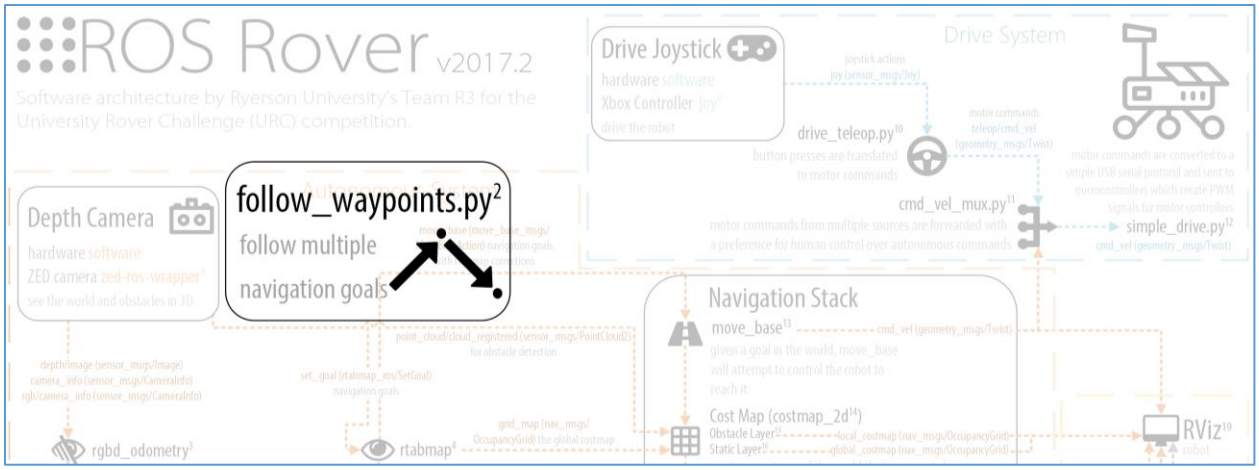

Fig. 7 ROS node `follow_waypoints` as seen in the larger architecture diagram. See section 4.1 for the full diagram.

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? The following tutorial documents a short script that will buffer move_base goals until instructed to navigate to them in sequence.

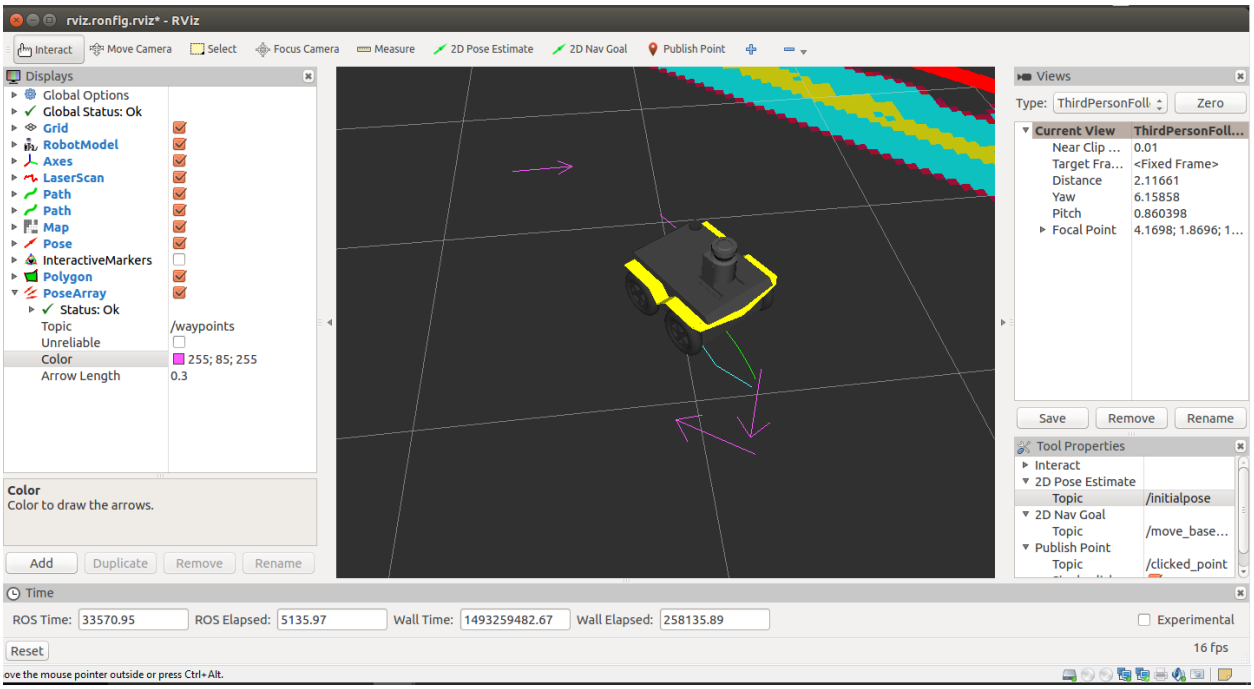

Fig. 8 A simulated Clearpath Jackal robot[8] navigating to one of several waypoints displayed as pink arrows.

Team R3 has developed the `follow_waypoints` ROS package which uses actionlib to send the goals to move_base in a robust manner.

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

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

Python state machine library in ROS called `smach`.[9][10] 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.

5.1.1 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 5.11)
- 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

5.1.2 Quick Start

1. Install:

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

2. Launch:

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

5.1.3 Usage

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. 7 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.

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

```
$ 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 -- {}
```

5.1.4 Normal Output

```
$ roslaunch rover_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 -- {}'
```

[9] SMACH state machine library for python http://wiki.ros.org/smach
[10] One alternative to SMACH is py-trees, a behavior tree library http://py-trees.readthedocs.io/en/devel/background.html

```
[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
```

5.2 Tutorial: Image Overlay Scale and Compass

Source code: https://github.com/danielsnider/image_overlay_scale_and_compass
Wiki page: http://wiki.ros.org/image_overlay_scale_and_compass

We have developed a ROS package `image_overlay_compass_and_scale that` can add an indication of scale and compass to images and video streams. Compass and scale values must be provided using standard Float32 messages. Alternatively, a command interface can be used without ROS.

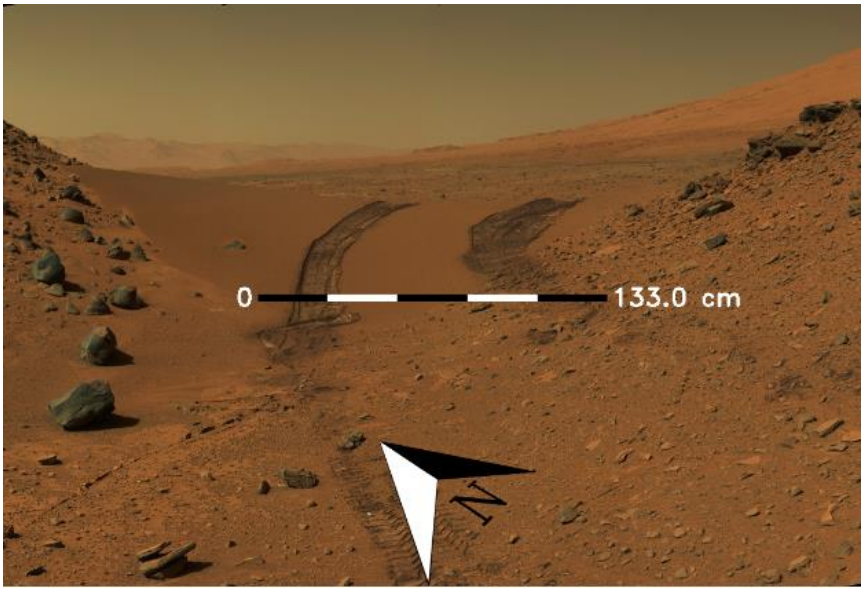

Fig. 9 Example of the `image_overlay_compass_and_scale` ROS package.

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.

This package uses OpenCV's python library to overlay text and overlay, resize, rotate, and warp images.

5.2.1 Quick Start

1. Install:

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

2. Launch 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
```

5.2.2 Command Line Interface (CLI)

Invoke once using the Command Line Interface (CLI) 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
```

CLI Options

```
Usage: image_overlay.py [OPTIONS]

Options:
--input-image TEXT  Path to input image file  [required]
--heading FLOAT     Current heading relative to north in degrees  [required]
--scale-text FLOAT  The value to be displayed on the right of the scale bar
                    in centimeters  [required]
--output-file TEXT  Output filename to save result to
                    [default='output.png']
--help              Show this message and exit.
```

5.2.3 Subscribed Topics

`camera/compressed` (sensor_msgs/CompressedImage)
: Image topic that will be overlaid with scale and compass indicators.

`heading` (std_msgs/Float32)
: Current heading relative to north in degrees.

`scale` (std_msgs/Float32)
: The value to be displayed on the right of the scale bar in centimeters.

5.2.4 Published Topics

`overlay/compressed` (sensor_msgs/CompressedImage)
: The resulting image overlaid with scale and compass indicators.

5.2.5 Parameters

`~framerate` (`int`, default: 3)
The rate at which to publish an image overlaid with scale and compass indicators.

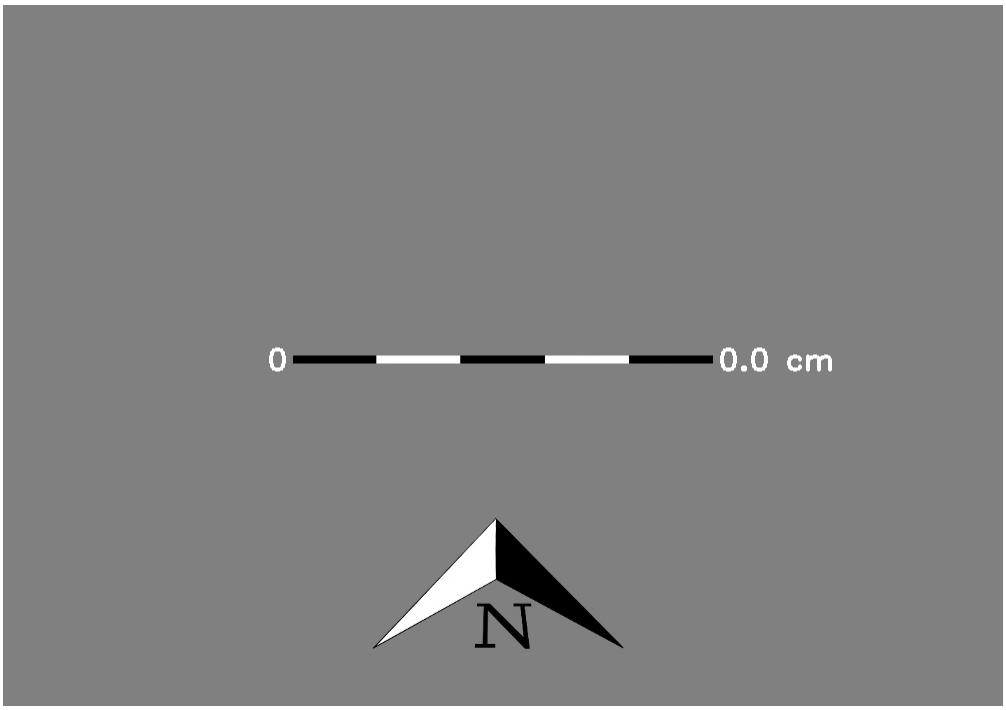

Fig. 10 This is what to expect when nothing is received by the node. This is the default published image.

5.3 Tutorial: A Simple Drive Software Stack

Source code: https://github.com/danielsnider/simple_drive
Wiki page: http://wiki.ros.org/simple_drive

A simple robot drive system that includes skid steering joystick teleoperation, control of a panning servo to look around the robot, and Arduino firmware.

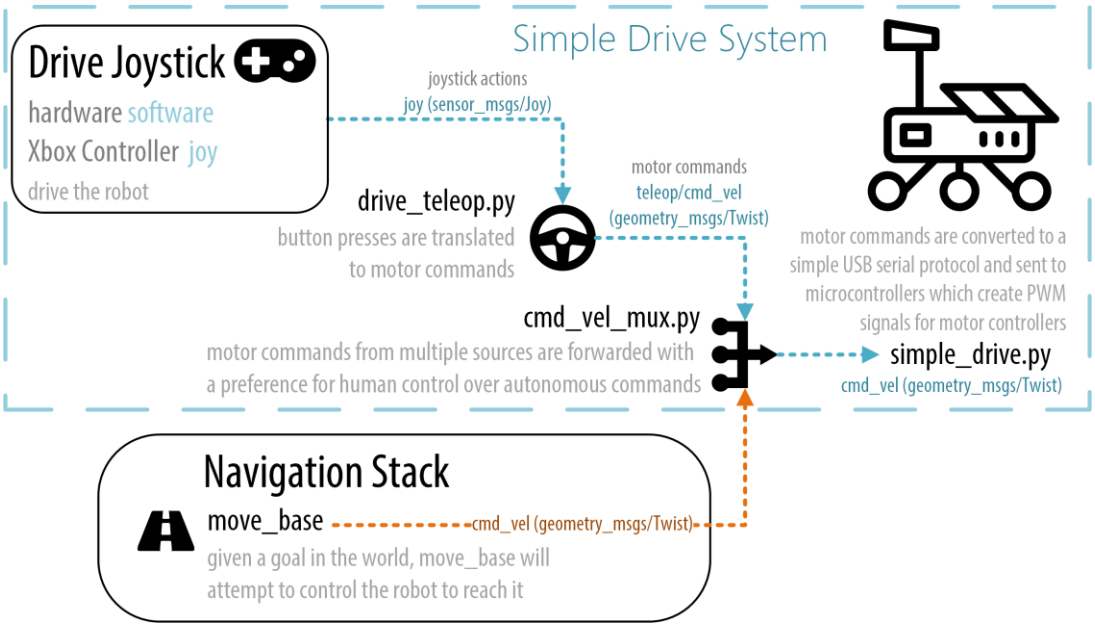

Fig. 11 Architecture of the drive software used by Team R3. See section 4.1 for a larger diagram.

Features include:

- skid steering joystick teleoperation with three drive speeds
- joystick control of a servo (left and right only)
- dedicated left and right thumbsticks for left and right wheel speeds
- a cmd_vel multiplexer to support a coexisting autonomous drive system
- simple Arduino firmware to talk to PWM motors and a servo

For the sake of simplicity, this package does not do the following:

- no tf publishing of transforms
- no wheel odometry publishing
- no PID control loop
- no integration with ros_control

This package is divided into four parts:

Nodes	Purpose
drive_teleop	Drive your robot with a joystick in skid steering mode (also known as diff drive or tank drive). Left joystick thumbstick controls left wheels, right thumbstick controls right wheels.
cmd_vel_mux	Motor commands from multiple sources are forwarded with a preference for human control over autonomous commands.
simple_drive	Send motor commands to your microcontroller.
drive_firmware	Microcontroller software that controls your motors. Not a ROS node.

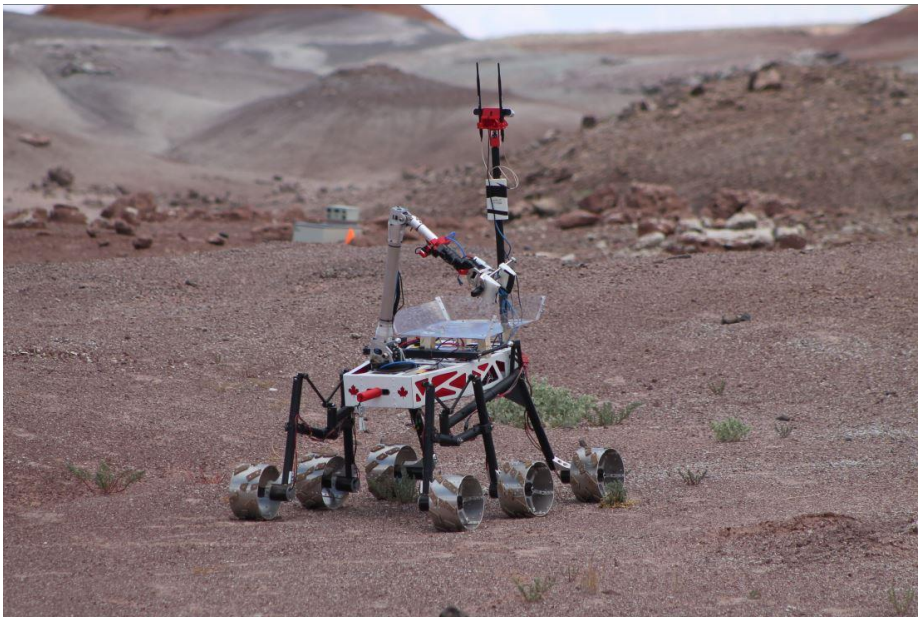

Fig. 12 Team R3's rover being teleoperated by `simple_drive`.

5.3.1 Quick Start

1. Install:

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

2. Launch:

```
$ 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
```

3. Install the drive_firmware onto a microcontroller connected to your motors and wheels by PWM. The microcontroller must also be connected to the computer running the `simple_drive` ROS node by a serial connection (eg. USB).

4. Drive your robot around.

5.3.2 drive_teleop ROS Node

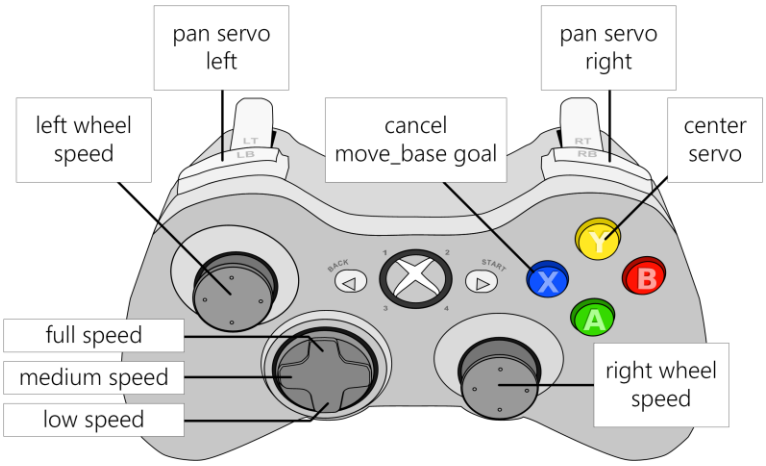

Fig. 13 The Xbox 360 joystick button layout for the `simple_drive` package (diagram is available in Visio format[11]).

This node converts sensor_msgs/Joy messages from the joy ROS node into a variety of commands to drive the robot at low, medium, and high speed, look around with a servo, and cancel `move_base` goals at any moment. The `drive_teleop` node simply sends commands to other nodes. Typically the servo is used to move a camera so that the teleoperator can look around the robot.

The button mapping was tested on an Xbox 360 controller and should require little or no modification for similar controllers.

Launch example

```
$ roslaunch simple_drive drive_teleop.launch joy_dev:=/dev/input/js0
```

5.3.2.1 Subscribed Topics

`joy` (sensor_msgs/Joy)
: Controller joystick button presses are received on this topic. See joy node for more info.

5.3.2.2 Published Topics

`teleop/cmd_vel` (geometry_msgs/Twist)
: Output motion commands that drive the robot.

`servo_pos` (std_msgs/Float32)
: Output servo commands that control the angle of a servo.

5.3.2.3 Parameters

`~servo_pan_speed` (`int`, default: 5)
: The degrees of servo motion per button press.

`~servo_pan_max` (`int`, default: 160)
: The maximum angle of servo rotation in degrees.

`~servo_pan_min` (`int`, default: 0)
: The minimum angle of servo rotation in degrees.

[11] https://github.com/danielsnider/ros-rover/blob/master/diagrams/simple_drive_Xbox_Controller.vsdx?raw=true

5.3.3 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 to a final topic for robot consumption. If any teleoperation command is received autonomous commands are blocked for a set time defined by the `block_duration` parameter.

Launch example

```
$ roslaunch simple_drive cmd_vel_mux.launch
```

5.3.3.1 Subscribed Topics

`teleop/cmd_vel` (geometry_msgs/Twist)
: Human generated movement commands from a teleoperation node such as the drive_teleop node.

`move_base/cmd_vel` (geometry_msgs/Twist)
: Computer generated movement commands from an autonomous system such as move_base or rtabmap_ros.

5.3.3.2 Published Topics

`cmd_vel` (geometry_msgs/Twist)
: Movement commands for consumption by the robot. Autonomous movement commands will be blocked for `block_duration` seconds if any teleoperation command is received.

5.3.3.3 Parameters

`~block_duration` (`int`, default: 5)
: The amount of time autonomous movement is blocked after any human movement is received.

5.3.4 simple_drive ROS Node

This node communicates with the drive_firmware using a custom serial protocol described below. An example serial data packet could be `0,0.5,0.5` which would mean drive motors forward at half speed and rotate at half speed.

Serial Protocol

Action	Serial data packet
Twist motor command	`(BYTE) 0, (FLOAT) LINEAR_VELOCITY, (FLOAT) ANGULAR_VELOCITY`
Servo position command	`(BYTE) 2, (FLOAT) SERVO_ANGLE`

 If your microcontroller supports subscribing to ROS `twist` messages (Arduinos can use rosserial_arduino) then it would be simpler to do that and skip this node. 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.

Launch example

```
$ roslaunch simple_drive simple_drive.launch serial_dev:=/dev/ttyACM0
```

5.3.4.1 Subscribed Topics

`cmd_vel` (geometry_msgs/Twist)
: Motor commands received on this topic are encoded and sent over serial to the drive_firmware.

`servo_pos` (std_msgs/Float32)
: Servo position commands received on this topic are encoded and sent over serial to the drive_firmware.

5.3.4.2 Parameters

`~serial_dev` (`string`, default: /dev/ttyACM0)
: The microcontroller, often an Arduino, as seen in linux, that this node will send motor commands to.

`~baudrate` (`int`, default: 9600)
: Sets the serial device data transmission rate in bits per second (baud).

5.3.5 drive_firmware

The `drive_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 output to be received by motor controllers.

We connected an Arduino by USB serial to our main robot computer. 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.

⚠ Caution! This software does not stop moving the robot if no messages are received for certain period of time. A pull request for this is very welcome.

PlatformIO

We deploy the `drive_firmware` to an Arduino microcontroller using PlatformIO. We like PlatformIO for their tagline, "Single source code. Multiple platforms." PlatformIO supports approximately 200 Embedded Boards and all major Development Platforms. Learn more on platformio.org.

Install and configure drive_firmware

This steps were tested on Ubuntu 16.04.

1. Install PlatformIO

```
$ 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
```

More PlatformIO install information: http://docs.platformio.org/en/latest/installation.html#super-quick-mac-linux

2. Create a PlatformIO project

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

More PlatformIO info: http://docs.platformio.org/en/latest/quickstart.html

3. Modify wiring if necessary

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

Depending on how you want to wire your microcontroller this wiring could be changed:

```
 1 // Pins to Left Wheels
 2 #define pinL1 13
 3 #define pinL2 12
 4 #define pinL3 11
 5 // Pins to Right Wheels
 6 #define pinR1 9
 7 #define pinR2 8
 8 #define pinR3 7
 9 // Pin to the Servo
10 #define pinServo 5
```

4. Modify PWM Settings

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

Edit the following section to match your motor controller PWM specs:

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

5. Deploy Code to Arduino

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

6. Drive your robot!

Have fun!

5.3.6 Other Implementation Details

Tank to Twist Calculation

When a left and right joystick inputs are received by the drive_teleop node, representing left and right wheel velocities (ie. skid steering or differential drive), a geometry_msgs/Twist with linear and rotational velocities is calculated as:

Linear velocity (m/s) =

```
(left_speed + right_speed) / 2.0
```

Angular velocity (rad/s) =

```
(right_speed - left_speed) / 2.0
```

Twist to Tank Calculation

When a geometry_msgs/Twist containing linear and rotational velocities is received by the drive_firmware, wheel velocities are calculated as:

Left wheel velocity (m/s) =

```
linear_speed + angular_speed
```

Right wheel velocity (m/s) =

```
linear_speed - angular_speed
```

5.3.7 Related Packages

Similar to **simple_drive**:

http://wiki.ros.org/differential_drive - Differential drive software with support for a velocity PID target, a small GUI to control the robot, and more.

http://wiki.ros.org/diff_drive_controller - Differential drive software that is real-time safe, integrates with ros_control, and more.

Similar to **drive_teleop**:

http://wiki.ros.org/teleop_twist_joy - This teleop node converts joy messages to twist messages.

http://wiki.ros.org/joy_teleop - This teleop node takes joy messages and publishes topics or calls actions according a configuration file.

Similar to **cmd_vel_mux**:

http://wiki.ros.org/twist_mux - Multiplex several velocity command topics with prioritization or disabling according to a configuration file.

5.4 Tutorial: Velocity Controlled Arm

Source code: https://github.com/danielsnider/simple_arm
Wiki page: http://wiki.ros.org/simple_arm

In this tutorial will we will look at teleoperation software and firmware for a simple, velocity controlled arm with 6 degrees of freedom.

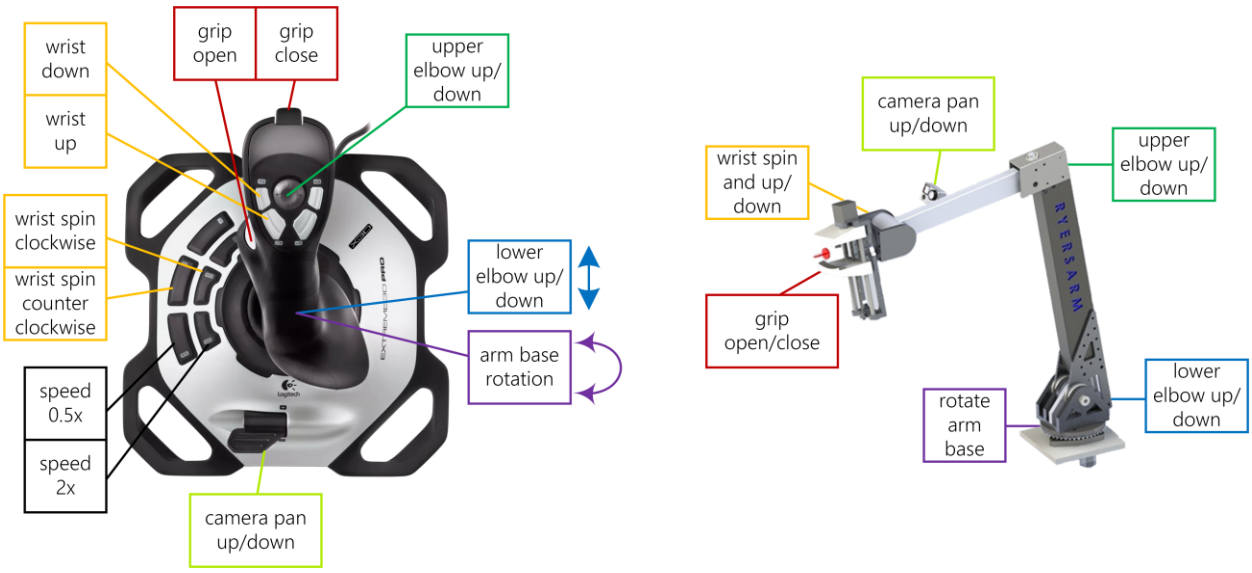

Fig. 14 The Logitech Extreme 3D Pro joystick button layout for the simple_arm package (diagram is also available in Visio format[12]).

Features include**:**

- velocity control of arm joint motors
- fast and slow motor speed modifier buttons
- open and close a gripper (single speed)
- control of a camera servo (up and down only)
- simple Arduino firmware to talk to PWM motors and a servo

For the sake of simplicity, this package does not do the following**:**

- no tf publishing
- no URDF
- no PID control loop
- no integration with ros_control or MoveIt!

5.4.1 Quick Start

1. Install:

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

2. Launch:

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

3. Install the arm_firmware onto a microcontroller connected to the arm joint motors by PWM. The microcontroller must also be connected to the computer running the simple_arm ROS node by a serial connection (ex. USB).

[12] https://github.com/danielsnider/ros-rover/blob/master/diagrams/simple_arm_joystick_diagram.vsdx?raw=true

4. Move your robot arm.

simple_arm ROS Node

This node converts sensor_msgs/Joy messages from the joy node into a variety of commands that are sent over serial to a microcontroller to drive the robot arm.

The button mapping was tested on a Logitech Extreme 3D Pro joystick and should only need small modifications for similar controllers.

5.4.2 Subscribed Topics

`joy_arm` (sensor_msgs/Joy)
: Joystick state is received on this topic. See the joy node for more info.

5.4.3 Parameters

`~microcontroller_serial_device` (`string`, default: /dev/ttyACM0)
: The microcontroller, often an Arduino, as seen in Linux, that this node will send motor commands to.

`~baudrate` (`int`, default: 9600)
: The serial device data transmission rate in bits per second (baud).

5.4.4 Serial Protocol

This node communicates with the arm_firmware using a simple serial protocol. Each serial motion command is a list of floats, one for each joint.

Serial data packet:

```
(FLOAT) GRIP,
(FLOAT) WRIST_ROLL,
(FLOAT) WRIST_PITCH,
(FLOAT) UPPER_ELBOW,
(FLOAT) LOWER_ELBOW,
(FLOAT) BASE_YAW,
(FLOAT) CAMERA
```

5.4.5 arm_firmware

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 output to be received by motor controllers.

We connected an Arduino by USB serial to our main robot computer. 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.

⚠ Caution! This software does not stop moving the robot if no messages are received for certain period of time. A pull request for this is very welcome.

PlatformIO

We deploy the `arm_firmware` to an Arduino microcontroller using PlatformIO. We like PlatformIO. Its tagline is "Single source code. Multiple platforms." PlatformIO supports approximately 200 Embedded Boards and all major Development Platforms. Learn more on platformio.org.

Install and configure arm_firmware

This steps were tested on Ubuntu 16.04.

1. Install PlatformIO

```
$ 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-u
dev.rules  > /etc/udev/rules.d/99-platformio-udev.rules
# After this file is installed, physically unplug and reconnect your board.
$ sudo service udev restart
```

More PlatformIO install info: http://docs.platformio.org/en/latest/installation.html#super-quick-mac-linux

2. Create a PlatformIO project

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

More PlatformIO info: http://docs.platformio.org/en/latest/quickstart.html

3. Modify wiring if necessary

```
$ vim src/main.cpp +9
```

Depending on how you want to wire your microcontroller this wiring can be changed:

```
 1 struct JOINTPINS{
 2   int wrist_roll = 9; // wrist roll pin
 3   int wrist_pitch = 10; // wrist pitch pin
 4   int upper_elbow = 11; // upper elbow pin
 5   int lower_elbow = 12;  // lower elbow pin
 6   int base_yaw = 13;  // base yaw pin
 7   int grip_enable = A4;  // gripper enable pin
 8   int grip_open = A5;  // gripper open pin
 9   int grip_close = A6;  // gripper close pin
10   int cam_tilt = 7; // camera servo pin
11 }pins;
```

4. Modify PWM Settings

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

Edit the following section to match your motor controller PWM specs:

```
1 // PWM specs of the Victor SP motor controller.
2 //      https://www.vexrobotics.com/217-9090.html
3 #define sparkMax 650 // Full-reverse input pulse
4 #define sparkMin 2350 // Full-forward input pulse
```

5. Deploy Code to Arduino

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

6. Move your robot arm! 😃

Have fun!

5.5 Tutorial: Autonomous Recovery after Lost Communications

Source code: https://github.com/danielsnider/lost_comms_recovery
Wiki page: http://wiki.ros.org/lost_comms_recovery

If your robot loses connection to the base station it will navigate to a configurable home. The base station connection check uses ping to a configurable list of IPs. The monitoring loop waits 3 seconds between checks. Beware that this node is not "real-time"[13] safe.

5.5.1 Quick Start

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
```

If move_base is running

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` param 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

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.

Important: Instead of relying on this this feature, you're better off with motor control software that sets zero velocity after a certain amount of time not receiving any new commands.

lost_comms_recovery ROS Node

5.5.2 Actions Called

`move_base` (move_base_msgs/MoveBaseAction)
The move_base action server performs the autonomous recovery navigation.

[13] Real time computing https://en.wikipedia.org/wiki/Real-time_computing

5.5.3 Subscribed Topics

`recovery_pose` (geometry_msgs/PoseWithCovarianceStamped)
: A pose that the robot will recover to if communication to the base station is lost. This will override the default recovery goal which is to the origin (0,0) of the `goal_frame_id` param.

5.5.4 Published Topics

`cmd_vel` (geometry_msgs/Twist)
: Send a zero `Twist` message on this topic when communication to the base station is lost.

`joy` (sensor_msgs/Joy)
: Send a zero `Joy` message on this topic when communication to the base station is lost.

5.5.5 Parameters

`~goal_frame_id` (`string`, default: map)
: The tf frame for `move_base` goals.

`~ping_fail_count` (`int`, default: 2)
: Number of pings that must fail for communication to be considered lost. Each ping waits 1 second for a response.

`~ips_to_monitor` (`string`, default: 192.168.1.2,192.168.1.3)
: Comma separated list of IPs to monitor using ping.

5.5.6 Normal 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.
[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.
```

5.6 Tutorial: Stitch Panoramas with Hugin

Source code: https://github.com/danielsnider/hugin_panorama
Wiki page: http://wiki.ros.org/hugin_panorama

Create panoramas using image snapshots or multiple video streams.

Fig. 15 Panorama by Senza Senso (cc).

Hugin photo stitching tool[14] is used to automatically stitch panoramas. Specifically, the Hugin command line tools[15].

5.6.1 Quick Start

1. Install:

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

2. Launch 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.

5.6.2 Live Panorama Mode

[14] http://hugin.sourceforge.net/
[15] http://wiki.panotools.org/Panorama_scripting_in_a_nutshell

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 (pull requests welcome).

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 you'll have to edit the simple python script (`rosed hugin_panorama stitch_loop.py`) and the launch file (`rosed hugin_panorama stitch_loop.launch`).

hugin_panorama ROS Node

5.6.3 Published Topics

`panorama/compressed` (sensor_msgs/CompressedImage)
: The resulting panorama.

5.6.4 Services

`stitch` (std_srvs/Empty)
: Create the panorama.

`reset` (std_srvs/Empty)
: Clear images waiting to be stitched.

5.6.5 Parameters

`~images_path` (`string`, default: "`$(find hugin_panorama)/images`")
: This is the working directory where input and output images for the panorama are stored.

image_saver ROS Node

This node provided by the image_view package allows you to save images as jpg/png file from a sensor_msgs/Image topic. The saved images are used when stitching the panoramas. The `image_saver` node should save images to the same location that the `hugin_panorama` node will look in when stitching.
See hugin_panorama.launch for a proper usage example. See the image_saver wiki for full node documentation.

5.7 Tutorial: Wireless Communication

2.4 GHz, 433 MHz, and a Unity simulation for blind driving the rover

Source code: https://github.com/itu-rover/communication

At Team ITU, communication was one of the things we were most afraid of having problems with so we really spent a good amount of time on this. In our earlier development time we were using Bullet M2s as our main link but after failing non-line of sight tests at long distances (400-500 meters) we searched a little bit more and found the Microhard pDDL2450, a 2.4 GHz radio, from a sponsor's advice. Luckily they had a few spares and donated them

to the team. Those were able to communicate at 1km, non-line of sight with standard 8 dBi omnidirectional antennas at both the ground station and rover.

As a backup to the main wireless Ethernet link, our second link was a lower radio frequency (RF) link in the UHF band. The RF device used in the both ends were 433 MHz LoRa modules[16]. The LoRa modules were wired to the standard RX-TX wires on our STM32F103 (similar to XBee wiring[17]) and uses UART communication. To make the RF link active, we just start communication with the LoRa in C# with:

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

And start receiving and transmitting data with: `_serialPort.ReadLine()` and `_serialPort.WriteLine()`.

Fig. 16 UHF LoRa radio module for long range communication.

For the autonomous mission we operated using the RF link because the rover would get very far away from the ground station and in a non-line of sight (NLOS) condition, so it seemed more logical to use it. We send the target coordinates to the low level system via RF link and then the low level system gives the coordinates to the Raspberry Pi and this is where ROS comes in to move the rover.

The wireless Ethernet link was connected to the Raspberry Pi and if wireless were to go down or we detect a problem with the Raspberry Pi, we would tell our software to close the Ethernet link and open the RF link. We had very good results and our communication link never went down during the competition.

Unity simulation for driving robot without camera feed

When in RF mode, using only GPS coordinates of the vehicle, we were able to drive the rover without any video stream. Using TerraUnity, we simulated the competition environment one-to-one. So the driver could just 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. This provided us a chance to blind drive the vehicle without the video feed. We never had to use it during the competition but it was working in our tests.

Lastly, we had a third backup system that took over if we lost all communications (RF and Ethernet together) the rover would navigate autonomously to the last point that any signal was received.

5.8 Tutorial: Autonomous Navigation by Team ITU

For a description of the requirements of the autonomous task for URC 2017, please see section 2.1.1 Autonomous Task of this chapter.

[16] LoRa URF radio module http://www.iotglobalnetwork.com/products/single/id/1187/wireless-technology-lora-433mhz-rf-module

[17] Wiring diagram http://www.pyroelectro.com/tutorials/arduino_pic_communication/theory2.html

Twist_mux, move_base and finding tennis balls

Firstly, Team ITU uses the `twist_mux`[18] ROS package to support simultaneous teleoperation and autonomous driving for the following reason, "When there are more than a single source to move a robot with a `Twist` message, it is important to multiplex all those input sources into a single one that goes to the motor controller (e.g. `diff_drive_controller`)". The `twist_mux` ROS package is even better for this than the `cmd_vel_mux`[19] ROS node included in the `simple_drive` package.

Team ITU uses `twist_mux` with a `configuration_file` (team ITU's config[20]) to combine the following three input velocities used to move the vehicle:

1. `joy_vel`: Velocity from joystick commands. Priority: 100
2. `cmd_vel`: Velocity from the autonomous system (move_base) to achieve the desired position. Priority: 10
3. `final_vel`: The velocity calculated to reach the tennis ball objective after finding it. Priority: 10

The `cmd_vel` and `final_vel` priorities are equal because we didn't want them to block each other as they should work together. In the competition, after reaching within 3 meters of the tennis ball objective's GPS coordinates, `cmd_vel` becomes 0 and accepts that it reached the objective. Then the system looks for the `final_vel` and if it's 0 as well the system confirms that it has reached the objective, then we talk with the judges to confirm we have reached our waypoint.

The `final_vel` topic, which is published by our `test_tennis_ball.cpp`[21] node which determines where the ball is and turns the vehicle towards the ball and goes forward until the ball is big enough, which means we are close enough to the ball.

The `twist_mux` node sends velocities to our `rover_velocity_controller.py`[22] node on `husky_velocity_controller/cmd_vel` and then by `serial_com.py`[23] to our low level system. The low level system (STM32F103) is written such that it just takes the velocity from ROS and navigates accordingly. The message we send to the low level from USB interface is constructed as B(Begin) - 0 or 1(0 for backward 1 for forward) - 00 to 99(linear velocity) - 0 or 1(left or right) - 00 to 99(angular velocity) - E(End). So a sample message would be "B152039E". Which means go forward with %52 of the full speed and go left with %39 of the full speed.

5.9 Tutorial: Autonomous Navigation by Team R3

For a description of the requirements of the autonomous task for URC 2017, please see section 2.1.1 Autonomous Task of this chapter.

[18] The twist_mux ROS package http://wiki.ros.org/twist_mux
[19] http://wiki.ros.org/simple_drive#cmd_vel_mux
[20] https://github.com/itu-rover/ros/blob/master/catkin_ws/src/rover_control/src/rover_twist_mux.yaml
[21] https://github.com/itu-rover/ros/blob/master/catkin_ws/src/rover_vision/src/test_tennis_ball.cpp
[22] https://github.com/itu-rover/ros/blob/master/catkin_ws/src/rover_control/src/rover_velocity_controller.py
[23] https://github.com/itu-rover/ros/blob/master/catkin_ws/src/rover_control/src/serial_com.py

ZED depth camera, RTAB-Map, and move_base

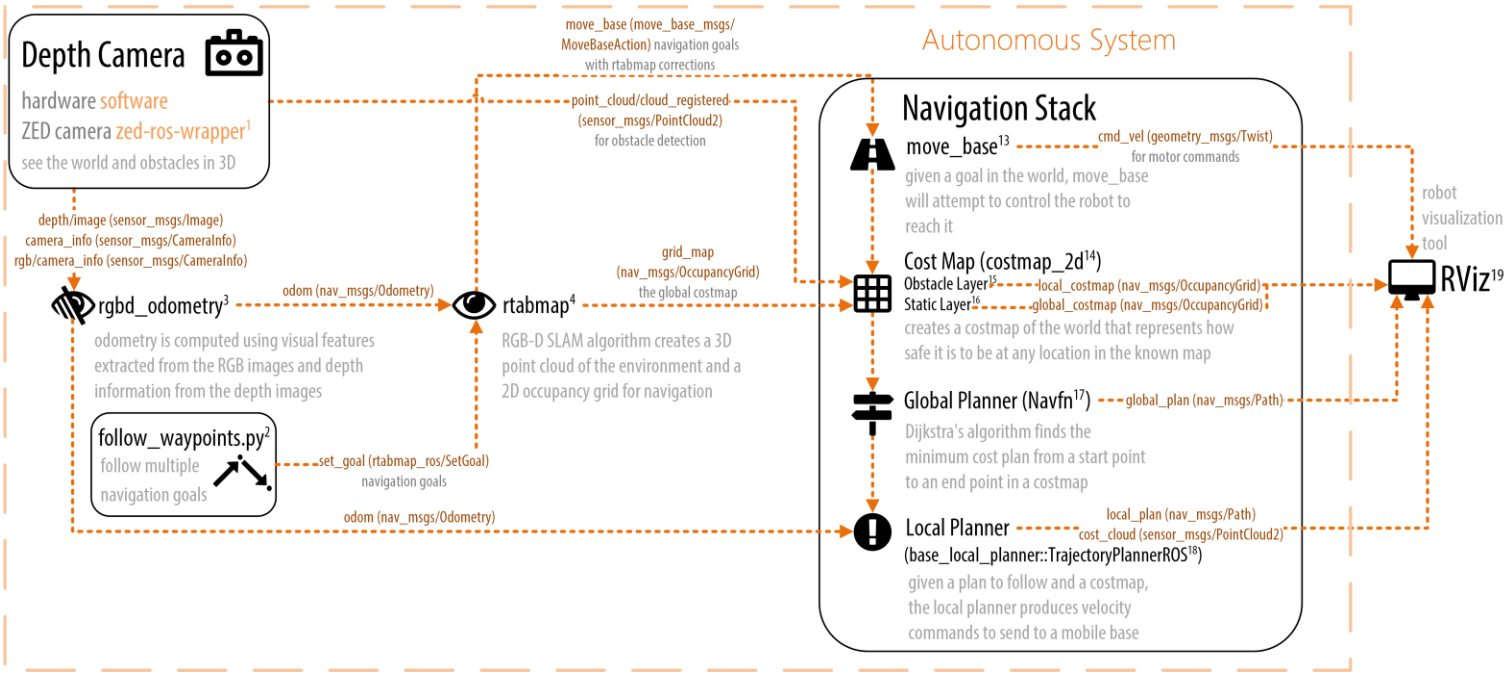

Fig. 17 Team R3's autonomous navigation system used at URC 2017 (diagram is also available in Visio format[24]). See section 4.1 for a larger diagram.

Diagram Documentation

Documentation links for each software component:

1. zed-ros-wrapper[1] – http://wiki.ros.org/zed-ros-wrapper
2. follow_waypoints.py[2] – http://wiki.ros.org/follow_waypoints
3. rgbd_odometry[3] – http://wiki.ros.org/rtabmap_ros#rgbd_odometry
4. rtabmap[4] – http://wiki.ros.org/rtabmap_ros
13. move_base[13] – http://wiki.ros.org/move_base
14. Cost Map (costmap_2d[14]) – http://wiki.ros.org/costmap_2d
15. Cost Map (Obstacle Layer[15]) – http://wiki.ros.org/costmap_2d/hydro/obstacles
16. Cost Map (Static Layer[16]) – http://wiki.ros.org/costmap_2d/hydro/staticmap
17. Global Planner (Navfn[17]) – http://wiki.ros.org/navfn
18. Local Planner (base_local_planner::TrajectoryPlannerROS[18]) – http://wiki.ros.org/base_local_planner
19. RViz[19] – http://wiki.ros.org/rviz

[24] https://github.com/danielsnider/ros-rover/blob/master/diagrams/team_r3_AUTO_Diagram.vsdx?raw=true

5.9.1 ZED Depth Camera

Fig. 18 The ZED depth camera.

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 (aka. performance) depth map quality. The ZED camera worked well outdoors on textured surfaces. Indoors you might want to attach 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
15 in:=rgb/image_rect_color out:=rgb_republished" />
16 </launch
```

Full zed launch file example: https://github.com/teamr3/URC/blob/master/rosws/src/rover/launch/zed_up.launch
More documentation: https://www.stereolabs.com/documentation/guides/using-zed-with-ros/ZED_node.html

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`[25] 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[26].

[25] http://wiki.ros.org/image_transport#republish
[26] https://github.com/teamr3/URC/blob/master/rosws/src/rover/src/low_res_stream.py

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.

5.9.2 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`[27] node provided by the `rtabmap` 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.

In the comments of the launch file below we show some tips that we used at URC 2017.

```
<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) -->
        <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>
```

Full rgbd_odometry launch file example:
https://github.com/teamr3/URC/blob/master/rosws/src/rover_navigation/launch/rgbd_odometry.launch
Full documentation: http://wiki.ros.org/rtabmap_ros#rgbd_odometry

5.9.3 RTAB-Map

Using depth data, RTAB-Map[28] creates a continuously growing point cloud of the world using SLAM[29]. 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, 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[30].

Recommended RTAB-Map tutorials: http://wiki.ros.org/rtabmap_ros#Tutorials

[27] http://wiki.ros.org/rtabmap_ros#rgbd_odometry
[28] http://wiki.ros.org/rtabmap_ros
[29] Simultaneous localization and mapping (SLAM)
[30] https://introlab.github.io/rtabmap/

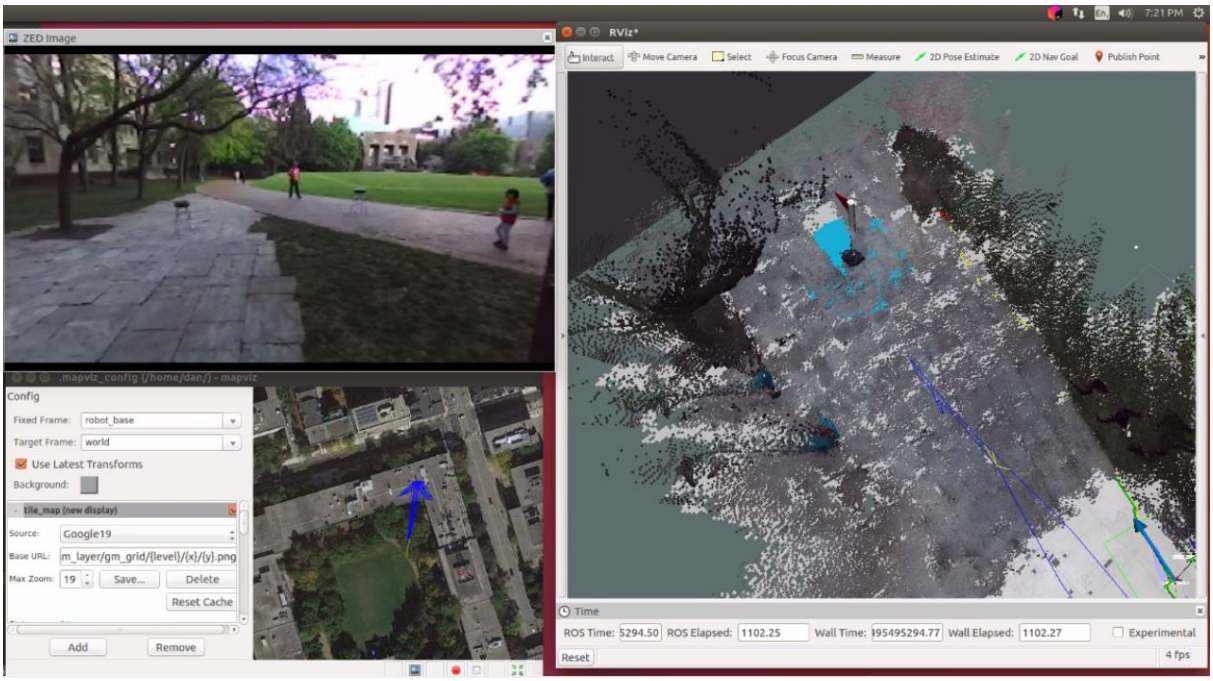

Fig. 19 Screenshot of R3's autonomous testing with RTAB-Map (video available on YouTube[31]).

In the launch file below on lines 3-8, we share tips for reducing 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 must also perform loop closures. Loop closure is the problem of recognizing a previously-visited location and updates the beliefs accordingly[32]. 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. On lines 10-18 we give some tips for increasing the chances of loop closures being detected.

```
1  <launch>
2      <node pkg="rtabmap_ros" name="rtabmap" type="rtabmap" output="screen">
3          <!-- Improve obstacle detection -->
4          <param name="Grid/MaxGroundAngle" value="110"/> <!-- Maximum angle between point's
5  normal to ground's normal to label it as ground. Points with higher angle difference are
6  considered as obstacles. Default: 45 -->
7          <param name="grid_eroded" value="true"/> <!-- remove obstacles which touch 3 or more
8  empty cells -->
9
10         <!-- Improve loop closure chances -->
11         <param name="RGBD/LoopClosureReextractFeatures" type="string" value="true"/> <!--
12 Extract features even if there are some already in the nodes, more loop closures will be
13 accepted. Defalut: false -->
14         <param name="Vis/MinInliers" type="string" value="10"/> <!-- Minimum feature
15 correspondences to compute/accept the transformation. Default: 20 -->
16         <param name="Vis/InlierDistance" type="string" value="0.15"/> <!-- Maximum distance
17 for feature correspondences. Used by 3D->3D estimation approach (the default approach).
18 Default: 0.1 -->
19     </node>
20 </launch>
```

[31] Team R3's autonomous navigation with rtabmap https://www.youtube.com/watch?v=p_1nkSQS8HE
[32] https://en.wikipedia.org/wiki/Simultaneous_localization_and_mapping#Loop_closure

Full rtabmap launch file example:
https://github.com/teamr3/URC/blob/master/rosws/src/rover_navigation/launch/rtabmap.launch
Full documentation: http://wiki.ros.org/rtabmap_ros

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[33] 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.

5.9.4 move_base Path Planning

The ROS navigation stack[34], also known as `move_base`[35], is a collection of components/plugins that are selected and configured by yaml configuration files.

```
<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>
```

Full move_base launch file example:
https://github.com/teamr3/URC/blob/master/rosws/src/rover_navigation/launch/move_base.launch

The most interesting configuration file for `move_base` is the `base_local_planner_params.yaml`. Given a path for the robot to follow and a costmap, the `base_local_planner`[36] 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 well as goal tolerance.

```
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
```

[33] RViz flag tool http://docs.ros.org/jade/api/rviz_plugin_tutorials/html/tool_plugin_tutorial.html
[34] http://wiki.ros.org/navigation
[35] http://wiki.ros.org/move_base
[36] http://wiki.ros.org/base_local_planner

```
  holonomic_robot: false

  xy_goal_tolerance: 1
  yaw_goal_tolerance: 1.39626 # 80 degrees

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

Full configuration file:
https://github.com/teamr3/URC/blob/master/rosws/src/rover_navigation/config/base_local_planner_params.yaml

5.10 Tutorial: MapViz Robot Visualization Tool

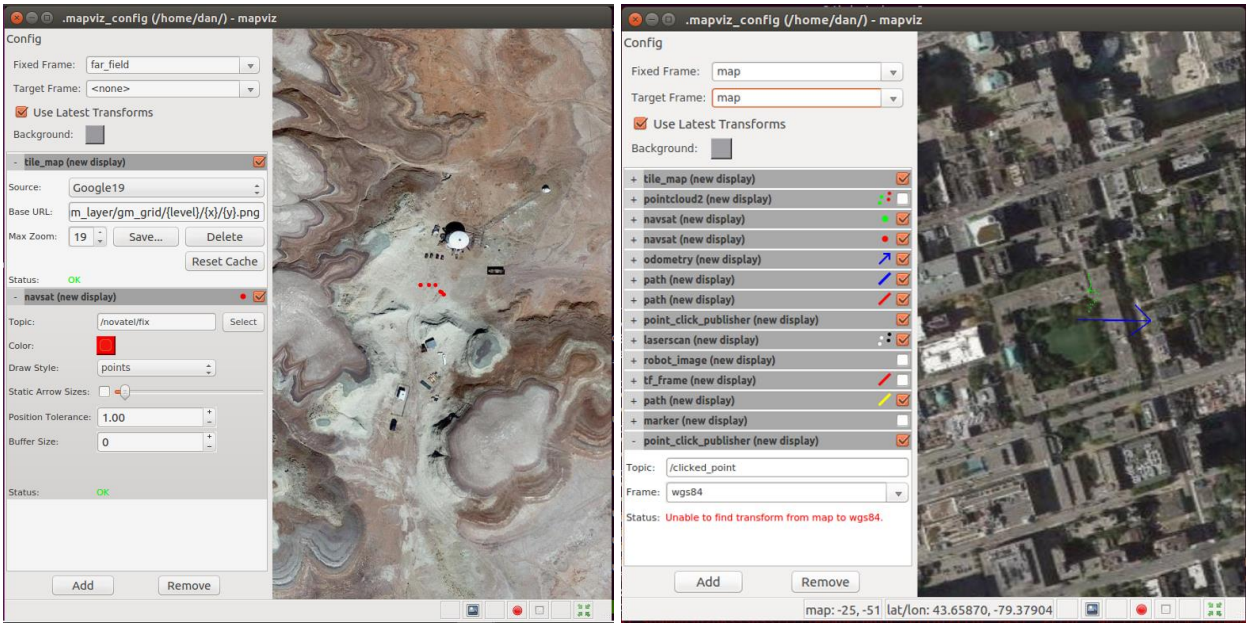

Fig. 20 Screenshots of MapViz ROS visualization tool.

Improve your situational awareness by using MapViz. Mapviz is a ROS-based visualization tool with a plug-in system similar to rviz focused on visualization 2D data[37]. Google Maps satellite view can be viewed by ROS's MapViz Tile Map plugin. After loading once, maps stay cached and are available without internet.

[37] https://github.com/swri-robotics/mapviz

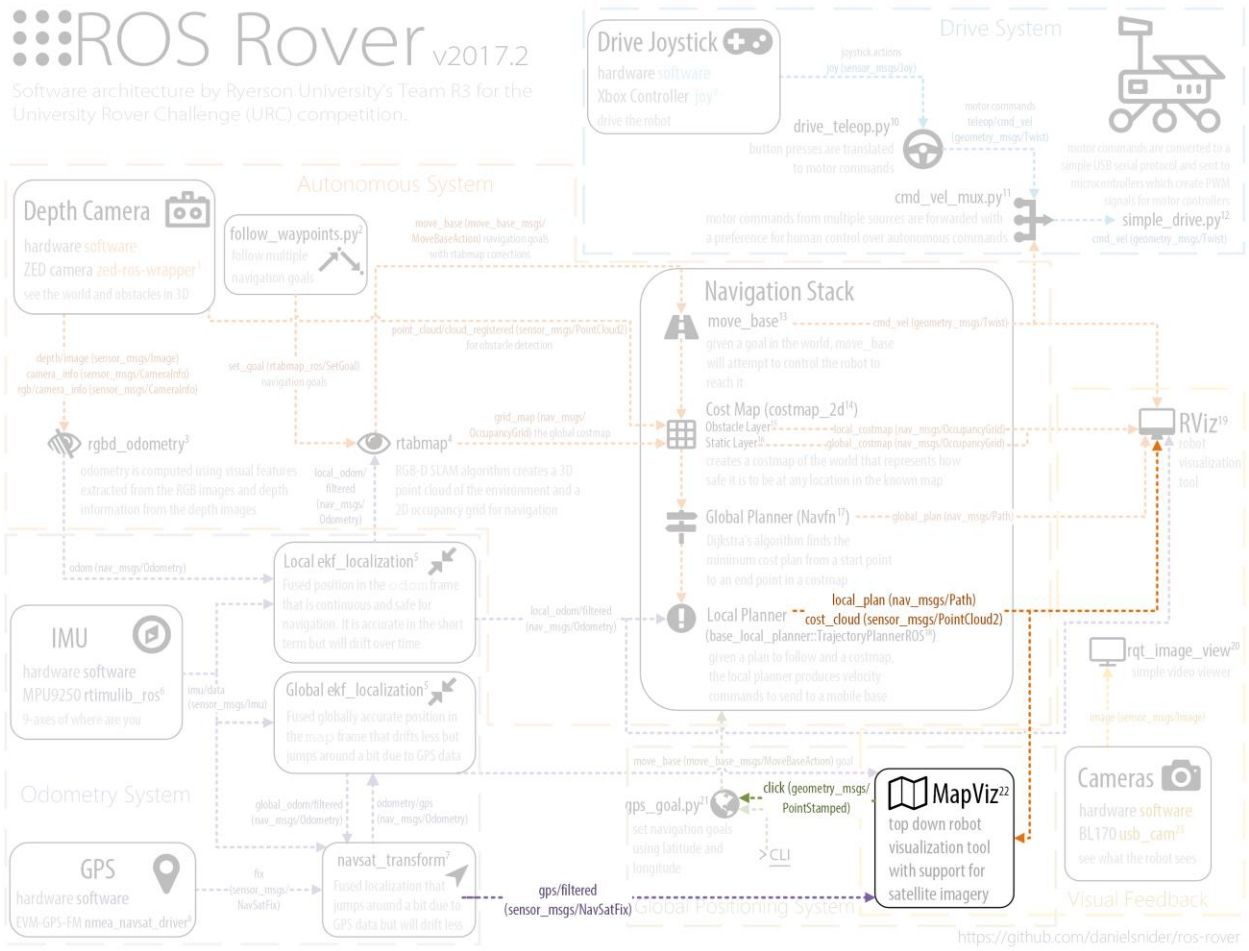

Fig. 21 MapViz ROS node seen within Team R3's rover system. See section 4.1 for the full diagram.

5.10.1 Quick Start

We will install and configure MapViz to display Google Maps.

1. Install:

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

2. Launch:

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

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

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

The `-v ~/mapproxy` option is a shared volume, a folder that it synced between your Docker container and your host computer. You will need to create the `~/mapproxy` folder, though it could be another location.

4. Confirm MapProxy is working by browsing to http://127.0.0.1:8080/demo/. You will see the MapProxy logo and if you click on "Image-format png" you will get an interactive map in your browser.

You can also see the first map tile 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...".

Congrats! You should now see Google Maps load in MapViz.

5.10.2 MapViz FAQ

Where are MapProxy's cached files?

```
~/mapproxy/cache_data
```

Any maps that you load will be cached to ~/mapproxy/cache_data and will be available offline.

How to set a default MapViz position (location in the world)?

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

How to publish GPS coordinates over ROS without a GPS?

```
$ rostopic pub /novatel/fix sensor_msgs/NavSatFix "{latitude: 38.406222, longitude: -
110.792027}"
```

5.11 Tutorial: GPS Navigation Goal

Source code: https://github.com/danielsnider/gps_goal
Wiki page: http://wiki.ros.org/gps_goal

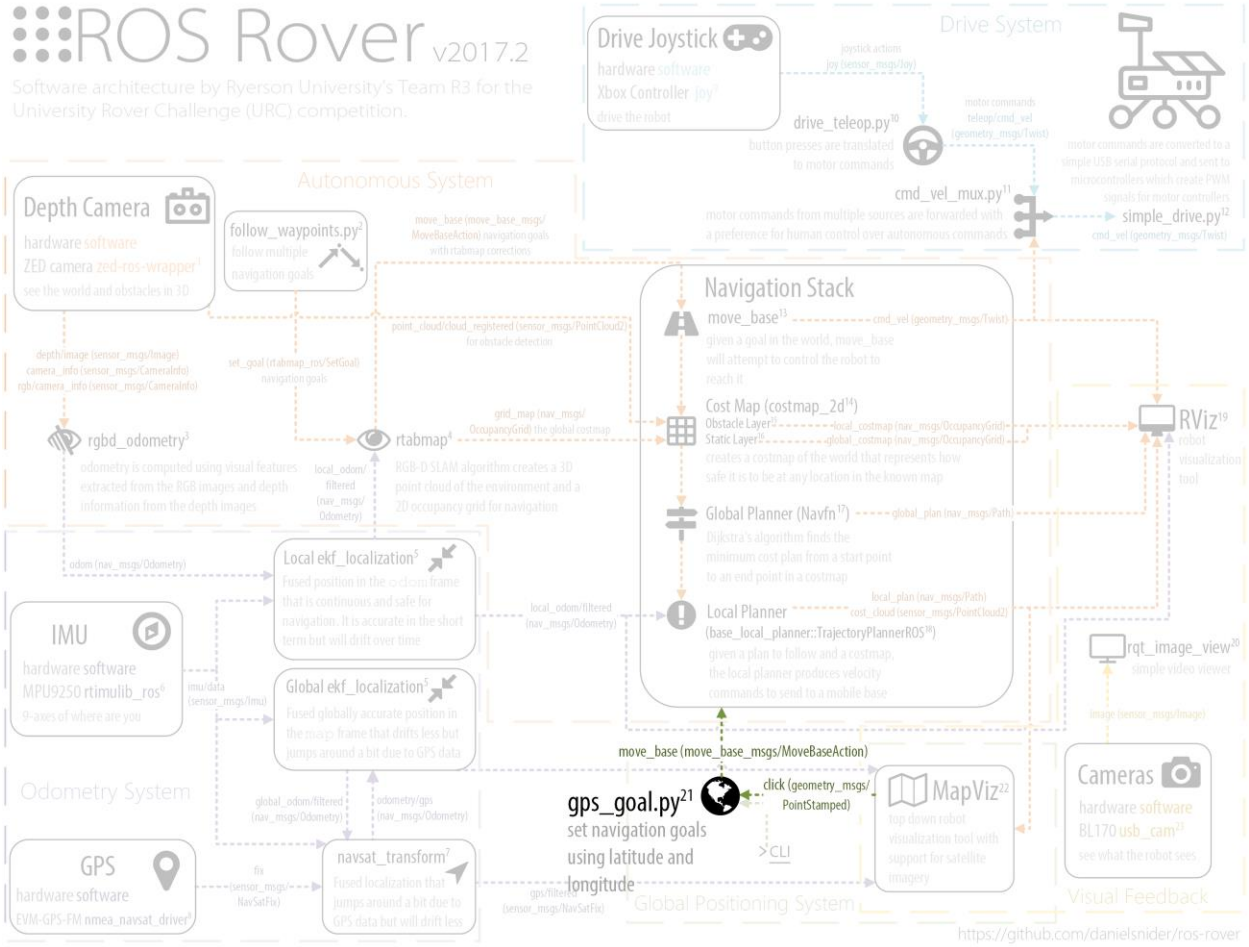

Fig 22 The GPS goal ROS node seen within Team R3's rover system. See section 4.1 for the full diagram.

The `gps_goal` ROS node will convert navigation goals in GPS coordinates to ROS frame coordinates. It uses the WGS84 ellipsoid[38] and geographiclib[39] python library for calculations.

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` parameter (typically `world`). This initial origin GPS location is best published using a helper `initialize_origin` node (see the initialize_origin section below).

5.11.1 Quick Start

1. Install:

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

2. Launch:

[38] https://en.wikipedia.org/wiki/World_Geodetic_System#A_new_World_Geodetic_System:_WGS.C2.A084
[39] https://geographiclib.sourceforge.io/

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

3. Initialize known starting GPS location:

This is needed to calculate the distance to the goal.

```
# use the next GPS coordinate (requires package ros-kinetic-swri-transform-util)
$ roslaunch gps_goal initialize_origin.launch origin:=auto
OR
$ rostopic pub /local_xy_origin geometry_msgs/PoseStamped '{ header: { frame_id: "/map" }, pos
e: { position: { x: 43.658, y: -79.379 } } }' -1
```

4. Set a goal using GPS:

```
$ 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
OR
$ 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
```

5.11.2 Command Line Interface (CLI)

You can invoke the `gps_goal` script once using the Command Line Interface (CLI).

```
$ roscd gps_goal
$ ./src/gps_goal/gps_goal.py --help

Usage: gps_goal.py [OPTIONS]

  Send goal to move_base given latitude and longitude

  Two usage formats:
  gps_goal.py --lat 43.658 --long -79.379 # decimal format
  gps_goal.py --lat 43,39,31 --long -79,22,45 # DMS format

Options:
  --lat TEXT         Latitude
  --long TEXT        Longitude
  -r, --roll FLOAT   Set target roll for goal
  -p, --pitch FLOAT  Set target pitch for goal
  -y, --yaw FLOAT    Set target yaw for goal
  --help             Show this message and exit.
```

gps_goal ROS Node

5.11.3 Actions Called

`move_base` (move_base_msgs/MoveBaseAction)
: The `move_base` action server performs navigation to the goal.

5.11.4 Subscribed Topics

`gps_goal_pose` (geometry_msgs/PoseStamped)

Set a GPS goal using a `PoseStamped` message. The x and y values will be considered latitude and longitude respectively. Z, roll, pitch, and yaw values will be respected.

`gps_goal_fix` (sensor_msgs/NavSatFix)
Set a GPS goal using a `NavSatFix` GPS message. The goal for roll, pitch, and yaw will be 0 degrees because they are not in this message type.

`local_xy_origin` (geometry_msgs/PoseStamped)
The topic to get the origin GPS location from. 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.

5.11.5 Parameters

`~frame_id` (`string`, default: map)
The tf frame for `move_base` goals.

5.11.6 initialize_origin ROS Node

The `initialize_origin` node will continuously publish (actually in a latched[40] manner) a geometry_msgs/PoseStamped on the `local_xy_origin` topic and this is better than 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. 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:=MRDS`".

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

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:

```
<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 recieves -->
  <remap from="gps" to="gps"/>
  <rosparam param="local_xy_origins">
    [{ name: MDRS,
```

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

```
      latitude: 38.40630,
      longitude: -110.79201,
      altitude: 0.0,
      heading: 0.0}]
  </rosparam>
</node>
```

5.11.7 Normal 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
```

5.12 Tutorial: Effective Robot Administration

5.12.1 ROS Master Helper Script

Team R3 has developed a script to automatically set your ROS_IP environment variable and to detect an online robot using ping and set your ROS_MASTER_URI environment variable (assumes a static IP for your robot). If your robot is not online your own ROS_IP will be used in your ROS_MASTER_URI. To use this script run 'source ~/set_robot_as_ROS_master.sh' or add to you '.bashrc'.

Source code: https://gist.github.com/danielsnider/13aa8c21e4fb12621b7d8ba59a762e75

5.12.2 tmux Terminal Multiplexer

What is tmux? Tmux[41] organizes many terminals into groups and will continue running if 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 see tmux as essential to their work[42].

Tmux's organization works harmoniously with ROS's modular design. Separate tmux windows can display different ROS components. Almost any ROS component can be launched, controlled, and debugged using ROS's command line tools. Using tmuxinator[43], you can codify the launching and debugging commands that you most often use into a repeatable layout.

[41] https://github.com/tmux/tmux/wiki
[42] https://robots.thoughtbot.com/a-tmux-crash-course
[43] https://github.com/tmuxinator/tmuxinator

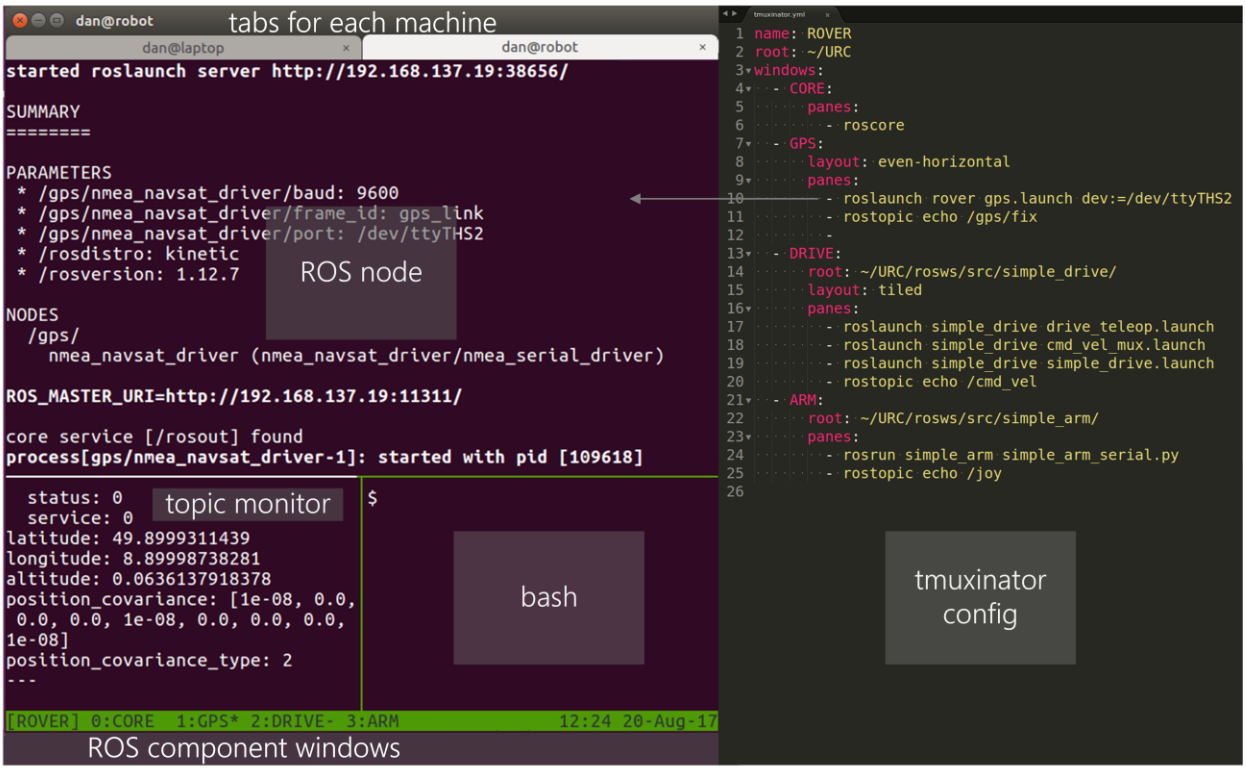

Fig. 23 An annotated example of tmuxinator's usefulness for ROS.

Tmuxinator is the quickest and simplest user interface that you could build to administer your robot (complemented by Rviz 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 of "pluming" and "glue code".

A small issue you may run into when using tmuxinator is if you start roscore and roslaunch at the same time, the problem will be that roslaunch will not find roscore because it is still starting. To solve this naively we have used a small wait time, for example "`sleep 3; roslaunch...`" in our tmuxiniator config.

The tmuxinator configuration used by Team R3 was split into two sides: the robot config[44], and the base station config[45]. The robot configuration launches all of the robot's software. The base station configuration launches all of the software needed to visualize and control the robot.

6 Further Collaboration

Ideally, the teams of URC should look past the competitive nature of the event and view collaborating and building better robots as the more important goal. We should build on a common core as much as possible and focus on the hardest parts. Here are a few ways to encourage collaboration:

- Discuss URC on: http://urchub.com/
- Discuss ROS on: https://discourse.ros.org/

[44] https://github.com/teamr3/URC/blob/master/.tmuxinator.yml
[45] https://github.com/teamr3/URC/blob/master/devstuff/dan/.tmuxinator.yml

- Contribute to the ROS core or a ROS package. Open an issue, feature request, or pull request.
- Contribute to a book like this.

Authors

Daniel Snider, Ryerson University, Hacklab.TO Toronto, Canada, Team R3

Matthew Mirvish, Bloor Collegiate Institute, Hacklab.TO Toronto, Canada, Team R3

Michał Barciś, University of Wrocław, Poland, Team Continuum

Vatan Aksoy Tezer, Istanbul Technical University, Turkey, ITU Rover Team

Advisor

Professor Michael R. M. Jenkin P.Eng., Electrical Engineering and Computer Science, York University, NSERC Canadian Field Robotics Network

Thank You to the Survey Respondents

Khalil Estell, San Jose State University, SJSU Robotics

Jacob Glueck, Cornell University, Cornell Mars Rover

Hunter D. Goldstein, Cornell University, Cornell Mars Rover

Akshit Kumar, Indian Institute of Technology, Madras, Team Anveshak

Jerry Li, University of Waterloo, UWRT

Jonathan Boyson, Missouri University of Science and Technology (Missouri S&T), Mars Rover Design Team

Michał Barciś, University of Wrocław, Team Continuum

Vatan Aksoy Tezer, Istanbul Technical University, ITU Rover Team

Gabe Casciano, Ryerson University, Team R3