Project 6: Reign of Gunters!

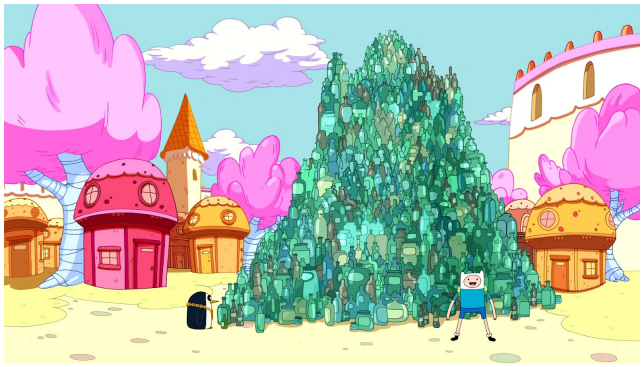
This project can be done in groups. Hand in this project by uploading the package via the ELMS website; there should be one submission per group.

Most of the guidelines (as well as starter code) are designed for Python. C++ developers will get some additional extra credit (+20%, as usual) for their implementations.

Note! This is a **very hard project**. Most of the software will not work out of the box and you will face a lot of issues not directly connected to your code (like internal segmentation faults, various library incompatibilities and just poor algorithm performance). For this project, **you are allowed to discuss your issues with anybody** (not only with team members) and **use any publicly available resources** to make the project work.

DELIVERABLES:

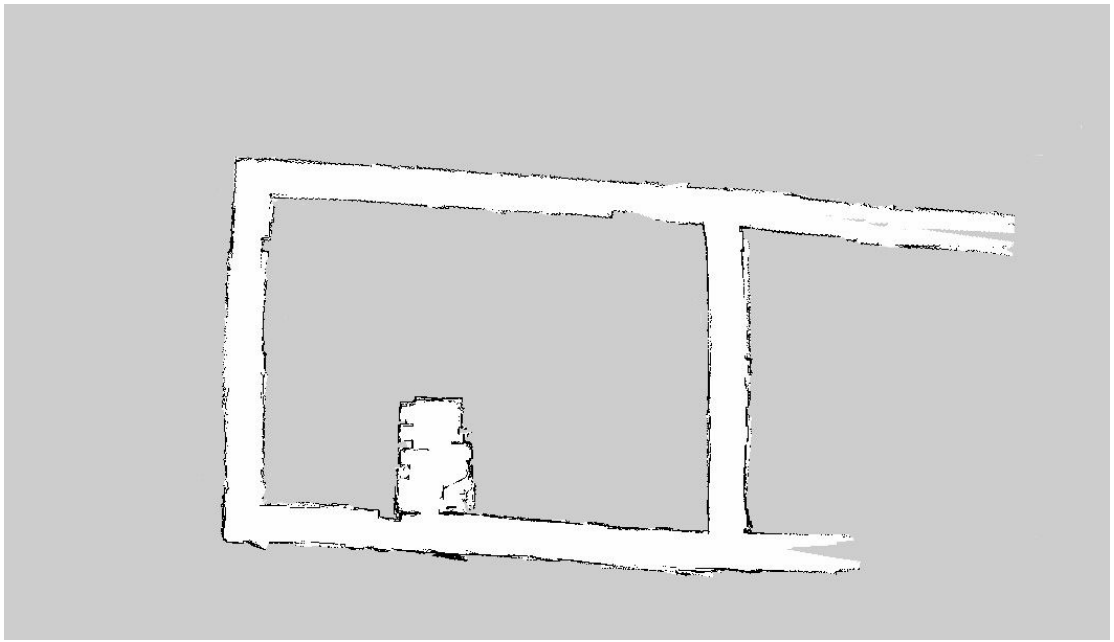
- ***project_6*** folder with your packages
- A link to a video of your robot executing the task
- A short report on methods you have used



Gunter loves to smash bottles! For this task, the robot will be placed in a maze. There will be several objects (bottles) placed along the walls in the maze. The robot must navigate through the maze, find the locations of the objects and hit them with the gripper so they would fall. The extra credit is to pick them up and smash them! To help you with this task the following information is provided:

1) The map of the maze. It will be used to navigate around the maze.
2) The objects (bottles) will be placed along the walls in several areas; the approximate coordinates of the areas will be given.
3) The images of the objects to be found. These will be compared to the images from the robot's camera to detect the search objects.

The sample of maze and the provided map:



1. Navigation

The package hierarchy is similar to the one in *project 5*. A slightly modified version of *adventure_gazebo* is provided. To navigate in the map we will be using the following ROS packages:

- *amcl* allows localizing a robot in the known map using laserscanner data
- *move_base* a set of planning and obstacle avoidance algorithms
- *actionlib* is a server client interface that allows sending navigation goals to *move_base*

Using these packages we can specify a goal pose in the map (x, y, θ) and these packages will take care of moving the robot to the desired position while avoiding obstacles along the way. To help you develop and test your navigational code we will be using Gazebo simulator.

Note! Play around with the simulator and the real robot to get a feeling of how robust and accurate (it is not) the path planning is. It will help you in development of the next parts of the project.

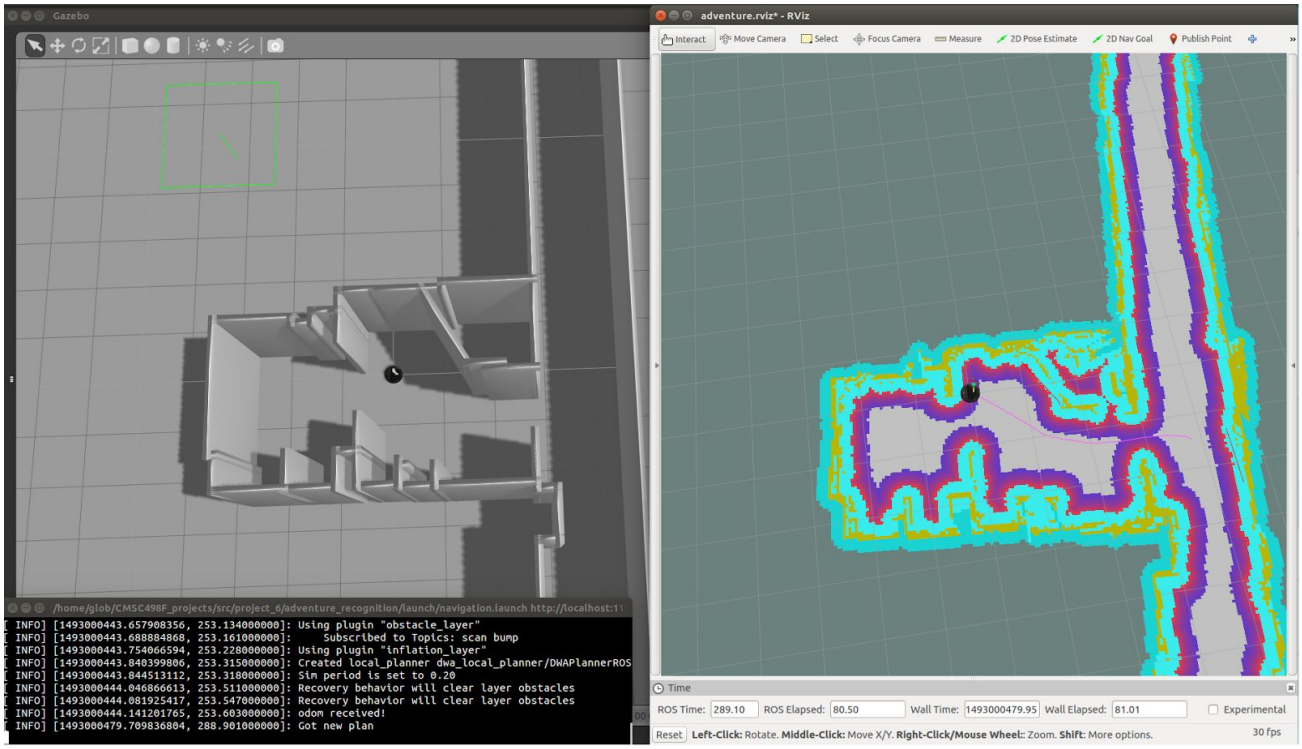
As usual, run the simulator:

roslaunch adventure_gazebo adventure_demo.launch

Note, that the only difference between *adventure_demo* and *adventure_world* is that the first one launches the laserscan, while the second just runs the simulation. Now launch:

roslaunch adventure_recognition navigation.launch

You will see something similar to:



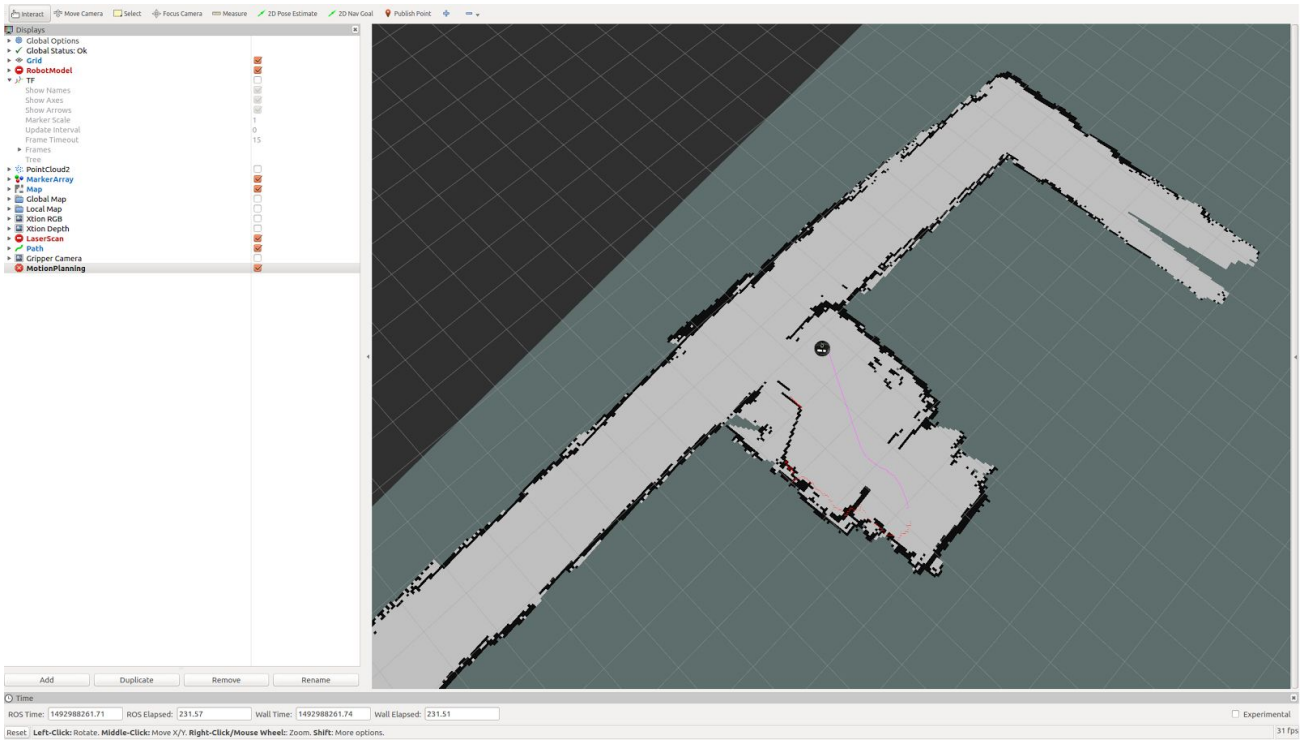
To test that navigation stack is working set a goal in the map using the '2D Nav Goal'. You should see the robot navigating to the specified point in the map. This ROS tutorial explains how to send goals to the actionlib programmatically from Python: (link)

Do not forget to experiment on a real robot. Run on the platform:

roslaunch adventure_bringup adventure.launch

And on your laptop (in separate terminals):

a) *roslaunch adventure_recognition navigation.launch*
b) *roslaunch adventure_gazebo adventure_monitor.launch*



A thin violet line shows the output of the path planning algorithm - robot will attempt to follow that trajectory.

Initially, you will be given three bottles to 'smash', the approximate coordinates are: (2.7, -1.6); (0.4, 0.0) and (0.7, -1.6). You will need to add some of your objects to this set later. You will have to be able to approach the area of interest pretty precisely, as object detection range is about 1 meter.

2. Object detection

Now given the coordinates we can move around, the only thing left - to get those coordinates! We will detect objects using SIFT features.

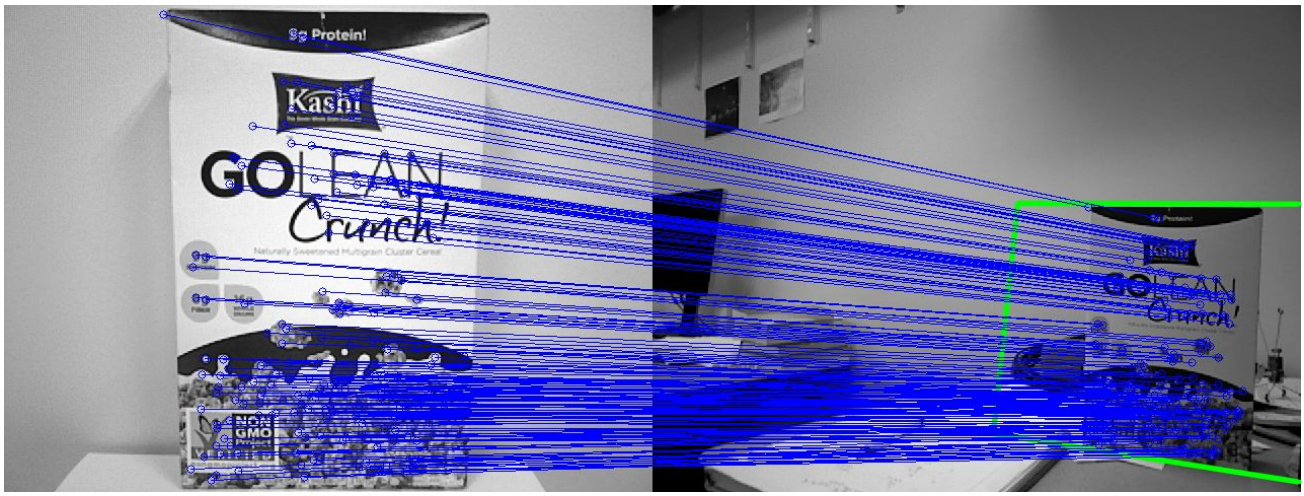
To detect objects in the image we will be using the approach of feature matching. The first step is to detect keypoints in an image. Keypoints are 'interesting' points in the image i.e. points that look distinguishable from their neighbours. Keypoints are then characterized using descriptors. Descriptors are signatures (usually 1D arrays of or binary numbers) that are used to measure similarity or distance between two keypoints. The idea of keypoint matching is that two images of the same object will have a lot of keypoints that are similar. Detecting these similarities allows us to detect the presence and position of objects on the image.

We will be using OpenCV library to implement this strategy. The following tutorial explains how to find an object in a cluttered scene and draw a bounding box around it: (link). You literally just need to copy the code and adjust it to ROS. The input images are in *project_6/adventure_recognition/images/train/*

The key steps of the algorithm are:

1. **Feature detection.** Detect SIFT keypoints and descriptors in both images.
2. **Matching.** For each keypoint from the first image find two keypoints from the second image that have the closest descriptors. To speed up the matching process a KD-tree data structure is used.
3. **Ratio test.** Reject all matches for which (distance of best feature match) / (distance of second best match) is smaller than some threshold (Lowe's paper suggests a threshold of 0.7). Note that here distance means similarity score between the features, not the distance in the image.
4. **Homography rejection.** Use the RANSAC algorithm to find a set of features that define a good transformation between the two images.

Feature matching example. Keypoints are shown with circles, lines denote the matches. Green polygon shows the alignment of the object image to the scene image:



You should write your detection code in *project_6/adventure_recognition/object_search.py*. The code provided for you contains the *ObjectSearch* class that has functions that subscribe to the image topic from the robot camera, visualize the live image stream, allow you to save individual frames from the stream to disk and visualize keypoint matched between two images. You can use this class as a template for your object search code. However, feel free to rewrite the file completely if you need to. To see what you have initially:

rosrun adventure_recognition object_search.py

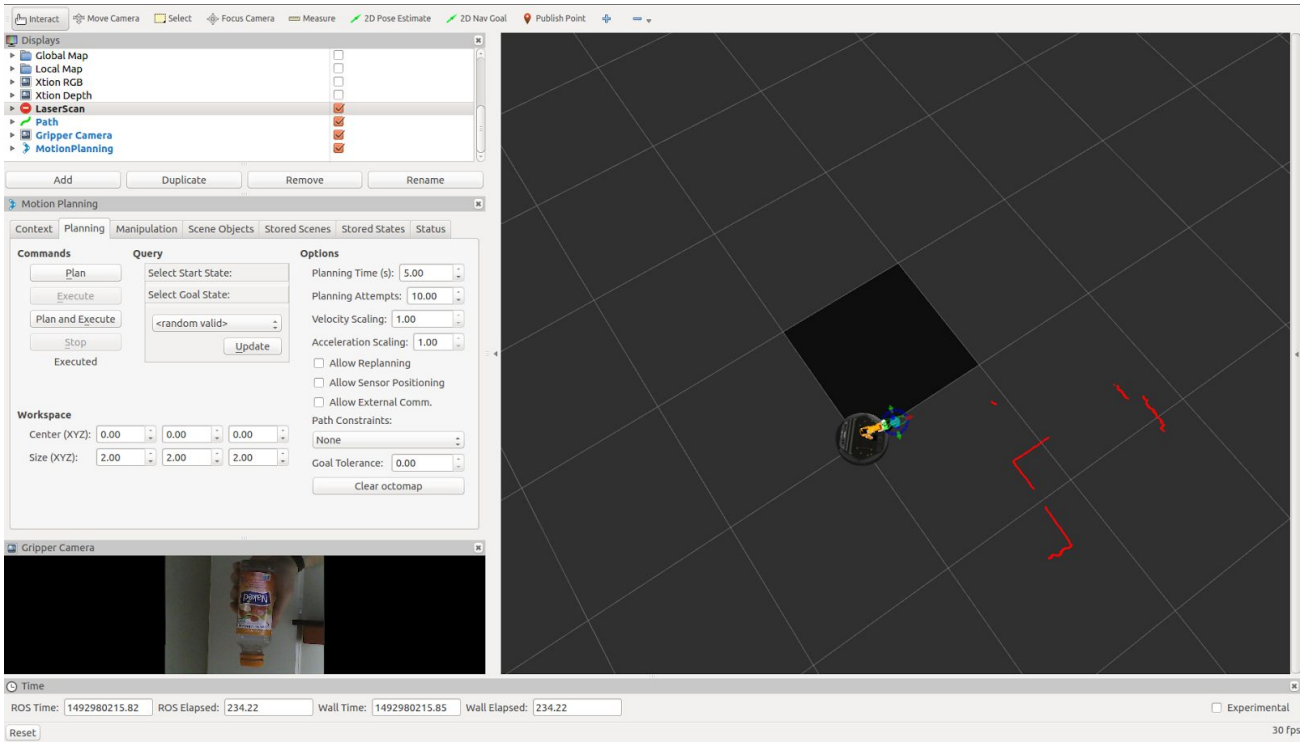
A piece of advice: Implement this module separately from ROS and get some objects (boxes, bottles, …) to experiment on and ease up debugging. Note, that performance with Python will be around 2 fps and

maximum detection distance will be around 1 meter. Experiment with both Xtion camera and Gripper camera - they have different fields of view and good for different ranges.

If you have issues: Now since you are about to use SIFT features for detection, you might face some issues with Python being unable to locate the SIFT module. The best way to work around that is to build new OpenCV from source (this is easy!). Just follow instructions here and **install OpenCV 3.0.0 rc1**. As of now, OpenCV 3.1 and 3.2 have serious issues with Python bindings (but C++ users should be fine).

Extra task: In the real experiment you will try to detect a number of bottles and hit them with the gripper so they would fall or (extra credit) grip objects (bottles) and smash them. In any case, you will need to know the approximate distance to the object. Try to estimate it based on the size of the bounding box. Keep in mind, that the bounding box may appear smaller than it is if the object is viewed at an angle. Also remember, you just need an estimate, you do not need to get a precise distance to the object!

On this image, you can see the output from the Gripper camera (bottom left), as well as the gripper manual controls (in the center) in Rviz:



3. Gripping

This is probably the hardest part of the project. The gripper is controlled by the MoveIt library, which is a very complicated piece of software designed for motion planning. MoveIt is standalone, but integrated with ROS - you can see the full system architecture here. The MoveIt for this project is already configured for you, your task will be only to figure out how to give the commands to the gripper from the code.

Note! The Gazebo support for the gripper is very limited - in the simulator, you will only see the movement of the gripper in Rviz, but you will not be able to actually interact with anything in Gazebo, neither will work the gripper camera. **Please, test your code on a real platform!**

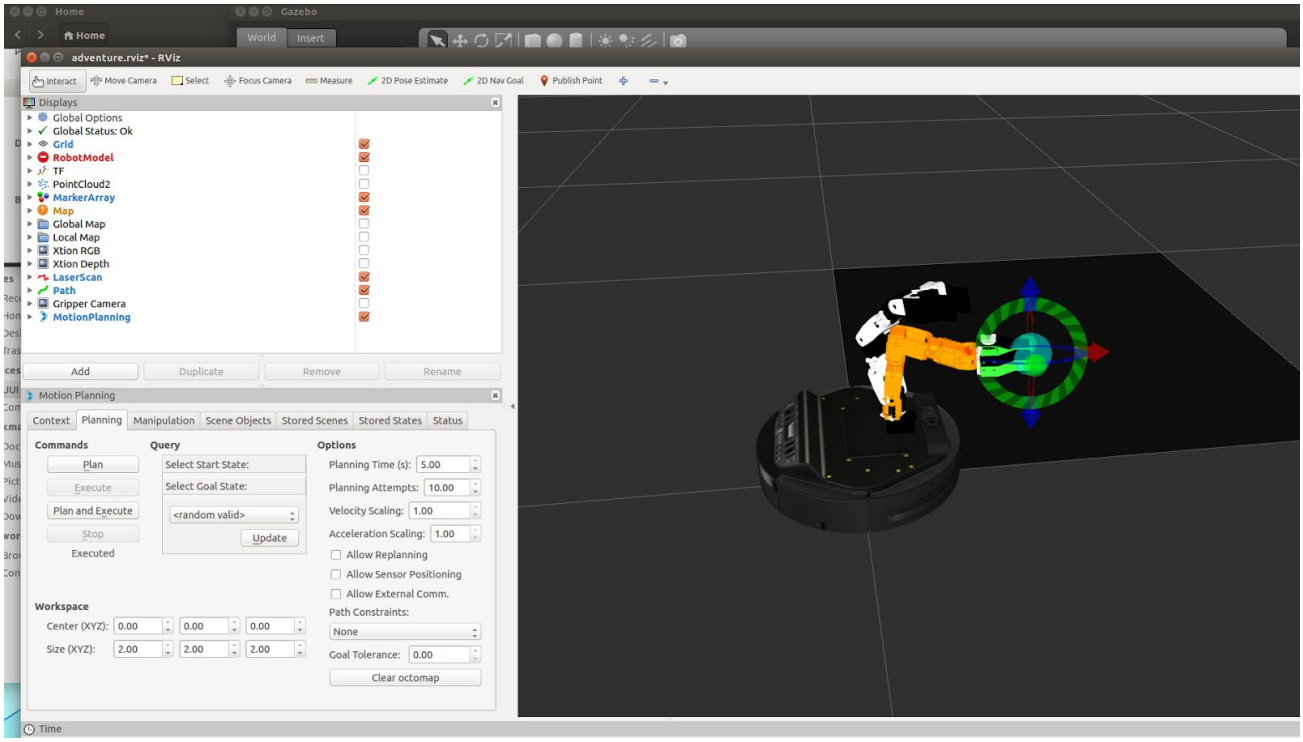
For the simulator, run the:

roslaunch adventure_gazebo adventure_demo.launch

On the real robot, run:

roslaunch adventure_bringup adventure.launch # on the platform
roslaunch adventure_gazebo adventure_monitor.launch # on your laptop

You will see something like on the picture:



To manually control the gripper, move around the blue sphere and then hit ‘Plan and Execute’ button. You will see how the arm moves in Rviz (if the motion is possible) and on the real platform the arm will move accordingly.

Task: Now we need to control the arm from the Python (C++) script. Please, check the sample code and a tutorial: link, link. If you see an error:

ABORTED: No motion plan found. No execution attempted.

That means that you try to move your gripper to the unreachable location.

4. Making it work all together

You will face several major issues during this project - the lack of simulator support, the unreliability of the software and high complexity of the algorithms involved: you need to use localization, planning, detection and gripping at the same time and all these modules have their issues. That is what makes robotics so much fun!

For this project you get a complete carte blanche - be creative! You have everything you need to complete the task, just get it done! Create new nodes, new launch files; change the configuration files and tweak the simulation. You just need to find and smash the bottles on a known map.

5. Extra credit

Just hitting the objects is way too easy for you? You can grip the bottle and throw it against the wall! You will get a partial credit for this part if you use magnets or velcro instead of gripping.

6. Grading

Testing navigation starter configuration	20 points
Object detection	25 points
Estimate distance to the detected object	5 points
Testing gripper starter configuration	10 points
Control gripper from code	30 points
Making everything work together	80 points
Extra credit	80 points
Report	30 points
Total	280 points

7. Tips

1) The version of OpenCV that comes with ROS is different from the one used in the OpenCV tutorials. Replace *cv2.LINE_AA* with *cv2.CV_AA* for drawing lines.
2) Use the *drawMatches* function provided in *ObjectSearch* class instead of *cv2.drawMatches* and *cv2.drawMatchesKnn*.
3) You can lookup C++ documentation for OpenCV here: (link)

8. Useful commands and configs:

8.1. Simulation

a) ***roslaunch adventure_gazebo adventure_demo.launch*** # launches Gazebo and Rviz, spawns the robot, and runs laserscan - this will be the most frequently used launch file for simulation
b) ***roslaunch adventure_gazebo adventure_world.launch*** # launches Gazebo and Rviz, spawns the robot, but does not launch any other nodes (might be useful from time to time)

8.2. Real robot

c) ***roslaunch adventure_bringup adventure.launch*** # should be executed on the platform; will run all the drivers, such as platform driver, Xtion driver, gripper camera driver and gripper driver
d) ***roslaunch adventure_gazebo adventure_monitor.launch*** # should be executed on the remote laptop; just runs the Rviz
e) ***roslaunch adventure_gazebo adventure_gmapping.launch*** # a demo launch file to run the ROS gmapping slam on our platform. This was used to build the map

8.3. Both

a) ***roslaunch adventure_recognition navigation.launch*** # runs amcl, move_base, map server and other nodes for navigation and motion planning
b) ***rosrun adventure_recognition object_search.py*** # runs the starter code for SIFT object detection
c) ***roslaunch adventure_teleop adventure_teleop.launch*** # standard teleoperation node, useful for debugging

8.4. Configs

a) ***adventure_gazebo/launch/adventure_world.launch*** and ***spawn_finn.launch*** # contain the spawn coordinate for the robot in Gazebo
b) ***adventure_recognition/launch/candy_kingdom_map.yaml*** # the description file for the map, contains the shift of the map in respect to the world
c) ***adventure_recognition/launch/navigation.launch*** # contains the initial position of the robot on the map

d) ***adventure_arm_bringup/config/arm.yaml*** # contains parameters for the arm, including minimum and maximum servo positions
e) ***adventure_arm_moveit_config/config/finn.srdf*** # contains the Moveit configuration for the arm and default positions (like 'right_up' and 'resting')

9. Known errors:

9.1. This error happens when gripper control Python script exits. Ignore this error;

terminate called after throwing an instance of
'boost::exception_detail::clone_impl<boost::exception_detail::error_info_injector<boost::lock_error> >'
what(): boost: mutex lock failed in pthread_mutex_lock: Invalid argument
Aborted (core dumped)

9.2. The following error is connected to malformed gripper semantic description, yet the gripper is fully functional:

Semantic description is not specified for the same robot as the URDF

9.3. The kobuki platform on *finn* and *jake* is known to emit this error. It does not affect the performance (*gunther* and *bmo* do not have this issue):

Kobuki : malformed sub-payload detected. [140][170][8C AA 55 4D 01 0F 48 F4]