Project 6: Treasure hunt!

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.

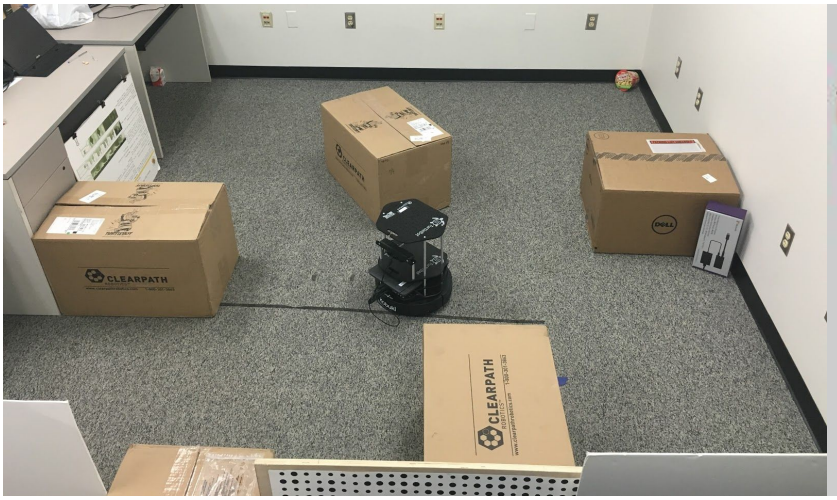
DELIVERABLES:

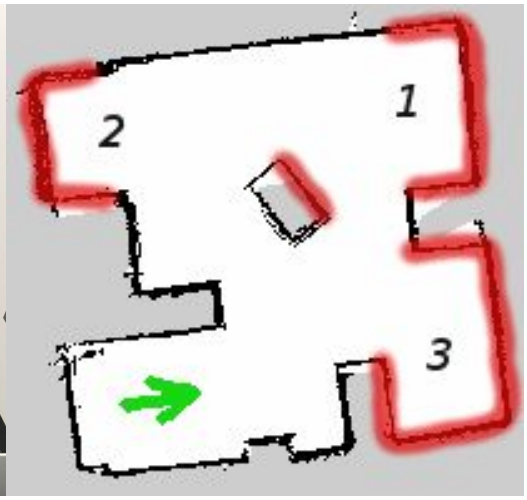
- ***project_6*** folder with your packages
- A link to a video of your robot executing the task
- A short report on methods you have used

The goal of this project is to get the robot to play the game of treasure hunt. The robot will be placed in a maze. There will be several objects placed along the walls in the maze. The robot must navigate through the maze, find the locations of the objects and pick them up! To help it 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 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. Approximate object locations are marked with red, green arrow denotes the initial pose of the robot:





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.

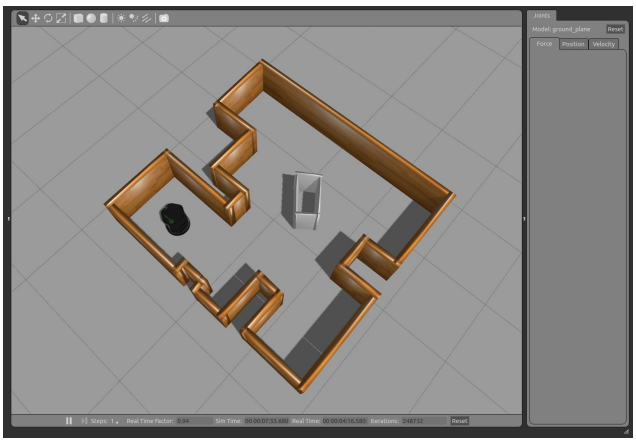
As usual, to run the simulator:

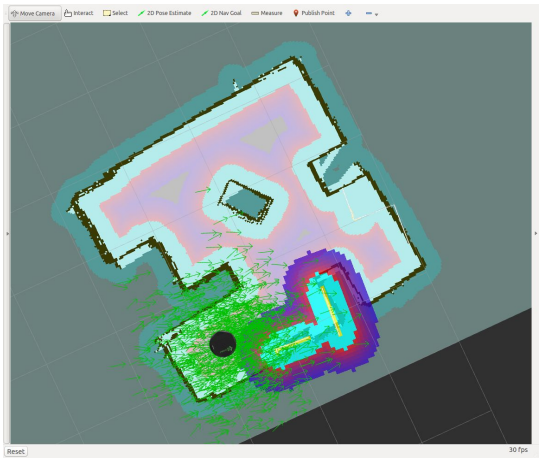
roslaunch adventure_gazebo adventure_demo.launch

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)

2. Object detection

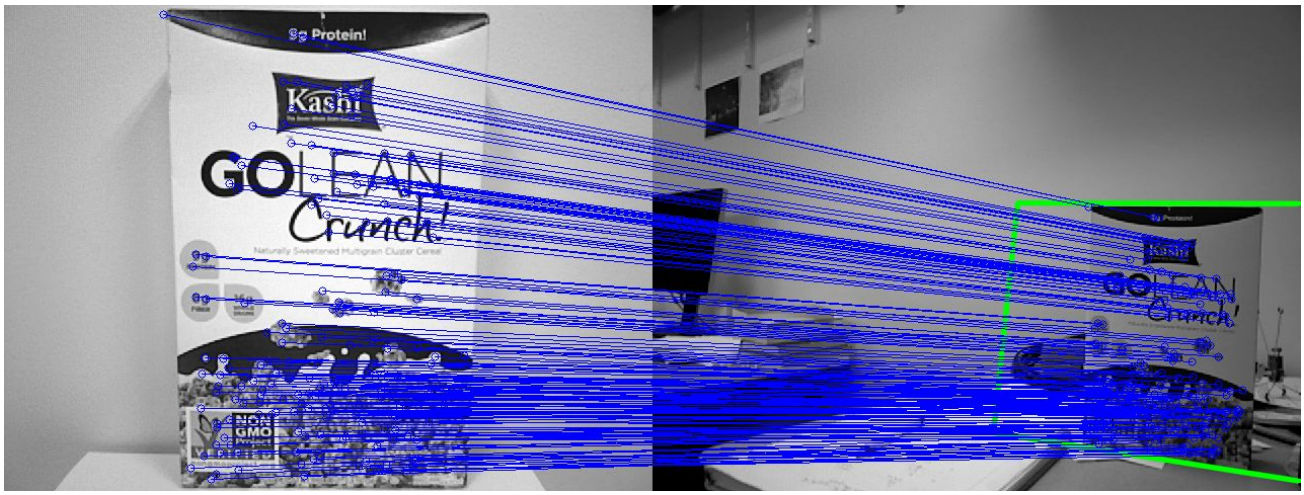
To detect objects in the image we will be using the approach of feature matching. The first step is to detect keypoints in an image. Keypoints are 'interesting' points in the image i.e. points that look distinguishable from their neighbours. Keypoints are then characterized using descriptors. Descriptors 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 of objects in 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). The input images are in *project_6/adventure_recognition/images/train/*

The key steps 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. Your goal is to detect the coordinate of the target object so that you could approach it and grip it!

3. Gripping

<TBA> Now that you know where the object is, grab it and put it in the basket.

4. Extra credit

<TBA>

5. Grading

navigation	20 points
detection	30 points
gripping	120 points
?	?
extra credit	?
Report	30 points

6. 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)