Robotics and Perception *CMSC 498F, CMSC 828K, Spring 2016*

Assignment 5: Treasure hunt!

NOTE: this assignment can be done in groups of up to 4 people.

The goal of this assignment is to get the robot to play the game of treasure hunt. The robot will be placed in a maze (Figure 1a). There will be several objects placed along the walls in the maze. The robot must navigate through the maze and find the locations of the objects. To help it with this task the following information is provided:

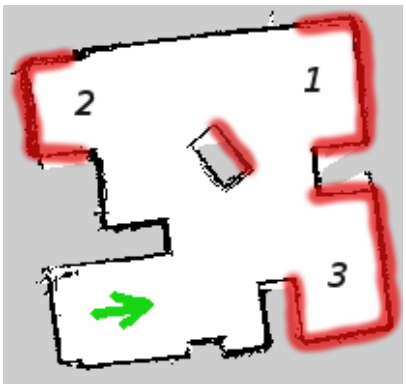
- The map of the maze. It will be used to navigate around the maze (Figure 1b).
- The objects will be placed along the walls in three areas in the map (Figure 1b). The map coordinates of these areas are given in Table 1.
- The images of the objects to be found. These will be compared to the images from the robot's camera to detect the search objects (Figure 4).

The simple approach to this problem is the following:

```
0:   angular_step = X degrees
1:   For number of areas:
2:   Go to the center of the area
3:   For 360 / angular_step
4:     rotate robot by angular_step degrees
5:     capture image from robot camera
6:     find objects in the image
```

The tools required to implement this simple approach are described in the sections below. Your goal is to implement this simple approach or use it as a starting point to come up with your own more sophisticated method for finding objects in the map.

(a)



(b)

Figure 1: (a) Image of a maze. (b) Map of the maze. Approximate object locations are marked with red, green arrow denotes the initial pose of the robot.

1 Navigation

To navigate in the map we will be using the following ROS packages:

- `amcl` allows localizing a robot in the map using laserscanner data.
- `move_base` a set of planning and obstacle avoidance algorithms.
- `actionlib` 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:

1. Tell ROS where to look for the map. The map files `assignment_5_map.pgm` and `assignment_5_map.yaml` are stored in `assignment_5/launch/` folder. Add the following line to your `~\.bashrc` file:

   ```
   export TURTLEBOT_MAP_FILE=<path_to_assignment_5>/launch/assignment_5_map.yaml
   ```

2. `roscore`
3. Launch simulation

   ```
   roslaunch assignment_5 simulation.launch
   ```

4. Launch ROS navigation packages

   ```
   roslaunch assignment_5 navigation.launch
   ```

5. Launch `rviz`

   ```
   roslaunch turtlebot_rviz_launchers view_navigation.launch
   ```

If everything works correctly you will see something similar to Figures 2 and 3. To test that navigation stack is working draw an arrow in the map using the "2D Nav Tool". 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:

`http://wiki.ros.org/actionlib_tutorials/Tutorials/Writing%20a%20Simple%20Action%20Client%20(Python)`

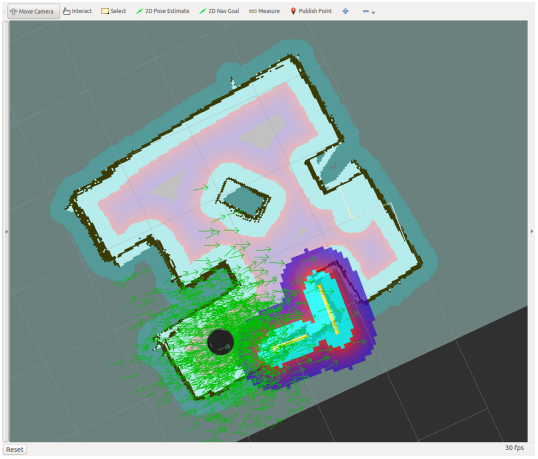


Figure 2: `rviz`

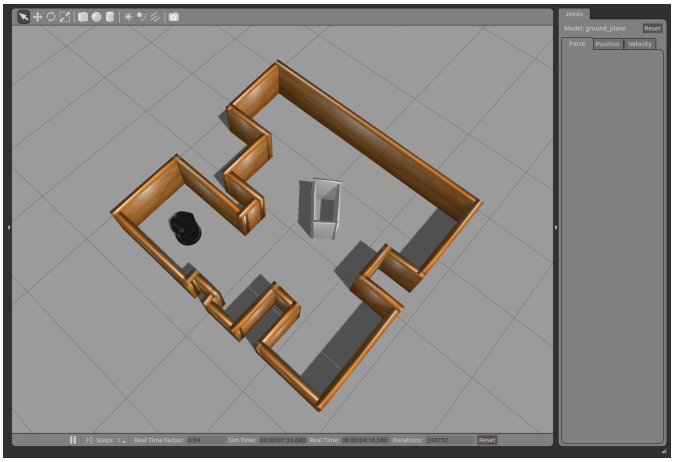


Figure 3: Gazebo simulator

Area ID	Coordinates
1	(4.0, 2.6)
2	(2.0, 2.4)
3	(4.5, 0.7)

Table 1: Approximate coordinates of the three search areas in the map.

(a) `cereal` (b) `dressing` (c) `kinect2` (d) `milk`

(e) `pringles` (f) `soup` (g) `straws`

Figure 4: Search objects.

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:

`http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_feature_homography/py_feature_homography.html#py-feature-homography`

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.

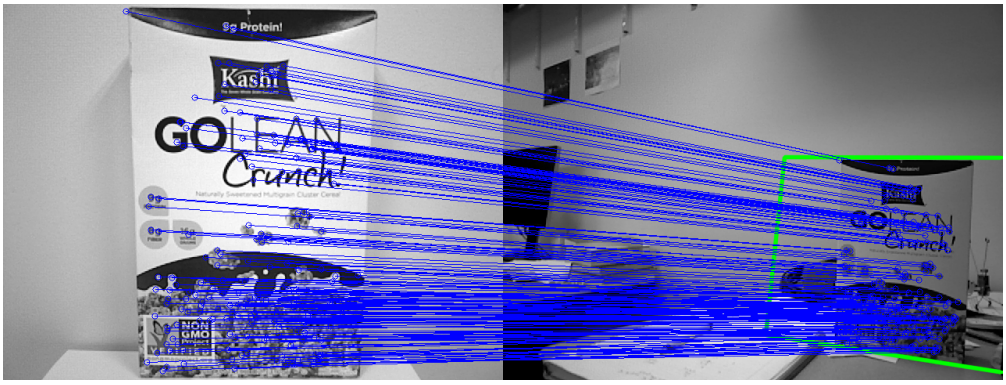


Figure 5: Feature matching example. Keypoints are shown with circles, lines denote the matches. Green polygon shows the alignment of the object image to the scene image.

At the end of the run your code should return a list of objects that were detected in the scene together with robot poses from which each object was detected. For extra credit augment this list by specifying the coordinates of each object found. You can use the depth camera on the robot to help you with that. You can get even more extra credit if you can overlay the detected object positions with the map.

You should write your code in `assignment_5/src/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 fell free to rewrite the file completely if you need to.

3 Deliverables and grading

The basic approach to navigation and object detection described above will not be sufficient to detect all of the objects in the map. You are encouraged to experiment with alternative methods. One possible strategy that has been demonstrated to give very good results, is to use first MSER feature detection and recognition. Then compute, using the MSER features a homography that warps the test image closely to the data-base image. In a second round then detect on the warped image, SIFT (or SURF) features, which are used for final matching (`http://www.cs.ubc.ca/~perfo/papers/forssen_iccv07.pdf`). Undergraduates will get full mark for correctly implementing the basic approach while graduates will need to implement alternative methods to get full mark.

You must demonstrate the robot executing your object search code to Alex. Submit the following via ELMS (a single person from your group should make the submission):

- `assignment_5` package
- a short report describing your approach
- a list of detected objects from the demonstration run
- a link to a video of your robot executing object search
- a link to a video showing the visual processes running on the robot (e.g. feature matching)
- a list of people in your group

The report should explain your navigation strategy and your object detection approach. It should discuss the performance of your approach i.e. which objects can be detected reliably, why are some objects harder to detect than the others and what steps did you use to overcome the limitations of the basic object search approach (if any).

4 Tips

- If you get a similar warning try restarting `navigation.launch`. If that doesn't work check that your have specified the path to your map file correctly.

```
[ WARN] [1461716483.096211062, 433.906000000]: Timed out waiting for trransform from
    base_footprint to map to become available before unning costmap, tf error: .
    canTransform returned after 0.101 timeout was 0.1.
```

- 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.
- Use the `drawMatches` function provided in `ObjectSearch` class instead of `cv2.drawMatches` and `cv2.drawMatchesKnn`.
- You can lookup C++ documentation for OpenCV here: `http://docs.opencv.org/2.4.9/modules/refman.html`