ROS perception #1: Vision Basics

cmvision package → blob tracking based on color tracking

rosrun rqt_image_view rqt_image_view

↳ to see camera images

rostopic list | grep ——

rostopic info /topic → Type → (cmvision / Blobs)

~~rostopic~~ → rosmsg show cmvision/Blobs

#2: Vision Basics

Most basic but powerful tool for perception → OpenCV

↳ Most extensive and complete library for image recognition

↳ Apply filters, post processing and working with images in any way you want

↳ Not a ROS library → but integrated via OpenCV bridge

Application: make robot follow a line

1° Get images from ROS topic and convert them into OpenCV format (BGR)
2° Process the images using OpenCV libraries to obtain the needed data
3° Move the robot along the yellow line based on the data

rosmsg show sensor_msgs/Image

```
:
uint32 height
:
uint8[] data
```

rgb8 → 8 bit integers

rostopic echo -n1 /camera/rgb/image_raw/height

topic parameter

↓ they won't appear if TAB TAB

chmod +X name.py

Raw images are useless unless you filter them to see only the color you want to track and crop irrelevant parts. To make the program faster

Before you start using images for detecting things:

- What are the dimensions?
- Cropping image → Minimize the size of the image as much as possible for the required task → Faster system

---

```
height, width, channels = cv_image.shape
descentre = 160
rows_to_watch = 20
crop_img = cv_image[(height)/2 + descentre : (height)/2 + (descentre + rows_to_watch)][1:width]
```

a) GET IMAGE INFO and CROP IMAGE
In this case you are interested in lines that are not too far away from the robot, nor too near. If you concentrate on lines too far, it won't follow them. If you concentrate on lines too close, won't give the robot time to adapt to changes in line.
It's also vital to optimize the region of the image as result of cropping

b) CONVERT from BGR to HSV
For differenciating colors it's not easy to work with BGR or RGB because they contain saturation info which is just noise for the purpose of color following. HSV allows to remove saturation
to light conditions always change

c) APPLY THE MASK → black and white (binary image)
It achieves ~~antigen~~ that you don't have contiguous / ambiguous detection

d) GET CENTROIDS → ~~point~~ represent points in space where mass concentrates
Applying the mask allows to calculate center of mass with a discrete function.
Otherwise, you would need a function considering fluctuations in color quantity

PID ROS package → integration in ROS

```
<launch>
  <include file = "$(find my_package)/launch/launcher.launch" />
  <node name = "" pkg = "" type = "" output = "" />
</launch>
```

3: Flat Surface and object recognition

- Recognize flat objects: Detect places where objects normally are, like tables and shelves. #1st step to searching for objects
- Recognize objects: Once you know where to look, you have to be able to recognize different object in the scene and localize where they are from your robot's location

catkin_create_pkg my_package rospy object_recognition_core

```
<launch>
  <arg name = "tabletop_ork_file" value = "$(find my package)/conf/detection.tabletop_
    _fetch.ros.ork" />
  <node pkg = "object_recognition_core" type = "detection" name = "tabletop_server_node"
    args = "-c $(arg tabletop_ork_file)" output = "screen">
  </node>
</launch>
```

(in binaries)

tabletop_ork_file → it's kind of yaml file with input sensors and values for the detection

1° create a conf directory in the package

2° launch

It is possible to pinpoint the location of any surface detected, or even filter the floor as we know the height of each surface (object pose)

2D Object finder

Basic approach to detecting objects although you can differenciate between them and localize them → find_object_2d package

Here you just use RGB camera information

Take picture in Find_Object and then crop to the desired object to recognize. Add objects from scene ← Edit

If you use Add objects from files → Mirror the pictures previous by !

You can add as many pictures of the same object turned around

Without the proper ~~object~~ filtering, the system will consider them to be different objects

Save_Objects → it will save taken imeges in a folder

save_session → the most compact way to do it although you won't have access to the images of the objects

---

```
<launch>
  <arg name="camera_rgb_topic" default="/head_camera/rgb/image_raw"/>
  <node name="find_object_2d" pkg="find_object_2d" type="find_object_2d"
    output="screen">
    <remap from="image" to="$(arg camera_rgb_topic)" />
    <param name="gui" value="true" type="bool"/>
```

if save_session → `<param name="session_path" value="$(find my_package)/item_session.bin" type="str" />`

if picture → `<param name="objects_path" value="$(find my_package)/folder_picture" type="str"/>`

```
    <param name="settings_path" value="~/.ros/find_object_2d.ini" type="str"/>
  </node>
</launch>
```

Delete model in gazebo : rosservice call gazebo/delete_model
"{model_name : coke_can}"

The image recognition algorithm is limited, it requires pictures from different angles of that certain object and if there is another version of that object it might not work (different model of backpack won't be recognized)

3D object detection

The difference now is the involved sensors and the fact that ObjectPoseStamped will be transformed into TFs

You have to create another round of session photos in the 3D system

---

```
<launch>
  <node name="find_object_3d" pkg="find_object_2d" type="find_object_2d"
  output="screen">
    <param name="gui" value="true" type="bool"/>
    <param name="settings_path" value="~/.ros/find_object_2d.ini" type="str"/>
    <param name="subscribe_depth" value="true" type="bool"/>
    <param name="session_path" value="$(find my_object_recognition_pkg)/
      /.../coke_session.bin type="str"/>
    <param name="objects_path" value="" type="str"/>
    <param name="object_prefix" value="object" type="str"/>
  <remap from="rgb/image_rect_color" to="/head_camera/rgb/image_raw"/>
  <remap from="depth_registered/image_raw" to="/head_camera/depth_registered/
  /image_raw"/>
    <remap from="depth_registered/camera_info" to="/head_camera/depth_registered/
    /camera_info"/>
</node>
```

```
<node name = "tf_example" pkg = "find_object_2d" type = "tf_example"
  output = "screen" >
  <param name = "map_frame_id" value = "/map" type = "string" />
  <param name = "object_prefix" value = "object" type = "str" />
</node>
</launch>
```

You will get the TF of the object detected published

If there are several pictures of the same object, multiple TF will be published nearby each other getting a clutter outlook

↳ The TFs appearing can be lowered by decreasing the time you consider a TF obsolete. Because most of these TFs are from previous detections that stay there for a while until they are old enough to be considered irrelevant

People tracking

ROS package for tracking people

There are various ways a person can be tracked:

- Detecting legs → laser patterns which have V shape → lots of false positives
- Detecting upper body → more robust
- Detecting pedestrians

None of them are as strong as all of them combined

cob_vision_create_pkg my_people_tracker_pkg geometry_msgs people_velocity_tracker

rospy

- Leg detector → leg_detector package ← /base_scan laser's reading data

```
<launch>
  <arg name="scan" default="/base_scan" />
  <arg name="machine" default="localhost" />
  <arg name="user" default="" />
  <node pkg="leg_detector" type="leg_detector" name="leg_detector"
    args="scan:=$(arg scan) $(find leg_detector)/config/trained_leg_detector.yaml"
    respawn="true" output="screen">
    <param name="fixed_frame" type="string" value="odom" />
  </node>
  <include file="$(find detector_msg_to_pose_array)/launch/to_pose_array.launch">
    <arg name="machine" value="$(arg machine)" />
    <arg name="user" value="$(arg user)" />
  </include>
</launch>
```

/to_pose_array/leg_detector is the transformation of /leg_tracker_measurements into PoseArray which will be needed when you combine all the systems together.

The end of the pipeline is → /people_tracker_measurements

- Detect upper body → publishes its estimated pose → Pose Stamped
  - " RGB image with a square overlapping where the person is

In the launcher:

- upper_body_detector.launch : makes the recognition
- ground_plane_estimated_launch: subscribed to the /ground_plane topic
  - allows to calculate and detect the distance to where the upper body is

/Bounding_box_center
/Closest_bounding_box_center } position detected people → used in case you want to follow the closest person in the initial moment all the way even if others appear

/Detections → person detections
/marker_array → same data but in marker format

limitations → problems detecting a person side ways

- Pedestrian detector

Uses detection of the ground to filter false positives and other algorithms to filter even more and make better predictions, including the upper body detections

mdl_people_tracker.launch works on the previous data of Upper Body and ground, plus visual_odometry

/mdl_people_tracker/image → cylinder around the detection
" /people_array
" /pose_array

rosrun rgt_image_view rgt_image_view

Combination of the 3 detectors

Once the required systems are working, you'll use that information and process it with the package spencer_people_tracking

To launch it, the most important thing is the configuration file in which you add all the detectors plus the detection by speed model

You unite this information through a Kalman filter (unscented one). Those are used to extract conclusions from sets of data that, by themselves are inaccurate, but combined with other data sources, are more precise.

detectors.yaml → my_people_tracker_pkg / config

↳ cartesian_noise_params → regulate the impact in the final decision that the detector makes

what values do you select?

Extract statistical values of each detector system

- You save X and Y values of the pose array. The more you have, the better the statistical values you'll get. Once done, generate mean and standard deviation of X and Y separately. σ_x and σ_y are the ones you have to use.

  ↳ not all scientific but very common in engineering tests

/people_tracker/people

" / pose

" / pose_array

" / positions

Point Cloud Sensor
- RGB images
- depth images
- PointCloud data

} Vital for object detection and people tracking

- minimum and maximum ranges

/camera/depth/image_raw → depth 2D image made through distance readings of the point cloud

rViz
- Camera Info → useful in perception to know if the robot should be seeing the object or not
- PointCloud 2
- Depth Cloud

Camera - Optic frame problem

Usually the frame of the place where the sensor is mounted is inverted from the sensor frame. Sensor readings might be inversed, and in order to visualise data in a coherent way, the sensor frames are reversed.