Lab 4: Computer Vision:
Camera Calibration, HSV Color Space and Depth Sensing
2.12: Introduction to Robotics
Fall 2016

Peter Yu, Ryan Fish, and Kamal Youcef-Toumi

Instructions:

1. **When your group is done with each task, call a TA to do your check-off.**
2. **After the lab, zip your files and make a backup using online storage/flash drive.**
3. **No need to turn in your code or answers to the questions in the handout.**

1 Introduction

In this lab you will experiment three computer vision techniques.

1. Calibrate the RGB camera on Kinect using a checkerboard.
2. Transform RGB images to HSV color space to detect an object based on color.
3. Use depth images from Kinect to find the 3D position of the object found in color image.

We will use OpenCV (Open Computer Vision) library to process images captured by Kinect to achieve Task 2 and 3.

2 Setting up the code

Open a terminal (Ctrl-Alt-T), and enter the following commands without the leading $.

```
$ cd ~                      # make sure we are at home folder
$ git clone https://github.com/mit212/me212lab4.git
$ cd me212lab4/catkin_ws    # go into the catkin_ws
$ catkin_make               # let ROS know our packages
$ rebash                    # reload bash resource file, ~/.bashrc
```

2.1 Folders and files

- `catkin_ws` : ROS catkin Workspace
  - `src` : source space, where ROS packages are located
    - `me212bot` : a folder containing a ROS package for the 2.12 moving platform.

    * `me212cv` : a folder containing a ROS package for the computer vision experiment in this lab.

We will use some contents from `me212bot` developed previously in the new package `me212cv`. Now let's focus on the content inside package `me212cv`, which is for our computer vision experiments.

- `scripts/object_detection.py` : a ROS node for detecting object by color and estimating object position in space.
- `launch/frames.launch` : a launch file for publishing necessary static coordinate transforms.

In this lab, you need to modify `object_detection.py`.

3 Task 1: Camera calibration

3.1 Introduction

Camera calibration is the process of finding parameters in the camera model. A camera model describes how a 3D point (x_c, y_c, z_c) with respect to the camera in the space projects to the image pixel coordinate (x_p, y_p).

Distortion Real cameras tend to have distortion in the images they take. Some lenses aren't very uniform, and compress the visual field near the edge of the frame. Other lenses, such as fish eye lenses, have a field of view wide enough that their image is more like a sphere than a rectangle. A line that is straight in the real world will appear curved because of distortions. We can account for these distortions with a camera model. Figure 1 shows an image from a fish eye camera on the MIT Atlas robot.

Some computer vision algorithms depend on identifying straight lines in the image. In these applications it is desirable to undistort them to straighten things out. However, not all applications require undistorting the image or the lens does not distort images much. Then we can skip the undistortion to save computational cost.

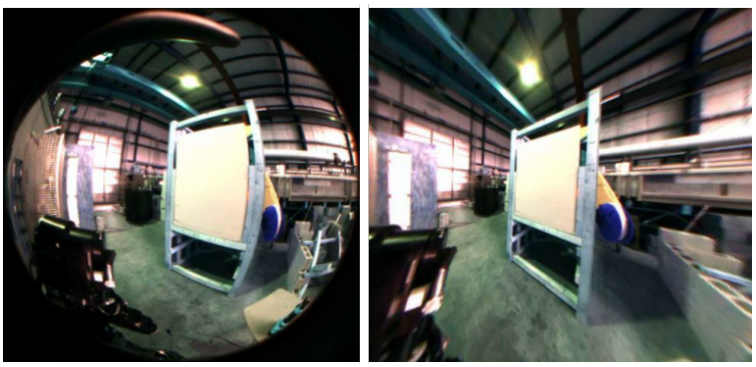

Figure 1: Left: image taken from a fish eye camera on the MIT Atlas robot. Right: rectified image. Observe the straight lines in the images. Adapted from [1]

We now describe the process of projecting a 3D point using a camera model. The description is adapted from [2].

First, pin hole camera projection. See Figure 2 for a visualization.

$$\mathbf{x}_n = [x_n, y_n] = [x_c/z_c, y_c/z_c] \tag{1}$$

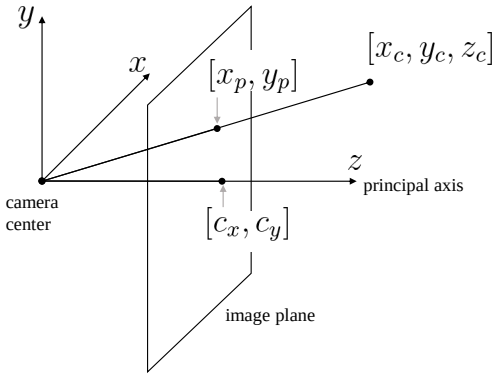

Figure 2: Camera model.

Second, include camera lens distortion. The distortion model used here is called Plumb Bob. Now, let $r = |\mathbf{x}_n|$. The distorted projection $\mathbf{x}_d$ is:

$$\mathbf{x}_d = (1 + k_1 r^2 + k_2 r^4 + k_5 r^6)\mathbf{x}_n + \delta, \tag{2}$$

where δ describes the tangential distortion, which happens when the lens and camera sensor are not perfectly parallel.

$$\delta = \begin{bmatrix} 2k_3(xy) + k_4(r^2 + 2x^2) \\ k_3(r^2 + 2y^2) + 2k_4(xy) \end{bmatrix} \tag{3}$$

Third, camera sensor mapping:

$$\begin{bmatrix} x_p \\ y_p \\ 1 \end{bmatrix} = \mathbf{K} \begin{bmatrix} x_d \\ y_d \\ 1 \end{bmatrix}, \tag{4}$$

where $\mathbf{K}$ is a homogeneous transform matrix:

$$\mathbf{K} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{5}$$

In summary we have the following parameters:

1. (f_x, f_y): focal distance expressed in number of horizontal and vertical pixels (pixel/meter). Sometimes the camera sensor cells are not square, so there then we separate them into two values;
2. (c_x, c_y): image center (pixel).
3. $(k_{1..5})$: distortion parameters. Their units are not important.

A normal camera calibration process involves taking pictures of the checkerboard in different poses. Then find the checkerboard corners in the images. Use them as measurement to a optimization problem: find the best set of parameters that gives the least amount of error given the constraints of checkerboard geometry, and camera model.

3.2 Instruction

We will use a ROS package `camera_calibration` to help us do the calibration process. The program is `cameracalibrator.py`. It simplifies the process: you don't need to capture the image explicitly. Instead, you just need to move the checkerboard *slowly* in front of the camera. The program will capture image and run the optimization by itself.

Type `pman` in terminal to launch `pman` GUI, and run the script `run-kinect-calibrate` to launch Kinect node and the following command.

```
$ rosrun camera_calibration cameracalibrator.py \
  --size 8x6 --square 0.025 image:=/camera/rgb/image
```

Here `\` means the two lines are essentially one line of command. Argument `size 8x6` is the number of *inner* corners of the checkerboard. Argument `square 0.025` is the dimension of one square in meter. Argument `image:=/camera/rgb/image` is a topic remapping. The node subscribes to a topic called `image`. However our camera publishes to `/camera/rgb/image`. To connect them together, we use `:=`.

In the calibration window, you will see the checkerboard detected if the image cover the whole checkerboard. Try to move it so that the program can capture the board in different poses in the view. Especially, pose it in extreme angles. See Figure 3.

Figure 3: Left: calibration interface. Right 3 images: example calibration images.

The `calibrate` button will become enabled when enough pictures are taken. Click it when it does. Wait for several seconds for the processing, then click "save." The calibration data will be saved at `/tmp/calibrationdata.tar.gz`. Open this file and copy the ost.yaml file in it to `~/.ros/camera_info`.

Hint 1: Folder Navigation Folder `/tmp` is a place for storing files temporarily. The content will be cleaned up after reboot. You can navigate to `/tmp` folder using one of the following methods.

- In a terminal, type "`open /tmp`". "`open`" is an alias for gnome-open, which ask Ubuntu to use the default program to open a file or directory.
- In a terminal, type "`cd /tmp`", and then type "`open .`". "`.`" means current folder.
- In a File Browser, press Ctrl-L. Then type in `/tmp` in the path bar.
- In a File Browser, click on `Computer` in the left menu bar. Then navigate to `tmp` folder

Hint 2: Hidden Folders and Files By convention, folders and files started with "." will be hidden, e.g. `.ros`. They usually contains configuration information. In a file listing such as in the File Browser, you may not see them. Try one of the following solutions.

- In a terminal, type `open ~/.ros`.
- In a File Broswer, press Ctrl-H. This will toggle showing or hiding hidden files.

You may need to create the `camera_info` folder in `~/.ros`. Rename the file to `rgb_SN.yaml`, where `SN` is your Kinect serial number. You can take a look at the Kinect printouts in `pman`. The serial number looks like `B00366623823047B`. After the file is copied, restarting the Kinect node will load the new parameters and publish it through topic `/camera/rgb/camera_info`. Note that cameras of the same model still have slight difference in the camera parameters because the camera sensors and lens are not placed exactly in the same way. That's why we use serial numbers to load the corresponding parameter file.

3.3 Undistortion

After calibration we get the parameters camera model, including projection and distortion. A common practice is get the image *undistorted*, or *rectified* before doing further processing. Undistortion is done in the Kinect package given the parameters we set. The original and undistorted images are published through two topics: `/camera/rgb/image_color` and `/camera/rgb/image_color_rect` respectively. We can run `rviz` to show undistorted image.

Bring up a `pman` GUI and run the script `run-object-detect`. To bring up `pman`, you only need to type `pman` in a terminal. On the left hand side of `rviz` you will see the original image on the top and undistorted image in the bottom. Note that the distortion for Kinect is relatively small. If you happen to see a fish eye camera, the effect will be large (Figure 1).

3.4 Inferring the 3D space from 2D images

The camera model projects a point in 3D space to the 2D image plane. However, we are more interested in the inverse problem: given an image pixel find its 3D point. Because we now have undistorted image, we can put the distortion effect aside to simplify the reasoning in the following text, i.e. assuming $k_{1..5}$ are 0 and $\mathbf{x}_d = \mathbf{x}_n$. Recovering 3D position is not trivial with only one camera image because as you can see in (1), the depth information, z_c, is merged with x_c and y_c. If we know z_c along with (x_p, y_p), we can recover (x_c, y_c, z_c). Some applications have known z_c, e.g. picking flat parts on a flat surface. Most of the time, we do not know other constraints so therefore the camera is only giving us *bearing* information of each pixel (angle relative to the principle axis). All possible points that can be projected to (x_p, y_p) forms a *cone* or a *square pyramid* with apex at the camera center extending infinitely in front of the camera. The reason for a cone/pyramid instead of a ray is that the pixels are discrete and have finite resolution. We can visualize it in Figure 4.

To visualize the pyramid, we can first assume a fixed z_c, e.g. 2 meters, and use Equation (1)-(5). We still assume no distortion factors. Please fill in the code in function `getXYZ()` in `object_detection.py`.

3.5 Testing

We can run the script to visualize the pyramid. In `pman` GUI, keeps other commands running, and restart `4-object-detection` command. Move your cursor in the `OpenCV_View` window and observe the pyramid in `rviz`. It should look like Figure 4.

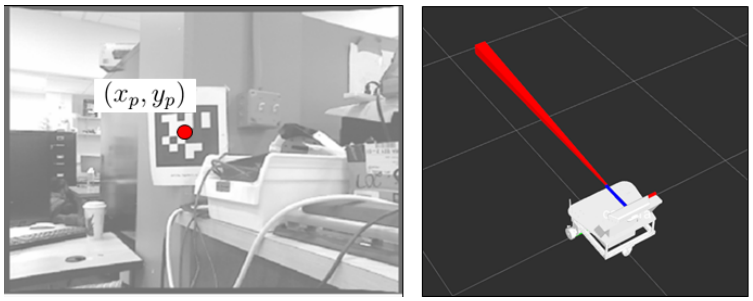

Figure 4: A pixel coordinate in the left image can be a projection of any points in the pyramid on the right image.

4 Task 2: HSV color space: find the metal box

4.1 Introduction

The most distinctive feature for object detection often lies in color. As described in class, HSV color space is a better representation to classify object color because it separate hue and saturation from value (lightness). Hue and saturation are often constant but lightness will change depending on the lighting strength in the room. So the detection criteria should be discriminative in the hue and saturation but invariant to lightness.

4.2 Instruction

In `object_detection.py`, change the `useHSV` to `True` in `main()` function in, and then fill in `HSVObjectDetection()` with the following code:

Step 1: Convert BGR image to HSV image. BGR means blue, green, and red. The BGR order is due to OpenCV convention. The image is 640×480.

```
hsv_image = cv2.cvtColor(cv_image, cv2.COLOR_BGR2HSV)
```

Step 2: Set the bound of HSV values of the target object. They are the upper and lower threshold for detecting the metal box given. See the Hint below to tune the numbers.

```
lower_red = np.array([170,50,50])
upper_red = np.array([180,255,255])
```

Step 3: Find a binary mask of size 640×480. If the HSV value at the pixel is in the range, then the mask at the pixel will be 1. Otherwise, it will be 0.

```
mask = cv2.inRange(hsv_image, lower_red, upper_red)
```

Step 4: Apply erosion to remove small positive region that usually correspond to noisy false positive.

```
mask_eroded = cv2.erode(mask, None, iterations = 3)
```

Step 5: Apply dilation to extend the eroded positive regions so that broken regions that actually belongs to one object will be connected.

```
mask_eroded_dilated = cv2.dilate(mask_eroded, None, iterations = 10)
```

Hint: The object detection will print out the HSV values of the center pixel, which is marked with a blue cross in the window named `image_with_cross` (inside `rviz`).

Question 1 *Change the number of iterations in* `erode` *and* `dilate`*. What will happen to the detection result?*

4.3 Testing

In `pman`, keep `roscore` and the `*.launch` commands running, and restart the `object_detecion` command. You will see 4 windows visualizing the image processing stages in `rviz` as shown in Figure 5. Now we detect the object center in pixel coordinate directly and a pyramid in `rviz` will visualize the object direction relative to the camera.

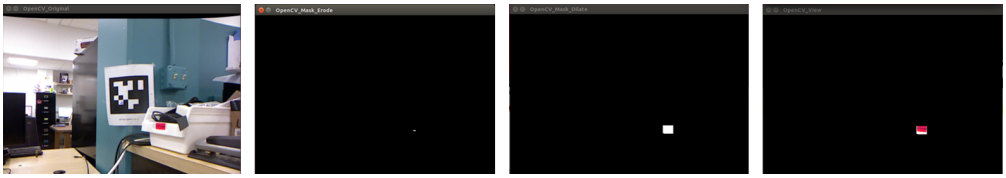

Figure 5: 4 stages of image processing to find red colored object. From left to right, 1) original picture, 2) applying inRange to form a mask and then erosion, 3) applying dilation to the mask, and 4) use the mask to crop the detected region in the original image. In a mask, black means 0 (negative) and white means 1 (positive).

5 Task 3: Depth sensing, estimate the metal box position

5.1 Introduction

Before we always assume a fixed distance from camera in order to visualize the pyramid the object may lies on. In 2010, the Kinect was a breakthrough in robotics because it provided dense depth images, with each pixel specifying the estimated z_c, at a very low cost per sensor. We will use the estimated z_c instead of the fixed value to pin point the 3D position of the object.

Briefly on Kinect depth sensing mechanism: it uses a infrared projector that projects well-designed patterns at its field of view, then uses an infrared camera to perceive the projected patterns. The light is infrared, so it's not visible to human eye or in RGB cameras. The Kinect runs an algorithm onboard to estimate depth based on the distortion of the perceived pattern. This depth estimation mechanism is called *structured light*.

5.2 Instruction

Set `useDepth` to `True` in `main()` function in `object_detection.py`. Now we will use `rosRGBDCallBack` function which takes approximately synchronized RGB and Depth images. We still apply HSV detection on RGB. The change is that we read the depth value z_c from the depth image.

5.3 Testing

Restart the `object_detecion` command in `pman` and observe the pyramid in `rviz`. On the left hand side of `rviz`, there is a `PointCloud2` Display. Enable it by checking it. Then you will see colored pointcloud captured by the Kinect. The pointcloud here is a set of 3D points associated with colors. You can compare your result of your pyramid with the pointcloud. The pyramid is supposed to end at your object in `rviz` normally.

Question 2 *Is the distance measurement of the object always accurate? If not, discuss the reason.*

Question 3 *Find what are the field of views of the color and depth camera. How does the resolution decrease over distance? What is the minimum and maximum distance in which depth sensing works?*

6 Future directions

In this lab, we built a very simple object detection program that uses color to classify object and uses Kinect depth sensing to find its 3D location. We can use the same architecture to build a more sophisticated object detection system. The color threshold can be better handled using machine learning. You can collect a number of representative pictures, and label the region that contains the target image, and use them to train a classifier. See [3] for a Python tutorial of classifying cats and dogs.

To get an idea of the state-of-the-art object detection system, see [4] for Team MIT-Princeton's approach to the Amazon Picking Challenge 2016. Again, the architecture is similar. The main improvements are: 1) exploit deep learning to label object in color image more accurately, and 2) take multiple views to obtain a more complete pointcloud to handle occlusion.

References

[1] M. Fallon, S. Kuindersma, S. Karumanchi, M. Antone, T. Schneider, H. Dai, C. P. D'Arpino, R. Deits, M. DiCicco, D. Fourie *et al.*, "An architecture for online affordance-based perception and whole-body planning," *Journal of Field Robotics*, vol. 32, no. 2, pp. 229–254, 2015.

[2] J. Y. Bouguet. Camera calibration toolbox for matlab. [Online]. Available: http://www.vision.caltech.edu/bouguetj/calib_doc

[3] A. Rosebrock. An intro to linear classification with python. [Online]. Available: http://www.pyimagesearch.com/2016/08/22/an-intro-to-linear-classification-with-python/

[4] A. Zeng, K.-T. Yu, S. Song, D. Suo, E. W. Jr., A. Rodriguez, and J. Xiao. Multi-view self-supervised deep learning for 6d pose estimation in the amazon picking challenge. [Online]. Available: http://www.cs.princeton.edu/~andyz/apc2016