Lab 3: Computer Vision

Learning Outcomes

- read images from ROS
- explore image representations
- blur images
- detect edges
- Hough Line Transform
- Image moments

Section 1: Requirements and Design

In this lab, you will explore a variety of techniques in Computer Vision. These methods can be used in robot systems to track objects, locate items, or detect obstacles.

Section 2: Installing the Lab

To perform this lab, you will need to get the lab3_computer_vision template into your catkin workspace. First ensure that your Jet has internet access by connecting it using WiFi or ethernet. Next ssh into Jet and enter the following command:

`wget http://instructor-url/lab3_computer_vision/lab3_computer_vision-code.zip`

Where the url should be replaced by the URL provided by your instructor. Now unzip the lab:

`unzip lab3_computer_vision-code.zip -d ~/catkin_ws/src/jetlabs/lab3_computer_vision`

Delete the zip file:

`rm lab3_computer_vision-code.zip`

To build the code, use the following command when you are in `~/catkin_ws/`:

`catkin_make --pkg lab3_computer_vision && source devel/setup.sh`

Section 3: Basic OpenCV

This section will use `lab3_computer_vision/src/basic_cv.cpp`.

The usb_cam rosnode is responsible for capturing video frames from the webcam and publishing them to a rostopic called `usb_cam/image_raw`. Any node that needs to access the video frames can subscribe to this topic. The ImageTransport framework is used to publish and subscribe to image topics. The code below shows how you can subscribe to the raw images and publish your own images. Often it is useful when building computer vision applications to visualize how your algorithm is performing. For example, if you are detecting faces, then you can draw circles around each face that is detected then publish this modified image to a new image topic.

```
ros::init(argc, argv, "line_follower");
ros::NodeHandle nh;

image_transport::ImageTransport it(nh);

//advertise the topic with our processed image
user_image_pub = it.advertise("/user/image1", 1);
```

```
//subscribe to the raw usb camera image
raw_image_sub = it.subscribe("/usb_cam/image_raw", 1, imageCallback);
```

Like other types of topics, you subscribe to image topics using a callback. The `imageCallback` function referenced above can be implemented like the code below.

```
void imageCallback(const sensor_msgs::ImageConstPtr& msg)
{
  try
  {

    src = cv_bridge::toCvShare(msg, "bgr8")->image;

    /*
     * INSERT CODE HERE
    */

    sensor_msgs::ImagePtr msg;
    msg = cv_bridge::CvImage(std_msgs::Header(), "bgr8", src).toImageMsg();

    user_image_pub.publish(msg);
  }
  catch (cv_bridge::Exception& e)
  {
    ROS_ERROR("Could not convert from '%s' to 'bgr8'.", msg->encoding.c_str());
  }
}
```

In this case, the `cv_bridge::toCvShare` function is used to convert the image topic message to a matrix that OpenCV can use. These OpenCV matrices are grids where each cell has blue, green, and red value. The code above does not modify the image, but merely republishes the same image using the `user/image1` topic.

You can compile the basic_cv node and run it as is with the launch script:

```
roslaunch lab3_computer_vision lab3_basic.launch
```

Then run image_viewer on your host machine:

```
rosrun image_view image_view image:=/user/image1
```

The image should be the same as the raw camera.

Topic 1: Basic Transformations

Grayscale

For some applications, it is easier to work with images that are grayscale rather than full-color. In OpenCV, the `cvtColor` function is used for this conversion. Add the lines below to the "INSERT CODE HERE" section of basic_cv.cpp. Then replace the variable `src` with `cdst` in the `cv_bridge::CvImage` function call.

```
cvtColor(src, gray, CV_BGR2GRAY);
cvtColor(gray, cdst, CV_GRAY2BGR);
```

This code snippet makes `dst` into a grayscale matrix representation of the original `src` matrix. `dst` has a single 8-bit value for each pixel, while `src` has three 8-bit values for each pixel (blue, green, red). The ROS ImageMessage type requires images to be represented as a matrix of three-tuples, so it is necessary to make a new matrix `cdst` that is a black and white representation using three-tuples (three 8-bit values).

Compile and run the `basic_cv` node and view the `user/image1` topic with image_view to see the grayscale image.

Blurring

Smoothing images is used in computer vision to reduce noise and make it easier to extract features. Blurring an image assigns each pixel to the weighted sum of the nearby pixels. There are a variety of averaging techniques that can be used. We will look at two of them - the normalized box blur and the Gaussian Blur.

In the normalized box blur, a simple numerical average of nearby pixels is used. The kernel size determines how many pixels are averaged for each value. The `blur` function is used to perform a normalized box blur. You can test it out using the following code segment. Replace the `cvtColor` functions from the previous section with the following code:

```
blur(src, cdst, Size(3,3));
```

Compile and run the code to view the results. The kernel is specified in the third argument to `blur`. This value must be an odd number; the larger the value, the more smooth the final image will be.

The Gaussian Blur averages nearby pixel values according to a Gaussian distribution, so the closest pixels are weighted more heavily than the distance pixels. Replace the normalized box blur with the Gaussian Blur function below to see the difference.

```
GaussianBlur(src, cdst, Size(3,3), 0, 0);
```

The `GaussianBlur` function's third argument is the kernel size, just like the normalized box blur. The fourth and fifth parameters are the standard deviations for the x and y directions; leaving these values zero means that they will be computed from the kernel size. Compile and run the `basic_cv` node with different kernel sizes (values must be odd but not necessarily the same).

Topic 2: Edge Detection

Edges in an image are the places where one colored region touches a different colored region. Edge detection is used to understand the structure of an image without selecting specific color values. The Canny Edge Detection algorithm is one of the most popular ways of extracting edges from an image. In OpenCV, this algorithm is implemented in the `Canny` function. Edge detection requires a black and white image. Edge detection can be susceptible to noise, so we will blur the image prior to applying the edge detection algorithm.

Remove the blurring function calls from the previous section. Then add a function call that creates a grayscale matrix called `gray` from the `src` matrix. Next, apply a normalized box blur to the `gray` image and save the resulting blurred image to the `dst` matrix. Now you can add the following code that computes the edges of the image and saves them to the `edges` matrix. Finally, convert the `edges` matrix to a BGR matrix called `cdst`. Compile and run the `basic_cv` node to see white lines representing edges in the original image.

```
Canny(dst, edges, 50, 200, 3);
```

The Canny algorithm has three important parameters (third, fourth, and fifth arguments, respectively): lower threshold, upper threshold, and the aperture size. The threshold parameters define which pixels should be included as edge pixels. The aperture size is used in the filtering of the image. Smaller threshold values cause more pixels to be classified as edges. Try changing the lower threshold to 10 and upper threshold to 100 and rerunning `basic_cv`.

Topic 3: Hough Transforms

Edges can be thought of as image features that are more abstract than raw pixel values because they are independent of the exact coloring. Even more abstract features like shapes can also be extracted from images. The Hough Line Transform is a popular technique for detecting the lines in an image. A similar procedure

can be used for detecting circles and ellipses. In this section, we will use OpenCV's `HoughLinesP` function to visualize the lines in an image.

Add the following code after the `Canny` function call. Compile and run the `basic_cv` node. Hold a straight edge in front of the camera; red lines indicate the lines that were detected by the algorithm.

```
HoughLinesP(edges, lines, 1, CV_PI/180, 80, 30, 10 );
for( size_t i = 0; i < lines.size(); i++ )
{
    line(cdst, Point(lines[i][0], lines[i][1]),
        Point(lines[i][2], lines[i][3]), Scalar(0,0,255), 2, 8 );
}
```

The `HoughLinesP` function saves each line to the lines vector. Each line entry is four-tuple, which has two points (a start and end of the line). Each line entry is composed of (x1, y1, x2, y2). The final three arguments are the important values for fine-tuning the line detection. The fifth value is the minimum threshold (the lower the value, the more lines will be found). The sixth value is the minimum line length, and the seventh value is the minimum gap between lines. Change the minimum line length to 5 to detect many more lines.

Section 4: Tracking

In this section you will use the `vision_tracking` node. You can run this node using roslaunch:

`roslaunch lab3_computer_vision vision_track.launch`

The `vision_tracking` node will identify and track a colored object. Choose an object that is brightly colored and easy to move (a small ball is a good choice).

Topic 1: HSV Segmentation

We have already looked at the BGR and grayscale representations of images, and now we will examine another way of representing images - HSV (Hue, Saturation, and Value). Like BGR, HSV matrices have three numbers. The first number, the Hue, is an 8-bit integer that corresponds to the color of the pixel. The second number, the saturation, is an 8-bit integer that corresponds to the whiteness of the pixel. The third number, the value, corresponds to the lightness of the pixel. Computer vision applications that must deal with colors generally prefer the HSV representation because the color of each pixel is encoded in a single number rather than three in the BGR representation.

Our first step in building a tracking program is to find the pixels that are in the object. To do this, we will perform color-based segmentation. Segmentation is the process of extracting certain components from the image. The code below creates the `HSV` matrix using the HSV representation of the original image. Then two scalar values are instantiated. Finally, the `inRange` function is called. The `inRange` function call sets the `mask` matrix's pixels to 255 when the corresponding `HSV` matrix pixel is within the range of the two thresholds and to 0 when it is outside the range. Add the code below to the `imageCallback` function.

```
//Convert the image to HSV
cv::cvtColor(src, hsv, CV_BGR2HSV);

//Define the range of blue pixels
cv::Scalar lower_thresh(hue_lower, sat_lower, value_lower);
cv::Scalar upper_thresh(hue_upper, sat_upper, value_upper);

//Create a mask with only blue pixels
cv::inRange(hsv, lower_thresh, upper_thresh, mask);

//Conert the image back to BGR
```

```
cv::cvtColor(mask, dst, CV_GRAY2BGR);
```

We left the hue, saturation, and value parameters for the upper and lower thresholds as variables, so we can dynamically change them while the node is running. To try this, launch the `vision_tracking` node:

```
roslaunch lab3_computer_vision vision_track.launch`
```

Then open the image_view on the host computer:

```
rosrun image_view image_view image:=/user/image1
```

Finally, start the dyanimic reconfigure gui on the host computer:

```
rosrun rqt_reconfigure rqt_reconfigure
```

Try updating the values of the hue, value, and saturation in the GUI until you are able to isolate the object. Start by adjusting only the hue values according to the color of your object. The list below provides some suggestions:

- Red: (0, 15), (160, 180)
- Yellow: (20, 45)
- Green: (50, 75)
- Blue: (100, 135)
- Purple: (140, 160)

Notice that the color red has two ranges because it occurrs where the number wrap around; if your object is red, test each range to see which one detects the object best. Keep manipulating the parameters until only the object is visible as white on the screen.

Topic 2: Image Moments and Center of Mass

After computing the pixels that are part of the object, the next step is to find the center of the object. The Image moments will be computed for the `mask` matrix. These moments will then be used to determine the "center of mass" of the object.

Add the following lines after the `inRange` function call. You can add a debug statement like `ROS_INFO("%f %f", center_of_mass.x, center_of_mass.y)` to see coordinates of the center of mass.

```
//Calculate moments of mask
moments = cv::moments(mask, true);

//Calculate center of mass using moments
center_of_mass.x = moments.m10 / moments.m00;
center_of_mass.y = moments.m01 / moments.m00;
```

OpenCV has a special datatype called `Moment` that stores the moments of an image. You can read more about the values stored in a `Moment` on OpenCV's documentation: http://docs.opencv.org/2.4/modules/imgproc/doc/structural_analy

To visualize the center of mass, add a function that draws a red circle where the center of mass is located. The OpenCV function to do this is `circle`. Add the red circle to the `dst` matrix after the initialization of `dst` from `mask`.

Now you can recompile and relaunch the `vision_tracking` node along with the reconfigure GUI and image_view. You should now see a red dot that is located at the center of your object. This system could be used by robots to find the ball in a game of soccer or locate fruit on a tree for harvesting. Try using a different colored object to see if you can locate it as well.