Lab 4: OpenCV Intro

Learning Outcomes

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

Section 1: Installing the Lab

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

```
wget http://instructor-url/lab4_opencv_intro/code.zip
```

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

```
unzip code.zip
```

Move the resulting folders into your catkin workspace:

```
mv lab4 ~/catkin_ws/src/ -r
```

Delete the zip file:

```
rm code.zip
```

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

```
catkin_make && source devel/setup.sh
```

You can execute the nodes with the following commands:

```
rosrun lab4 basic_cv
rosrun lab4 line_follower
```

Section 2: Getting Started with OpenCV

The cv_camera rosnode is responsible for capturing video frames from the webcam and publishing them to a rostopic called `cv_camera/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("/cv_camera/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 operate 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. Then open the dashboard and select `user/image1` as the video source. The image should be the same as the raw camera. In the subsequent sections, we will look at various OpenCV techniques for manipulating images.

Section 3: 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` source in the dashboard to see a gray-scale 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).

Section 4: 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`.

Section 5: 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 8: Line Following

In this section you will use the line_follower node. After completing this section, the Jetbot will be able to follow a line on the ground. To prepare

This portion requires some setup and configuration. First, the camera on the robot should be rotated to face towards the ground. Next, tape will be used to construct a course that the robot must navigate. The tape pieces should be straight and the corners of the course should be less than 30 degrees. Choose a tape color that is different from the ground color (blue masking tape is a good choice).

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 line following program is to find the pixels that are in the line. 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.

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

//Define the range of blue pixels
cv::Scalar lower_thresh(H1, S1, V1);
cv::Scalar upper_thresh(H2, S2, V2);

//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);
```

If your tape is a different color from the ground then you can locate an appropriate `H1` and `H2` values from the list below and set the Saturation range to (10, 255) and the Value range to (10, 255). Notice that the color red has two ranges because it occurrs where the number wrap around; if your tape is red, test each range to see which one detects the tape best.

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

If your tape is white, black or gray, set the Hue range to (0, 180). If the tape is white set the Saturation range to (200, 255) and the Value range to (200, 255). If the tape is black or gray, choose a low range for both the Saturation range and the Value range.

Compile and run the `line_follower` node to view the thresholding results. The line should appear white against a black background. If the line does not appear or it is not a crisp band of white, then try adjusting the threshold values.

Image Moments and Center of Mass

After computing the pixels that are part of the tape, the next step is to find the center of the tape. The Image moments will be computed for the `mask` matrix. These moments will then be used to determine the "center of mass" of the tape. The horizontal component of the center of mass should be in the center of the image if the robot is directly on top of the tape. The velocity of the robot will need to be adjusted when the center of mass is not aligned in the center of the image.

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`.

Changing the Velocity

The angular velocity of the robot should be proportional to the distance of the center of mass from the center of the image. Set the angular.z velocity component to a value between -1.0 and 1.0 that is proportional to the difference between the x coordinate of the center of mass and the x coordinate of the center of the image. *Hint: src.cols is the number of columns in the image.* After setting the angular velocity, publish the velocity message.

Place the robot on a block or box so that the wheels do not touch the ground. Run the `line_follower` node to see if the wheels spin appropriately. Now modify the linear.x component of the velocity in the `main` function, so the robot is always moving forward. Recompile and launch the node again to test the solution. You may need to adjust the linear component or scale the angular component. It may be helpful to divide the angular velocity by a constant (like 3.0) if the robot turns to sharply.