DUCKIETOWN

💻🚙 05 - Visual control

The view from a Duckiebot when centered in its lane.

Now, we use the image processing tools that we have explored to design a policy for lane-following. Consider the above image taken from the Duckiebot's camera when the Duckiebot was centered in its lane. Our goal is to design a control policy that uses only images streamed from the Duckiebot's camera to keep it in the lane as it drives forward at a fixed velocity.

Rather than design a controller that reasons over the raw image, we will process the image to enhance aspects of the image that are helpful for lane-following. In particular, the dashed-yellow and solid-white lane markings provide valuable cues that we can use to guide the robot to stay in its lane.

Assume that the Duckiebot starts out in the center of its lane, facing in the direction of travel. Assume also that the Duckiebot is traveling forward at a nominal speed. We can imagine using the location of the dashed-yellow and solid-white lane markings in the image to vary the rotational velocities of the left and right wheels in order to stay in the lane. For example, if the dashed-yellow lane markings appear in the left- and right-halves of the image, we might command a negative rotational velocity (i.e., counter-clockwise) as a correction.

Of course, this requires that we can detect the dashed-yellow and solid-white lane markings and estimate their orientations. Thus, we want to apply a filter that makes the lane markings "pop out" in the image, while supressing the other distracting content of the image. Lane markings are brightly colored against a dark background (the road).

We can identify the location of lane markings in the image by applying finite-difference filters h_x and h_y (e.g., using the Sobel operator) to estimate the horizontal and vertical intensity gradients

$$\nabla I[u,v] = \begin{bmatrix} G_x[u,v] \\ G_y[u,v] \end{bmatrix}$$

where

$$G_x = h_x * I \tag{1}$$
$$G_y = h_y * I \tag{2}$$

We can then look for regions of the image where the magnitude of the intensity gradient is large as a measure of the strength of each edge.

$$|\nabla I[u,v]| = \sqrt{\left(G_x[u,v]\right)^2 + \left(G_y[u,v]\right)^2}$$

Assuming that these edges correspond to the lane boundaries, the direction of the gradients can then be used to estimate the desired orientations.

$$\theta[u,v] = \operatorname{atan2}\left(G_y[u,v], G_x[u,v]\right)$$

In this exercise, we will develop this idea by investigating different ways of processing the images to extract information about the robot's pose relative to the lane. We will use these results to implement a reactive image-space controller that we will validate on a simulated and/or real Duckiebot.

In [ ]:
```
## Run this cell to import relevant modules
%load_ext autoreload
%autoreload 2
%matplotlib inline
%pylab inline

from matplotlib import pyplot as plt
import numpy as np
import cv2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
%pylab is deprecated, use %matplotlib inline and import the required librari
es.
Populating the interactive namespace from numpy and matplotlib
```

Let's load the image as well as the homography for the corresponding camera into Python.

In [ ]:
```
# Load in the image and generate hsv and grayscale versions
imgbgr = cv2.imread('../../assets/images/visual_control/pic1_rect.png')

# OpenCV uses BGR by default, whereas matplotlib uses RGB, so we generate an
imgrgb = cv2.cvtColor(imgbgr, cv2.COLOR_BGR2RGB)

# Convert the image to HSV for any color-based filtering
imghsv = cv2.cvtColor(imgbgr, cv2.COLOR_BGR2HSV)

# Most of our operations will be performed on the grayscale version
img = cv2.cvtColor(imgbgr, cv2.COLOR_BGR2GRAY)

# The image-to-ground homography associated with this image
H = np.array([-4.137917960301845e-05, -0.0001144585419146805, -0.1595567007
              0.0008382870319844166, -4.141689222457687e-05, -0.251820163817
              -0.00023561657746150284, -0.005370140574116084, 0.999999999999

H = np.reshape(H,(3, 3))
Hinv = np.linalg.inv(H)

print(Hinv)
```

```
[[-1.60814783e+03  1.12073254e+03  2.56322894e+01]
 [-8.98794107e+02 -9.11232359e+01 -1.66355291e+02]
 [-5.20555699e+00 -2.25281421e-01  1.12688097e-01]]
```

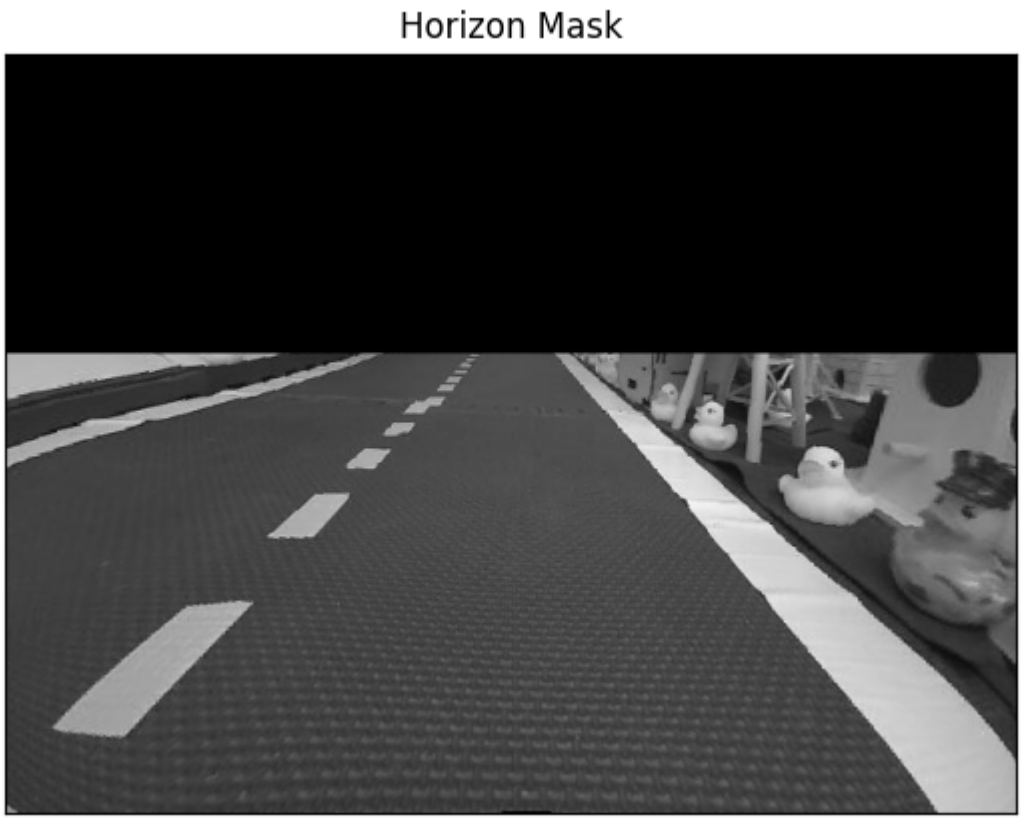
Finding the Horizon

In Duckietown, the ground is planar and in other self-driving and robotics domains, it is often assumed that the ground is locally planar. We can exploit this as we search for lane markings by not searching above the horizon. In the world frame relative to which we estimated the homography (i.e., a reference frame with the origin centered between the drive wheels with the positive x-axis pointing forward and the positive y-axis to the left), the horizon corresponds to large x-coordinates. In Duckietown, we are only interested in the road a few meters in front of the Duckiebot.

Note: You do not need to include this in the implementation of the functions that will be run in simulation and on the physical Duckiebot. For this activity, the robot will automatically crop out the horizon.

In [ ]:
```
# TODO: Use the homography (technically Hinv) to mask out the horizon
# When generating masks that will be used with OpenCV try and

# IDEA: use a point in the world with Y=0 (horizon) and translate into image
# Choose a value for X (distance in front of the robot to look for lane mark
X = 2.0  # You can adjust this value based on your preferences

# Project the point (X, 0) in the world frame to the image frame using Hinv
point_in_world = np.array([X, 0, 1])
point_in_image = Hinv @ point_in_world # @ is matrix multiply
```

```
# Normalize the coordinates to get homogeneous coordinates
point_in_image /= point_in_image[2]
horizon_y = int(point_in_image[1])

# Create a mask based on the projected coordinates to identify the region be
mask_ground = np.ones(img.shape)
mask_ground[:horizon_y,:] = 0

# Convert the boolean mask to a matrix of 1s and 0s
mask_ground = mask_ground.astype(np.uint8)

fig = plt.figure(figsize = (30,20))
ax1 = fig.add_subplot(1,4,1)
ax1.imshow(img*mask_ground,cmap = 'gray')
ax1.set_title('Horizon Mask'), ax1.set_xticks([]), ax1.set_yticks([]);
```

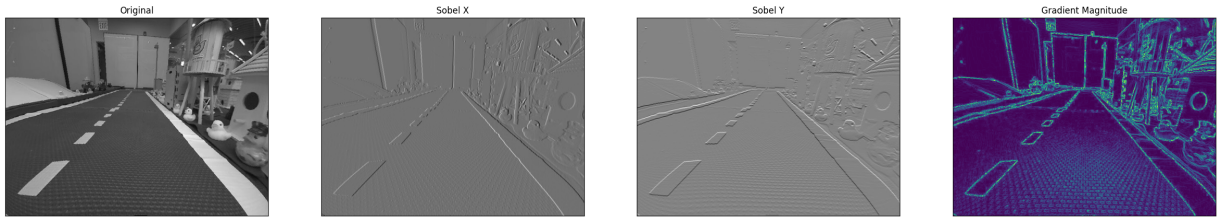


Detecting Lane Markings using Sobel Edge Detection

Now, let's apply the Sobel operator to the image to identify image gradients.

In [ ]:
```
# Convolve the image with the Sobel operator (filter) to compute the numeric
sobelx = cv2.Sobel(img,cv2.CV_64F,1,0)
sobely = cv2.Sobel(img,cv2.CV_64F,0,1)

# Compute the magnitude of the gradients
Gmag = np.sqrt(sobelx*sobelx + sobely*sobely)

# Compute the orientation of the gradients
Gdir = cv2.phase(np.array(sobelx, np.float32), np.array(sobely, dtype=np.flo
```

```
fig = plt.figure(figsize = (30,20))
ax1 = fig.add_subplot(1,4,1)
ax1.imshow(img,cmap = 'gray')
ax1.set_title('Original'), ax1.set_xticks([]), ax1.set_yticks([])
ax2 = fig.add_subplot(1,4,2)
ax2.imshow(sobelx,cmap = 'gray')
ax2.set_title('Sobel X'), ax2.set_xticks([]), ax2.set_yticks([])
ax3 = fig.add_subplot(1,4,3)
ax3.imshow(sobely,cmap = 'gray')
ax3.set_title('Sobel Y'), ax3.set_xticks([]), ax3.set_yticks([]);
ax4 = fig.add_subplot(1,4,4)
ax4.imshow(np.uint8(Gmag))
ax4.set_title('Gradient Magnitude'), ax4.set_xticks([]), ax4.set_yticks([]);
```



Filtering the image has enhanced the location of the lane markings, but has also emphasized other locations where there are rapid changes in intensity. These include the duckies, buildings in Duckietown, the road surface, and elements in the background (e.g., the lights in the upper-right corner of the image). Note that we are using a non-grayscale colormap for the gradient magnitudes in order to more clearly convey locations that are enhanced by the Sobel operator.

We can reduce the occurrence of some of these features by first blurring the filter using a Gaussian kernel.

Gaussian Blurring

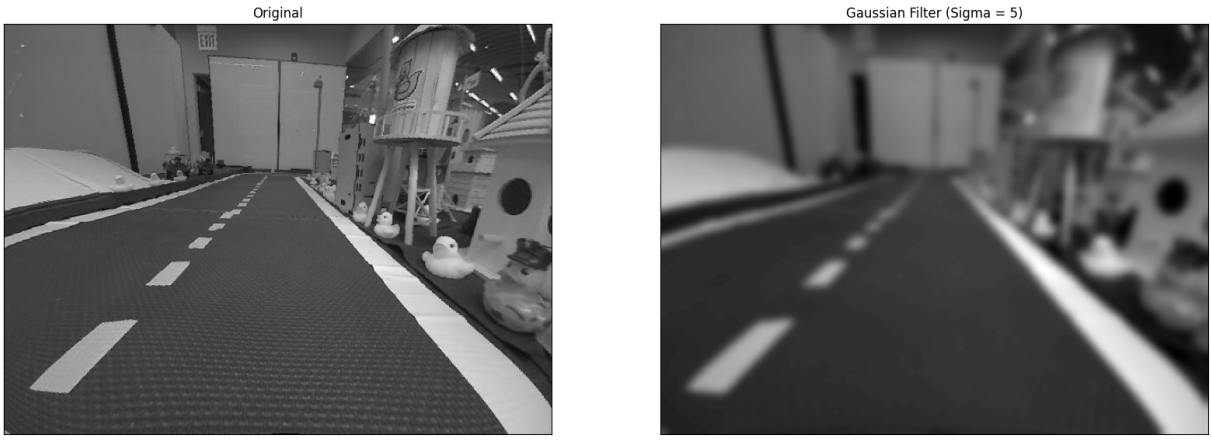
The Sobel operator picks up on noise in the image as well as texture in the scene that we don't particularly care about. We can blur the image with a Gaussian kernel to reduce these spurious detections.

Example: Gaussian Blurring

Using your experience from the previous exercise, identify a setting for the standard deviation of the Gaussian kernel that removes noise and local texture (e.g., that of the road surface), without sacrificing too much valid content (namely, the edges associated with lane markings)

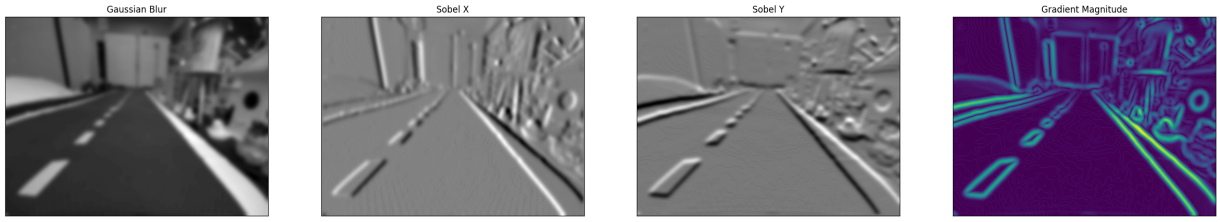
In [ ]:
```
# TODO: Identify a setting for the standard deviation that removes noise whi
sigma = 5 # CHANGE ME

# Smooth the image using a Gaussian kernel
```

```
img_gaussian_filter = cv2.GaussianBlur(img,(0,0), sigma)

# Visualize the filtered image alongside the original image.
fig = plt.figure(figsize = (20,20))
ax1 = fig.add_subplot(1,2,1)
ax1.imshow(img,cmap = 'gray')
ax1.set_title('Original'), ax1.set_xticks([]), ax1.set_yticks([])
ax2 = fig.add_subplot(1,2,2)
ax2.imshow(img_gaussian_filter,cmap = 'gray')
ax2.set_title('Gaussian Filter (Sigma = ' + str(sigma) +')'), ax2.set_xticks
```



Now, let's look at the gradients of the blurred image. For this exercise, experiment with different settings for the standard deviation above and see how they affect the image gradients. In particular, compare the gradients for two extremes of the standard deviation (e.g., $\sigma = 1$ and $\sigma = 10$).

In [ ]:
```
# Convolve the image with the Sobel operator (filter) to compute the numeric
sobelx = cv2.Sobel(img_gaussian_filter,cv2.CV_64F,1,0)
sobely = cv2.Sobel(img_gaussian_filter,cv2.CV_64F,0,1)

# Compute the magnitude of the gradients
Gmag = np.sqrt(sobelx*sobelx + sobely*sobely)

# Compute the orientation of the gradients
Gdir = cv2.phase(np.array(sobelx, np.float32), np.array(sobely, dtype=np.flo

fig = plt.figure(figsize = (30,20))
ax1 = fig.add_subplot(1,4,1)
ax1.imshow(img_gaussian_filter,cmap = 'gray')
ax1.set_title('Gaussian Blur'), ax1.set_xticks([]), ax1.set_yticks([])
ax2 = fig.add_subplot(1,4,2)
ax2.imshow(sobelx,cmap = 'gray')
ax2.set_title('Sobel X'), ax2.set_xticks([]), ax2.set_yticks([])
ax3 = fig.add_subplot(1,4,3)
ax3.imshow(sobely,cmap = 'gray')
ax3.set_title('Sobel Y'), ax3.set_xticks([]), ax3.set_yticks([]);
ax4 = fig.add_subplot(1,4,4)
ax4.imshow(np.uint8(Gmag))
ax4.set_title('Gradient Magnitude'), ax4.set_xticks([]), ax4.set_yticks([]);
```



A Varient of Non-maximal Supression

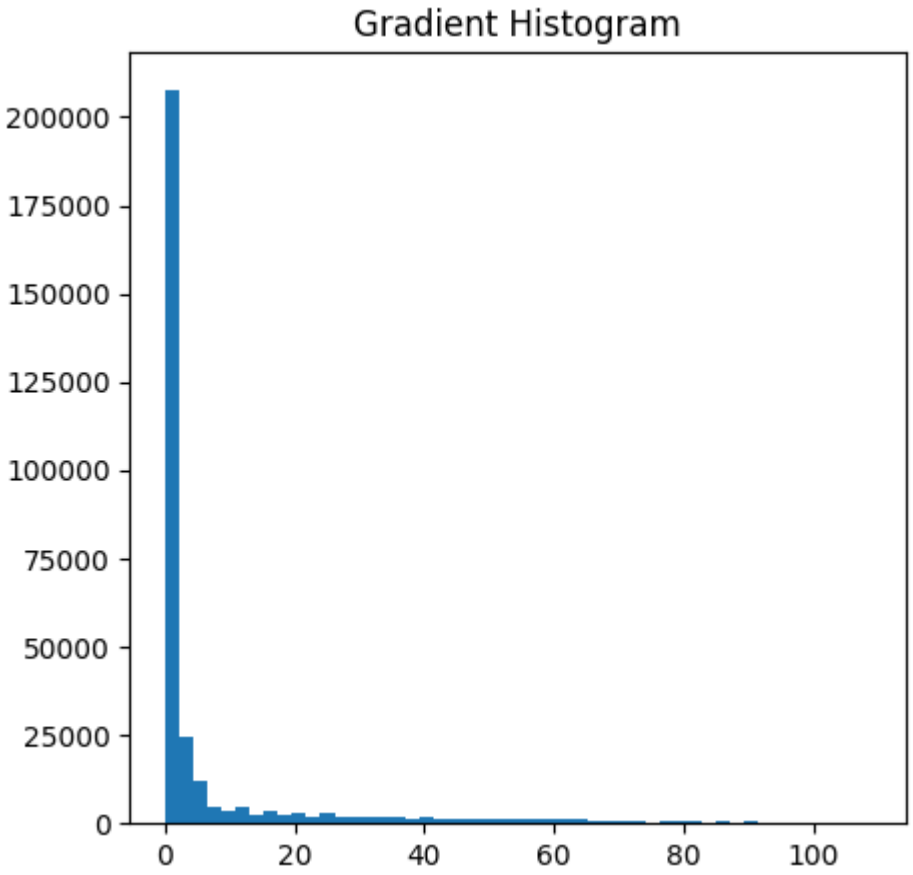
With a suitable choice for the standard deviation, we have smoothed out some of the noise and texture in the image (e.g., the road), reducing the number of candidate edges while preserving most of the edges that corresopnd to lane markings.

Under the assumption that the edges associated with lane markings are stronger than most of the edges in the image (in terms of the magnitude of the gradients), we can further eliminate distractors by only keeping edges whose gradient magnitude is greater than a threshold.

Example: A form of non-maximal supression

In this exercise, we will take a look at the histogram over gradient magnitudes to better understand the distribution of gradient magnitudes. Based on the histogram, we can choose a magnitude threshold that we will use to filter out weaker edges.

In [ ]:
```
# Visualize the histogram over gradient magnitudes
fig = plt.figure(figsize = (5,5))
ax1 = fig.add_subplot(1,1,1)
ax1.hist((Gmag*mask_ground).flatten(), bins=50)
ax1.set_title('Gradient Histogram');
```



In [ ]:
```
# TODO: Use the histogram above to choose the minimum threshold on the gradi
#       Edges whos gradient magnitude is below this threshold will be filter
threshold = 50 # CHANGE ME

mask_mag = (Gmag > threshold)

fig = plt.figure(figsize = (20,10))
ax1 = fig.add_subplot(1,2,1)
ax1.imshow(Gmag, cmap = 'gray')
ax1.set_title('Gradient Magnitude'), ax1.set_xticks([]), ax1.set_yticks([])
ax2 = fig.add_subplot(1,2,2)
ax2.imshow(mask_mag*Gmag,cmap = 'gray')
ax2.set_title('Gradient Magnitude (Thresholded)'), ax2.set_xticks([]), ax2.s
```

Color-based Masking

Having identified a candidate set of edges, we can now try to design a set of masks that isolate the edges associated with the dashed-yellow and solid-white lane markings.

Perhaps the most obvious thing to do is to create masks that filter out pixels whose color differs from that of the yellow and white lines.

Example: Color-based Masking

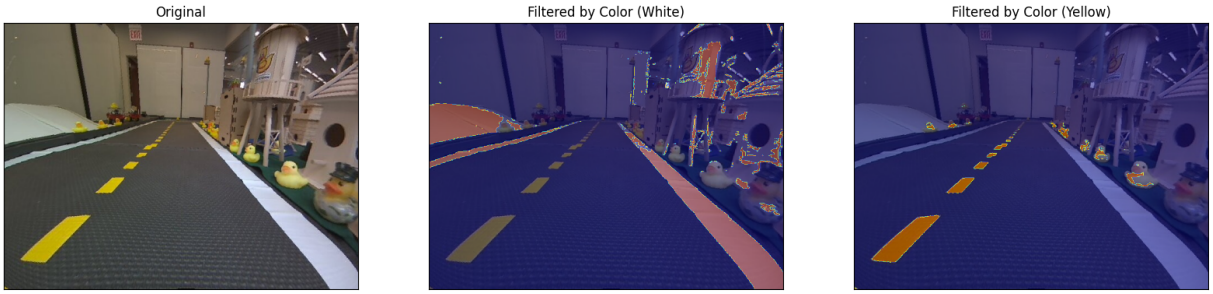
We can select upper- and lower-bounds on the HSV values for the two lines. One option is to use this online color picker, which allows us to get the HSV values for particular pixels.

Note: When we go to use these bounds, remember that OpenCV uses uses the convention that $\mathrm{H} \in [0, 179]$, $\mathrm{S} \in [0, 255]$, and $\mathrm{V} \in [0, 255]$, while other tools may use different ranges (e.g., Gimp uses $\mathrm{H} \in [0, 360]$, $\mathrm{S} \in [0, 100]$, and $\mathrm{V} \in [0, 100]$). Regardless of the color picker you use, always double-check whether it uses a different convention for HSV ranges. If it does, make sure to convert the result to the OpenCV's convention.

Note: While the HSV color space provides a better representation (e.g., compared to RGB) for color-based detection, the appearance of the yellow and white lane markings will change due to variations in illumination, shading, etc. for In order to improve the generalizability of these bounds, you are encouraged to also consider the `../../assets/images/visual_control/pic3_rect.png` image and define bounds that are appropriate for both.

In [ ]:
```
# Using the above tool, we can identify the bounds as follows
# TODO: Identify the lower and upper HSV bounds for the white and yellow lan
#       These values represent the maximum range, so they don't filter out a
white_lower_hsv = np.array([0, 0, 135])      # CHANGE ME
white_upper_hsv = np.array([125, 42, 255])   # CHANGE ME
yellow_lower_hsv = np.array([22, 114, 165])  # CHANGE ME
yellow_upper_hsv = np.array([29, 255, 236])  # CHANGE ME

mask_white = cv2.inRange(imghsv, white_lower_hsv, white_upper_hsv)
mask_yellow = cv2.inRange(imghsv, yellow_lower_hsv, yellow_upper_hsv)

fig = plt.figure(figsize = (20,10))
ax1 = fig.add_subplot(1,3,1)
ax1.imshow(imgrgb)
ax1.set_title('Original'), ax1.set_xticks([]), ax1.set_yticks([])
ax2 = fig.add_subplot(1,3,2)
ax2.imshow(imgrgb)
ax2.imshow(mask_white, cmap='jet', alpha=0.5)
ax2.set_title('Filtered by Color (White)'), ax2.set_xticks([]), ax2.set_ytic
ax3 = fig.add_subplot(1,3,3)
```

```
ax3.imshow(imgrgb)
ax3.imshow(mask_yellow, cmap='jet', alpha=0.5)
ax3.set_title('Filtered by Color (Yellow)'), ax3.set_xticks([]), ax3.set_yti
```



Edge-based Masking

Now, we will explore possible ways to mask out unwanted edges based upon their orientation in the image.

Assuming that the Duckiebot hasn't deviated too much from its lane, most of the dashed-yellow line should lie in the left-half of the image, while most of the solid-white line should lie in the right-half of the image. We can use this to reason separately over dashed-yellow and solid-white lane markings.

Note: While this masking may seem reasonable when the Duckiebot is oriented close to the direction of travel, it may not be what we want when the Duckiebot's orientation is off. In these cases, we would expect some of the dashed-yellow lane markings to be in the right-half of the image or some of the solid-white lane markings to be in the left-half of the image. These would be valuable cues to use. Thus, as you experiment with the included images as well as in simulation and on your physical Duckiebot, you may want to try disabling these masks.

Example: Edge-based Masking

In [ ]:
```
# Let's create masks for the left- and right-halves of the image
width = img.shape[1]
mask_left = np.ones(sobelx.shape)
mask_left[:,int(np.floor(width/2)):width + 1] = 0
mask_right = np.ones(sobelx.shape)
mask_right[:,0:int(np.floor(width/2))] = 0
```

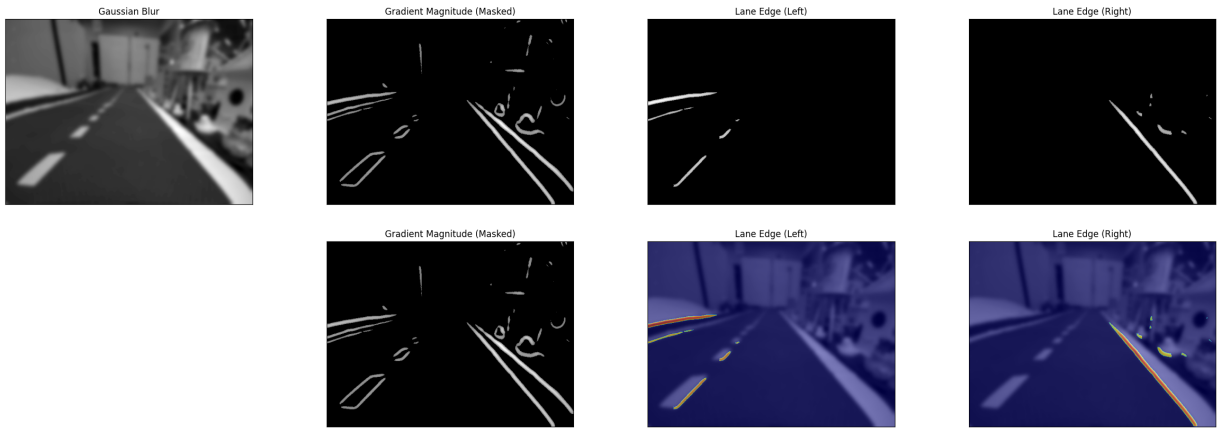
The inner edge of the dashed-yellow line corresponds to a negative gradient in the x- and y-directions, while the inner-edge of the solid-white line corresponds to a positive gradient in the x-direction and negative gradient in the y-direction. We can use this to further mask out unwanted edges.

In [ ]:
```
# In the left-half image, we are interested in the right-half of the dashed
# In the right-half image, we are interested in the left-half of the solid w
# Generate a mask that identifies pixels based on the sign of their x-deriva
```

```
mask_sobelx_pos = (sobelx > 0)
mask_sobelx_neg = (sobelx < 0)
mask_sobely_pos = (sobely > 0)
mask_sobely_neg = (sobely < 0)
```

Now, lets combine these masks to see their effect on the edge detections, but let's look at the effects of the gradient-based masks before including those based on color.

In [ ]:
```
# Let's combine these masks with the gradient magnitude mask
mask_left_edge = mask_ground * mask_left * mask_mag * mask_sobelx_neg * mask
mask_right_edge = mask_ground * mask_right * mask_mag * mask_sobelx_pos * ma

fig = plt.figure(figsize = (30,10))
ax1 = fig.add_subplot(2,4,1)
ax1.imshow(img_gaussian_filter,cmap = 'gray')
ax1.set_title('Gaussian Blur'), ax1.set_xticks([]), ax1.set_yticks([])
ax2 = fig.add_subplot(2,4,2)
ax2.imshow(mask_mag*Gmag,cmap = 'gray')
ax2.set_title('Gradient Magnitude (Masked)'), ax2.set_xticks([]), ax2.set_yt
ax3 = fig.add_subplot(2,4,3)
ax3.imshow(Gmag * mask_left_edge, cmap = 'gray')
ax3.set_title('Lane Edge (Left)'), ax3.set_xticks([]), ax3.set_yticks([])
ax4 = fig.add_subplot(2,4,4)
ax4.imshow(Gmag * mask_right_edge,cmap = 'gray')
ax4.set_title('Lane Edge (Right)'), ax4.set_xticks([]), ax4.set_yticks([]);

ax6 = fig.add_subplot(2,4,6)
ax6.imshow(mask_mag*Gmag,cmap = 'gray')
ax6.set_title('Gradient Magnitude (Masked)'), ax6.set_xticks([]), ax6.set_yt
ax7 = fig.add_subplot(2,4,7)
ax7.imshow(img_gaussian_filter,cmap = 'gray')
ax7.imshow(Gmag * mask_left_edge, cmap='jet', alpha=0.5)
ax7.set_title('Lane Edge (Left)'), ax7.set_xticks([]), ax7.set_yticks([])
ax8 = fig.add_subplot(2,4,8)
ax8.imshow(img_gaussian_filter,cmap = 'gray')
ax8.imshow(Gmag * mask_right_edge, cmap='jet', alpha=0.5)
ax8.set_title('Lane Edge (Right)'), ax8.set_xticks([]), ax8.set_yticks([]);
```



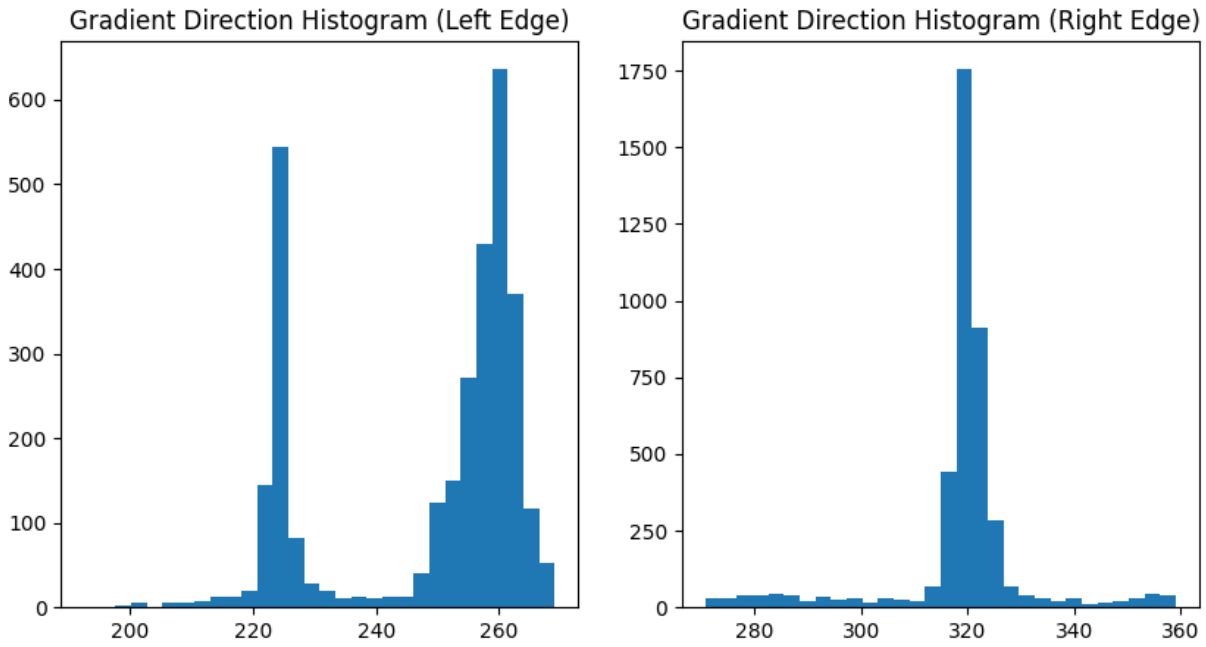
We see that the inclusion of (a simple version of) non-maximal supression together with masking based on the image gradients has helped to filter out many of the distractor edges. However, the "Lane Edge (Left)" gradients include edges that don't correspond

to lane markings (i.e., the bright-to-dark transition at the edge of the road). The "Lane Edge (Right)" gradients better isolate the edge caused by the solid-white line.

Example: Dominant Gradient Orientations

Now, we want to estimate the gradient orientations for the dashed-yellow and solid-white lane markings. To that end, let's look at the gradient histogram for the left edge and right edge images.

In [ ]:
```
# Now, let's apply the mask to our gradient directions
fig = plt.figure(figsize = (10,5))
ax1 = fig.add_subplot(1,2,1)
ax1.hist(np.extract(mask_left_edge, Gdir).flatten(), bins=30)
ax1.set_title('Gradient Direction Histogram (Left Edge)')
ax2 = fig.add_subplot(1,2,2)
ax2.hist(np.extract(mask_right_edge, Gdir).flatten(), bins=30)
ax2.set_title('Gradient Direction Histogram (Right Edge)');
```

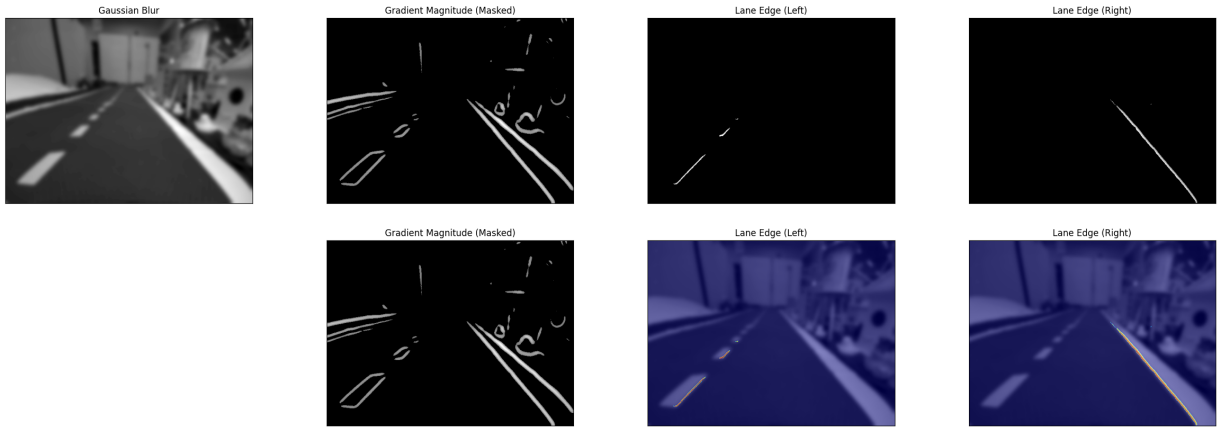


Looking at the gradient histogram for the "Right Edge", we see that the large majority of the edges have an orientation around 315 degrees, while there are few outliers with orientations between 0 and 100 degrees and between 250 and 350 degrees. This is consistent with the gradients that we visualize above in the "Lane Edge (Right)" images above. The ~315 degree orientation is consistent with the dominant edge associated with the solid-white lane marking (i.e., the gradient points to the upper-right).

Now, let's incorporate the color-based masks.

In [ ]:
```
# Let's generate the complete set of masks, including those based on color
mask_left_edge = mask_ground * mask_left * mask_mag * mask_sobelx_neg * mask
mask_right_edge = mask_ground * mask_right * mask_mag * mask_sobelx_pos * ma
```

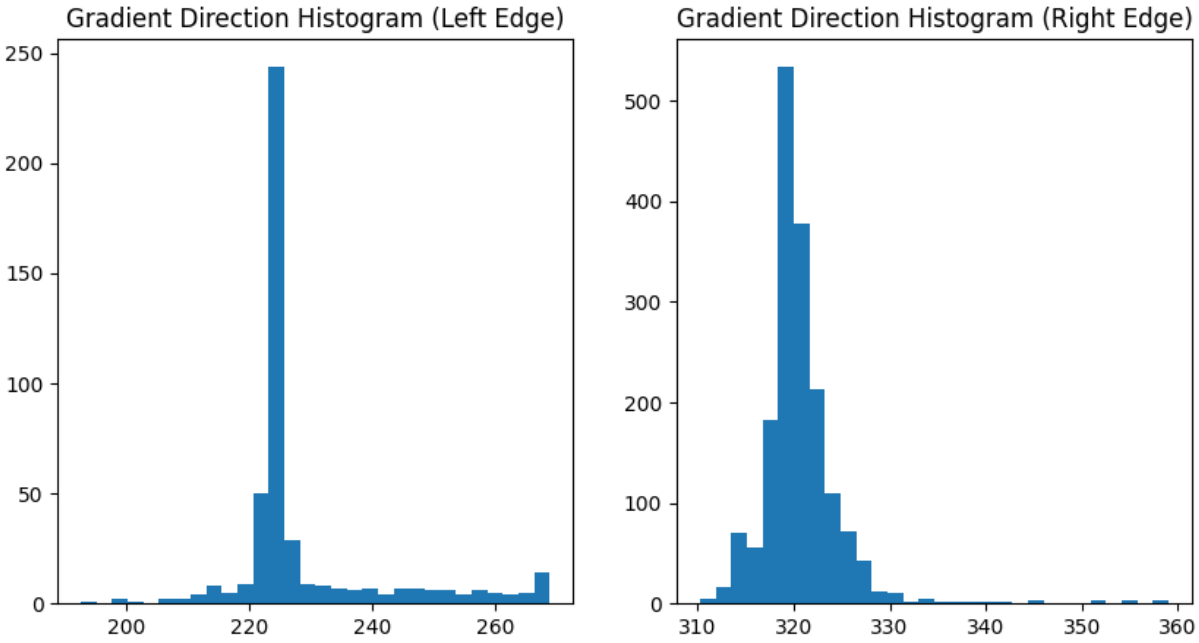
```
fig = plt.figure(figsize = (30,10))
ax1 = fig.add_subplot(2,4,1)
ax1.imshow(img_gaussian_filter,cmap = 'gray')
ax1.set_title('Gaussian Blur'), ax1.set_xticks([]), ax1.set_yticks([])
ax2 = fig.add_subplot(2,4,2)
ax2.imshow(mask_mag*Gmag,cmap = 'gray')
ax2.set_title('Gradient Magnitude (Masked)'), ax2.set_xticks([]), ax2.set_yt
ax3 = fig.add_subplot(2,4,3)
ax3.imshow(Gmag * mask_left_edge, cmap = 'gray')
ax3.set_title('Lane Edge (Left)'), ax3.set_xticks([]), ax3.set_yticks([])
ax4 = fig.add_subplot(2,4,4)
ax4.imshow(Gmag * mask_right_edge,cmap = 'gray')
ax4.set_title('Lane Edge (Right)'), ax4.set_xticks([]), ax4.set_yticks([]);

ax6 = fig.add_subplot(2,4,6)
ax6.imshow(mask_mag*Gmag,cmap = 'gray')
ax6.set_title('Gradient Magnitude (Masked)'), ax6.set_xticks([]), ax6.set_yt
ax7 = fig.add_subplot(2,4,7)
ax7.imshow(img_gaussian_filter,cmap = 'gray')
ax7.imshow(Gmag * mask_left_edge, cmap='jet', alpha=0.5)
ax7.set_title('Lane Edge (Left)'), ax7.set_xticks([]), ax7.set_yticks([])
ax8 = fig.add_subplot(2,4,8)
ax8.imshow(img_gaussian_filter,cmap = 'gray')
ax8.imshow(Gmag * mask_right_edge, cmap='jet', alpha=0.5)
ax8.set_title('Lane Edge (Right)'), ax8.set_xticks([]), ax8.set_yticks([]);
```



We can see that incorporating the color-based masks has removed most if not all of the outliers, but has also removed some of the valid edges, particularly those associated with the solid-white lane markings.

Let's look at the corresponding historgrams.

In [ ]:
```
# Now, let's apply the full set of masks to our gradient directions
fig = plt.figure(figsize = (10,5))
ax1 = fig.add_subplot(1,2,1)
ax1.hist(np.extract(mask_left_edge, Gdir).flatten(), bins=30)
ax1.set_title('Gradient Direction Histogram (Left Edge)')
ax2 = fig.add_subplot(1,2,2)
ax2.hist(np.extract(mask_right_edge, Gdir).flatten(), bins=30)
ax2.set_title('Gradient Direction Histogram (Right Edge)');
```



Consistent with what we saw above, incorporating the color-based masks has effectively removed the large number of edges that lead to a dominant mode around 260 degrees in the previous histogram for the left edge. Now, there is a single dominant mode for each of the edges.

Now, you might be wondering why we don't use these angles for control. Afterall, you might expect that we could use any deviations in the gradient orientations associated with the left and right lane markings in order to steer the vehicle back into the lane. However, the image-space orientations don't change significantly as the vehicle's orientation relative to the lane changes, implying that there isn't as much useful signal here as we might expect

💻🚙 Write the lane-following function

Now that we have an understanding of how we might go about detecting the left (dashed-yellow) and right (solid-white) lane markings, we can integrate these components into a reactive controller that seeks to keep the Duckiebot in its lane.

In particular, if the Duckiebot is commanded to travel forward at a fixed velocity, we can imagine controlling the steering (angular rate) as follows:

```
steering  = np.sum( STEER_LEFT_LM * left_lane_markings_img) +
np.sum( STEER_RIGHT_LM * right_lane_markings_img)
```

where `STEER_LEFT_LM` and `STEER_RIGHT_LM` are weight matrices that map the image-space locations of the detected left (dashed-yellow) lane markings to the steering (angular rate) commands.

In doing so we ask you to define two sets of functions:

1. A set of two functions that define these weight matrices;
2. A function that takes as input the image from the Duckiebot's camera and outputs two images, one corresponding to the image-space detections of the left (dashed-yellow) and right (solid-white) lane markings.

These functions will then be combined to control the Duckiebot.

Define the left and right lane marking matrices

We will control the steering according to a weighted combination of the left and right lane markings. Implement the functions `get_steer_matrix_left_lane_markings()` and `get_steer_matrix_right_lane_markings()` inside the file visual_servoing_activity.py. These functions create weight matrices for the left and right lane markings respectively.

Detecting the left and right lane markings

Based on what we have learned above, implement the function `detect_lane_markings()` inside the file visual_servoing_activity.py that takes as input the image from the Duckiebot's camera (in the BGR color space) and outputs two images, one corresponding to the filtered left (dashed-yellow) lane markings and the other for the right (solid-white) lane markings. These images can be in the form of the binary masks that we developed above.

The two lane marking images will then be used together with the weight matrices defined above to control the Duckiebot.

Test the `detect_lane_markings()` function

As we have seen, unit tests are valuable in confirming that a particular piece of code works as intended when provided with an expected input.

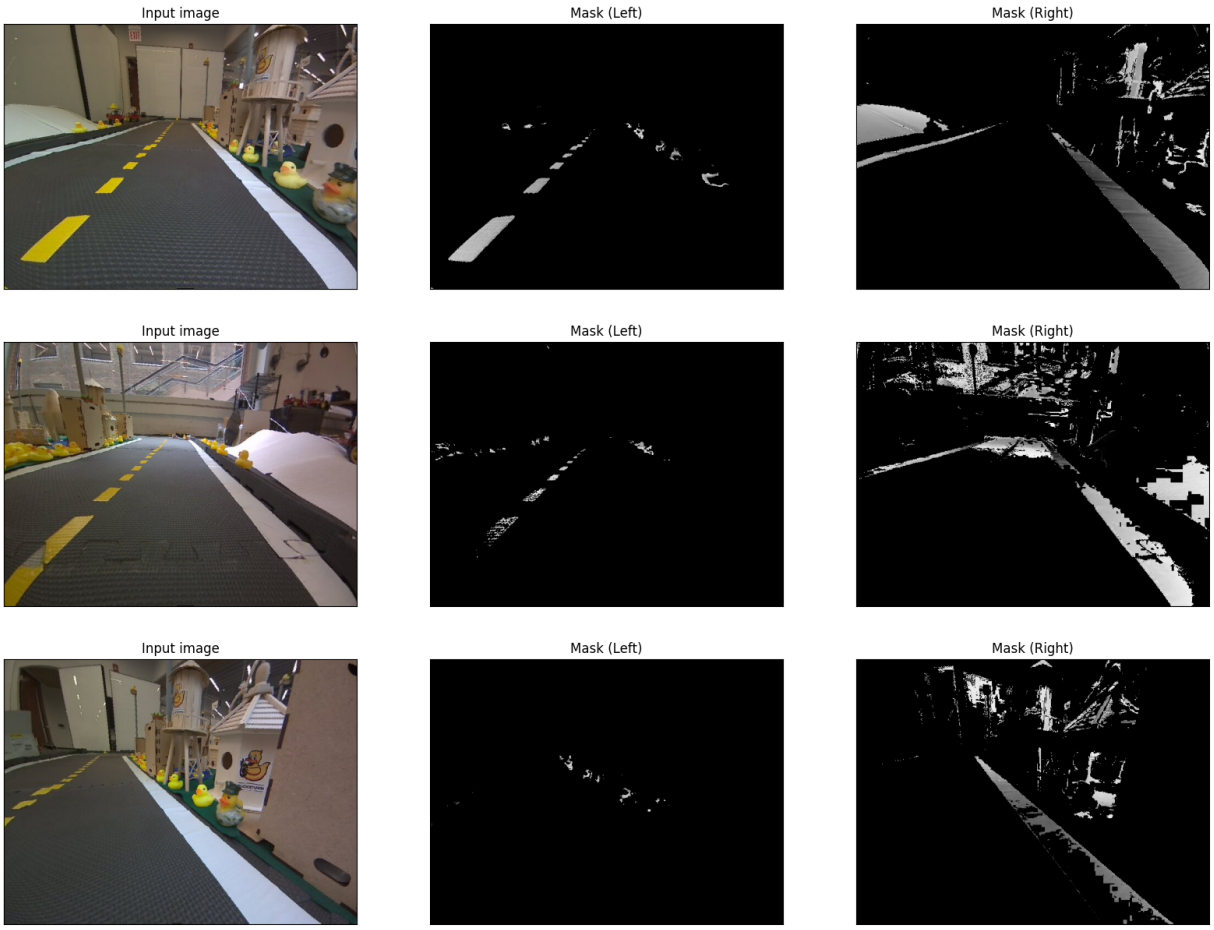
Let's see if the function you wrote above passes the following test!

Note: you might need to reload the kernel for the notebook to detect the changes you make.

In [ ]:
```
from tests import UnitTestDLM
import solution.visual_servoing_activity as solution

# The test provides an image to your detect_lane_markings function
# and visualizes the left and right masks that it produced.
```

```
UnitTestDLM(solution.detect_lane_markings)
```

Out[ ]: <tests.unit_test.UnitTestDLM at 0xffff36c7fca0>



💻 Test your controller in the simulator

1. Open a terminal on your computer, and type

   ```
   dts code build
   ```

2. Wait for the build to finish, then type:

   ```
   dts code workbench --sim
   ```

3. Open VNC on you browser and click on the `VLS - Visual Lane Servoing Exercise` icon on your desktop. You will see the following open (it might take ~30 seconds or more, depending on the specifications of your computer):

   - A preconfigured RVIZ: to visualize your detections
   - A GUI with buttons labeled `Go`, `Calibration`, and `Stop`.

Note: Clicking on the icon will also bring up an LX terminal, which may print out an error message that stating "Tried to advertise a service that is already advertised in this node ...". You can ignore this error.

On the right side of the RVIZ window, you should see three images: the top is the current image from the Duckiebot's camera, the middle is a visualization of your right lane marking detections, and the bottom is a visualization of your left lane marking detections.

On the terminal on your computer where you launched the activity, you will see some some initial instructions as well as some debugging information.

Once everything is running, you should see instructions on your terminal to calibrate the visual servoing controller. As described above, the `STEER_LEFT_LM` and `STEER_RIGHT_LM` matrices that you defined above will be multiplied by your left and right detection masks, respectively, and the result is then summed to arrive at a scalar value for steering. In practice, the magnitude of this value will be large and unknown. As part of the calibration procedure, you rotate the Duckiebot to different orientations in its lane and the extremes of the resulting steer values will be used to define the range of values.

Once you have finished calibrating, you can click the `Go` button to start your controller. You can hit the `Stop` button at any point to stop the Duckiebot. The Duckiebot will start driving when you hit `Go`.

To test different solutions, change the `get_steer_matrix_left_lane_markings`, `get_steer_matrix_right_lane_markings`, and `detect_lane_markings` functions above. Each time you do, make sure to save this file (`Ctrl-S`) and then re-run the activity with `dts code workbench --sim`.

🚙 Test the controller on your Duckiebot

1. Open a terminal on your computer, and type

```
dts code build
```

2. Wait for the build to finish, then type:

```
dts code workbench -b ROBOTNAME
```

3. Follow the same instructions at point 3 for the simulation-based evaluation above.

Note: we suggest to start at very slow speeds with the physical Duckiebot, and get a hang of the interface first. Going wasted in simulation is just a matter of re-setting the

instance. Having the Duckiebot go wasted in the physical world might be significantly more time consuming!

To test different solutions, change the `get_steer_matrix_left_lane_markings`, `get_steer_matrix_right_lane_markings`, and `detect_lane_markings` functions above. Each time you do, make sure to save this file (`Ctrl-S`) and then re-run the activity with `dts code workbench -b ROBOTNAME`.

Local evaluation and remote submission of your homework exercise

Local evaluation

If you want (this is not necessary) you can evaluate your submission locally before shipping it to the cloud. This will provide you access to detailed performance metrics of your implementation on various episodes. Note that this will take a while to run (~30-60 minutes).

1. Open a terminal, navigate to the exercise folder and run:

   ```
   dts code evaluate
   ```
2. The result of the simulation can be visualized in realtime at the link printed by the evaluator, for example:

3. The evaluation output is saved locally at the end of the evaluation process.

Remote submission

You can submit your agent for evaluation by:

1. Opening a terminal on your computer, navigating to the exercise folder and running:

   ```
   dts code submit
   ```
2. The result of the submission can be visualize on the AIDO challenges website:

After some processing, you should see something like this:

```
~        ## Challenge lx22-viscontrol - MOOC - Visual Lane
Servoing
~
~                Track this submission at:
~
~
https://challenges.duckietown.org/v4/humans/submissions/SUBMISSION-
```

```
NUMBER
~
~                You can follow its fate using:
~
~                    $ dts challenges follow --submission
SUBMISSION-NUMBER
~
~                You can speed up the evaluation using your
own evaluator:
~
~                    $ dts challenges evaluator --submission
SUBMISSION-NUMBER
~
~                For more information, see the manual at
https://docs-old.duckietown.org/daffy/AIDO/out/
~
```

💡 Reflecting on the experience

The above strategy is a reasonable first approach to steering your Duckiebot in order to stay in its lane. Depending on your impelementation and your environment, you may even have found that it does a decent job staying in the lane, even around curves. However, as you have likely seen, there are a number of limitations of the approach:

- Our strategy for detecting lane markings and designing weight matrices limits the the road geometries that it the Duckiebot can handle. While it may do a decent job navigating gradual curves, it likely can't handle sharp turns or T-intersections, among others;
- The detection algorithm is sensitive to changes in illumination, which can result in failed detections (e.g., valid pixels are being filtered out as a result of color masking), and clutter, which can result in erroneous detections (e.g., your Duckiebot may hallucinate lane markings on the wall, on duckies, or elsewhere in the town) that cause the Duckiebot to steer in unpredictable ways.

Nonetheless, hopefully this introduced you to some of the core concepts of image filtering and how you might use it to control the motion of your Duckiebot. Hopefully, it also provided you with first-hand knowledge of some of the challenges of distilling information from images and using them for control, which should set the stage for upcoming topics in the class.