In [ ]:
```
%load_ext autoreload
%autoreload 2
```

The objective in this exercise is to build the functions that you will need to create your histogram filter. The histogram filter represents the state as a fixed set of hypotheses that correspond to the centroids of evenly spaced cells. Each one of these hypothesis has an associated weight, and the weights should sum to one. As a result, the histogram corresponds to a valid belief distribution over the state space. In math we can write this as a weighted sum of Dirac delta functions:

$$bel(x_t) = \sum_{i=1}^{N} w_t^i \delta(x_t - x^i)$$

The API for this (and really any) filter will comprise three functions: `prior()`, `predict()` and `update()`. The `prior()` function sets up the initial belief weights, w_0 over the histogram.

The `predict()` function propagates forward the belief weights based on the motion model and the control, u_t. This amounts to propagating the centroids of each of the cells forward and then adding all of the weight up that lands in each bin.

$$\overline{bel}(x_t) = \sum_{i=1}^{N} \sum_{j=1}^{N} w_{t-1}^j p(x^i | x^j, u_t) \delta(x_t - x^i)$$

In our case we will be using the odometry as a proxy for the control input so that we may use a simple kinematic model of the robot.

Finally, the `update()` function takes a measurement and uses it to update the weights of the histogram bins based on the incoming measurement, z_t. This is achieved by multiplying the weight in each bin by the likelihood that the measurement was generated by the state corresponding to the centroid of that bin:

$$bel(x_t) = \sum_{i=1}^{N} \frac{\overline{w}_t^i p(z_t | x^i)}{\displaystyle\sum_{j=1}^{N} \overline{w}_t^j p(z_t | x^j)} \delta(x_t - x^i)$$

In this notebook we will proceed by loading one image and using the line detection and ground projection algorithms (we can consider them as a black box here) to detect all of the white and yellow line segments. Each line segment will contribute a vote in the measurement likelihood.

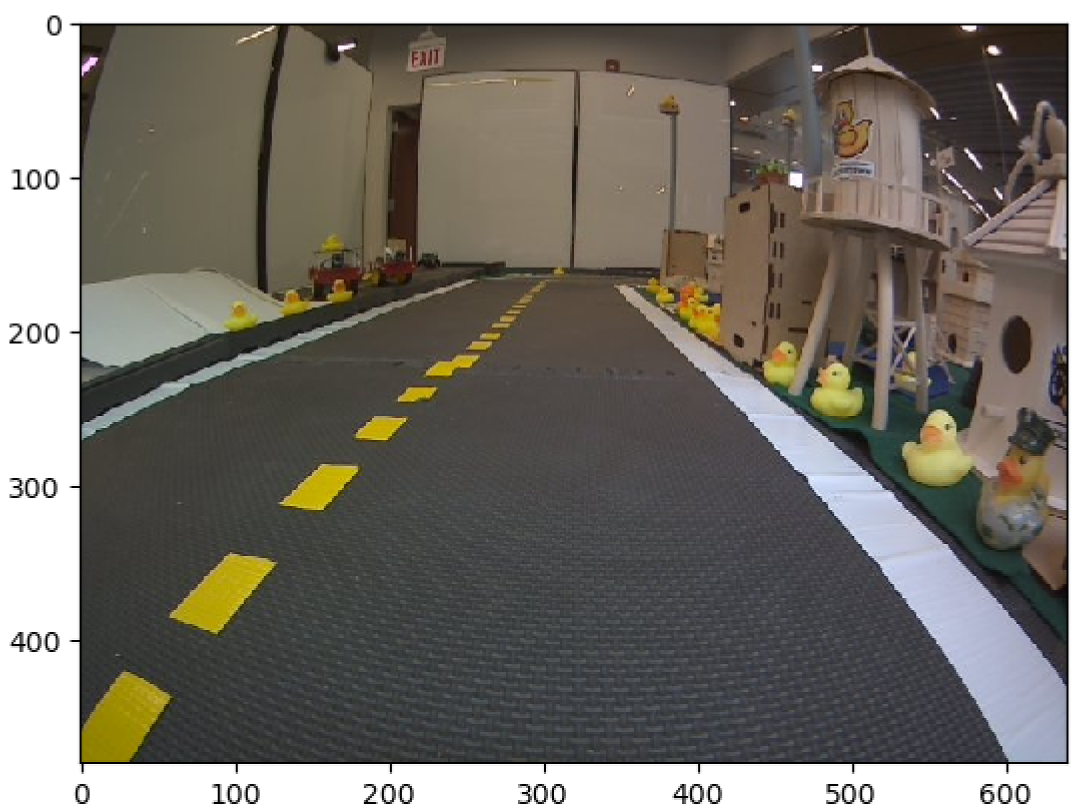
After completing the notebook, apply the same concepts in the functions within histogram_filter.py (fill in the TODOs).

After running `dts code build` these functions can be used on the simulated or real Duckiebot using `dts code workbench --sim` or `dts code workbench -b <ROBOT_NAME>`. This will use the real (or simulated) data coming from your camera instead of the single image that we loaded here.

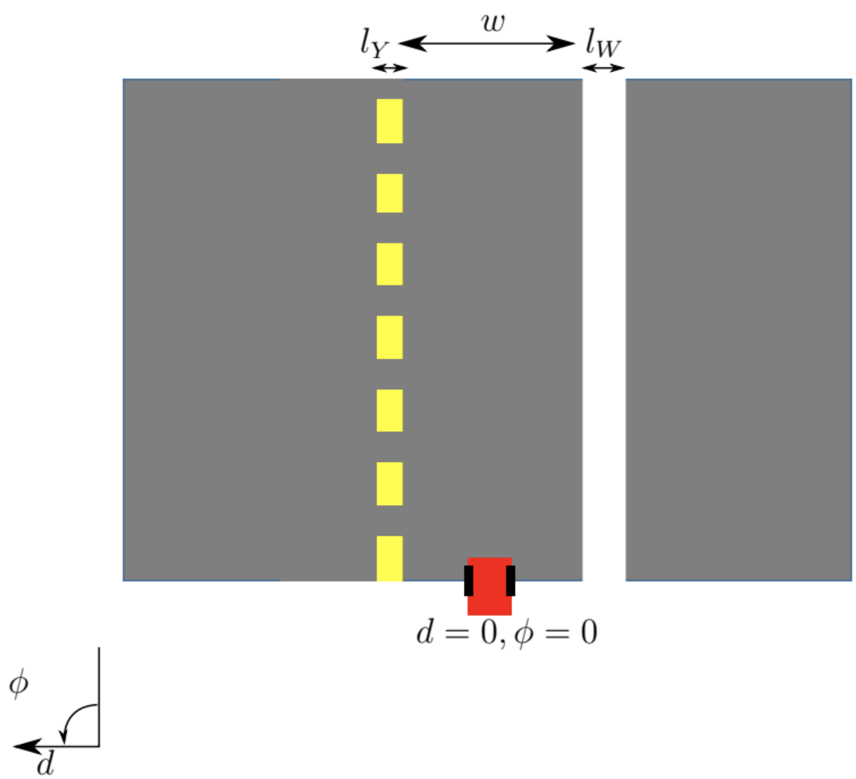
In [ ]:
```
# start by importing some things we will need
import cv2
import matplotlib
import numpy as np
from scipy.ndimage.filters import gaussian_filter
from scipy.stats import entropy, multivariate_normal
from math import floor, sqrt
```

In [ ]:
```
# Now let's load the image that we will use. Feel free to change it,
# but the calibrations in the setup/calibrations folder should correspond to
# that took the image
from matplotlib.pyplot import imshow
%matplotlib inline
img = cv2.imread("../../assets/images/pic1.png")
imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
```

Out[ ]:
```
<matplotlib.image.AxesImage at 0xffff647b19a0>
```



We will define the state here to be the comprised of the distance from the center of the lane d and the angle relative to the lane ϕ.



In [ ]:

```
# Now we will load parameters from the configuration file
# These are the same parameters that will be loaded when we do
# dts code workbench. Feel free to experiment with different values
# for any of the parameters
import yaml
with open("../../packages/histogram_lane_filter/config/histogram_lane_filter
    try:
        params = yaml.safe_load(stream)
    except yaml.YAMLError as exc:
        print(exc)
hp = params["lane_filter_histogram_configuration"]
print(hp)

d, phi = np.mgrid[hp['d_min'] : hp['d_max'] : hp['delta_d'], hp['phi_min'] :

# We are going to organize them into some data structures so that they are e
grid_spec = {
    "d": d,
    "phi": phi,
    "delta_d": hp['delta_d'],
    "delta_phi": hp['delta_phi'],
    "d_min": hp['d_min'],
    "d_max": hp['d_max'],
    "phi_min": hp['phi_min'],
    "phi_max": hp['phi_max'],
}
road_spec = {
    "linewidth_white": hp['linewidth_white'],
    "linewidth_yellow": hp['linewidth_yellow'],
    "lanewidth": hp['lanewidth'],
```

```
}
robot_spec = {
    "wheel_radius": hp['wheel_radius'],
    "wheel_baseline": hp['wheel_baseline'],
    "encoder_resolution": hp['encoder_resolution'],
}

# The "cov_mask" is effectively the process model covariance
cov_mask = [hp['sigma_d_mask'], hp['sigma_phi_mask']]
belief = np.empty(d.shape)
mean_0 = [hp['mean_d_0'], hp['mean_phi_0']]
cov_0 = [[hp['sigma_d_0'], 0], [0, hp['sigma_phi_0']]]
```

```
{'mean_d_0': 0, 'mean_phi_0': 0, 'sigma_d_0': 0.1, 'sigma_phi_0': 0.1, 'delt
a_d': 0.02, 'delta_phi': 0.1, 'd_max': 0.3, 'd_min': -0.3, 'phi_min': -1.5,
'phi_max': 1.5, 'linewidth_white': 0.05, 'linewidth_yellow': 0.025, 'lanewid
th': 0.23, 'sigma_d_mask': 1.0, 'sigma_phi_mask': 2.0, 'range_min': 0.2, 'ra
nge_est': 0.45, 'range_max': 0.6, 'encoder_resolution': 135, 'wheel_radius':
0.0318, 'wheel_baseline': 0.1}
```

In [ ]:
```
# Now let's define the prior function. In this case we choose
# to initialize the historgram based on a Gaussian distribution around [0,0]
def histogram_prior(belief, grid_spec, mean_0, cov_0):
    pos = np.empty(belief.shape + (2,))
    pos[:, :, 0] = grid_spec["d"]
    pos[:, :, 1] = grid_spec["phi"]
    RV = multivariate_normal(mean_0, cov_0)
    belief = RV.pdf(pos)
    return belief
```

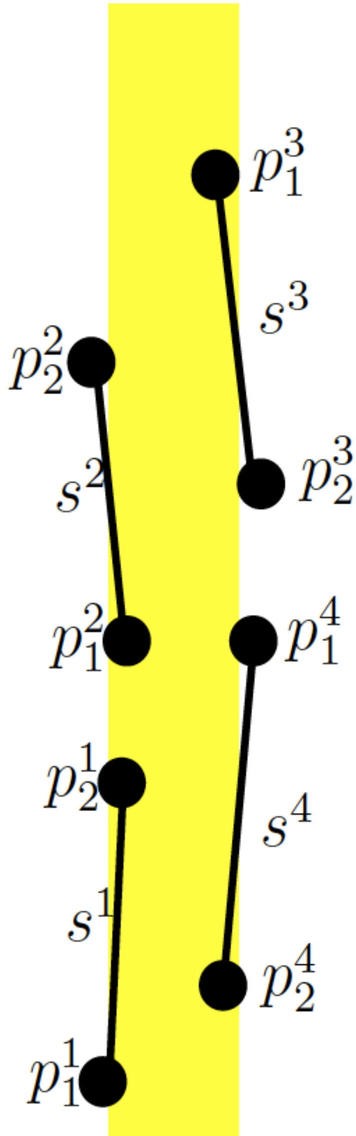
In [ ]:
```
# Now let's define the predict function


def histogram_predict(belief, left_encoder_ticks, right_encoder_ticks, grid_
        belief_in = belief

        # TODO propagate each centroid forward using the kinematic function
        # Extracting robot specifications
        wheel_radius, wheel_baseline, encoder_resolution = robot_spec['wheel
        # Convert encoder ticks to distances traveled by each wheel
        left_distance = (2 * np.pi * wheel_radius * left_encoder_ticks) / er
        right_distance = (2 * np.pi * wheel_radius * right_encoder_ticks) /
        # Calculate the forward displacement (d_t) and angular displacement
        d_t = (left_distance + right_distance) / 2
        phi_t = (right_distance - left_distance) / wheel_baseline

        d_t = d_t + grid_spec['d'] # replace this with something that adds t
        phi_t = phi_t + grid_spec['phi'] # replace this with something that

        p_belief = np.zeros(belief.shape)

        # Accumulate the mass for each cell as a result of the propagation s
        for i in range(belief.shape[0]):
            for j in range(belief.shape[1]):
                # If belief[i,j] there was no mass to move in the first plac
                if belief[i, j] > 0:
```

```python
                # Now check that the centroid of the cell wasn't propaga
                if (
                    d_t[i, j] > grid_spec['d_max']
                    or d_t[i, j] < grid_spec['d_min']
                    or phi_t[i, j] < grid_spec['phi_min']
                    or phi_t[i, j] > grid_spec['phi_max']
                ):
                    continue

                # TODO Now find the cell where the new mass should be ac
                i_new = int((i + d_t[i, j] / grid_spec['delta_d']) % bel
                j_new = int((j + phi_t[i, j] / grid_spec['delta_phi']) %

                p_belief[i_new, j_new] += belief[i, j]

    # Finally we are going to add some "noise" according to the process
    # This is implemented as a Gaussian blur over the histogram
    s_belief = np.zeros(belief.shape)
    gaussian_filter(p_belief, cov_mask, output=s_belief, mode="constant"

    if np.sum(s_belief) == 0:
        return belief_in
    belief = s_belief / np.sum(s_belief)
    return belief
```

Now we are going to work on building the measurement likelihood. We will have as an input a list of segments. Each segment has endpoints, a normal vector, and an associated color. For each segment, we will use basic geometry to figure out what position (d) and orientation (ϕ) the robot would have had to have been at to detect the specific segment assuming that it did in fact come from a road marking.

There is a bit of annoying detail here since we can detect lines on either side of the actual lane markings. We use the normals to determine which side of the lane marking the line was on. The following shows the `lanewidth` and the `linewidth_yellow` and `linewidth_white` parameters.

The following image shows a representations of how the detected line segments sit on an actual lane



In [ ]:

```
# We will start by doing a little bit of processing on the segments to remov
# or a color not equal to yellow or white

def prepare_segments(segments):
    filtered_segments = []
    for segment in segments:

        # we don't care about RED ones for now
        if segment.color != segment.WHITE and segment.color != segment.YELLO
            continue
        # filter out any segments that are behind us
        if segment.points[0].x < 0 or segment.points[1].x < 0:
            continue

        filtered_segments.append(segment)
    return filtered_segments
```

Now for each segment we will generate a vote according to:

Algorithm 1 Generate_Vote

Input: A single segment in the body frame s^i

Output: The corresponding position and angle in the lane indicated by the segment: d^i,ϕ^i.

$d^i = 0.5(x_1^i + x_2^i)$

if $s^i.color == WHITE$ **then**

  $d^i - = w/2$

  if $x_2^i > x_1^i$ **then**

    $d^i - = l_W$

  end if

else

  $d^i + = w/2$

  if $x_2^i > x_1^i$ **then**

    $d^i + = l_Y$

  end if

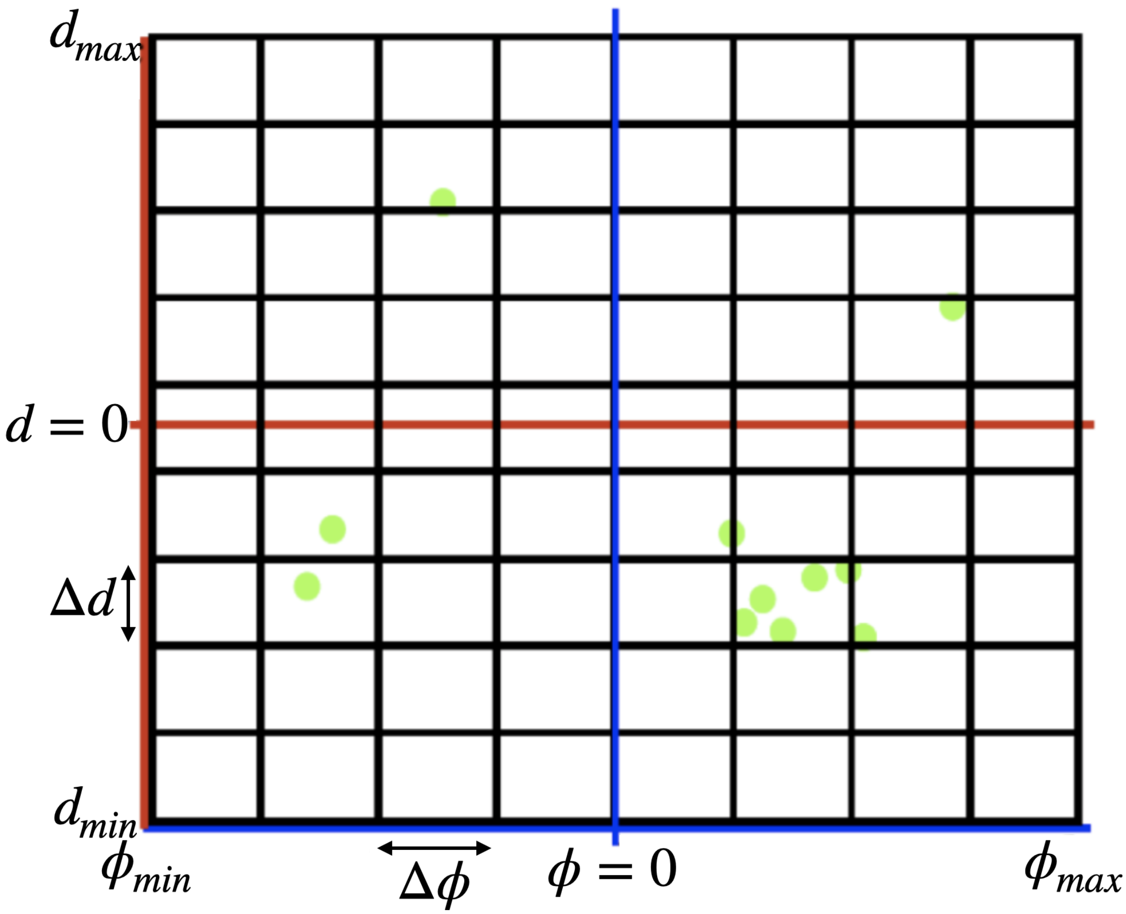
end if

$\phi_i = \pi/2 - \operatorname{atan2}\left(\frac{|x_2^i - x_1^i|}{y_2^i - y_1^i}\right)$

In [ ]:
```python
def generate_vote(segment, road_spec):
    p1 = np.array([segment.points[0].x, segment.points[0].y])
    p2 = np.array([segment.points[1].x, segment.points[1].y])
    t_hat = (p2 - p1) / np.linalg.norm(p2 - p1)
    n_hat = np.array([-t_hat[1], t_hat[0]])

    d1 = np.inner(n_hat, p1)
    d2 = np.inner(n_hat, p2)
    l1 = np.inner(t_hat, p1)
    l2 = np.inner(t_hat, p2)
    if l1 < 0:
        l1 = -l1
    if l2 < 0:
        l2 = -l2

    l_i = (l1 + l2) / 2
    d_i = (d1 + d2) / 2
    phi_i = np.arcsin(t_hat[1])
    if segment.color == segment.WHITE:  # right lane is white
        if p1[0] > p2[0]:  # right edge of white lane
            d_i -= road_spec['linewidth_white']
        else:  # left edge of white lane
            d_i = -d_i
            phi_i = -phi_i
        d_i -= road_spec['lanewidth'] / 2

    elif segment.color == segment.YELLOW:  # left lane is yellow
        if p2[0] > p1[0]:  # left edge of yellow lane
            d_i -= road_spec['linewidth_yellow']
            phi_i = -phi_i
        else:  # right edge of white lane
            d_i = -d_i
```

```
        d_i = road_spec['lanewidth'] / 2 - d_i

    return d_i, phi_i
```

Now we generate the entire measurement likelihood by generating a vote for each line segment in the list that we received. The measurement likelihood will itself be a histogram:



In [ ]:
```
def generate_measurement_likelihood(segments, road_spec, grid_spec):

    # initialize measurement likelihood to all zeros
    measurement_likelihood = np.zeros(grid_spec['d'].shape)

    for segment in segments:
        d_i, phi_i = generate_vote(segment, road_spec)

        # if the vote lands outside of the histogram discard it
        if d_i > grid_spec['d_max'] or d_i < grid_spec['d_min'] or phi_i < g
            continue

        # TODO find the cell index that corresponds to the measurement d_i,
        i = int(round((d_i - grid_spec['d_min']) / grid_spec['delta_d'])) #
        j = int(round((phi_i - grid_spec['phi_min']) / grid_spec['delta_phi'

        # Add one vote to that cell
        measurement_likelihood[i, j] += 1
```

```
    if np.linalg.norm(measurement_likelihood) == 0:
        return None
    measurement_likelihood /= np.sum(measurement_likelihood)
    return measurement_likelihood
```

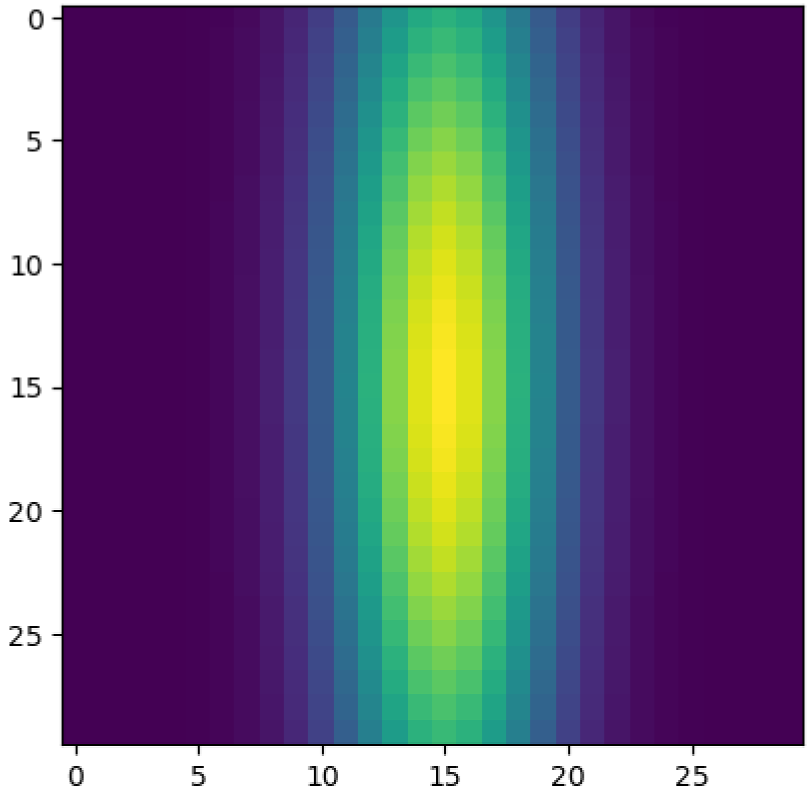
Now we have everything we need for the update function.

In [ ]:
```
def histogram_update(belief, segments, road_spec, grid_spec):
    # prepare the segments for each belief array
    segmentsArray = prepare_segments(segments)
    # generate all belief arrays

    measurement_likelihood = generate_measurement_likelihood(segmentsArray,

    if measurement_likelihood is not None:
        # TODO: combine the prior belief and the measurement likelihood to g
        # Don't forget that you may need to normalize to ensure that the out
        posterior_belief = belief * measurement_likelihood
        # Normalize the posterior belief to ensure it is a valid probability
        posterior_belief /= np.sum(posterior_belief)
        belief = posterior_belief # replace this with something that combine
    return (measurement_likelihood, belief)
```

Now we have defined the `prior()`, `predict()` and `update()` functions. We will test one cycle of the filter here to see if things look reasonable.

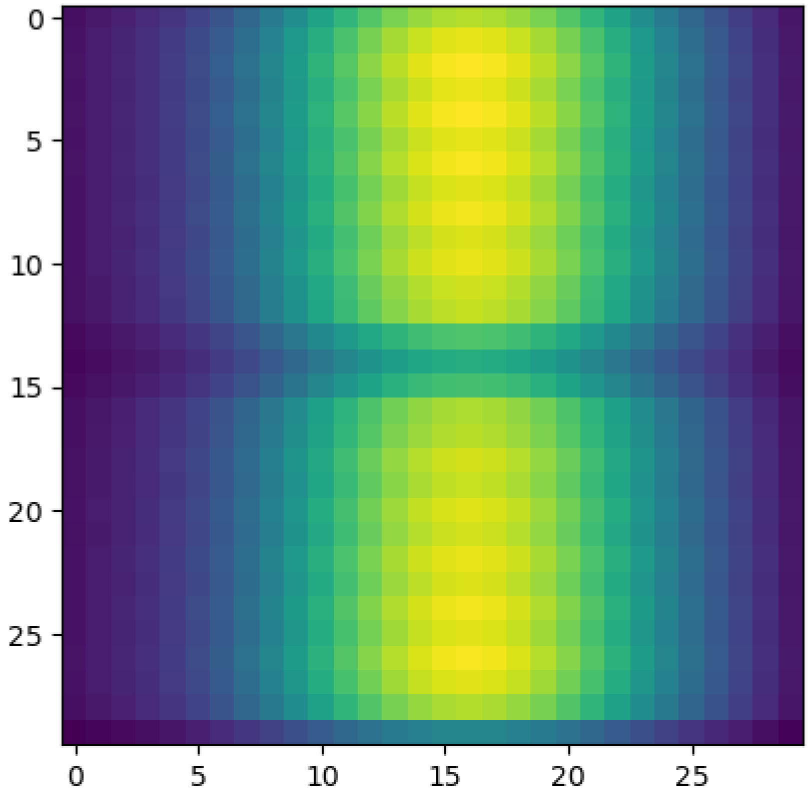
In [ ]:
```
# Let's start initializing the belief:
belief = histogram_prior(belief, grid_spec, mean_0, cov_0)
imshow(belief)
```

Out[ ]: `<matplotlib.image.AxesImage at 0xffff2ef553a0>`



In [ ]:
```
# Now let's generate some fake encoder data and do one step of the predictic

left = 10 # left ticks
right = 20 # right ticks
belief = histogram_predict(belief, left, right, grid_spec, robot_spec, cov_m
imshow(belief)
```

Out[ ]: <matplotlib.image.AxesImage at 0xffff2ef2bdc0>



In [ ]:
```
!echo $LD_PRELOAD
```

In [ ]:
```
from solution.segments import detect_line_segments
import cv2

# This function will take the image that we loaded, detect the line segments
# We don't need to worry too much about details here.
sg = detect_line_segments(img)
```

```
DEBUG:commons:version: 6.2.4 *
DEBUG:typing:version: 6.2.3
DEBUG:geometry:PyGeometry-z6 version 2.1.4 path /usr/local/lib/python3.8/dis
t-packages
```

```
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
Cell In[14], line 1
----> 1 from solution.segments import detect_line_segments
      2 import cv2
      4 # This function will take the image that we loaded, detect the line
segments, and project them onto the ground plane.
      5 # We don't need to worry too much about details here.

File /code/state-estimation/packages/solution/segments.py:9
      7 from ground_projection.segment import rectify_segments
      8 from image_processing.more_utils import get_robot_camera_geometry
----> 9 from line_detector2.image_prep import ImagePrep
     10 from line_detector_interface import FAMILY_LINE_DETECTOR
     11 from line_detector_interface.visual_state_fancy_display import vs_fa
ncy_display

File /code/catkin_ws/src/state-estimation/packages/dt-core/packages/complete
_image_pipeline/include/line_detector2/image_prep.py:6
      4 from duckietown_msgs.msg import Segment, SegmentList
      5 from .fuzzing import fuzzy_segment_list_image_space
----> 6 from .ldn import toSegmentMsg
      9 class ImagePrep:
     10     FAMILY = "image_prep"

File /code/catkin_ws/src/state-estimation/packages/dt-core/packages/complete
_image_pipeline/include/line_detector2/ldn.py:3
      1 import cv2
      2 import numpy as np
----> 3 from anti_instagram import AntiInstagram
      5 import duckietown_code_utils as dtu
      6 from cv_bridge import CvBridge

File /code/catkin_ws/src/state-estimation/packages/dt-core/packages/complete
_image_pipeline/include/anti_instagram/__init__.py:8
      5 logger = logging.getLogger("anti_instagram")
      6 logger.setLevel(logging.DEBUG)
----> 8 from .anti_instagram_imp import *
      9 from .kmeans import *
     10 from .utils import *

File /code/catkin_ws/src/state-estimation/packages/dt-core/packages/complete
_image_pipeline/include/anti_instagram/anti_instagram_imp.py:2
      1 import cv2
----> 2 from .kmeans import getparameters2, identifyColors, runKMeans
      3 from .scale_and_shift import scaleandshift
      4 from anti_instagram.kmeans import CENTERS, CENTERS2

File /code/catkin_ws/src/state-estimation/packages/dt-core/packages/complete
_image_pipeline/include/anti_instagram/kmeans.py:2
      1 from collections import Counter
----> 2 from sklearn import linear_model
      3 from sklearn.cluster import KMeans
      4 import cv2

File /usr/lib/python3/dist-packages/sklearn/__init__.py:83
```

```
     81     from . import __check_build  # noqa: F401
     82     from .base import clone
---> 83     from .utils._show_versions import show_versions
     85     __all__ = ['calibration', 'cluster', 'covariance', 'cross_decomp
osition',
     86                'datasets', 'decomposition', 'dummy', 'ensemble', 'ex
ceptions',
     87                'experimental', 'externals', 'feature_extraction',
   (...)
     96                'clone', 'get_config', 'set_config', 'config_contex
t',
     97                'show_versions']
    100 def setup_module(module):

File /usr/lib/python3/dist-packages/sklearn/utils/_show_versions.py:12
      9 import sys
     10 import importlib
---> 12 from ._openmp_helpers import _openmp_parallelism_enabled
     15 def _get_sys_info():
     16     """System information
     17 
     18     Return
   (...)
     22 
     23     """

ImportError: /lib/aarch64-linux-gnu/libgomp.so.1: cannot allocate memory in
static TLS block
```

In [ ]:

```
%matplotlib inline

# Finally we can take the ground projected segments and call our update func
(measurement_likelihood, belief) = histogram_update(belief,sg.segments, road
imshow(belief)
```

After completing the notebook, apply the same concepts in the functions within histogram_filter.py (fill in the TODOs).

💻 Test your histogram filter 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
```

🚙 Test the histogram filter 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
   ```

Local evaluation and remote submission of your homework exercise

Local evaluation

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

   ```
   dts code evaluate
   ```
2. 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-state-estimation - MOOC - State
Estimation
~
~                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/
~
```