ENPM 809T

UMCP, Mitchell, Summer 2019

Disclaimers

1. There are many approaches to identify lane lines

   ...this is one incarnation

2. Not (yet) using Python best practices in terms of coding

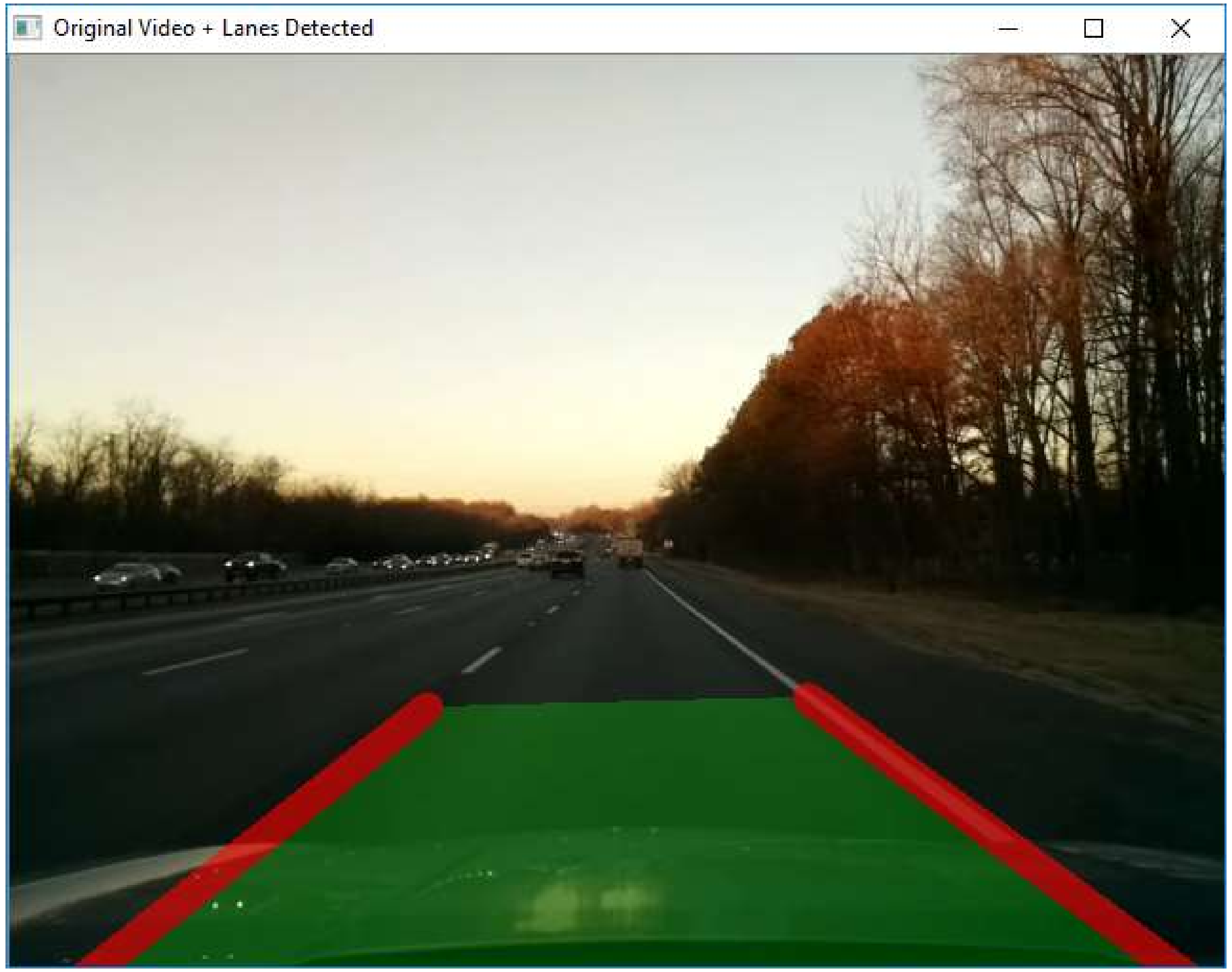
   ...we'll focus on that now

3. This is a basic approach to identifying lane lines

   ...have fun with it!... think of ways to enhance

Lane Detection Pipeline

1. Load image
2. Snip region of interest
3. Mask region of interest
4. Detect lane line type
5. Greyscale then B/W
6. Blur
7. Detect edges
8. Detect lines
9. Plot lines on image
10. Plot lane on image



Codes available via GitHub - https://github.com/oneshell/enpm809T

Lane Detection

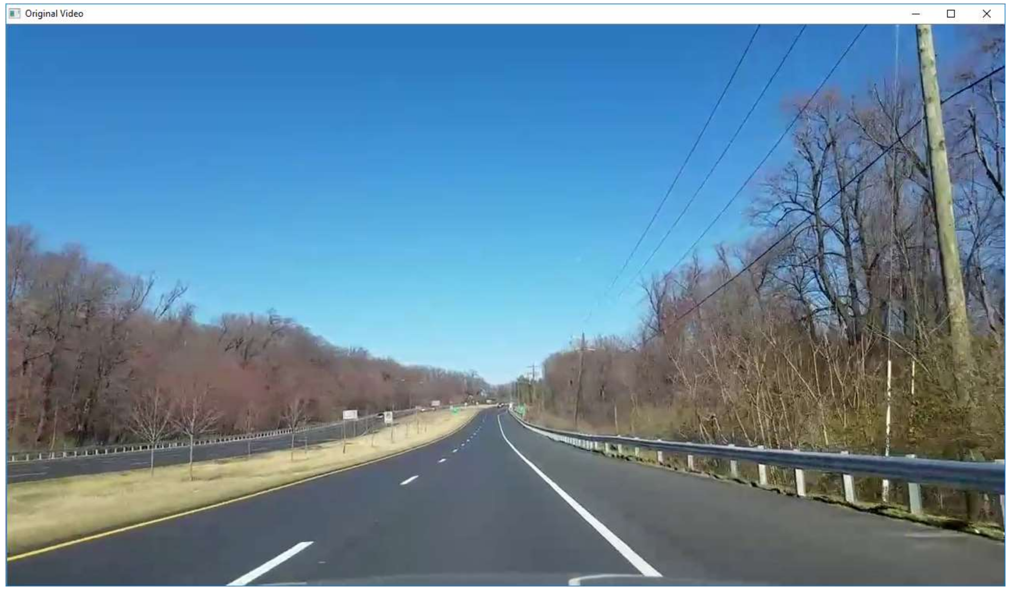
- Import required packages

```
import numpy as np
import cv2
import imutils
```

Lane Detection

- Load video & show original video feed on screen

```
# Identify filename of video
camera = cv2.VideoCapture('20190130-073331.mp4')

# Grab single frame from video stream
def grab_frame(camera):
    ret, frame = camera.read()
    # frame = cv2.flip(frame, -1)   #
    return frame

# Loop through until entire video file is played
while True:
    # print "Frame number: ", counter, "\n"
    counter = counter + 1

    frame = grab_frame(camera)
    final_output = frame.copy()
```



Lane Detection

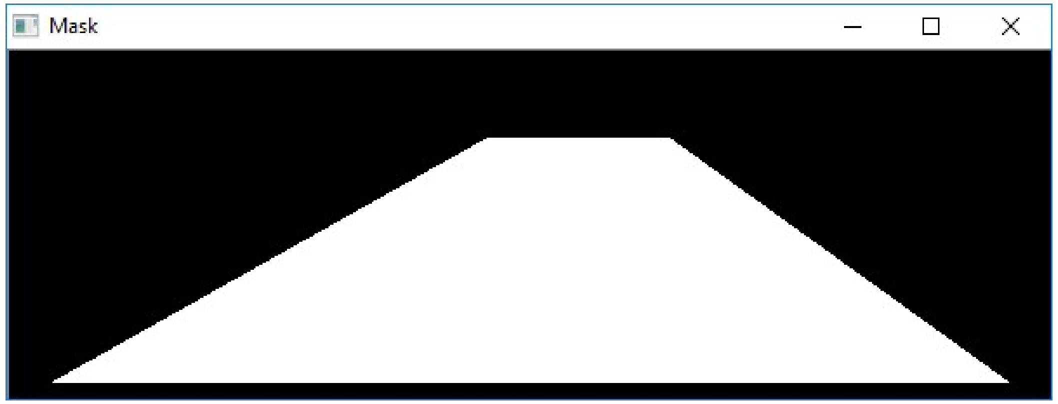
- Snip region of interest


```
# Snip region of interest in video frame
def snip_image(img):
    # cv2.line(img, (img.shape[1]/2, 0), (img.shape[1]/2, img.shape[0]), (0, 255, 0))  # plots
    # cv2.line(img, (0, img.shape[0]/2), (img.shape[1], img.shape[0]/2), (0, 255, 0))  # plots
    cv2.rectangle(img, (0, img.shape[0] - cc), (img.shape[1], img.shape[0] - aa), (0, 0, 255))
    snip = img[(img.shape[0] - aa):(img.shape[0] - cc), (0):(img.shape[1])]
    return snip
```

```
snip = snip_image(frame)
# cv2.imshow("Region of Interest", frame)
# cv2.imshow("Snip", snip)
```

Lane Detection

- Create polygon mask




```
# Create & apply polygon (trapezoid) mask to selected region of interest
def mask_image(img):
    mask = np.zeros((img.shape[0], img.shape[1]), dtype="uint8")
    pts = np.array([[90, img.shape[0]-25], [90, img.shape[0]-35], [img.shape[1]/2-30, 100], [img.shape[1]/2+30, 100],
                    [img.shape[1]-90, img.shape[0]-35], [img.shape[1]-90, img.shape[0]-25]], dtype=np.int32)
    cv2.fillConvexPoly(mask, pts, 255)
    masked = cv2.bitwise_and(img, img, mask=mask)
    # cv2.line(masked, (img.shape[1]/2, 0), (img.shape[1]/2, img.shape[0]), (0, 255, 0))  # plots crosshairs on the mask
    # cv2.line(masked, (0, img.shape[0]/2), (img.shape[1], img.shape[0]/2), (0, 255, 0))  # plots crosshairs on the mask
    return masked
```

Lane Detection

- Apply mask to snipped video


```
# Create & apply polygon (trapezoid) mask to selected region of interest
def mask_image(img):
    mask = np.zeros((img.shape[0], img.shape[1]), dtype="uint8")
    pts = np.array([[90, img.shape[0]-25], [90, img.shape[0]-35], [img.shape[1]/2-30, 100], [img.shape[1]/2+30, 100],
                    [img.shape[1]-90, img.shape[0]-35], [img.shape[1]-90, img.shape[0]-25]], dtype=np.int32)
    cv2.fillConvexPoly(mask, pts, 255)
    masked = cv2.bitwise_and(img, img, mask=mask)
    # cv2.line(masked, (img.shape[1]/2, 0), (img.shape[1]/2, img.shape[0]), (0, 255, 0))  # plots crosshairs on the mask
    # cv2.line(masked, (0, img.shape[0]/2), (img.shape[1], img.shape[0]/2), (0, 255, 0))  # plots crosshairs on the mask
    return masked
```

```
masked = mask_image(snip)
cv2.imshow("Masked", masked)
```

AND logic gate

Gate

Truth Table

A	B	Z
0	0	0
0	1	0
1	0	0
1	1	1

Notation

$$Z = AB$$
$$= A \cdot B$$

"logic multiplication"

- Output is 1 if BOTH inputs are 1

Lane Detection

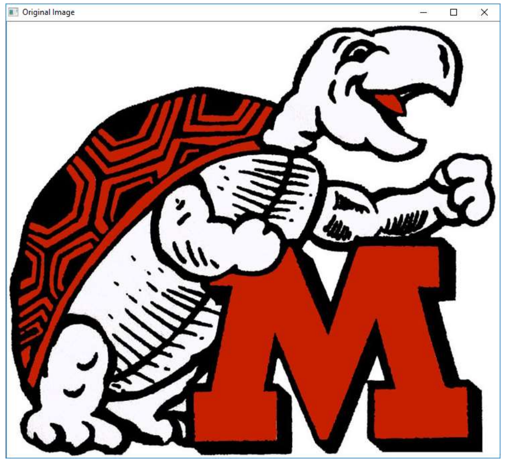
- Convert to grayscale

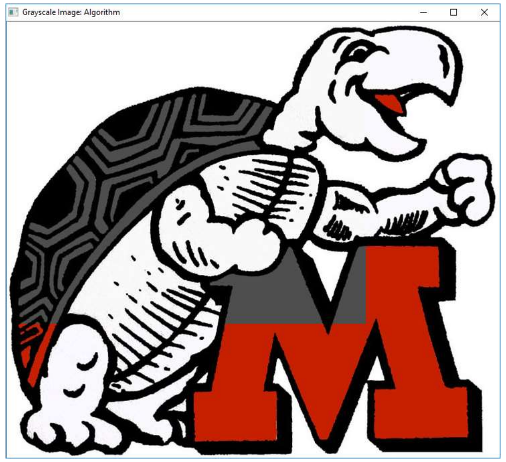
```
# Mask frame for color of lane lines, convert to grayscale then black/white to binary image
def thres_image(img):
    hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    yellowLower = np.array([0, 65, 164]); yellowUpper = np.array([255, 255, 255])
    whiteLower = np.array([30, 0, 65]); whiteUpper = np.array([255, 255, 150])
    yellow_mask = cv2.inRange(hsv, yellowLower, yellowUpper)
    white_mask = cv2.inRange(hsv, whiteLower, whiteUpper)
    full_mask = cv2.bitwise_or(yellow_mask, white_mask)
    frame = cv2.bitwise_or(gray, full_mask)
    thresh = 100
    frame = cv2.threshold(frame, thresh, 255, cv2.THRESH_BINARY)[1]
    return frame
```

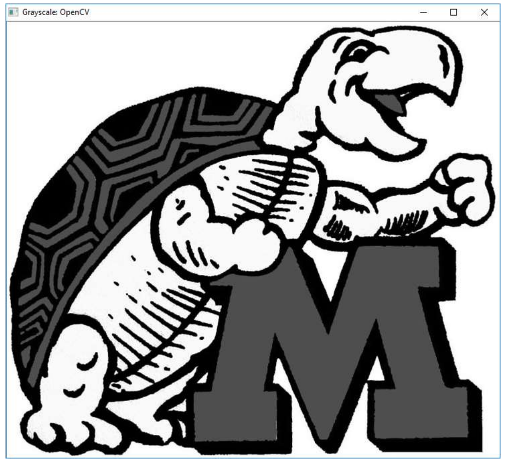
```
frame = thres_image(masked)
# cv2.imshow("Thresholded to B/W", frame)
```

GitHub: grayscale.py

```python
import numpy as np
import cv2
import imutils

# Load & show original image
image = cv2.imread("testudo.jpg")
true = image.copy()

cv2.imshow("Original Image", image)

x_lim = image.shape[0]
y_lim = image.shape[1]

for x in range (0,x_lim-1):
    for y in range (0,y_lim - 1):
        (b, g, r) = image[x, y]
        # image[x, y] = (0.33 * b + 0.33 * g + 0.33 * r)
        image[x, y] = (0.11 * b + 0.59 * g + 0.3 * r)

cv2.imshow("Grayscale Image: Algorithm", image)

image = cv2.cvtColor(true, cv2.COLOR_BGR2GRAY)
cv2.imshow("Grayscale: OpenCV", image)

cv2.waitKey(0)
```







Lane Detection

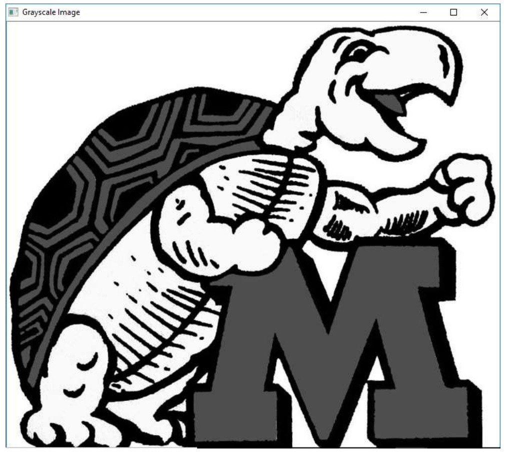
- Threshold image to black/white

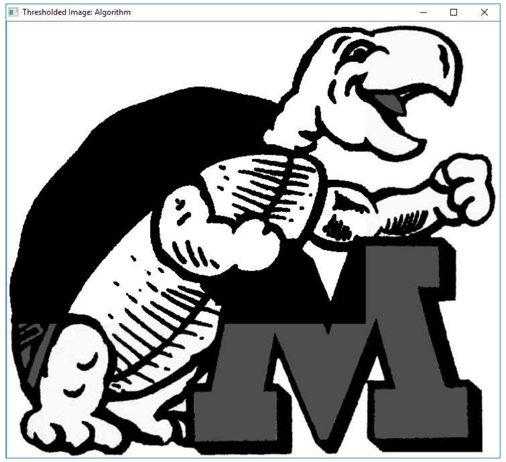
```
# Mask frame for color of lane lines, convert to grayscale then black/white to binary image
def thres_image(img):
    hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    yellowLower = np.array([0, 65, 164]); yellowUpper = np.array([255, 255, 255])
    whiteLower = np.array([30, 0, 65]); whiteUpper = np.array([255, 255, 150])
    yellow_mask = cv2.inRange(hsv, yellowLower, yellowUpper)
    white_mask = cv2.inRange(hsv, whiteLower, whiteUpper)
    full_mask = cv2.bitwise_or(yellow_mask, white_mask)
    frame = cv2.bitwise_or(gray, full_mask)
    thresh = 100
    frame = cv2.threshold(frame, thresh, 255, cv2.THRESH_BINARY)[1]
    return frame
```

```
frame = thres_image(masked)
# cv2.imshow("Thresholded to B/W", frame)
```

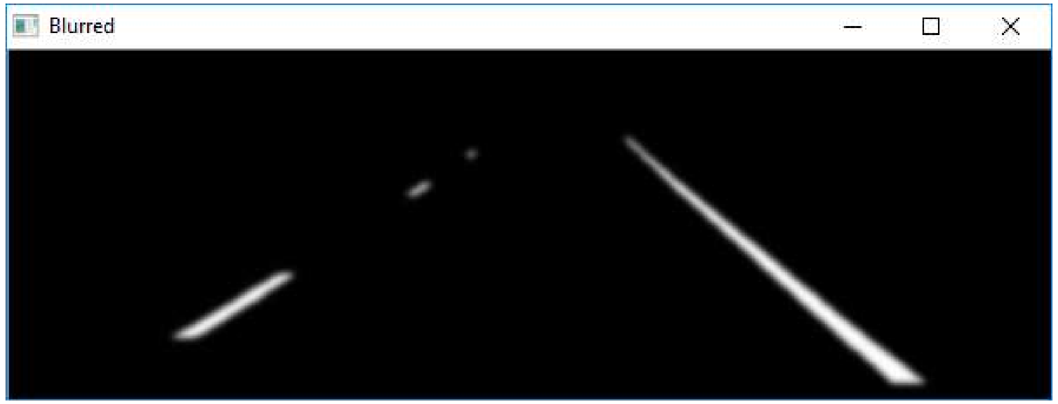
GitHub: threshold.py

```python
import numpy as np
import cv2
import imutils

# Load & show original image
image = cv2.imread("testudo.jpg")
cv2.imshow("Original Image", image)

# Grayscale image using cv2.COLOR_BGR2GRAY
image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
cv2.imshow("Grayscale Image", image)

true = image.copy()

# Define threshold, in the range 0-255
thresh = 150

for x in range (0,image.shape[0]-1):
    for y in range (0,image.shape[1]-1):
        if image[x, y] > thresh:
            image[x, y] = 255
        else:
            image[x, y] = 0

cv2.imshow("Thresholded Image: Algorithm", image)

# Threshold image using cv2.THRESH_BINARY
frame = cv2.threshold(true, thresh, 255, cv2.THRESH_BINARY)[1]
cv2.imshow("Thresholded Image: OpenCV", frame)

cv2.waitKey(0)
```





Lane Detection



- Blur image
- …which blur is optimal?

```
# Blur image to assist edge detection
def blur_image(img):
    return cv2.GaussianBlur(img, (21, 21), 0)    # (img, kernel size)
```

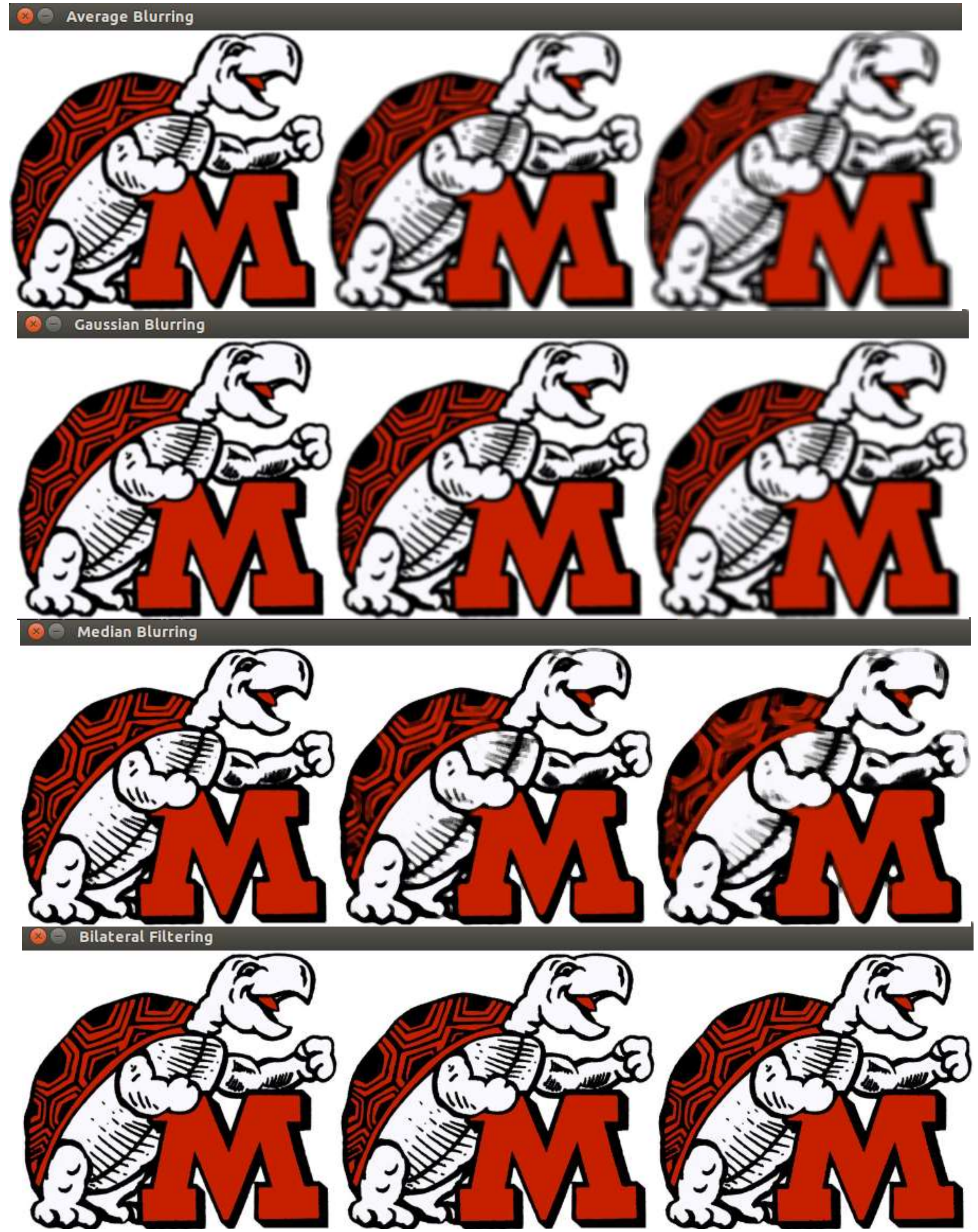
```
blurred = blur_image(frame)
# cv2.imshow("Blurred", blurred)
```

H $*$ F

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	0	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	0	0	0	0	0	0	0
0	0	90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

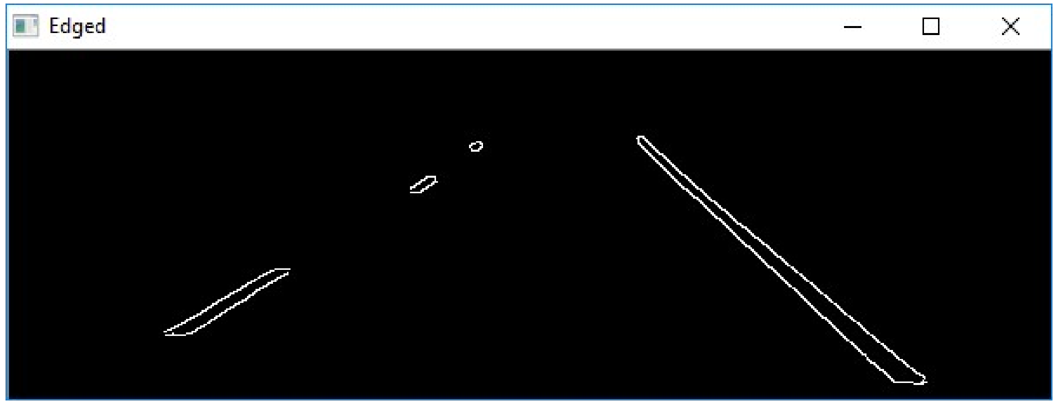
$=$ G

	0	10	20	30	30	30	20	10	
	0	20	40	60	60	60	40	20	
	0	30	60	90	90	90	60	30	
	0	30	50	80	80	90	60	30	
	0	30	50	80	80	90	60	30	
	0	20	30	50	50	60	40	20	
	10	20	30	30	30	30	20	10	
	10	10	10	0	0	0	0	0	



Lane Detection

- Identify edges




```
# Identify edges & show on screen
def edge_image(img):
    return cv2.Canny(img, 30, 150)  # (img, low thresh, high thres)
```

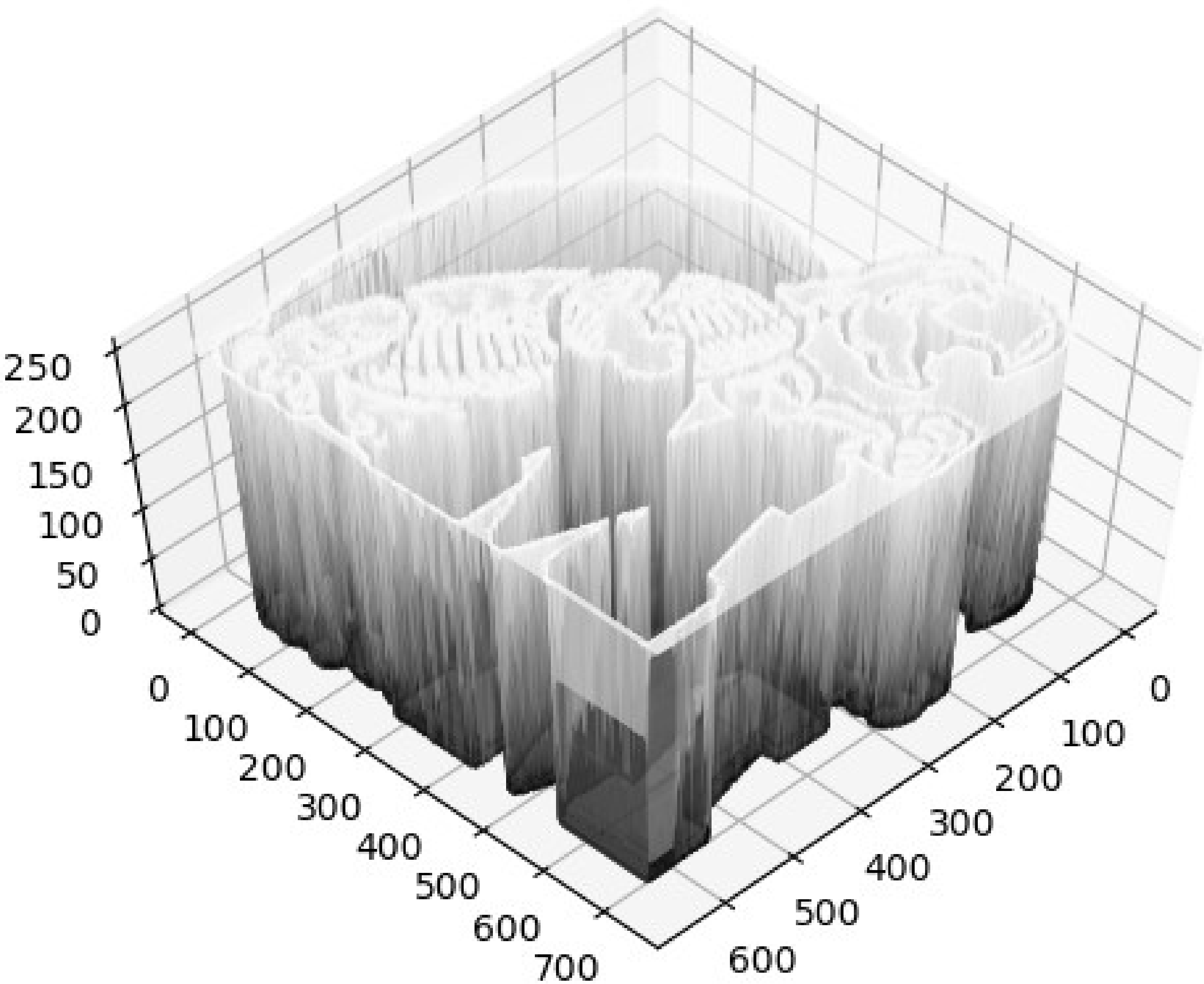
```
edged = edge_image(blurred)
cv2.imshow("Edged", edged)
```

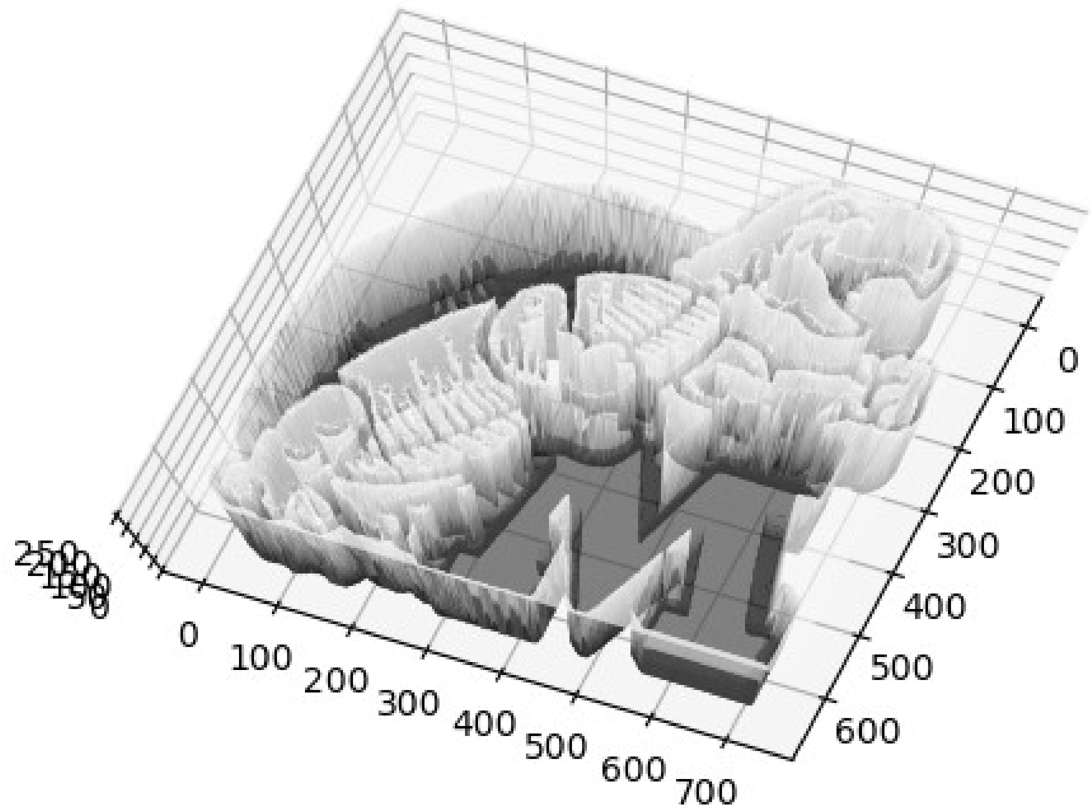
https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_imgproc/py_gradients/py_gradients.html

- Motivation: it is clear that Testudo is in both images at right

- And yet, in the bottom image, there is not much information in each pixel, only the edges as 0 or 255 in grayscale

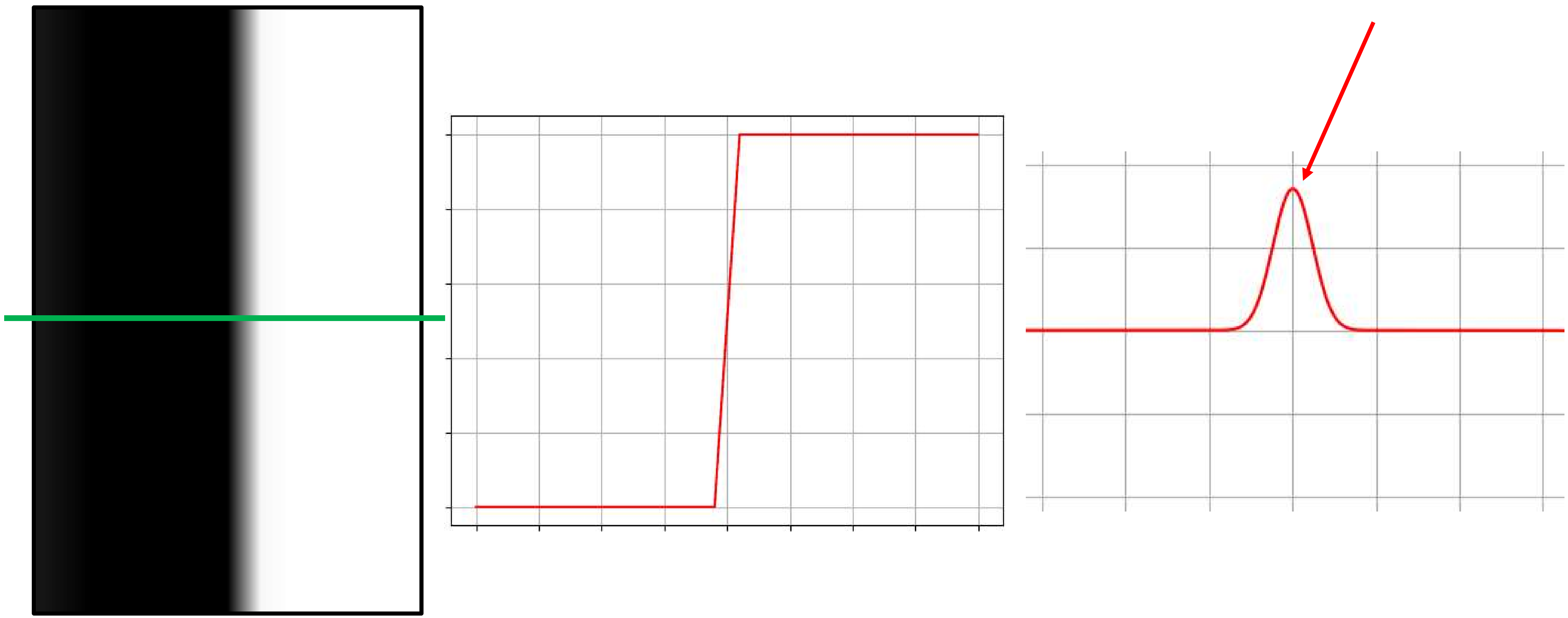
- Edges appear as steep cliffs in a 3D image-as-a-function plot



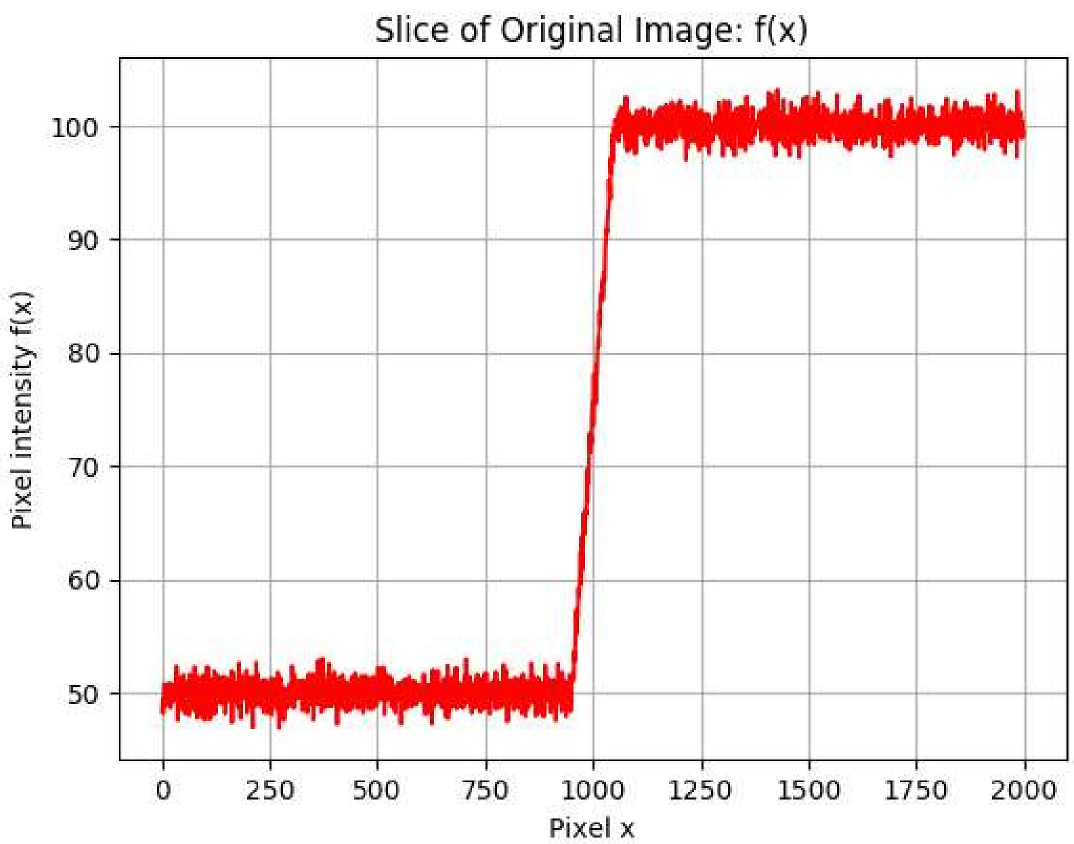


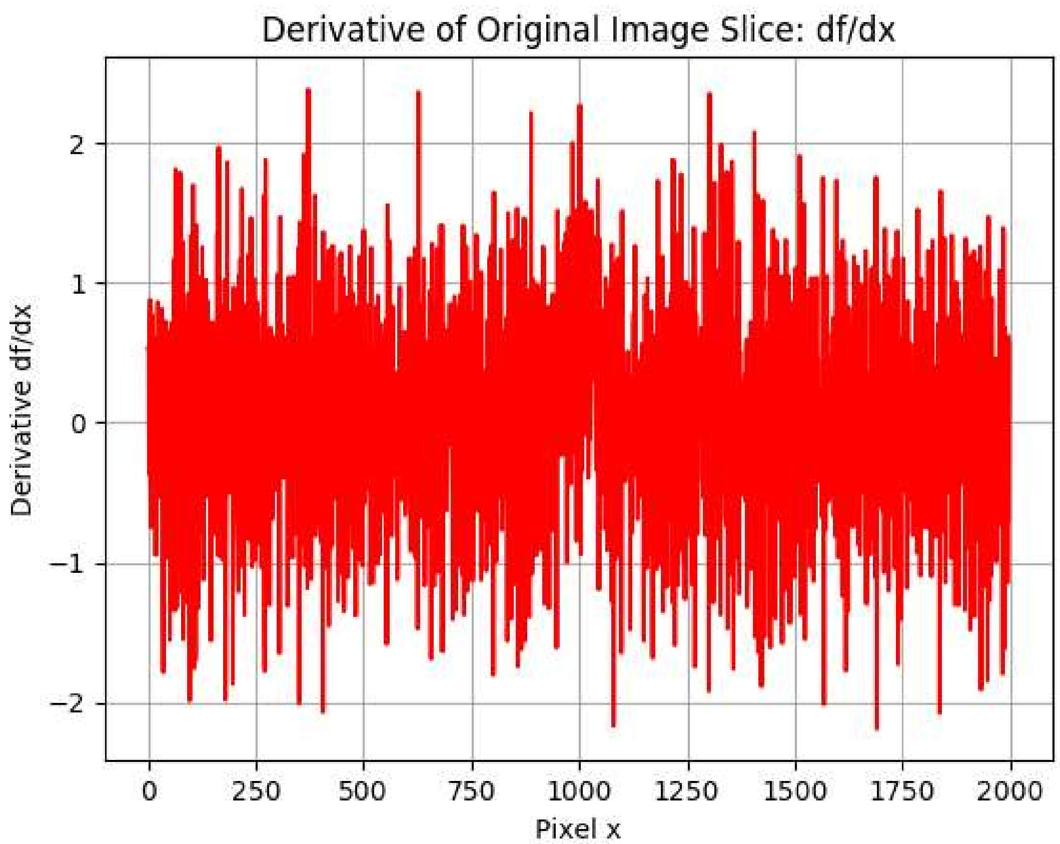
- Change in images is described by derivatives
- An edge is an instance of rapid change in the image intensity (i.e. pixel values)

Image | Intensity (Pixel Value) | 1st Derivative



- In reality, performance can be limited by presence of noise
- This results in positive & negative derivatives

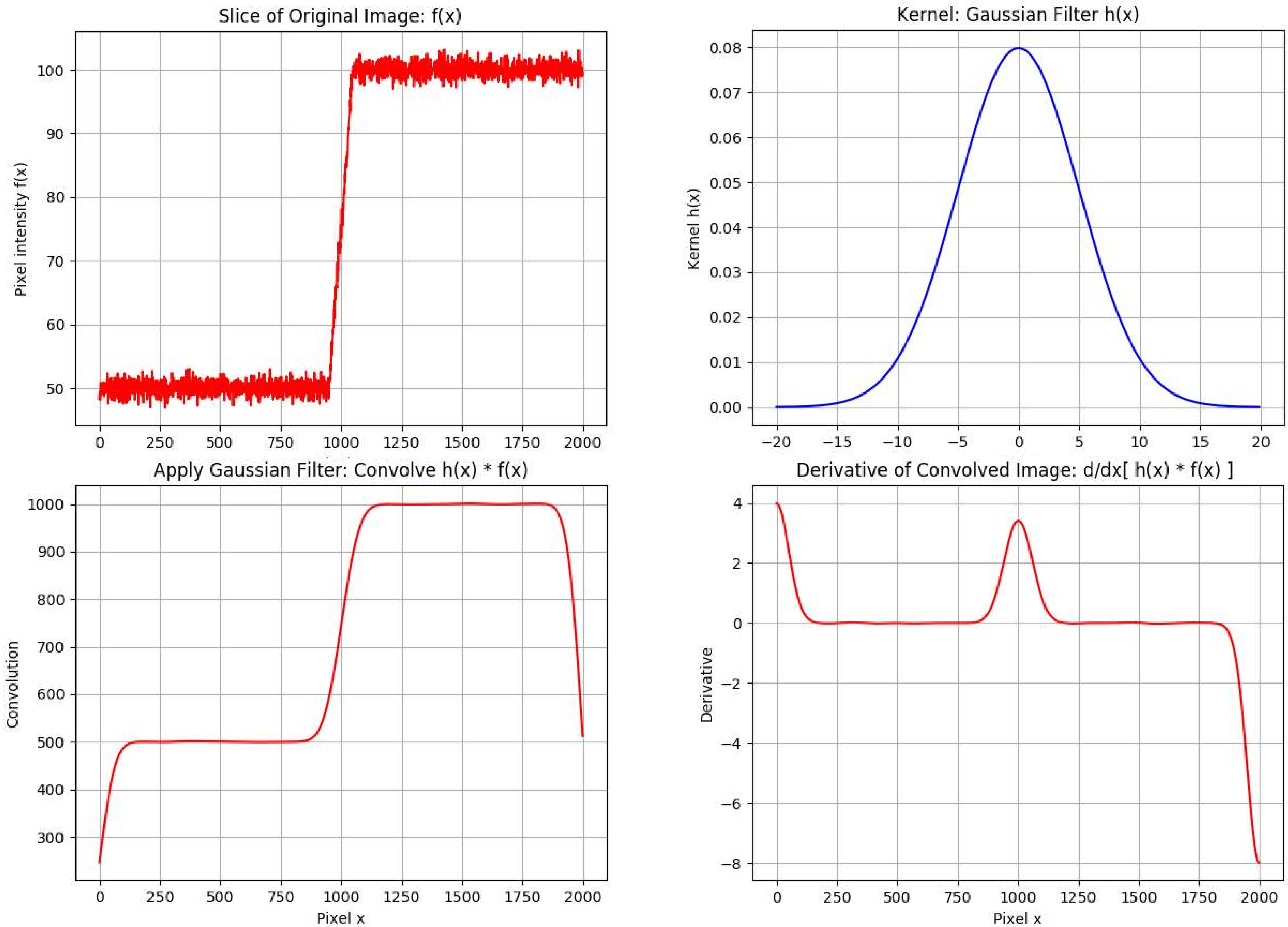


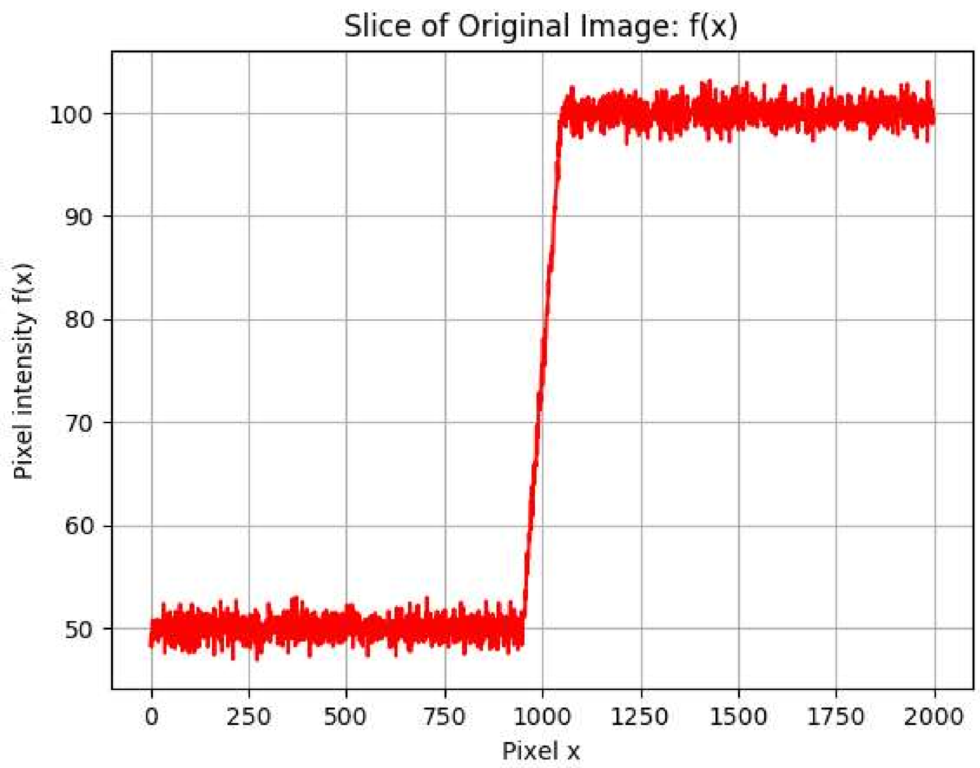


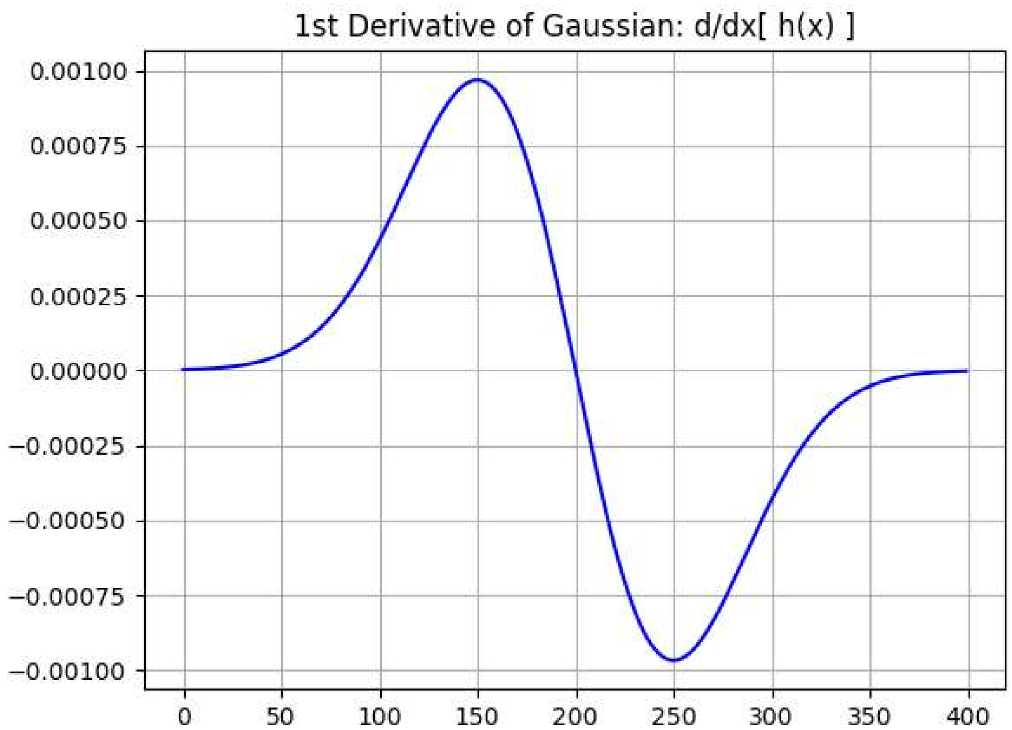
GitHub: edgedetection.py

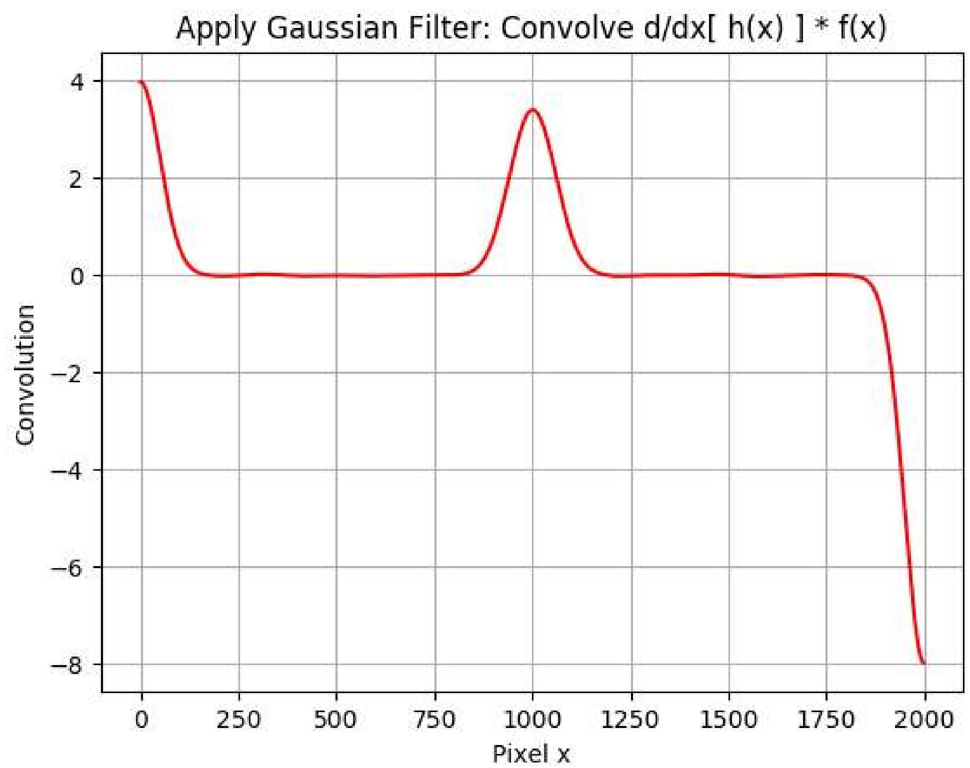
- Solution:

1. Apply smoothed gradients (filtering)
2. Find peaks

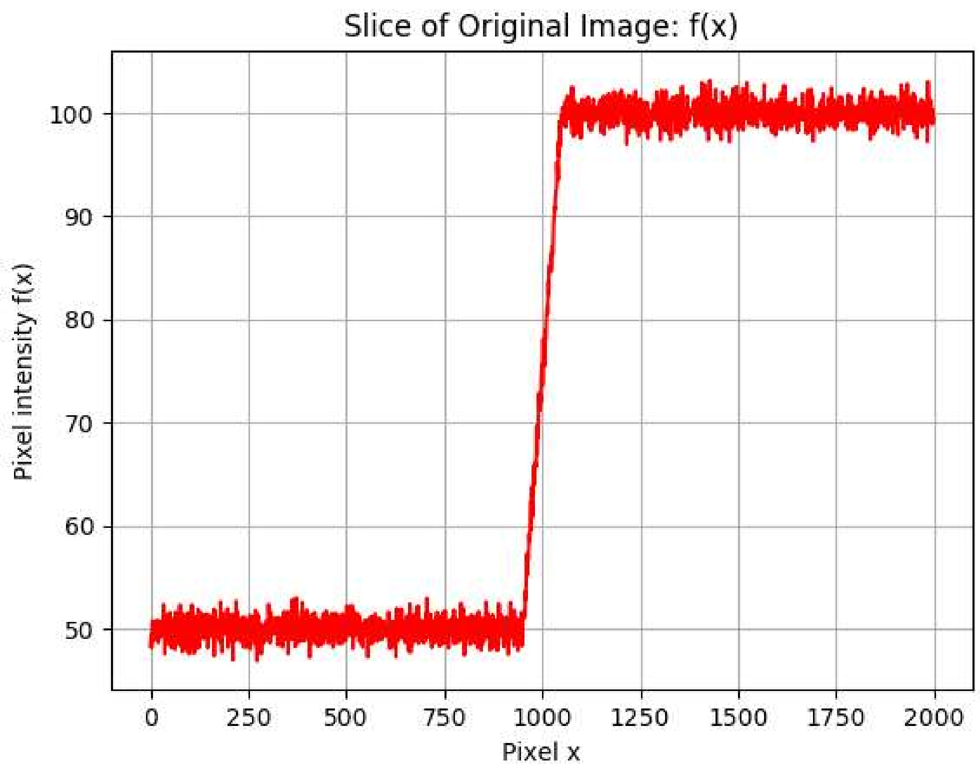

Slice of Original Image: f(x)
Pixel intensity f(x)
Kernel: Gaussian Filter h(x)
Kernel h(x)
Apply Gaussian Filter: Convolve h(x) * f(x)
Convolution
Pixel x
Derivative of Convolved Image: d/dx[h(x) * f(x)]
Derivative
Pixel x

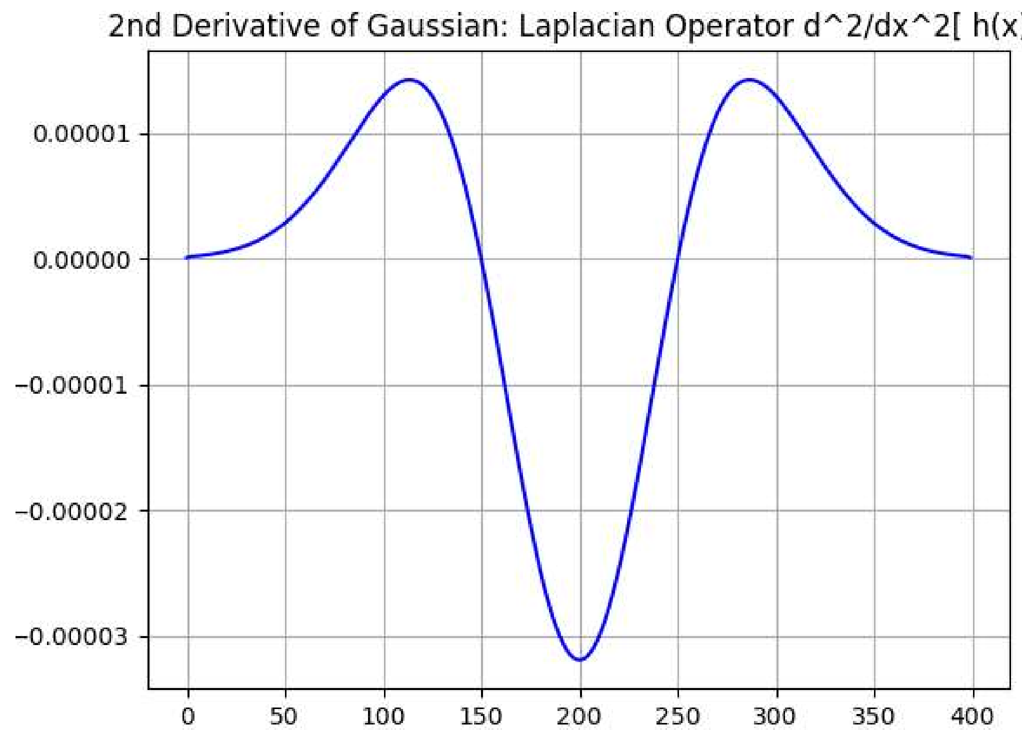




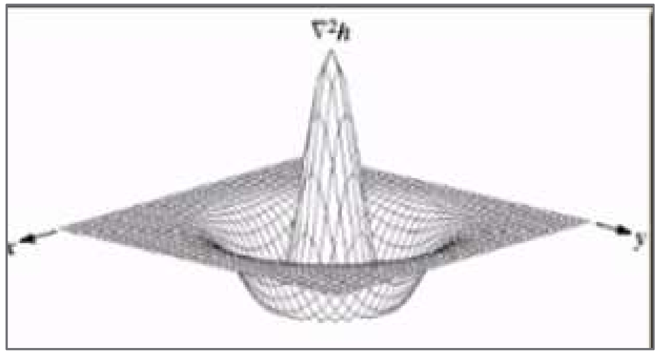


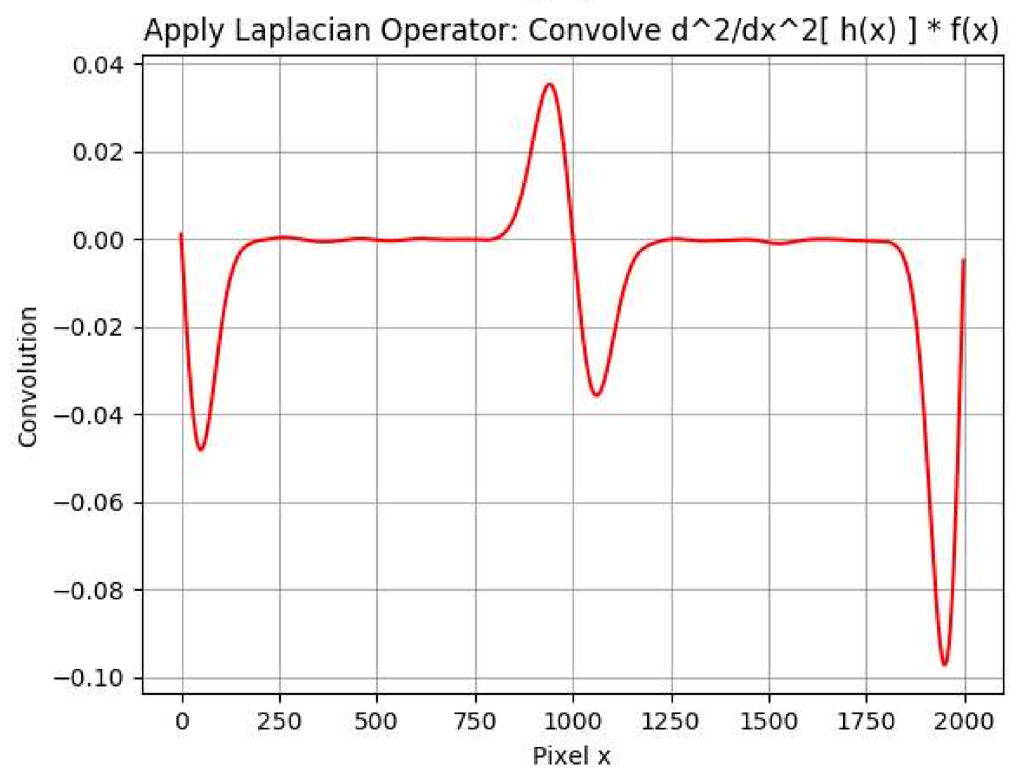
- Opportunity here to reduce number of required operations
- Important for fast processing

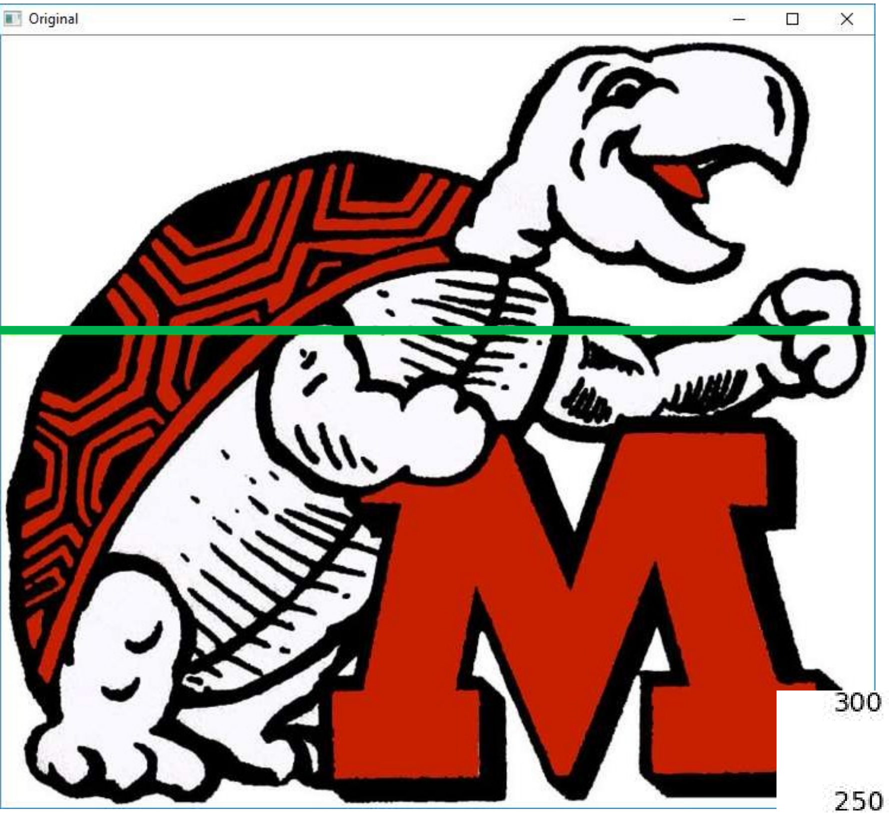


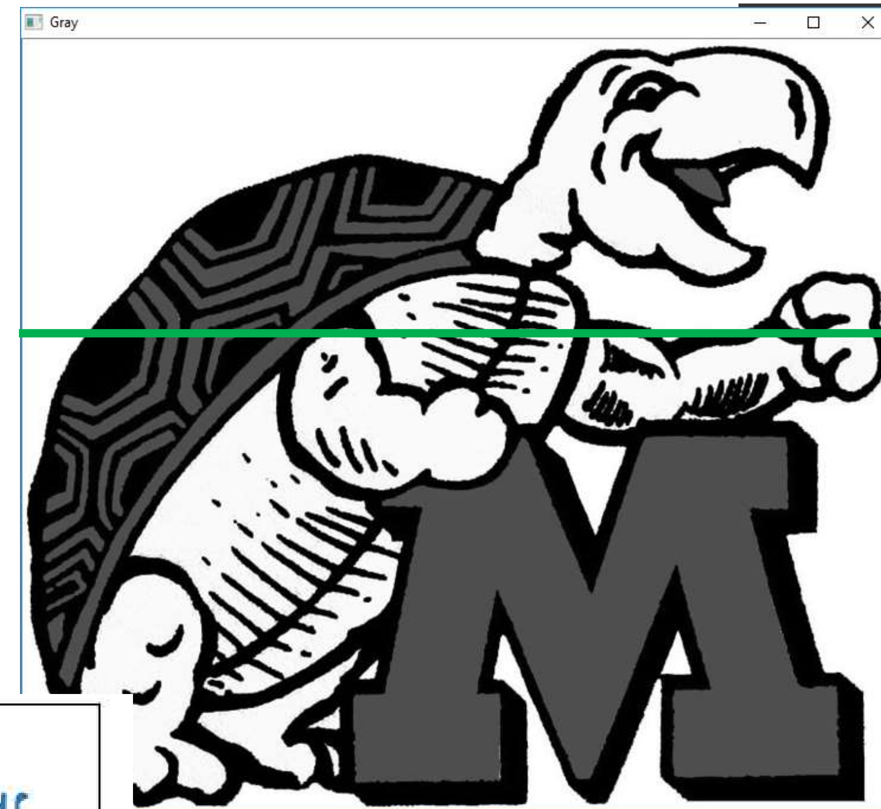


- Laplacian operator: second derivative of Gaussian
- Zero-crossings are edges



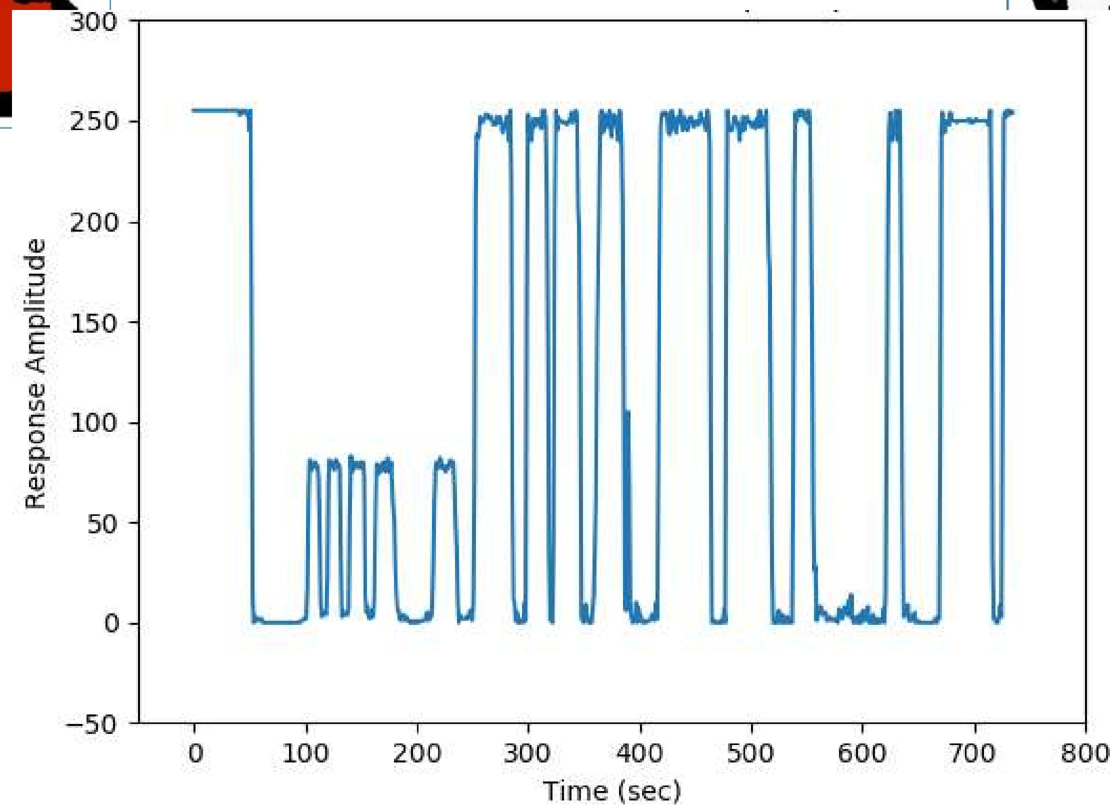


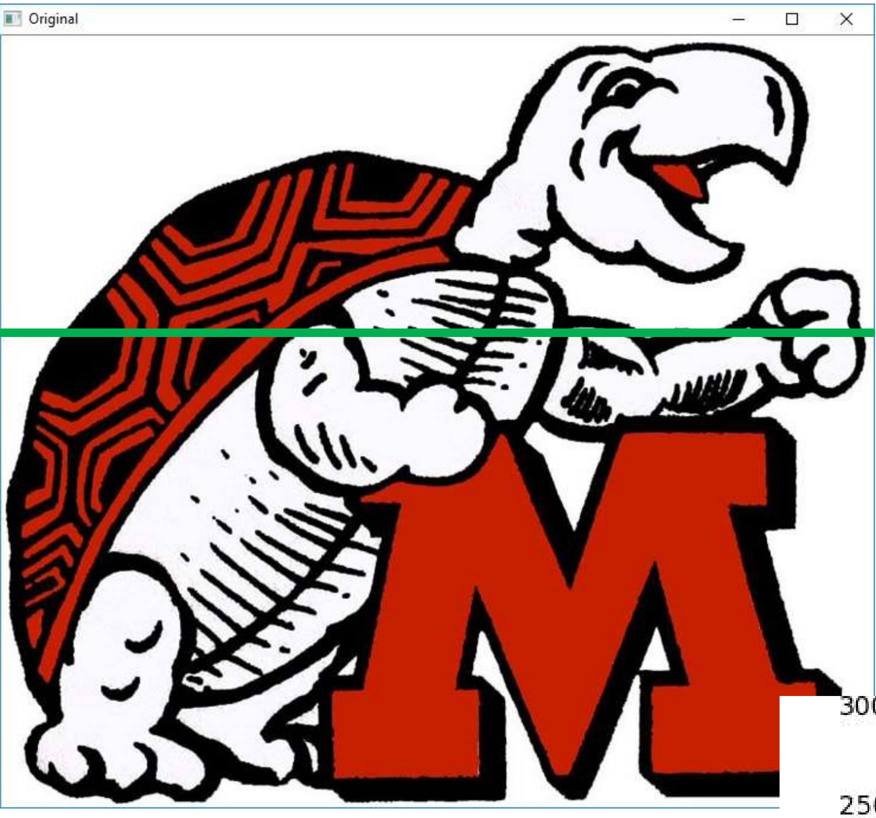


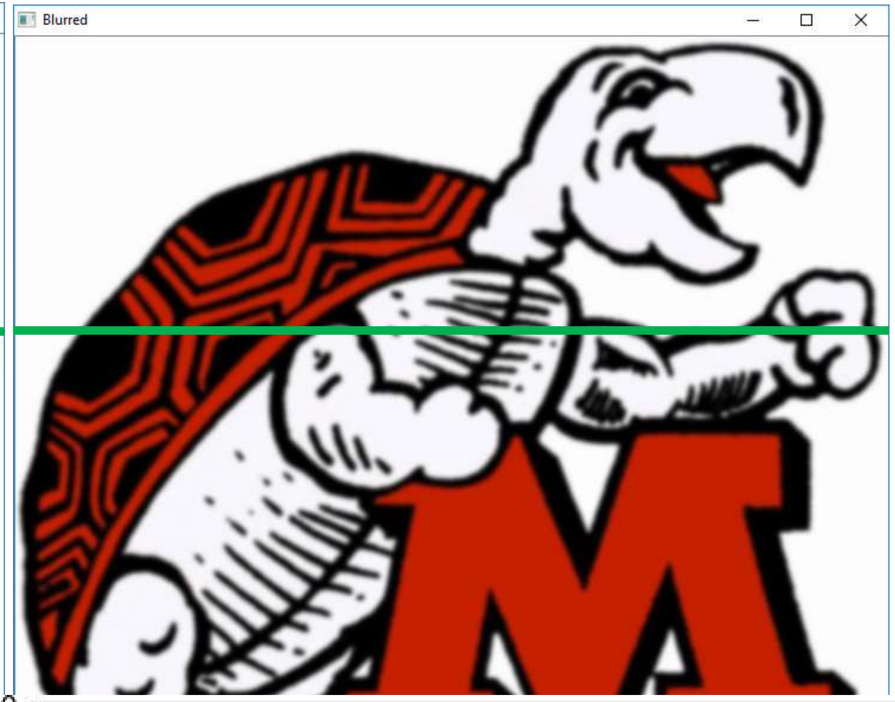


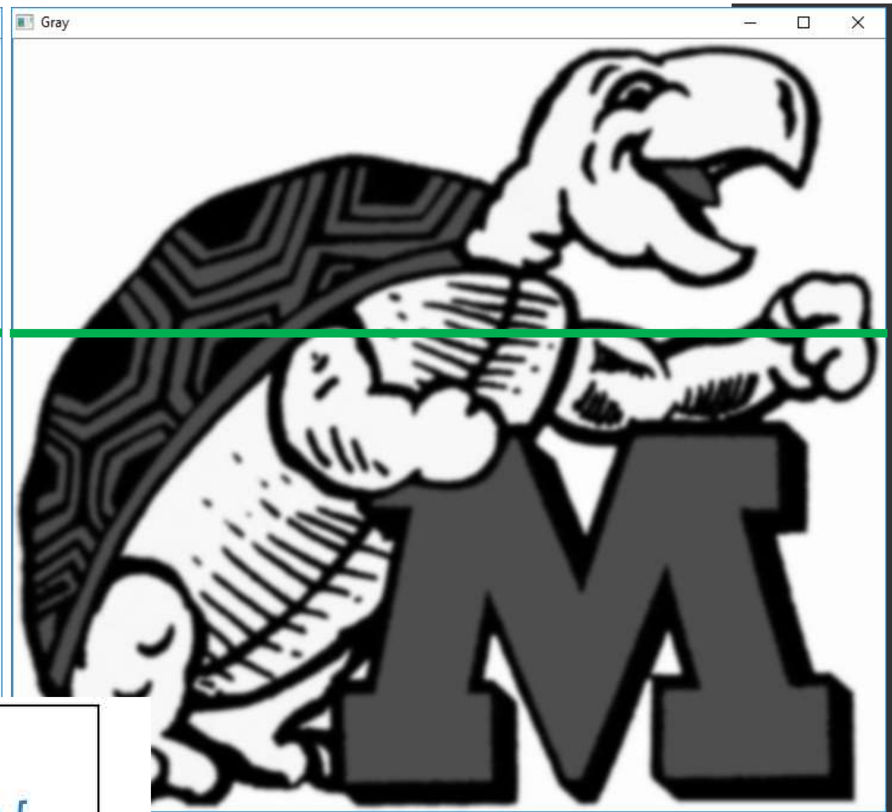
- Plot of **green** row

- Original signal contains high-frequency content





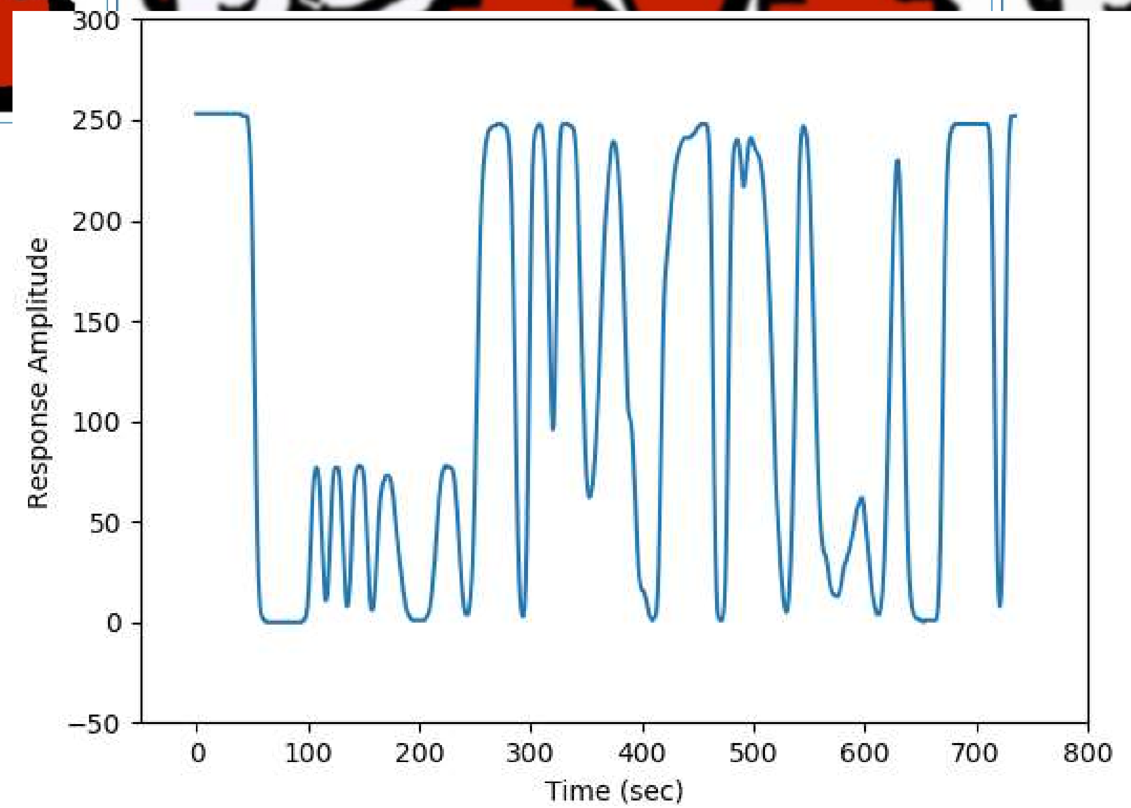




- Plot of **green** row

- Blurring removes high-frequency content

- Easier to identify edges



H

$*$

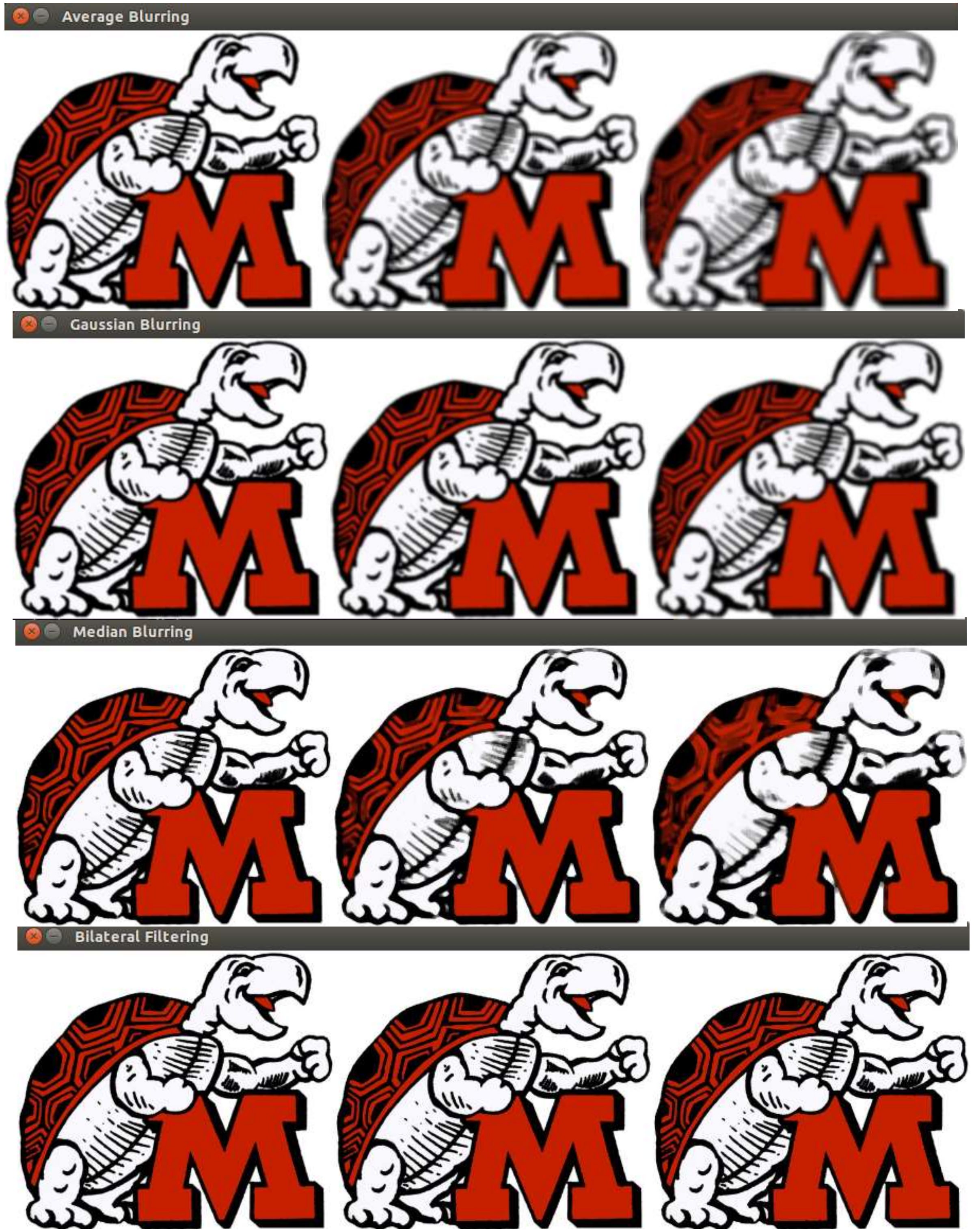
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	0	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	0	0	0	0	0	0	0
0	0	90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

F

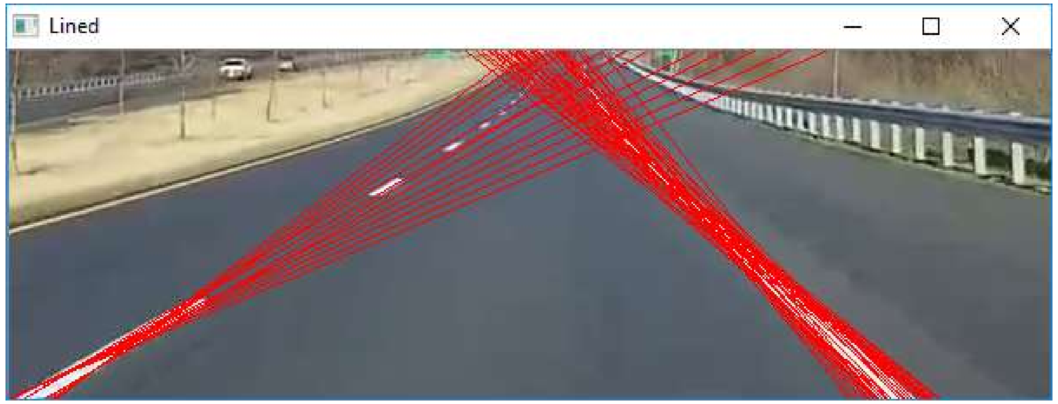
$=$

	0	10	20	30	30	30	20	10	
	0	20	40	60	60	60	40	20	
	0	30	60	90	90	90	60	30	
	0	30	50	80	80	90	60	30	
	0	30	50	80	80	90	60	30	
	0	20	30	50	50	60	40	20	
	10	20	30	30	30	30	20	10	
	10	10	10	0	0	0	0	0	

G



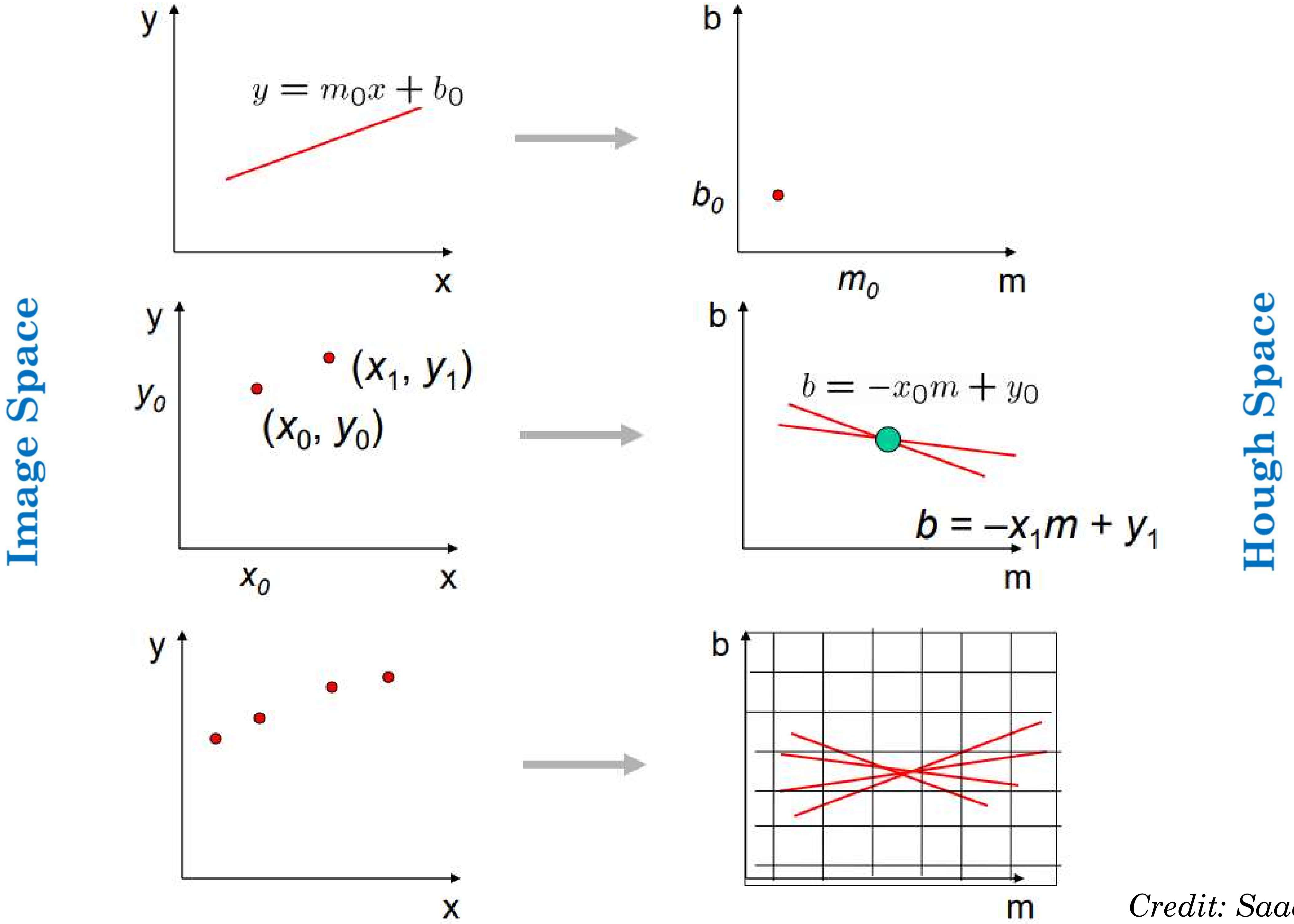
Lane Detection



- Hough Transform to identify lane lines

```
# Perform full Hough Transform to identify all possible lane lines
def line_image(img):
    return cv2.HoughLines(img, 1, np.pi / 180, 30)
```

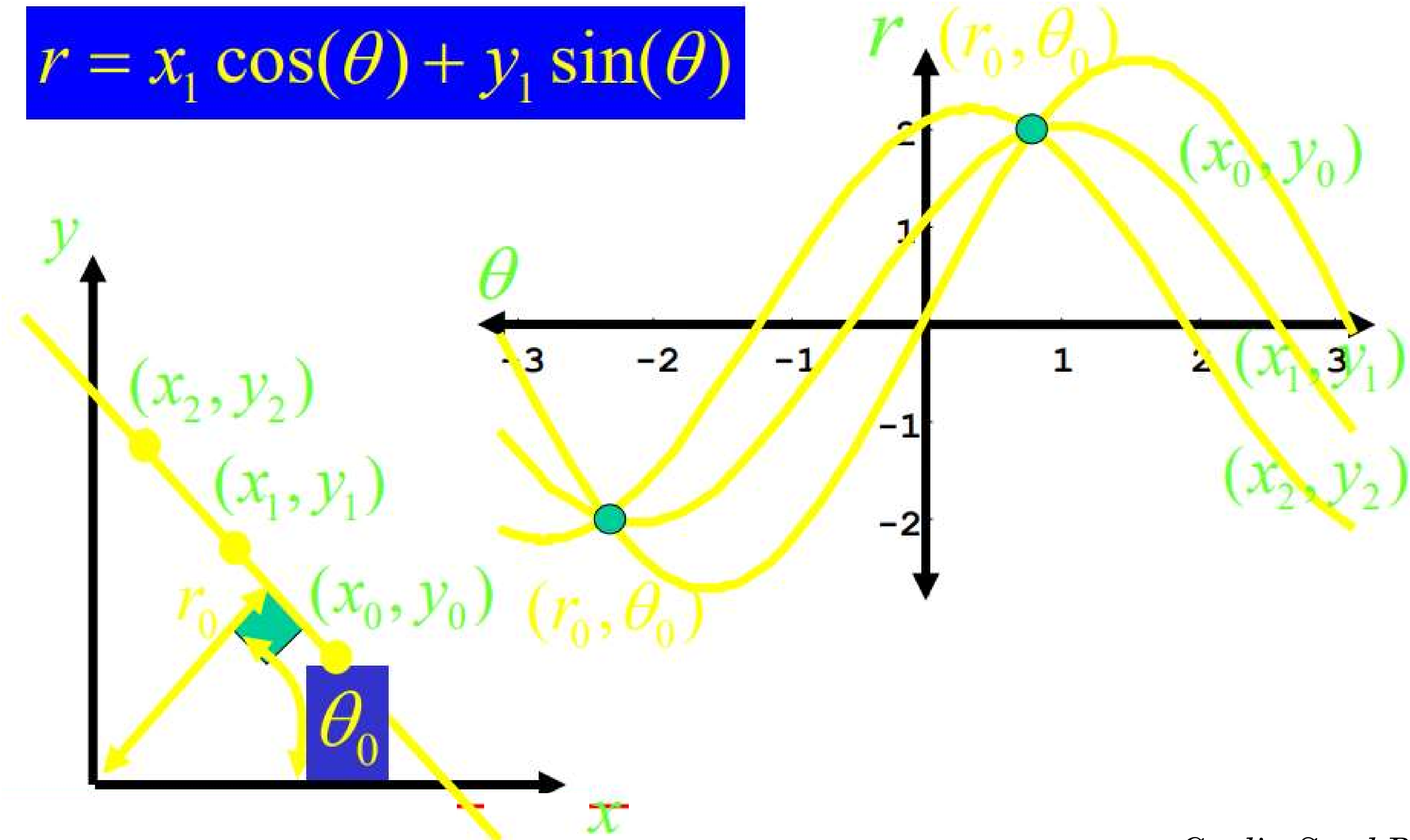
```
lines = line_image(edged)
```



Credit: Saad Bedros

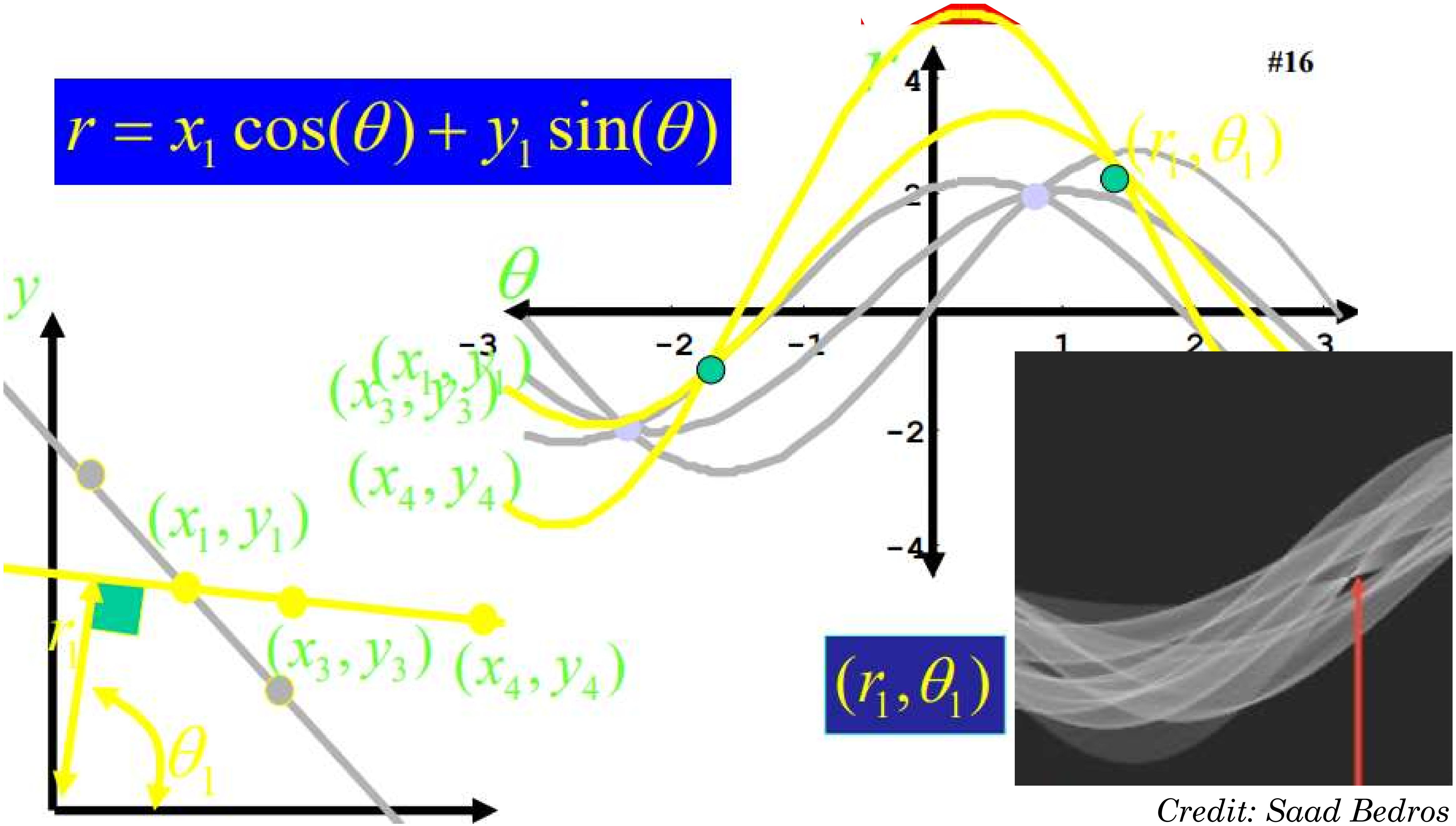
- Issues with vertical lines (infinite slope)

$$r = x_1 \cos(\theta) + y_1 \sin(\theta)$$



Credit: Saad Bedros

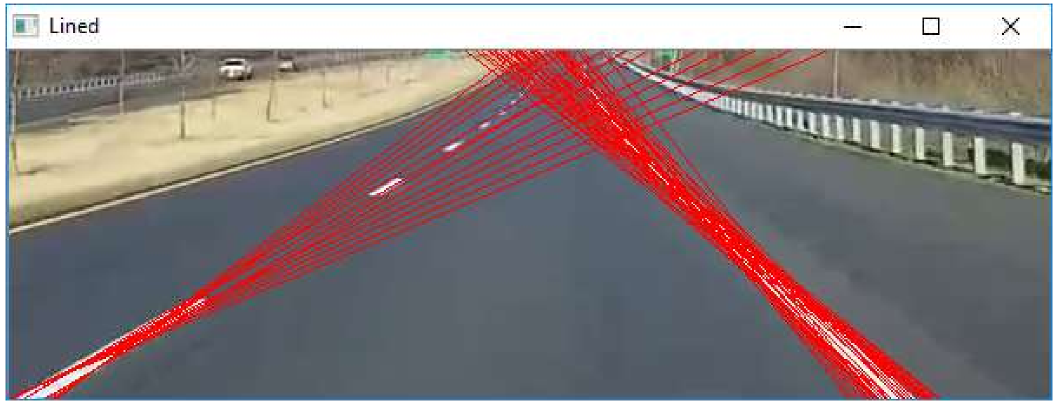
- Issues with vertical lines (infinite slope)



Credit: Saad Bedros

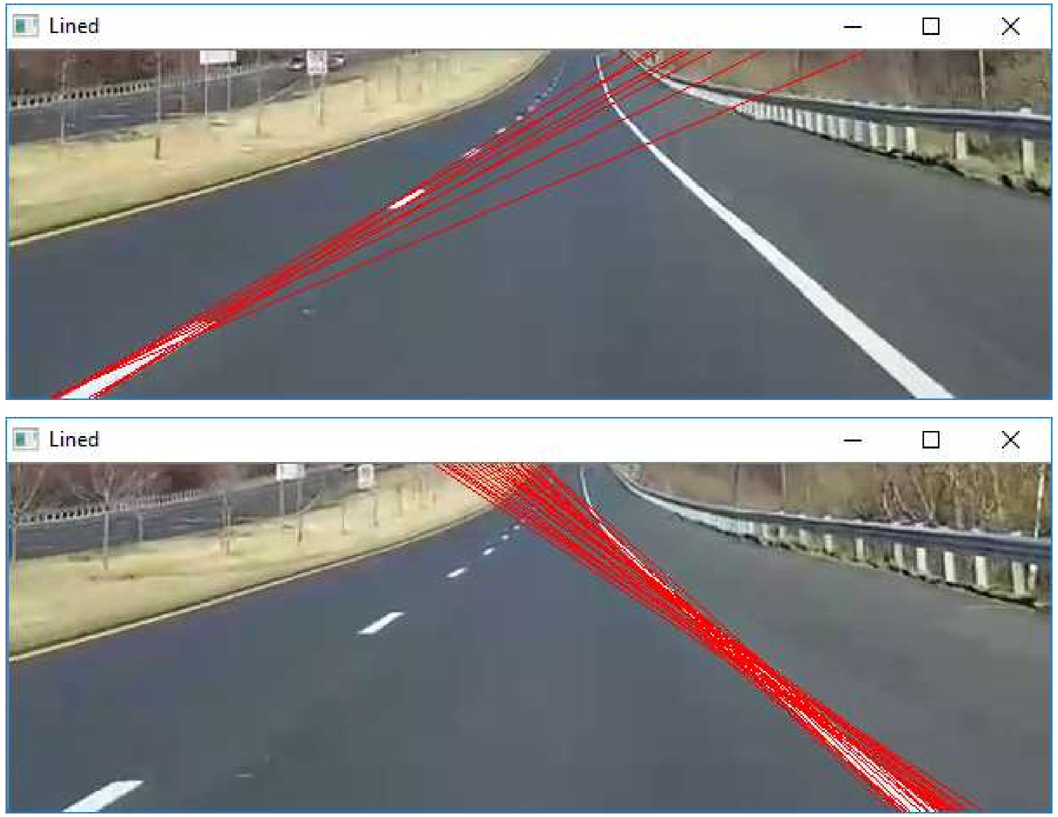
Lane Detection

- Ensure minimum one line has been identified
- Loop through all lines found in Hough Transform



Lane Detection

- Evaluate each row of lines array
- Identify left vs. right lane lines via angle

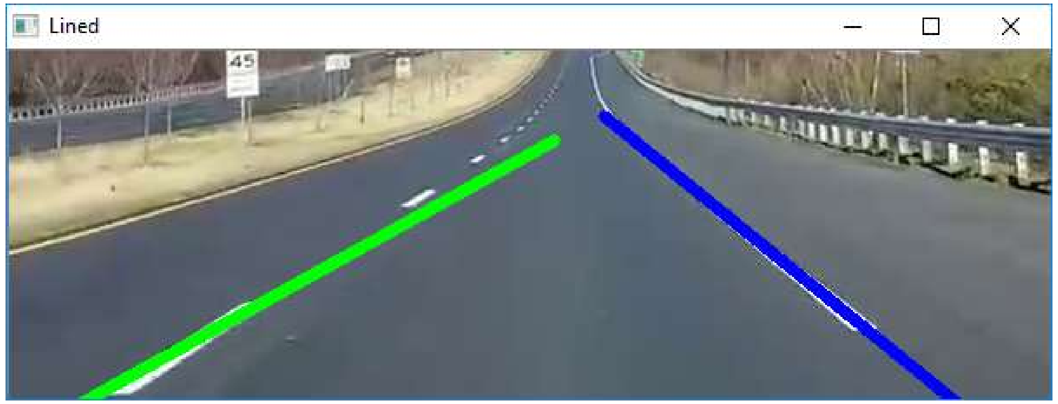


Lane Detection

- Statistics to identify the median lane line
- Linear regression & plot lane line atop original snip

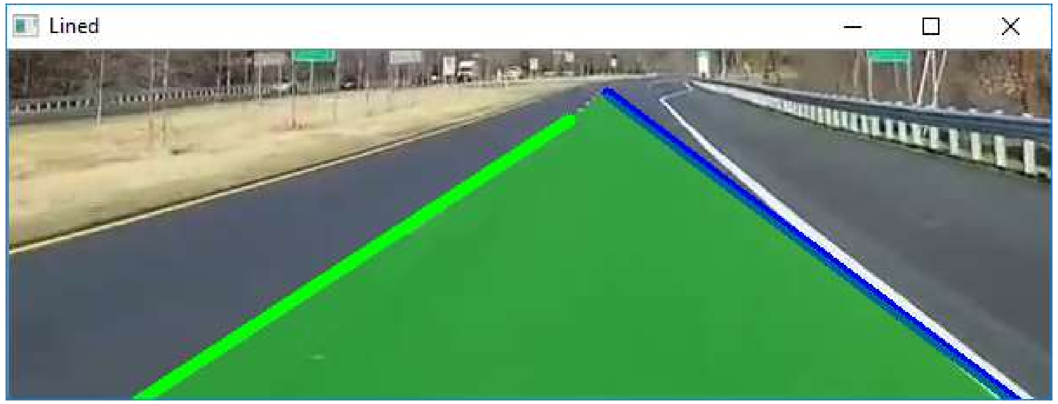
Lane Detection

- Statistics to identify the median lane line
- Linear regression & plot lane line atop original snip



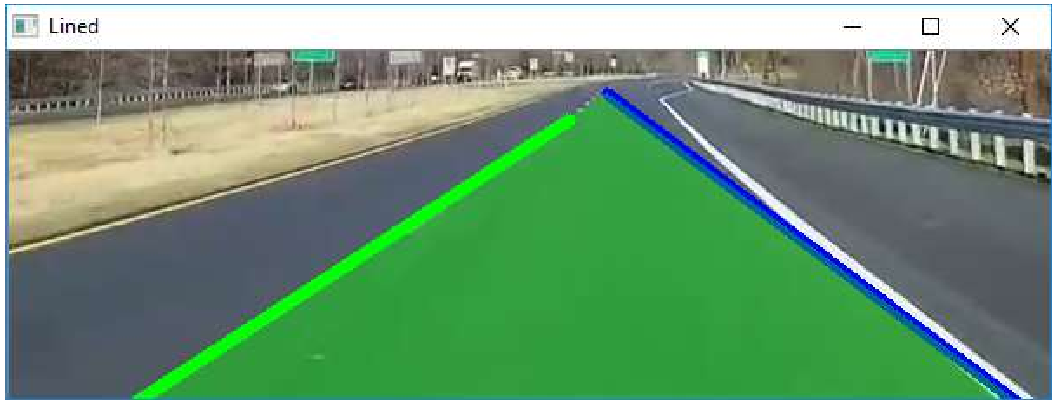
Lane Detection

- Plot semi-transparent lane identifier



Lane Detection

- Plot semi-transparent lane identifier
- Shown here without lane lines



References

- *OpenCV Tutorials*
  - http://docs.opencv.org/2.4/doc/tutorials/tutorials.html
- *Practical Python and OpenCV*, Rosebrock 2016
- *Grayscale to RGB conversion*
  - https://www.tutorialspoint.com/dip/grayscale_to_rgb_conversion.htm
- *Image Kernels Explained Visually*
  - http://setosa.io/ev/image-kernels/
- *Hough Transform*
  - https://www.youtube.com/watch?v=uDB2qGqnQ1g
- *Hough Transform & Thresholding*
  - http://me.umn.edu/courses/me5286/vision/Notes/2015/ME5286-Lecture9.pdf
- OpenCV Python Tutorial – Find Lanes for Self-Driving Cars
  - https://www.youtube.com/watch?v=eLTLtUVuuy4&feature=youtu.be