COMPUTER VISION LAB

Lane Detection

We will do a lane detector… by hand!

The algorithm is VERY easy to understand but has several steps.

Tackle each step and print your result.

The actual Algorithm

1. **Open the video file "Lane Detection Test Video 01.mp4" and play it!** This code should work:

```python
import cv2


cam = cv2.VideoCapture('Lane Detection Test Video 01.mp4')

while True:

    ret, frame = cam.read()

    # ret (bool): Return code of the `read` operation. Did we get an image or not?
    #             (if not maybe the camera is not detected/connected etc.)

    # frame (array): The actual frame as an array.
    #                Height x Width x 3 (3 colors, BGR) if color image.
    #                Height x Width if Grayscale
    #                Each element is 0-255.
    #                You can slice it, reassign elements to change pixels, etc.

    if ret is False:
        break

    cv2.imshow('Original', frame)

    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

cam.release()
cv2.destroyAllWindows()
```

Notes:

a. **cv2** is the **OpenCV** library

b. **VideoCapture** objects can get video from files or webcams!

c. **VideoCapture.read()** returns whether **cv2** was able to get a frame, and the actual frame (which is a Numpy array)

   Video processing works by editing each frame as they come, inside the **while** loop is the code that will be applied to each frame.

d. The frame is a regular **Numpy** array (documentation for methods)

   The elements are of type **unsigned char** (or **uint8**). We will need to specify the type of element for new arrays and for some methods.

e. You can index **a Numpy array** like this **arr[row, column]**. Slicing works like normal: elements from every row until the 5th and from columns 2 to 4 (excluding column 4) is written like this: **arr[:5, 2:4]**.

You can add **numbers** to **Numpy arrays** (**n + my_array**) to add the number **to each element**, same with **multiplication** with a number **etc.**

You can **also add 2 Numpy arrays** together (**arr1 + arr2**) to add them **element by element**, same with **multiplication etc**.

f. **cv2.imshow(title_of_window, frame_array)** displays an image.

g. **cv2.waitKey(n)** waits **n** ms for a key to be pressed and returns the code of that key.

cv2.waitKey(n) & 0xFF gives the **ASCII** code of the letter (so the if is executed when we press "q").

2. **Shrink the frame!**

We don't need a full HD image.

More pixels to process = more work = slower execution = bad.

Shrink the frame until you could fit around 12 separate windows of that size on your screen.

HINTS:

a. To resize use **cv2.resize(frame, (new_width, new_height))**, it returns the resized frame.

b. To get the size of the frame use **frame.shape** (tuple of (height, width))

3. **Convert the frame to Grayscale!**

We don't need the colors. Too much work. Get rid of them.

Think about how to convert a 3-tuple of colors to a single "color" because you will have to explain it even if you use a **cv2** function.

HINTS:

a) There is a **cv2** function that does exactly this (**cvtColor**)

OR

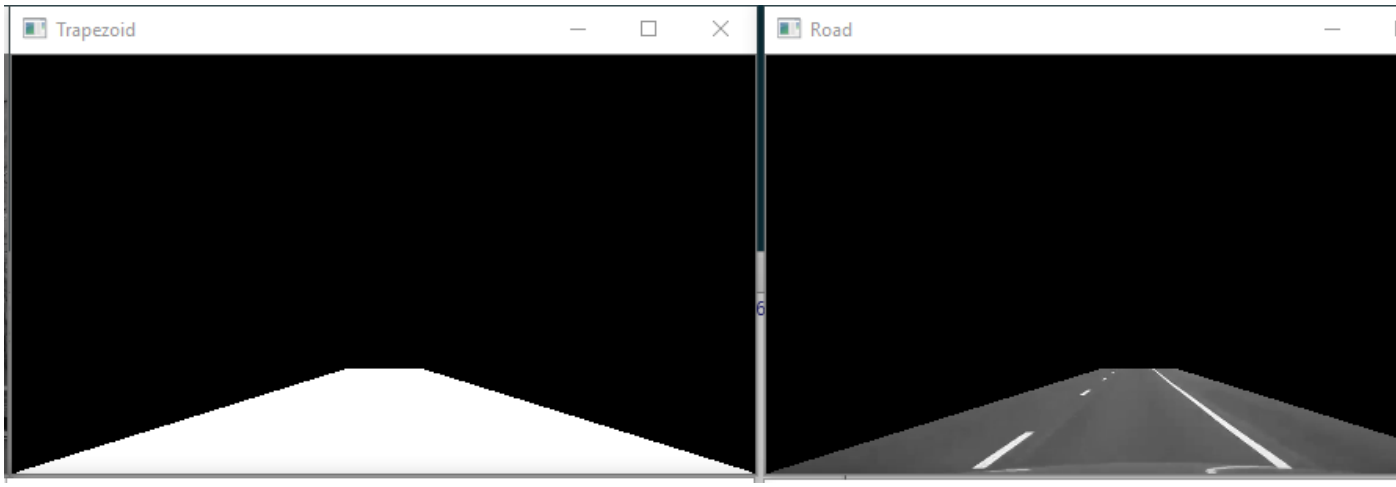
a) You could create a **separate blank frame** and set each pixel manually to be a shade of gray according to the color version of the image.
Color images are 3D matrices because each pixel/element is a 3-element vector
Grayscale matrices are 2D matrices because each element is simply a shade of gray between black and white (0 and 255)

b) To create a new frame you can use **np.zeros((width, height), dtype = np.uint8)** to create an array full of 0's (completely black), the 0's being **C-style unsigned char**s.
(here is where we need to specify that **uint8** data type because we need the array to be an array of **unsigned char**s)

c) Now iterate over it with a couple of **for loops** and set each pixel manually according to the colored frame
(or you could try an easier way by taking whatever you did to the individual color values to turn a pixel into a shade of gray and applying that method directly to the color image).

4. Select only the road!

The road is almost always in the lower part of the image and almost always in the shape of a trapezoid. We can use this knowledge to select only the road.



a. To do this you need to create a black frame of the same size as your main frame but full of 0's (using **np.zeros** explained in ex. 2 and **frame.shape** explained in ex. 1).

Then create a trapezoid shape in it made of 1's like in the left example.
NOTE: 1 is very close to 0, which is absolute black, if you try to imshow the trapezoid directly it will just show a black image. There is a hint below on how to display an image like the left example.

i. To create the trapezoid you simply need the coordinates of the corners, in trigonometrical order (upper right, upper left, lower left, lower right).
Each of corner point is a normal tuple of 2 ints **point = (x, y)**.

The coordinates can be fractions of the image dimensions.
For example the **y** of the **upper left** corner might be just a little below the middle of the image.
In this case that **y** might be **y = int(height * 0.55)** since the top is 0.0 and the bottom is 1.0, so 0.55 is just a bit lower than the half-point.

Compute the 4 coordinates of the trapezoid in 4 variables:

```
51      upper_left  =
52      upper_right =
53      lower_left  =
54      lower_right =
```

ii. Once you have the bounds of the trapezoid (**(x, y)** coordinates of the corners) place them in a Numpy array in the specified order: **np.array([pt1, pt2, pt3, pt4], dtype = np.int32)**.
This array is an **int32** array since they are not pixels of an image but coordinates of

points.

iii. Once you have the array of the points of the trapezoid you should draw them onto an empty frame.
Create an empty black frame using **np.zeros** explained in ex. 2 and **frame.shape** explained in ex. 1.
Then use **cv2.fillConvexPoly(frame_in_which_to_draw, points_of_a_polygon, color_to_draw_with)** to draw the trapezoid onto the frame.
The points are a Numpy array as defined at **ii.**
The color is going to be 1 since we want an image of 1's and 0's.
Now display the trapezoid using **cv2.imshow**.

HINT: To display the image (like the example on the left, only a white trapezoid) you need to do **cv2.imshow("Trapezoid", trapezoid_frame * 255)** since we have an array of 0 and 1 the color 1 is VERY close to pure black (0), and 255 is pure white (and clearly visible).

b. After you have an array of **all 0's** and **some 1's** in the shape of a trapezoid you can **multiply** each element in the grayscale frame with the corresponding element in the trapezoid frame.

HINT: To multiply 2 matrices element-by-element you can simply use the multiplication operator **mat1 * mat2**.

Once you do that, if you display the image it should look like the example on the right, with only the street visible. Note that you might have to adjust the proportions of the trapezoid to select only the street (you may not want to use 0.55 for the y, maybe 0.57 is better, or 0.52, etc.). At the end *no grass should be visible* on the edges of the street.

5. Get a top-down view! (sometimes called a birds-eye view)

We want to detect the lane markings on the street, and distant markings are just as important as closer ones! However, in the image they are smaller because they are farther away.
To make the far away markings just as significant for our algorithm as the closer lane markings, we need to make them equally large.
So, once we have the street "cropped" we need to "stretch" it over the entire screen.
Simply imagine "grabbing" the top corners of the trapezoid and pulling them up and apart from each other so that you "stretch" the trapezoid until you get a shape that covers the entire screen.

As you can see, the top half is more blurry because the top part of the trapezoid is pretty small, it doesn't have many pixels, and those few pixels were stretched until they covered the entire top half of the screen, resulting in a more blurry top half. This is normal.

To do this "stretching" we only need to call 2 functions!

a. You need the **corners of the area of the screen you want to stretch** (in this case the coordinates of the trapezoid) and the **corners of the area you want to stretch it to** (in our case the whole screen).

You already have the coordinates of the trapezoid corners (the bounds of the trapezoid) in trigonometrical order as a Numpy array from the previous exercise.

The coordinates of the screen corners are simple to do. The upper left point is **(0, 0)**, the upper right is **(frame_width, 0)** similar for the bottom corers. You still have the frame dimensions from **ex. 1**.
Don't forget to place them in *trigonometrical order starting with the top right corner*!

Once you have both sets of corners in trigonometrical order you need to convert them to **float32** since the stretching works with floats.
To do this you can simply do **trapezoid_bounds = np.float32(trapezoid_bounds)**. Similar with the frame bounds.

b. Now simply use **cv2.getPerspectiveTransform(bounds_of_current_area, bounds_of_area_you_want_to_stretch_to)**
In our case the current area is the trapezoid, so you will use the **Numpy array** with the trapezoid bounds from the previous exercise.

And the area we want to stretch it over is the entire frame so its bounds are the corners of the screen.

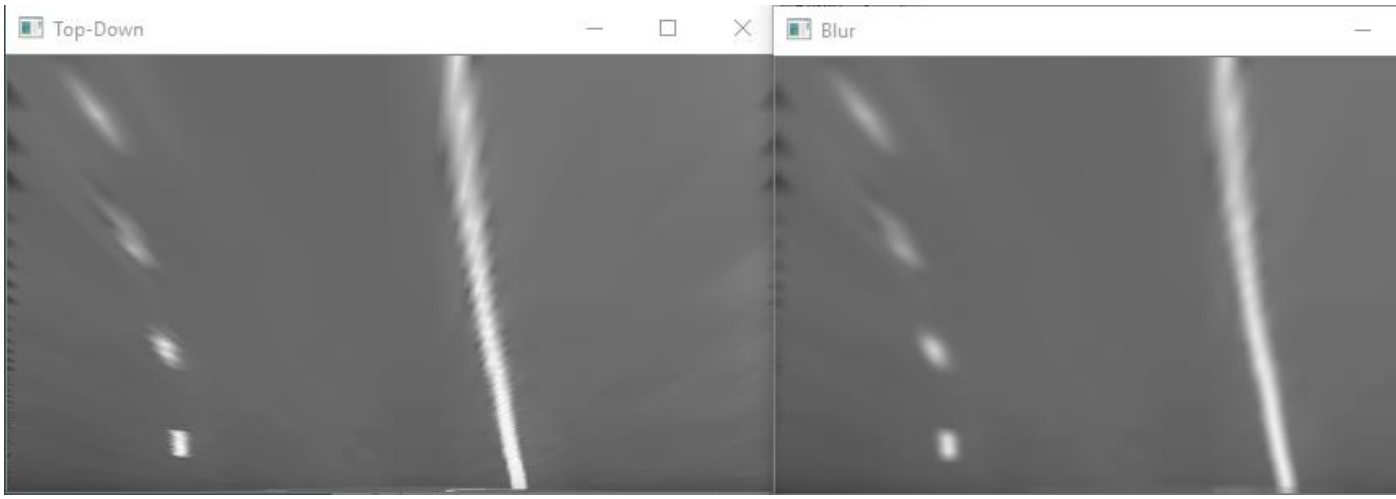
From **getPerspectiveTransform** you will get back a magical matrix that you can use to "stretch" the trapezoid.

c. This magical matrix will be used for the actual "stretching" using **cv2.warpPerspective(some_image, magic_matrix, (new_width, new_height)).**
In our case the image we want to "stretch" is the frame with the cropped street (right example from the previous exercise) and the dimensions are the same as every frame we have produced until now.

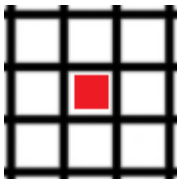
warpPerspective should return the "stretched" image like in the example.

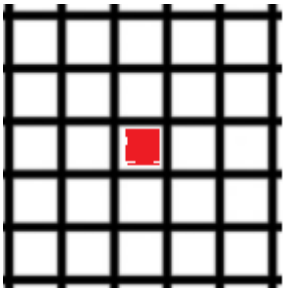
6. Add a bit of blur!

Just like with any picture no two pixels are exaaaactly the same, they are slightly different.
This can be a problem if we want to differentiate between 2 clear zones: *the lane markings* and ***not*** *the lane markings*.



To blur an image we need to make each pixel to be the average value of its neighbors.
If we use more neighbors (figure b.) the blur will be stronger (more blurry), if we use less neighbors (figure a.) the blur will be weaker ("clearer").

a) 

b) 

We use a simple **n x m** matrix (for example 3 x 3). We move this matrix over the frame so that each pixel will at some point be in its center (**n** and **m** must be odd numbers like 3, 5, 7, etc.) and that pixel become the average of the elements in the matrix.

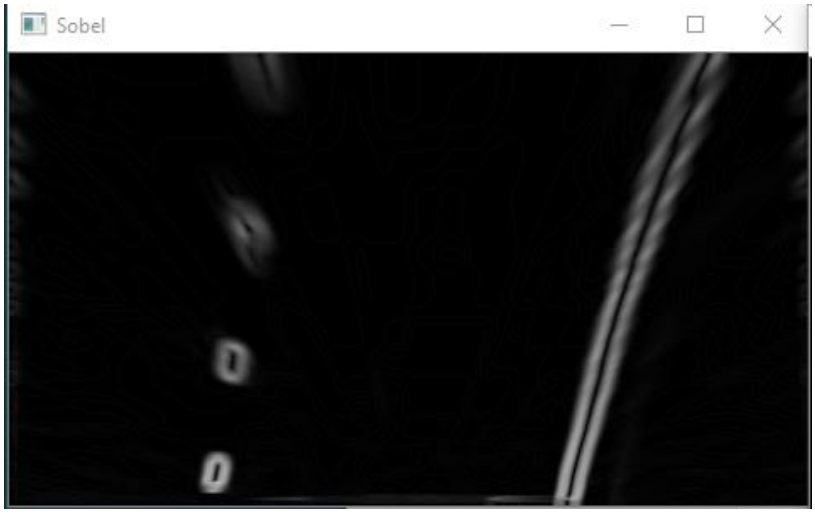
HINTS:

a. Use **cv2.blur(frame, ksize = (n, n))**.
Ksize is a tuple with the dimensions of the blur area (also called a "**kernel**").
Usually **ksize** is a small square matrix **like (3, 3), (5, 5), (7, 7)**.

cv2.blur should return the blurred frame.

7. Do edge detection!

There are A LOT of ways of doing edge detection, from matrix-magic to machine learning.
We will use an older and very simple method: **the Sobel filter**!



A filter is very similar to how we did blurring. The idea is like this (you won't need to code it, but you need to understand it):

First, we have a matrix, called a "**filter**". Let's say this one:

-1	0	1
-2	0	2
-1	0	1

Then we move this small matrix over the frame (using a **for** loop for example) so that each pixel in the frame gets to be in the center of this matrix at some point.
For each pixel, multiply each of the neighbors with the corresponding number in the matrix and then sum up the results, this becomes the new value of the center-pixel.

NOTE:

The pixels on the edges of the image do not have all the 8 surrounding neighbors so they won't ever be in the center of the filter.

The output image will be just a bit smaller than the input one because the pixels on the outer border of the image will be lost.

Demo here: http://setosa.io/ev/image-kernels/

With different filters we get different results.
The example filter above is a classic filter for edge detection, called the horizontal Sobel.
It detects horizontal differences in color very well.

If the 2 pixels in the middle left and middle right are kind of equal (they are not part of an "edge", which is usually a more drastic change in color) they will cancel each other out.
Suppose the left and right pixels in the middle row are both more or less some value **n**.
Then applying the Sobel filter they will be multiplied with **-2** and **+2** so we get **-2n** and **+2n**, and when we sum them up we will get almost 0.
This means that all areas where there is not a more drastic change in color… the frame will become black, and where there is a more pronounced change that part will light up like in the examples.

a. What you need to do is simply create the vertical and horizontal Sobel matrices:

```
sobel_vertical = np.float32([[-1, -2, -1],
                             [ 0,  0,  0],
                             [+1, +2, +1]])
```

The vertical Sobel has the change from "**-**" to "**+**" going from the top to the bottom while the horizontal Sobel goes from "**-**" to "**+**" from the left to the right.

The horizontal version is simply the vertical one transposed.

HINTS:

i. **np.transpose(my_matrix)** returns the transpose of a matrix.

b. Now that you have the 2 Sobel filters (vertical and horizontal) in 2 variables you need to **apply** each of them to the image separately.
This means that you will end up with one resulting frame for each of the 2 Sobel filters.

HINTS:

i. **cv2.filter2D(frame_as_float_32, -1, filter_matrix)** applies a filter to a frame and returns the result.
We need to convert the frames to **float32** because since the frames are **uint8** (**unsigned**) they don't play well with negative values.
ii. To see how to convert the frame (which is a Numpy array) to **float32** check out ex. **5.a.**
iii. You will need to apply the Sobel filters to 2 separate copies of the frame. Do **NOT** apply the 2nd Sobel filter over the result of the 1st one!

You should now have 2 frames (Numpy **float32** matrices) like the first 2 examples of this exercise.
You will see that each of the Sobel filters either detected all the vertical or all the horizontal edges that separate the street from the actual markings.

IMPORTANT (ONLY in case you want to display your images):

The matrices are **float32**.

You can only show **uint8** matrices as images.

To convert the **float32** matrices to **uint8** matrices (so you can display them) without losing quality you can use **cv2.convertScaleAbs(my_matrix)**.

So to show your current images pass it trough **cv2.convertScaleAbs(my_matrix)**. Think of this function as a „toString()".

Then you can pass the result into **imshow()**.

The next part of this exercise (**c.**) works with **float32** matrices.

If you used **cv2.convertScaleAbs(my_matrix)** to show the result of the Sobel filters make sure you also keep the original **float32** matrices too.

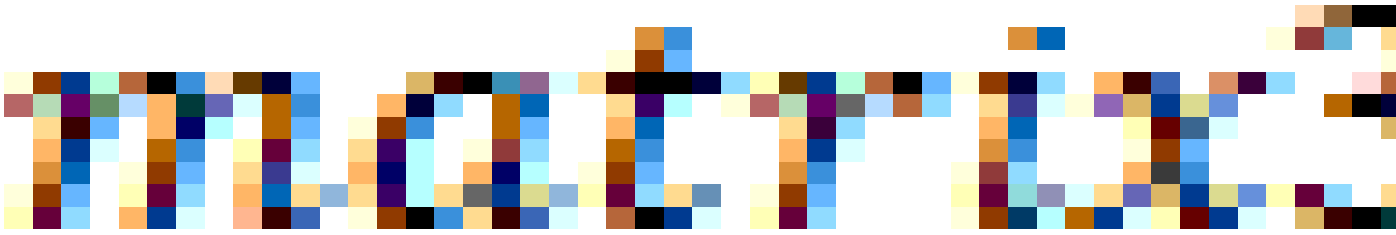
c. To combine the 2 images and get something like the final example you simply apply the following formula to the 2 frames you got during the previous exercise:
$matrix3 = \sqrt{(matrix1)^2 + (matrix2)^2}$.
It normally needs to be computed for each element (the first element of $matrix3$ is the result of applying the formula first element from each of the 2 resulting matrices from **b.** and so on…)

HOWEVER…
Numpy matrices overload their regular operators like addition, multiplication, power etc.
mat1 +/-/* mat2 and **mat1 ** n** perform these operations **element-wise.**
Adding/subtracting/multiplying each element of the first matrix with the corresponding element of the 2nd matrix, so doing matrix operations is **really** easy in Numpy.
Also for the square root you can use **np.sqrt(mat)**.

At the end apply **cv2.convertScaleAbs(my_matrix)** to the final matrix. We are done using **float32** for now.

You should now have a matrix that look like the final example for this exercise.

8. **Binarize the frame!**
This is one of the easier steps. It simply means to either convert each pixel to absolute black (0) or absolute white (255).



This is as easy as defining a variable called **threshold** (with some initial value like **int(255/2)**) and then iterating over the frame.
Each pixel below the threshold becomes absolute black, every pixel above the threshold becomes absolute white.

The threshold might need to be tuned a little (a bit higher or lower than 255/2) to get rid of some random pixels that are not part of the actual road markings.

You should now have an image like the example on the right.

HINT: There might be multiple ways of doing this using either **Numpy** or **OpenCV** (filtering, slicing, **cv2.threshold**), both of which are 100-1000+ times faster than for-loops.

9. Get the coordinates of street markings on each side of the road!

To get nice lane detection going we will need to divide the screen into the left and right half.
Each half will most likely contain only one of the 2 lane markings, left or right.

No example here as we will not produce an image juuust yet.

a. First we need to get rid of some more noise.
You may have noticed that there are some white pixels near the edges of the screen in the video you get by watching the result of **ex. 8**.
We need to get rid of those as we know it's just the edge of the road, maybe some random grass or markings for the edge of the road.

Make a copy of the frame (so we don't lose the initial frame) using **my_frame.copy()** and make the **first 5% and last 5%** of the columns of the frame be completely black (0).

HINT:

If you don't know how to get the first… 10% of the columns for example, you can do **width * 0.10** to get 10% of the total number of columns.

Also, you can assign a single number to entire parts of Numpy matrices: **my_matrix[2:5, 3:7] = 13** this will assign 13 to all the elements in the specified part of my_matrix.

Remember to convert **width * percentage** to int, you can't use floats as indices.

b. Now you need the **(x, y) coordinates** of each white point (because those are the points that are part of the road markings) in each of the 2 halves of the frame. So in the end you need to have these variables:

```
left_xs =
left_ys =

right_xs =
right_ys =
```

You can either do this by iterating over the image with a couple for-loops and storing the **(x, y)** (or **(col, row)**) coordinates. Remember that you need to have the coordinates of white points in the left and right halves of the image separately.

OR…

Instead of doing it “by hand” using for loops which are *slow*) you can use **np.argwhere**.
Argwhere returns a Numpy array with the indices of the elements you are asking for:

```
>>> x
array([[0, 1, 2],
       [3, 4, 5]])
>>> np.argwhere(x>1)
array([[0, 2],
       [1, 0],
       [1, 1],
       [1, 2]])
```

In this example we have an array, **x**, and we want the indices of all the positions in that that are **>1**.
As you can see, **np.argwhere** returns an **array** where every row contains the coordinates of a position in the initial matrix where the corresponding element satisfied our condition (**x>1**).
The coordinates are in the form **(row, col)**, which means the form of **(y, x)**.

To use **np.argwhere** in this exercise you will probably need to use slicing to separate the frame into the 2 halves and call **np.argwhere** on each half to get the coordinates of the points for each half
(*WARNING*: **np.argwhere** gives the coordinates as **(y, x)** not **(x, y)**).

Also since you are creating a new array from the right side of the image, the first pixel on the right side, which had **x = width//2** in the original frame, now has **x = 0** as it is the first pixel in its row. Hence you will need to add something like **width//2** to each **x** coordinate in the right half.

* * *

After you get the coordinates (in any of the 2 ways presented, loops or **np.argwhere**) you need to separate the **x** and **y** coordinates into 2 different arrays (if you placed them in a list by using the first method then construct a **Numpy array** from that list by using **np.array(my_list)**).

Whatever method you choose (loops or **np.argwhere**) you need to end up with 4 variables:

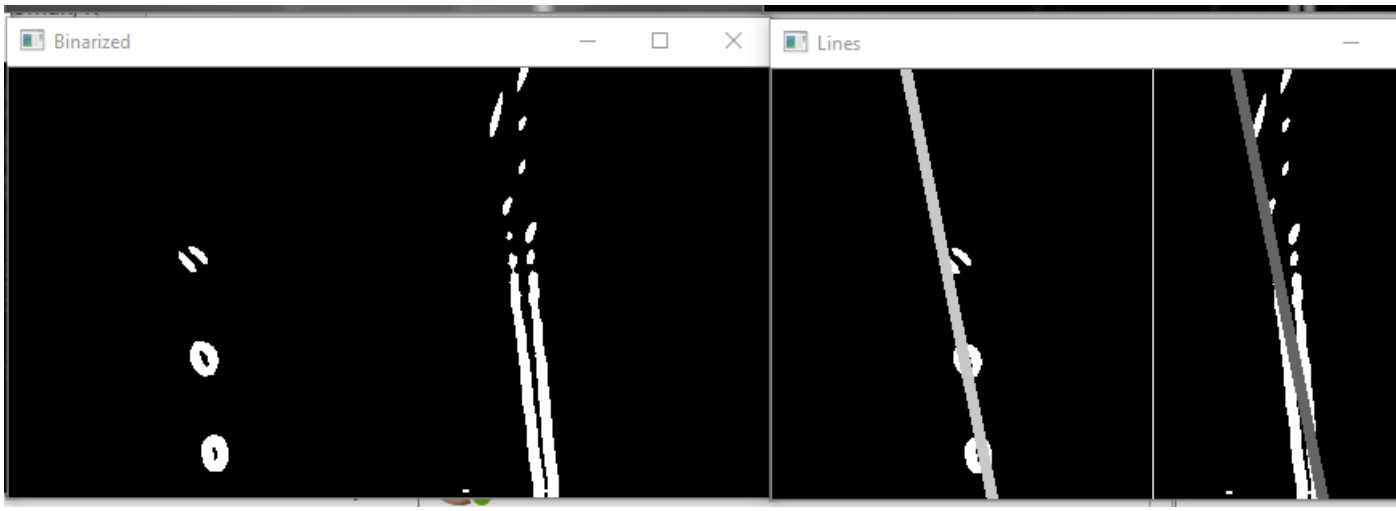
```
left_xs =
left_ys =

right_xs =
right_ys =
```

i. The **X** coordinates of the white points on the **left** side of the frame
ii. The **Y** coordinates of the white points on the **left** side of the frame
iii. The **X** coordinates of the white points on the **right** side of the frame
iv. The **Y** coordinates of the white points on the **right** side of the frame

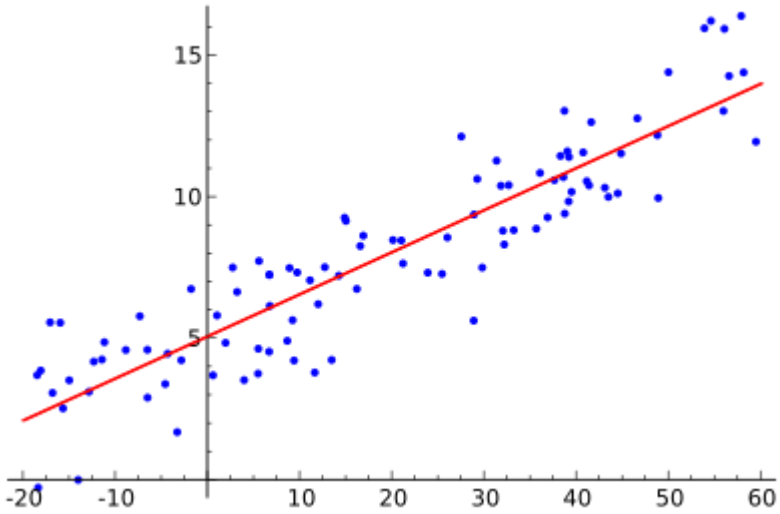
10. **Find the lines that detect the edges of the lane!**

Next we will find the lines that best fit the lane markings we have found.



A simple way to do this is for each half of the screen to just find the line that best passes through all the white points on that half of the screen.

Finding the line that best passes trough some set of points is called **regression**.



Don't worry, *we will not be doing the math behind it.*

a. The function **np.polynomial.polynomial.polyfit(x_list, y_list, deg = 1)** gives us the line (a polynomial of **deg**ree 1, get it?) that best passes through the points determined by **x_list** and **y_list**.
It returns the **b** and **a** from the equation **y = ax + b**, in that order, in a **Numpy array**.

NOTE: You might find the function **np.polyfit()** on the internet. *This function is deprecated and might give wrong results*.

Get the lines that best pass through the points left and right sides of the screen.

b. Now that you have the lines the fun part starts: *draw them*!
To draw a line in **OpenCV** you simply need **2 points** from that line and **OpenCV** draws a line between the 2 points.
So we need 2 points from the line on the left side of the screen and 2 points from the line on the right side of the screen.

A simple way to choose 2 points is to see that both lines tend to be *somewhat vertical* so for each line you could take the points where that line intersects with the top and bottom borders of the screen.
This means the points where **y = 0** and **y = height** for both lines.
Simply solve the equations for **y = ax + b** since you already know **y** and you need **x**.

You should end up with something like:

```
left_top_y =
left_top_x =

left_bottom_y =
left_bottom_x =

right_top_y =
right_top_x =

right_bottom_y =
right_bottom_x =
```

c. Normally we can just compute the points (converting everything to int) like:

```
left_top = int(left_top_x), int(left_top_y)
```

However…

Sometimes (because of an unclear image) the white pixels in the image are not detected quite right, which means our line will not be almost vertical but rather almost horizontal, leading our **x** values to be somewhere around $10^{30} - 10^{50}$.
Obviously these are bad values so we can just ignore them.

If one of the **x** values found at **b.** is not in $[-10^8, +10^8]$ then you should use the **(x, y)** point from the last frame instead of using the bad value to create a new point.
Remember that we are in a loop, simply don't update one of the values and let it remain the same as in the last iteration.
In case the first frame is one such bad frame you will need to initialize the **4 points** (**left_top**, **left_bottom**, etc.) near the top of the program, before the loop.
You can initialize them all with **0**.

d. To draw the actual lines we just use **cv2.line(frame, point1, point2, color, width)**.
Frame is the frame we want to draw over, in our case the binarized frame.
Point1 and **point2** are the top and bottom points in our case.
Color is a **3-tuple of ints** denoting the color of the line.
Since our image is grayscale you don't need to think about fancy colors, something like **(200, 0, 0) for one line and (100, 0, 0) for the other** (so they have different colors) will work.
Width is the width of the line **in pixels**. In the example I have used a width of 5.

NOTE:

The vertical line separating the 2 halves in the example is added just for visual effect.

It is a simple line (using **cv2.line**) going from the middle point of the top edge of the frame to the bottom edge.

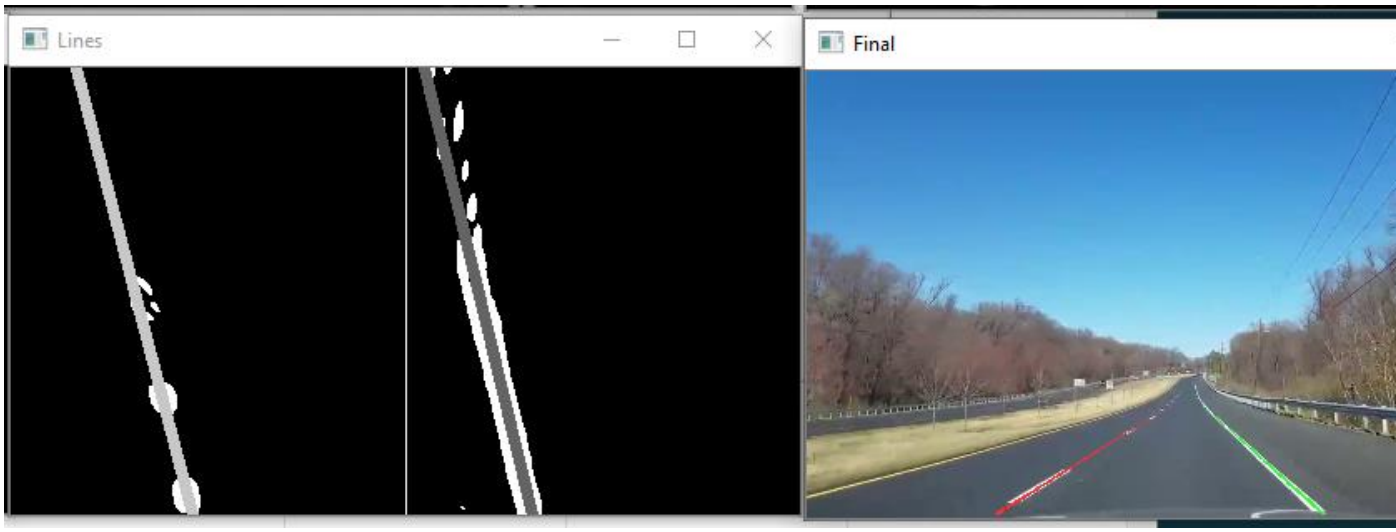
I have used color (255, 0, 0) and a width of 1.

Now the shown frame should look like the example of the right!

11. Create a final visualization!

Normally we could stop here and consider the car to be of some width w centered in the middle of the frame generated at the previous step.
We could give warnings to the driver (beeps etc.) when the car approaches one of the 2 lines and be done with it.

But what about the iconic lane detection image where the lanes are also displayed?



This part is much simpler than the previous parts since we are not doing math-magics anymore (just repeating one of the previous steps in reverse).

Since we need to create 2 lines in the final image (the right example) we will first get the pixels we need to color in green and then the pixels we need to color in red.

a. Create a blank frame (like in exercise **3.**).

b. Onto the blank frame draw only the line that you obtained from the points on the **left** side of the frame (like in exercise **10.**).
Use **color = (255, 0, 0)** and **width = 3** (you can use a larger width if you want).

c. Map this top-down frame to the trapezoid obtained in exercise **5.**
Use **cv2.getPerspectiveTranform** again to get the magic matrix, only this time the **current bounds** are the **frame bounds** and the **bounds you want to stretch to** are the ones found for **the trapezoid** in exercise **5.**

d. After getting the magic matrix simply use **cv2.warpPerspective** again like in exercise **5.**
The frame you want to transform is the one you just created at **b.**

You should get a frame where the left line instead of being from the top to the bottom of the frame, it will go from the top to the bottom of the trapezoid (the frame will still be blank with only a white line but the line will be in the general area where the road was in the original frame from exercise **1.**).

e. Get the coordinates of the white pixels for this left line in the frame obtained at **d.** (like in ex. **9.**).

f. Repeat steps **a-e** for the right line *on a separate blank frame (you will end up with 2 different frames in 2 different variables)*.
You should now have the coordinates of the left and right lines in the trapezoid. All that is left is to color them!

g. Make a copy of the original frame (**my_frame.copy()**), the frame you got from **cam.read()**.
Color in **red** (50, 50, 250) the pixels at the coordinates of the **left** line and in **green** (50, 250, 50) the pixels at the coordinates of the **right** line using the coordinates you found at step **e.** (you can use other colors if you want).

You should now obtain the right example.
***Congratulations**, you have created a functional lane detector!*

12. Make the lane detector run in real time.

Your visualization should not run in slow motion.
Don't worry if you do not have a good PC, these image processing tasks could easily be handled by a 10-15 year old PC.
The video normally runs at about 24-30 FPS (frames per second). If yours runs a bit slower (~15 FPS) there is no problem.
We just do not want it to run in "slow motion".