Setting Up OpenCV

Thursday, August 4, 2016 3:48 PM

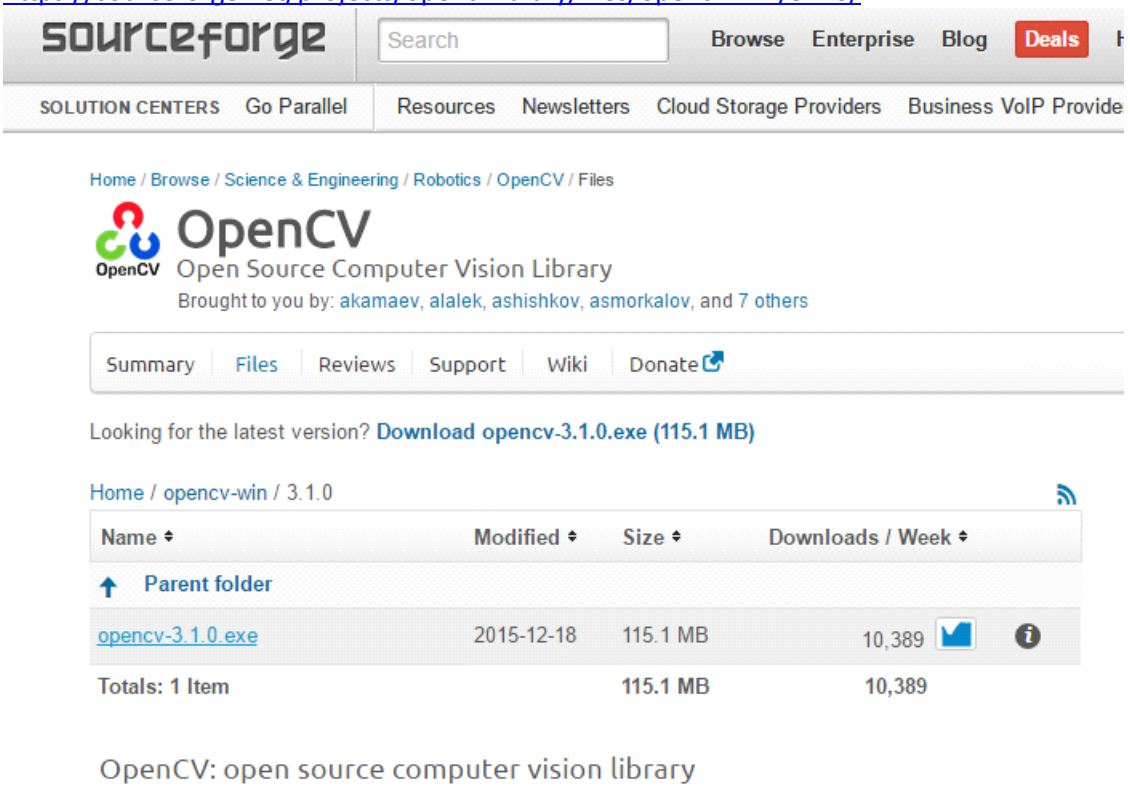
What is OpenCV?

- OpenCV is a collection of open-source libraries for computer vision

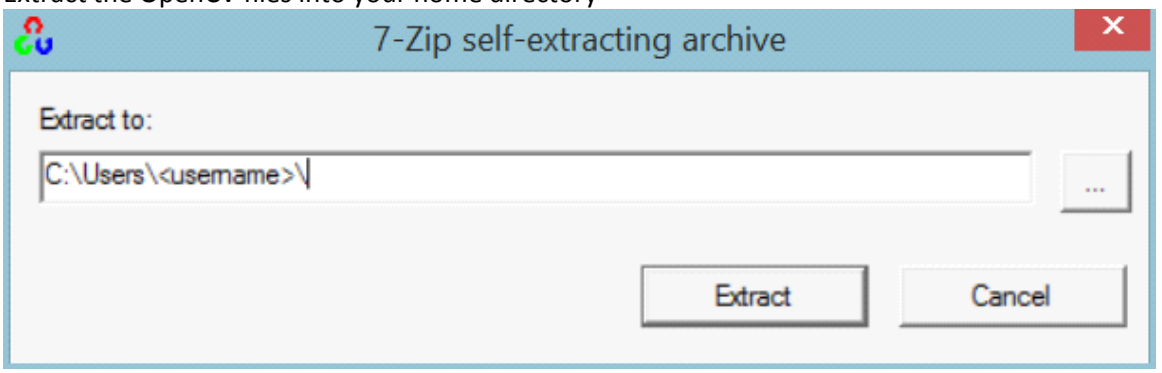
It supplies tools for everything from digital photography to video analysis and object detection

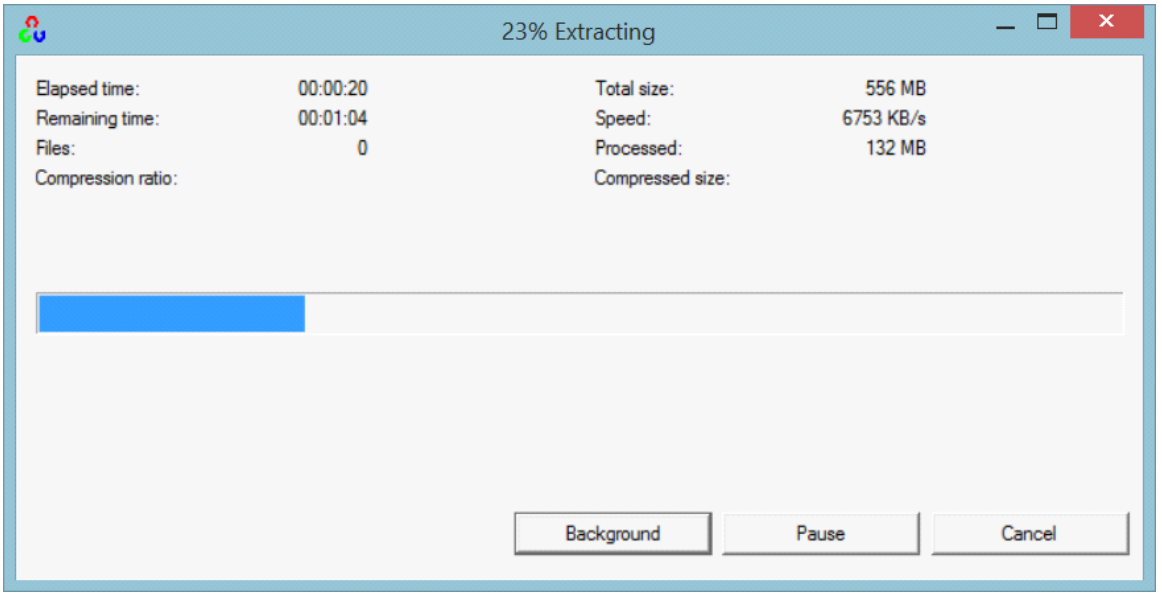
Getting started with OpenCV

- Make sure you have Python version 2.7.x installed
  This can be found by typing
  python -V
  into the command line
- Install *numpy* and *matplotlib* using
  pip install numpy matplotlib
- Download the OpenCV 3.1 installer from
  https://sourceforge.net/projects/opencvlibrary/files/opencv-win/3.1.0/



- Extract the OpenCV files into your home directory





- Once the opencv folder is extracted, go to
  C:\Users\username\opencv\build\python\2.7\x64
  and copy **cv2.pyd** into
  C:\Python27\Lib\site-packages

- Create a new file and name it **hello.py**
  - First, import the OpenCV library with
    import cv2
  - Next, create a **VideoCapture** object that will retrieve video from your webcam:
    cap = cv2.VideoCapture(0)
  - Create a *while* loop that will grab a new frame with every iteration:
    while(True):
    ret, frame = cap.read()
  - Display the frame:
    cv2.imshow("Hello World!", frame)
  - If the **q** key is pressed, exit the loop:
    if cv2.waitKey(1) == ord('q'):
    break
  - Lastly, release the video capture and close all windows:
    cap.release()
    cv2.destroyAllWindows()

  Your code should look like this:

```
import cv2
cap = cv2.VideoCapture( 0 )
```

```
while( True ):
    ret, frame = cap.read()
    cv2.imshow( "Hello World!", frame )
    if cv2.waitKey(1) == ord('q'):
        break
cap.release()
cv2.destroyAllWindows()
```

- Run the file with from the terminal:

  python hello.py

  and press **q** to exit
- Congratulations, you have created your first OpenCV application!

Summary

- In this lab, you have learned how to:
  - Install OpenCV dependencies
  - Install OpenCV Python bindings
  - Create a Hello World application that displays the video taken from the web cam

OpenCV Basics

Thursday, August 4, 2016 4:33 PM

Loading, Displaying, and Writing Images

- Here, we will display and write an image from a file
- Begin by creating a new file, ***image.py***
- First, the OpenCV library is imported into Python using the **cv2** package:
  import cv2
- Images can be read by the program using:
  img = cv2.imread('image.jpg')
- They are then displayed using:
  cv2.imshow("Image", img)
  the first argument is the name of the window in which the image is displayed, and the second is the actual image we have read
- When displaying an image, the function **cv2.waitKey(*n*)** must be called. It is a keyboard binding function that processes a variety of GUI events. It waits for the specified number of milliseconds, ***n***, for a keyboard press. If 0 is given, it will wait indefinitely.
  Therefore, when we show an image, we assign:
  cv2.waitKey(0)
- An image can be written to a file using:
  cv2.imwrite("new_image.jpg", img)
- Lastly, all of the OpenCV windows can be closed using:
  cv2.destroyAllWindows()
- To summarize, display the image by running:

```
import cv2

img = cv2.imread( 'image.jpg' )
cv2.imshow( "Image", img )

cv2.waitKey( 0 )
cv2.imwrite( "new_image.jpg", img )
cv2.destroyAllWindows()
```

Loading, Playing, and Writing Video

- Create a new script, ***video.py***
- A video is a sequence of frames, each of which is an image. Therefore, working with video is similar to working with images, except that the frames must be looped through.
- As always, import **cv2** and **numpy** into the script:
  import cv2
  import numpy as np
- To begin a video, we create a **VideoCapture** object. If the argument is a string, the object retrieves video from a filename or URL. If it is a number, video is taken from a camera with the respective label. In our case, we retrieve our video from the computer's default webcam, enumerated **0**:
  cap = cv2.VideoCapture(0)
- Similarly, the **VideoWriter** object can be used to create a new video file. First, the video codec must be specified using the **VideoWriter_fourcc** object:
  fourcc = VideoWriter_fourcc(*'MJPG')
  MJPG, or Motion JPEG, is a codec that is supported by a large number of systems
- Next, the **VideoWriter** object is initialized. The last two parameters are the framerate and size of

the new video:

fps = 24
width, height = (640, 480)
out = cv2.VideoWriter('output.avi', fourcc, fps, (width, height))

- To work with the video, we retrieve its contents frame-by-frame. This can be done with an while loop.

while(True):

- We get the frame using the **cap.read()** method, which returns a Boolean value of whether the frame is available and the frame itself:

ret, frame = cap.read()

- We can write to **output.avi** using the **out.write** method:

out.write(frame)

- Like images, frames are displayed using the **cv2.imshow** function:

cv2.imshow("Video", frame)

- The video loop can be ended by using the **cv2.waitKey** function:

k = cv2.waitKey(0)

- We can then check whether a key has been pressed by comparing the value of **k** to the desired key:

if k == ord('q'):
break

- At the end of processing the video, the **cap** video capture and **out** video writer handles must be released:

cap.release()
out.release()

- And the windows destroyed:

cv2.destroyAllWindows()

- In summary, you can record video from your webcam by running:

```
import cv2
import numpy as np

cap = cv2.VideoCapture( 0 )
fourcc = cv2.VideoWriter_fourcc(*'MJPG')
fps = 24
width, height = ( 640, 480 )
out = cv2.VideoWriter( 'output.avi', fourcc, fps, ( width, height ) )
while( True ):
    ret, frame = cap.read()
    out.write( frame )
    cv2.imshow( "Video", frame )
    k = cv2.waitKey(0)
    if k == ord( 'q' ):
        break

cap.release()
out.release()
```

Drawing on Images in OpenCV

- OpenCV supports a variety of drawing functions. These features make it easier to visualize elements in a frame.
- The file will be named ***drawing.py***
- First, specify a 640x480 blank image:

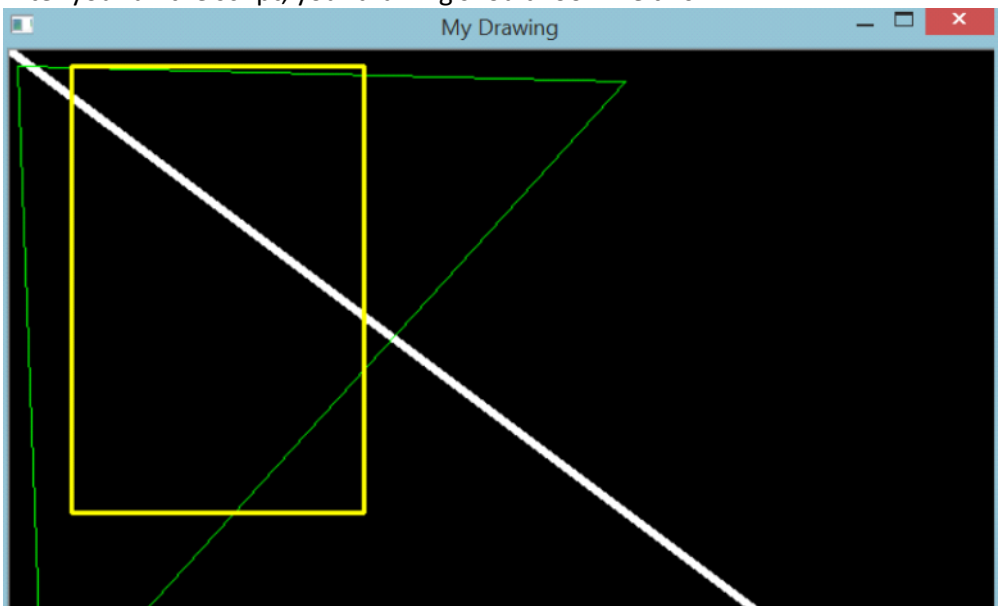
img = np.zeros((480, 640, 3), np.uint8)

Note that the image is a 3-channel Numpy array… more on this later.

- To draw a line, specify the image that it is painted on, initial and final *(x, y)* coordinates, BGR color, and pixel width
  For example, a white line 3 pixels wide that goes diagonally through the screen is drawn with:
  cv2.line(img, (0, 0), (639, 479), (255, 255, 255), 3)
- Multiple lines that connect a sequence of points can be drawn using **cv2.polylines**:
  points = np.array([(5, 10), (20, 400), (55, 400), (400, 20)], np.int32)
  points = points.reshape((-1, 1, 2))
  cv2.polylines(img, [points], True, (0, 255, 0))
  The Boolean argument specifies whether the shape is closed
- A rectangle can be made by providing the upper left and lower right corner coordinates
  cv2.rectangle(img, (40, 10), (230, 300), (0, 255, 255), 2)
- Text strings can be inserted into the image by specifying the bottom left corner coordinate, font, size, color, thickness, and interpolation type:
  font = cv2.FONT_HERSHEY_SIMPLEX
  cv2.putText(img, 'Hello Worlds!', (200, 460), font, 5, (255, 255, 0), 2, cv2.LINE_AA)
- Display the image with:
  cv2.imshow('My Drawing', img)
  cv2.waitKey(0)
- The script will ultimately be:

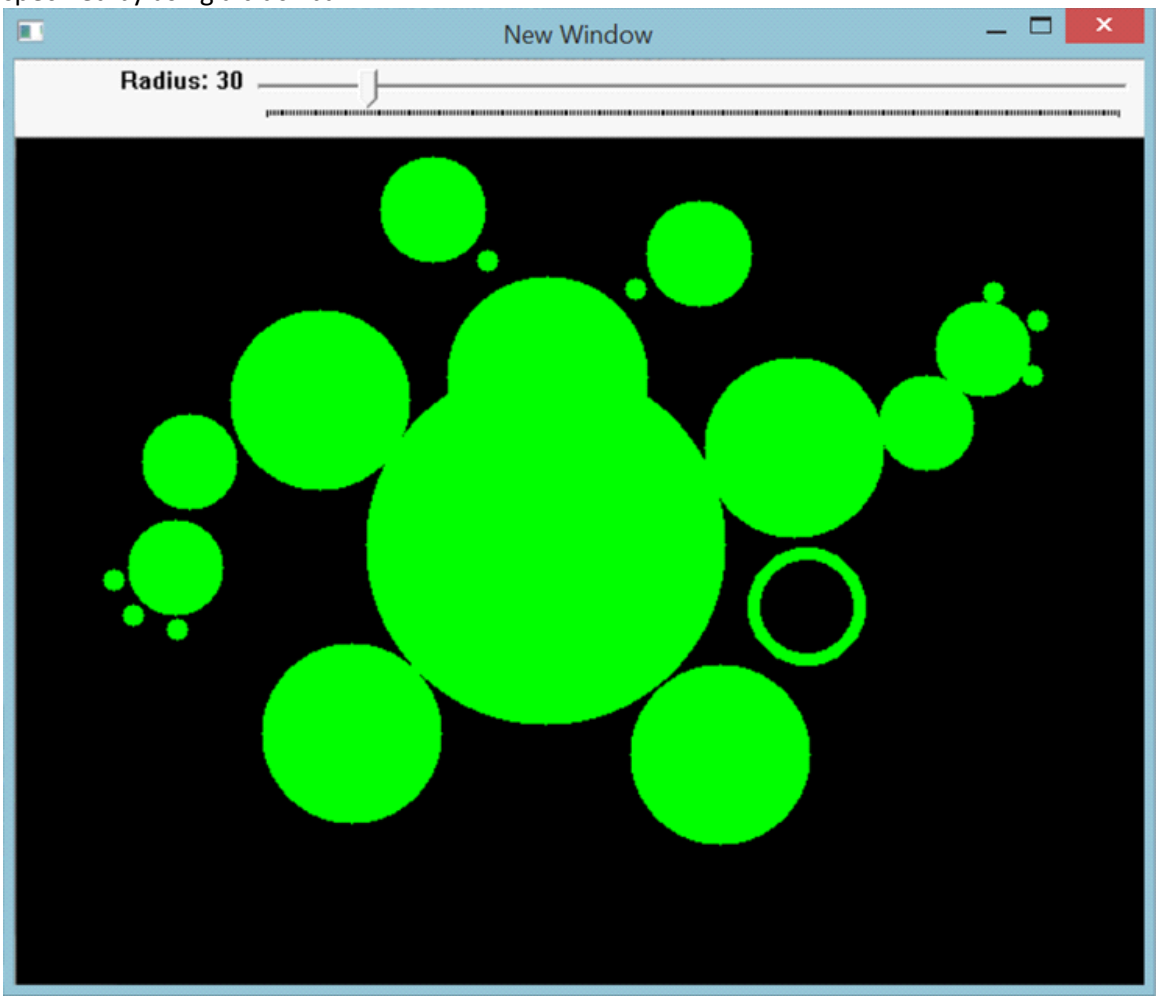
```
import cv2
import numpy as np

img = np.zeros( ( 480, 640, 3 ), np.uint8 )
cv2.line( img, (0, 0), (639, 479), (255, 255, 255), 3 )
points = np.array( [ (5, 10), (20, 400), (55, 400), (400, 20) ], np.int32 )
points = points.reshape( ( -1, 1, 2 ) )
cv2.polylines( img, [points], True, ( 0, 255, 0 ) )
cv2.rectangle( img, (40, 10), (230, 300), (0, 255, 255), 2 )
font = cv2.FONT_HERSHEY_SIMPLEX
cv2.putText( img, 'Hello Worlds!', (200, 460), font, 2, (255, 255, 0), 2, cv2.LINE_AA )

cv2.imshow( 'My Drawing', img )
cv2.waitKey(0)
```

- After you run the script, your drawing should look like this:



Interacting with the GUI

- OpenCV provides a number of tools for interacting with the application. These tools allow developers to build graphical interfaces, resulting in finer user controls.
- In this example, we will build a program that allows users to draw green circles of variable radii. The circles will be drawn at the locations where the user clicks the screen, and the radii will be specified by using a track bar.



- Begin the program by starting a script ***gui.py*** importing **cv2** and **numpy**

  import cv2

  import numpy as np
- Next, we will begin creating callback functions that will be triggered by events caused by the user. First, create the drawCircle function. **drawCircle** will be attached to the mouse events in the image window, and will tell the program to draw a circle either temporarily, if the mouse is moved, or permanently, if the mouse is clicked. **drawCircle** is passed information about the mouse event in its arguments:

  def drawCircle(event, x, y, flags, param):
- Next, we will define variables **tempCircle** and **circlesList** as global. Both variables will contain dicts of circle information, with position and radius data, but **tempCircle** will only contain information

about a single circle, which is updated every time that the mouse is moved, while **circlesList** will contain permanent circle information about all circles created and the list will be augmented after each click

```
global tempCircle, circlesList
circ = {
    "coord": (x, y),
    "radius": param
}
if event == cv2.EVENT_LBUTTONUP:
    circlesList.append(circ)
    print circlesList
elif event == cv2.EVENT_MOUSEMOVE:
    tempCircle = circ
```

- The other callback that we will define is **setRadius**. It will be called every time that the track bar value is changed and will pass the new radius parameter into **drawCircle**:

```
def setRadius( radius ):
    cv2.setMouseCallback( "New Window", drawCircle, radius )
```

- Next, we give default values to **tempCircle**, **circlesList**, and **radius**

```
tempCircle = False
circlesList = []
radius = 5
```

- We can then create a new window with

```
cv2.namedWindow( "New Window" )
```

- We can then attach a mouse callback to the window:

```
cv2.setMouseCallback( "New Window", drawCircle, radius)
```

- And create a trackbar:

```
cv2.createTrackbar('Radius', 'New Window', 0, 255, setRadius)
```

- Now, **drawCircle** will be executed whenever a mouse event occurs on the window, and **setRadius** will be called whenever the trackbar position is updated.
- In order to update our drawing in real-time, we will draw the circles inside a while loop

```
while( True ):
```

- First, we create a black image:

```
img = np.zeros( (480, 640, 3), np.uint8 )
```

- And draw an open circle from **tempCircle** if the mouse has been moved in the window

```
if tempCircle:
    cv2.circle( img, tempCircle["coord"], tempCircle["radius"], (0, 255, 0), 5 )
```

- Then, draw all of the circles contained in the **circlesList**

```
for circle in circlesList:
    cv2.circle( img, circle["coord"], circle["radius"], (0, 255, 0), -1 )
```

- You can then display the image:

```
cv2.imshow( "New Window", img )
```

- If the *ESC* key is pressed, exit the program:

```
k = cv2.waitKey( 1 )
if k == 27: # ESC
    break
```

- If the ***s*** key is pressed, save a copy of the image

```
elif k == ord('s'):
    cv2.imwrite("new_drawing.jpg", img)
```

- If loop exits, destroy all of the windows

```
cv2.destroyAllWindows()
```

- Overall, the code looks like:

```
import cv2
```

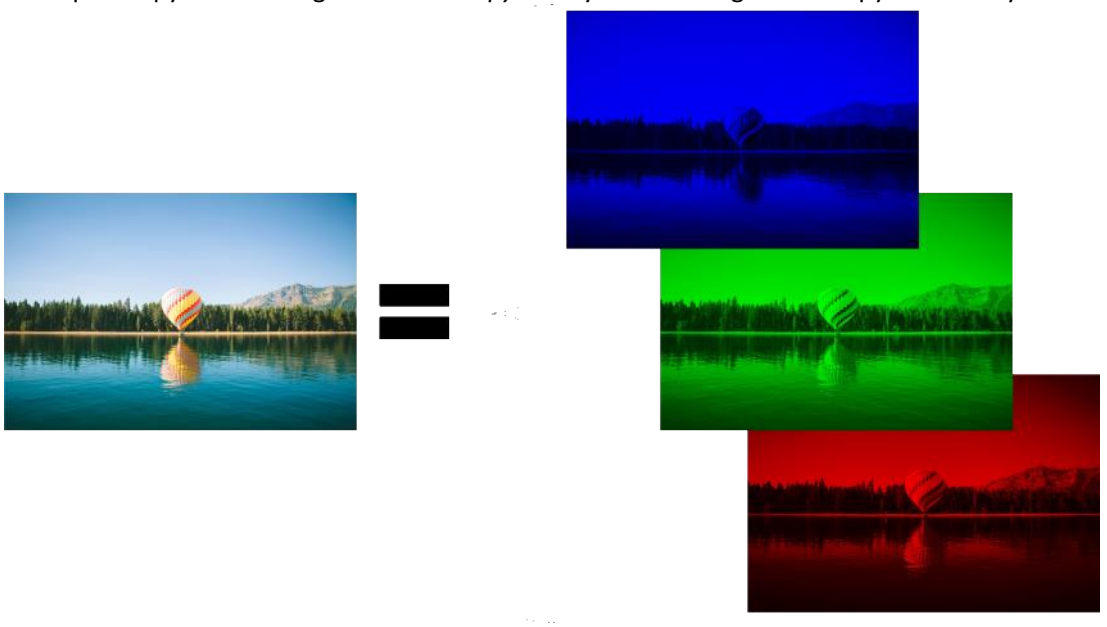
```
import numpy as np

def drawCircle( event, x, y, flags, param ):
    global tempCircle, circlesList
    circ = {
        "coord": (x, y),
        "radius": param
    }
    if event == cv2.EVENT_LBUTTONUP:
        circlesList.append(circ)
        print circlesList
    elif event == cv2.EVENT_MOUSEMOVE:
        tempCircle = circ

def setRadius(radius):
    cv2.setMouseCallback( "New Window", drawCircle, radius)

tempCircle = False
circlesList = []
radius = 5

cv2.namedWindow( "New Window" )
cv2.setMouseCallback( "New Window", drawCircle, radius)
cv2.createTrackbar('Radius', 'New Window', 0, 255, setRadius)

while(True):
    img = np.zeros( (480, 640, 3), np.uint8 )
    # cv2.getTrackbarPos('Radius', 'New Window')
    if tempCircle:
        cv2.circle( img, tempCircle["coord"], tempCircle["radius"], (0, 255, 0), 5 )
    for circle in circlesList:
        cv2.circle( img, circle["coord"], circle["radius"], (0, 255, 0), -1 )
    cv2.imshow( "New Window", img )

    k = cv2.waitKey(1)
    if k == 27: # ESC
        break
    elif k == ord('s'):
        cv2.imwrite("new_drawing.jpg", img)

cv2.destroyAllWindows()
```

- Summary
    - In this lab, you have learned how to:
        - Load, display, and write images
        - Load, play, and record video
        - Draw shapes
        - Create graphical user interfaces using mouse events and trackbars

Computer Vision Fundamentals

Friday, August 5, 2016 3:23 PM

Images

- As we begin computations on images, we ask what an image is in the first place.
- A digital image is a 3-channel array of pixel intensities. In OpenCV, images are loaded as unsigned 8-bit 3-channel arrays, where every element represents the intensity of the pixel on the specific channel. The channels are listed in Blue, Green, Red order.
- The OpenCV python bindings use the *Numpy* library to load images as numpy **uint8** arrays
- 

Color Spaces

- The BGR colorspace is used by default, as it is very convenient for both capturing and displaying pixels. While it is useful at the lower level, it is difficult to use BGR for specifying ranges of colors, as a range has to be given to in each of the three channels. Initially, however, every pixel loaded into OpenCV contains a three-dimensional BGR vector.
- For the purposes of describing colors with common characteristics, like a similar level of brightness or tone, the Hue, Saturation, and Value (HSV) colorspace becomes far more intuitive. The *hue* is specified as a value between 0 and 179 degrees, where every degree represents a color. The *saturation* describes how vividly the color is expressed, on a scale from 0 to 255, and the *value* states the brightness of the color, ranging from 0 to 255. OpenCV provides a function **cvtColor** for converting between color spaces, and a conversion from BGR to HSV can be done by passing the attribute **cv2.COLOR_BGR2HSV**. For example, to convert blue, which is **(255, 0, 0)** in BGR to HSV, we write:

  blue = np.uint8([[[0, 255, 0]]])

  blue_hsv = cv2.cvtColor(blue, cv2.COLOR_BGR2HSV)

  blue_hsv yields a numpy array containing **[[[120, 255, 255]]]**

  Note that the triple brackets around the values are due to this function being intended for use on an entire image, not just a single color vector.
  - We will see that working between different color spaces can be a useful technique in image processing. For example, suppose we have an image named **gorge.jpg** that has a dull color set:

- We can make the colors more vivid by increasing the saturation in the image. If we are dealing with BGR channels, however, the task becomes difficult. Thanks to color spaces, however, we can convert the color vectors from BGR to HSV space, operate on the HSV vectors, and then transform the image back into the BGR space to be displayed by OpenCV.
  - Begin by importing **cv2** and **numpy**, which will allow us to perform mathematical operations on the image array:

    import cv2
    from numpy import np
    img = cv2.imread("gorge.jpg")
  - Next, simply convert the BGR array into HSV space by using **cv2.cvtColor**:

    img_hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

    This returns a 3-channel array, where the channels correspond to hue, saturation, and intensity value, respectively.
  - We can now extract specifically the saturation channel and perform computations on it.

    sat = img_hsv[… , 1]
  - There are several ways to increase the saturation of this image; naively, we could add a value to the array, or we could multiply it by a factor. Because the array elements are loaded by default in the **np.uint8** type, and different pixels contain varying saturation levels, we run the risk of exceeding the 255 limit of unsigned 9-bit integers; if that is the case, Numpy will loop back, setting the value of $element$ to $element \ mod \ 255$. A better way to increase the saturation is to calculate the difference of the pixel saturation from 255, multiply the difference by a factor, and then add it to the array. This way, the new pixel saturation will never exceed 255.

    scale = 0.1
    augmentSat = np.uint8(scale * (255 - sat))
    img_hsv[… , 1] = sat + augmentSat
  - We then convert the image back into the BGR colorspace

    img_new = cv2.cvtColor(img_hsv, cv2.COLOR_HSV2BGR)
  - Because the image is in BGR format, we can now display it using **cv2.imshow**:

    cv2.imshow("Vivid Image", img_new)

- Thus, using several color spaces allows us to make convenient modifications to the images. Say, in the above example, you also wanted to give the image a sepia effect. [How would you do that? -- can serve as challenge question]. As it is known that at the zero point of the hue circle and yellow is 1/6th of the way around, and that we are specifying a value from 0-179, the sepia tint will have a value of 15. Inserting the line:

  img_hsv[…, 0] = 15

  before we convert back to BGR, we get the sepia effect:

- Sometimes, we only need for the pixels of an image to be mapped into one dimension. For example, the 1-d
  vector may specify the absolute distance between the two images, or may be the probability that an object is at that location. In this situation, the grayscale color space proves useful. The only value in the color vector at each pixel is the **intensity**, given as a value from 0 to 255. Black is shown as 0 and white as 255. In OpenCV, an image **img**, which consists of a BGR array, can be drawn in Grayscale by calling:

gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

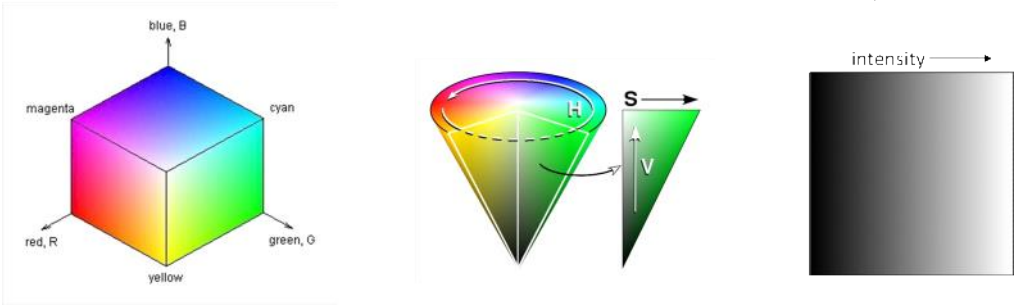
Furthermore, an image can be loaded in grayscale by default by calling

gray = cv2.imread("image_filename.jpg", 0)

the **0** instructs OpenCV to import the image as grayscale.

$$(Blue, Red, Green) \leftrightarrow (Hue, Saturation, Value) \rightarrow (Intensity)$$

BGR HSV Grayscale



Thresholding

- In computer vision, thresholding allows us to select pixels based on their intensity. This is done on a grayscale image, and returns an array of binary values that serves as a mask. This mask can be applied to other images using bitwise operations. In OpenCV, thresholding is done using the **cv2.threshold** function.
- In this example, we will use OpenCV thresholding to make an Intel intern jump over a whale. We will use two concepts, **masking** and the **region of interest (ROI)**, to combine parts from two images
- We begin with two images: one contains a wakeboarder, and the other contains a whale tail.
- The images are as follows:

- Because there is a strong contrast between the wakeboarder and the background, we can use thresholding to get an image mask of the wakeboarder's shape. We will extract a binary mask, shrink down the mask to a rectangular region of interest around the body shape, and then use the ROI to replace points in a corresponding region in the whale image with the wakeboarder.
- 

- We begin by importing **cv2** and **numpy** and reading both images:

  import cv2
  import numpy as np

  wake = cv2.imread("wakeboard.jpg")
  whale = cv2.imread("whale.jpg")
- Because thresholding is done on grayscale images, we need to perform a color conversion on the **wake** frame

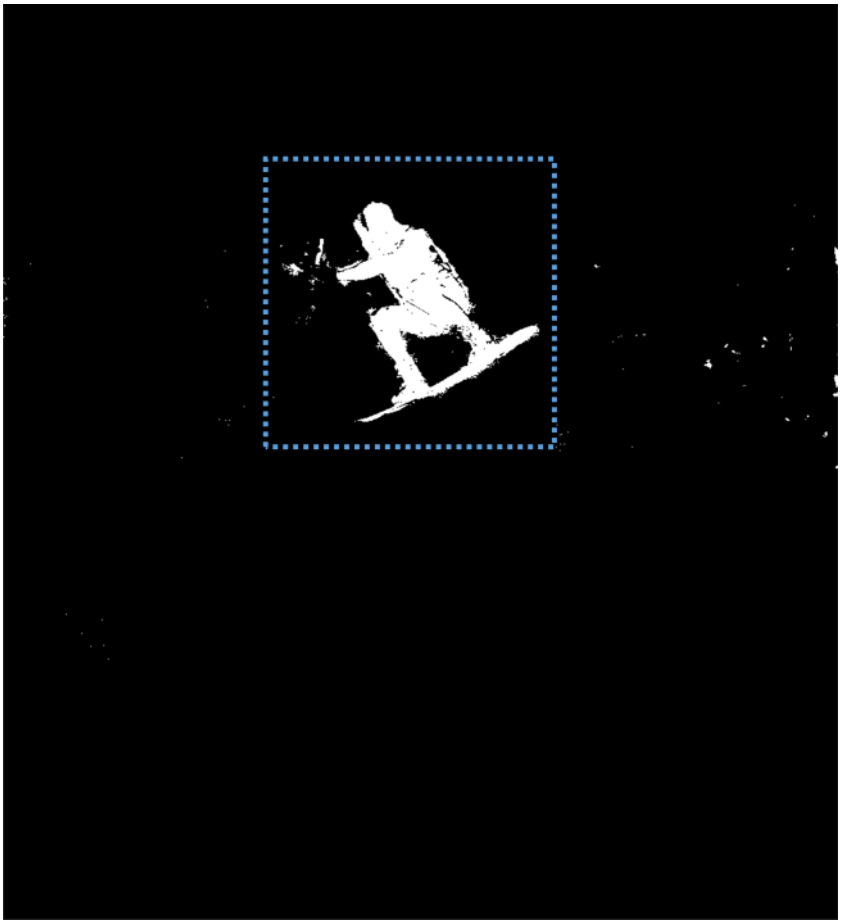
  gray = cv2.cvtColor(wake, cv2.COLOR_BGR2GRAY)
- We now take the threshold. In order to do so, we find a pixel intensity such the amount of the wakeboarder's silhouette is maximized and the background noise is minimized. OpenCV provides means for finding this value or adapting values to smaller sections of the image; for our purposes, however, **130** will work well. Therefore, we write:

  ret, mask = cv2.threshold(gray, 130, 255, cv2.THRESH_BINARY_INV)

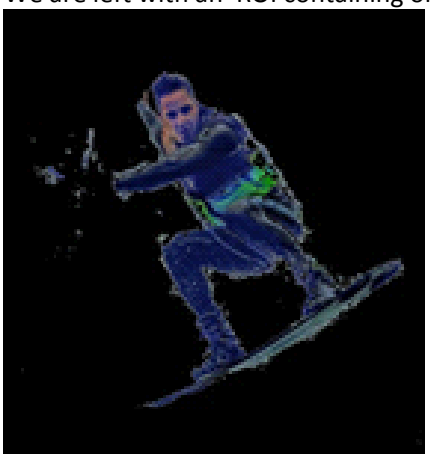
  This function makes every pixel in **gray** whose value is under **130** become **255**. If we had set **cv2.THRESH_BINARY** instead of **cv2.THRESH_BINARY_INV**, everything *but* the wakeboarder, who is darker than the background, would be selected.
- Next, we continue to defining a **region of interest (ROI)**. There are two reasons for doing so:
  - First, the image contains points that are darker than 130 and not on the wakeboarder's body shape. This causes extraneous blotches to be in our mask, and therefore will result in noise being transferred into the final region.
  - Second, defining a narrow ROI increases our control of where the image of the wakeboarder will be placed in the whale photo.

  We define a rectangular ROI by selecting the left upper and right bottom corners; we then find the width and height of the rectangle:

lu = (280, 170)
rb = (600, 490)
w = rb[0] - lu[0]
h = rb[1] - lu[1]



- We now extract the ROI from both the mask and the wake image, and then apply the mask lying inside of the ROI to the wake image ROI.
  To extract the masks:
  roi_mask = mask[lu[1]:rb[1], lu[0]:rb[0]]
  roi_wake = wake[lu[1]:rb[1], lu[0]:rb[0]]
  In order to apply the mask, we perform a **bitwise and** between **roi_wake** and **roi_mask**:
  roi_wake = cv2.bitwise_and(roi_wake, roi_wake, mask=roi_mask)
  We are left with an ROI containing only the body of the wakeboarder!



- Next, we can begin transferring the wakeboarder image into whale photo. First, we specify the location of the desired shape, marked by the top left coordinate of where the rectangular ROI will

be inserted. We then extract a region from the whale image with the same dimensions as the ROI:

loc = (270, 60)

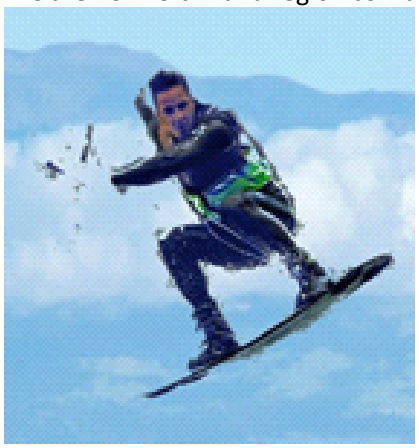
roi_whale = whale[loc[1]:loc[1]+ h , loc[0]:loc[0] + w]

- We will now apply a mask that keeps everything *but* the silhouette of the wakeboarder by performing a **bitwise not** on the ROI mask, bitwise and-ing the new mask with **roi_whale**, and adding the pixel values of both masked regions together with **cv2.add**.

  roi_whale = cv2.bitwise_and(roi_whale, roi_whale, mask=cv2.bitwise_not(roi_mask))

  roi_whale = cv2.add(roi_whale, roi_wake)

  We are now left with a region combining elements from both images:



- Lastly, replace the section containing the ROI in the final image with the edited **roi_whale**:

  whale[loc[1] : loc[1]+ h , loc[0]:loc[0] + w] = roi_whale

- Display the image with **cv2.imshow**:

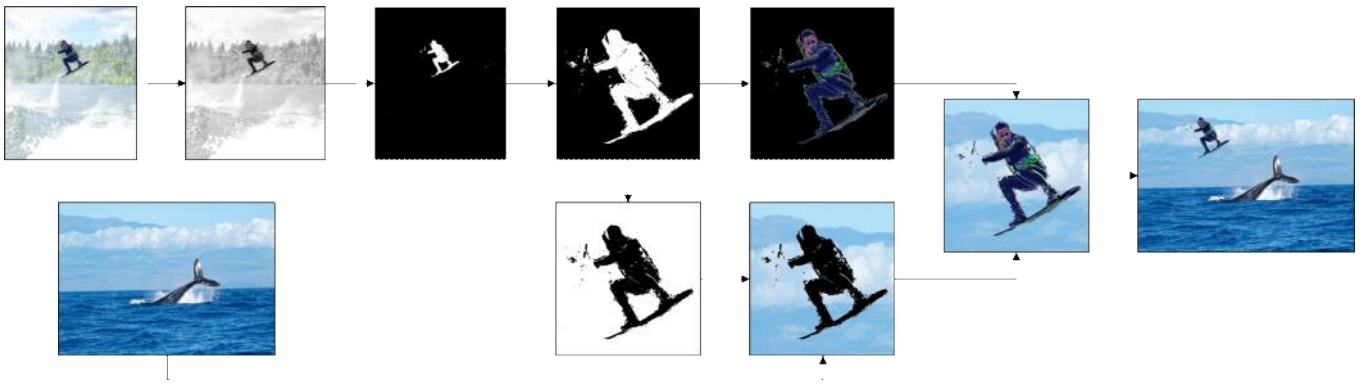
  cv2.imshow("Internship 2016", whale)

  cv2.waitKey(0)

  cv2.destroyAllWindows()

- The resulting image should look like this:

- Our process can be visualized by looking at the intermediary frames:



Often, it is useful to insert **cv2.imshow** statements throughout the script in order to better visualize what is happening.

- The whole script is as follows:

```
import cv2
import numpy as np

wake = cv2.imread( "wakeboard.jpg" )
whale = cv2.imread( "whale.jpg" )

gray = cv2.cvtColor( wake, cv2.COLOR_BGR2GRAY )
ret, mask = cv2.threshold( gray, 130, 255, cv2.THRESH_BINARY_INV )
# ROI
lu = (280, 170)
rb = (600, 490)
w = rb[0] - lu[0]
h = rb[1] - lu[1]

roi_mask = mask[ lu[1]:rb[1], lu[0]:rb[0] ]
roi_wake = wake[ lu[1]:rb[1], lu[0]:rb[0] ]
roi_wake = cv2.bitwise_and( roi_wake, roi_wake, mask=roi_mask )

# upper left corner of region in whale image where we will place wakeboarder
loc = (270, 60)
roi_whale = whale[ loc[1]:loc[1]+ h , loc[0]:loc[0] + w ]
roi_whale = cv2.bitwise_and( roi_whale, roi_whale, mask=cv2.bitwise_not(roi_mask) )
roi_whale = cv2.add( roi_whale, roi_wake )
whale[ loc[1] : loc[1]+ h , loc[0]:loc[0] + w ] = roi_whale

cv2.imshow("Internship 2016", whale)
cv2.waitKey(0)

cv2.destroyAllWindows()
```

In future sections, we will see better ways to both create and modify image masks.

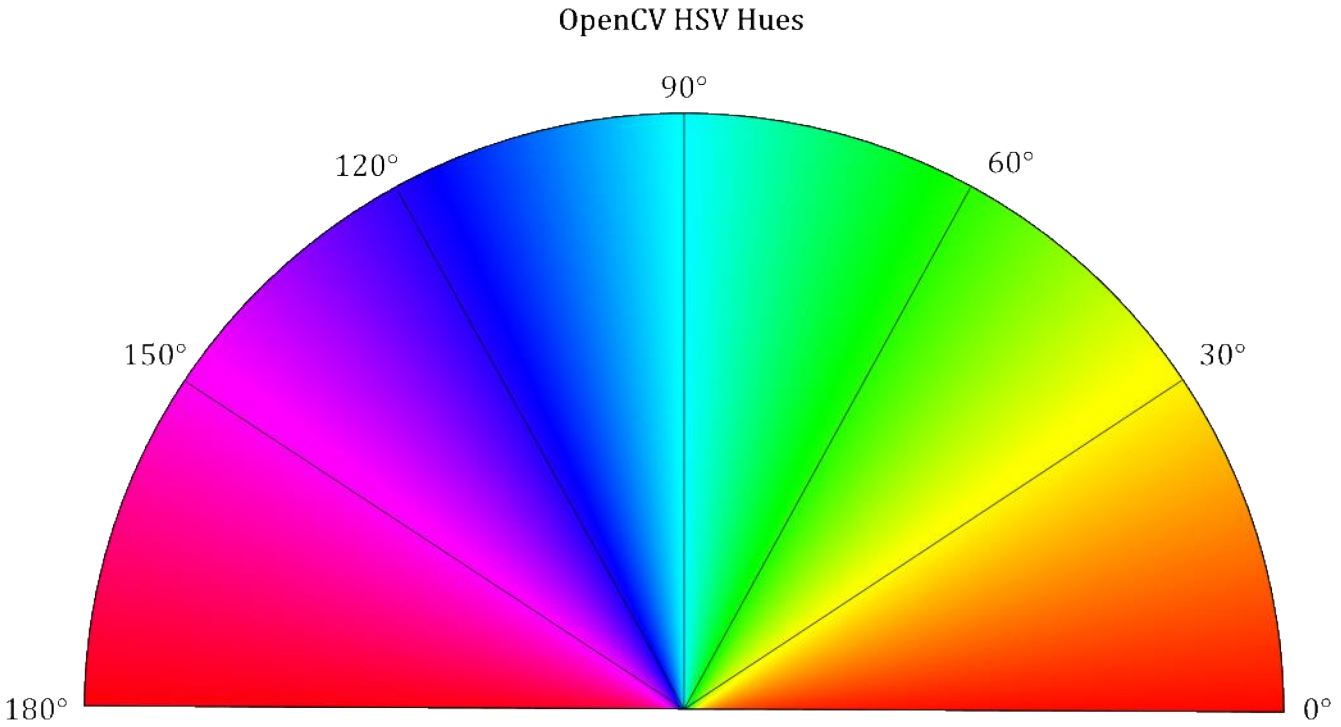
Selecting ranges of colors

- As you can see, thresholding is a very useful tool when we analyze data in a single dimension; as seen in the last example, images can have their light and dark elements be extracted using this technique. Furthermore, multiple thresholds and bitwise operations can allow us to select specific tones in a grayscale image.
- A similar method can be applied to colored images in order to select items within a specific color range. For instance, a digital image can have multiple shades of red, and it would therefore be impractical to index for individual colors. OpenCV provides the **cv2.inRange** function, which

returns a mask of values that are within a specified range. This is best used in HSV color space, for a small range of hues can be specified and then the object's colors will just be saturation and value variations of these hues. Thus, **cv2.inRange** can be used as follows:

```
imgHSV = cv2.cvtColor( img, cv2.COLOR_BGR2HSV )

lowerRange = np.array( (35, 100, 5) )
upperRange = np.array( (88, 255, 255) )

mask = inRange( imgHSV, lowerRange, upperRange )
```

Note that **lowerRange** and **upperRange** are HSV vectors. The hue is set to a value from 0 to 179, and the saturation and value are set from 0 to 255. The HSV coordinate can be determined by finding the color on an HSV color wheel.

- 

- Being able to both perform masking and modify images in HSV space provides us with very powerful image processing tools. For example, suppose we are given an image of a red car:

and wish to change the car color to blue. We can apply the techniques we have just learned to select the red colors of the car, apply masking and HSV transformations to give the car a blue color, and then insert the modified car into the original image.

- We begin by importing the necessary packages and reading the image from **car.jpg:**

  import numpy as np
  import cv2
  img = cv2.imread('car.jpg')

- We then convert **img** into the HSV color space:

  imgHSV = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

- We're now ready to create a mask from pixels in the car's red color range. Unfortunately, we run into an issue: red hues appear at both the very beginning and the very end of the hue spectrum, where the values are closer to orange and to violet, respectively. Because the car incorporates both of these values, we must calculate for both values separately, and then perform a **bitwise or** on the two masks. Another issue we find is that the driver's skin tones are also on the red side of the Hue spectrum. To avoid including him in the mask, we simply raise the minimum saturation value in the range.

  mask_r1 = cv2.inRange(imgHSV, np.array((0, 100, 5)), np.array((10, 255, 230)))
  mask_r2 = cv2.inRange(imgHSV, np.array((161, 100, 20)), np.array((179, 255, 200)))
  mask = cv2.bitwise_or(mask_r1, mask_r2)

  At this point, we have created a mask containing just the values of the car.

- Next, we alter the hue and saturation layers of **imgHSV** to set the hue to blue and the saturation to maximal.

  imgHSV[…, 0] = 120
  imgHSV[…, 1] = 255

- We now convert the blue car image from HSV to BGR color space and apply the mask we have created from the red color range. This will extract solely the blue car from the image.

  imgBlue = cv2.cvtColor(imgHSV, cv2.COLOR_HSV2BGR)
  imgNew_car = cv2.bitwise_and(imgBlue, imgBlue, mask=mask)

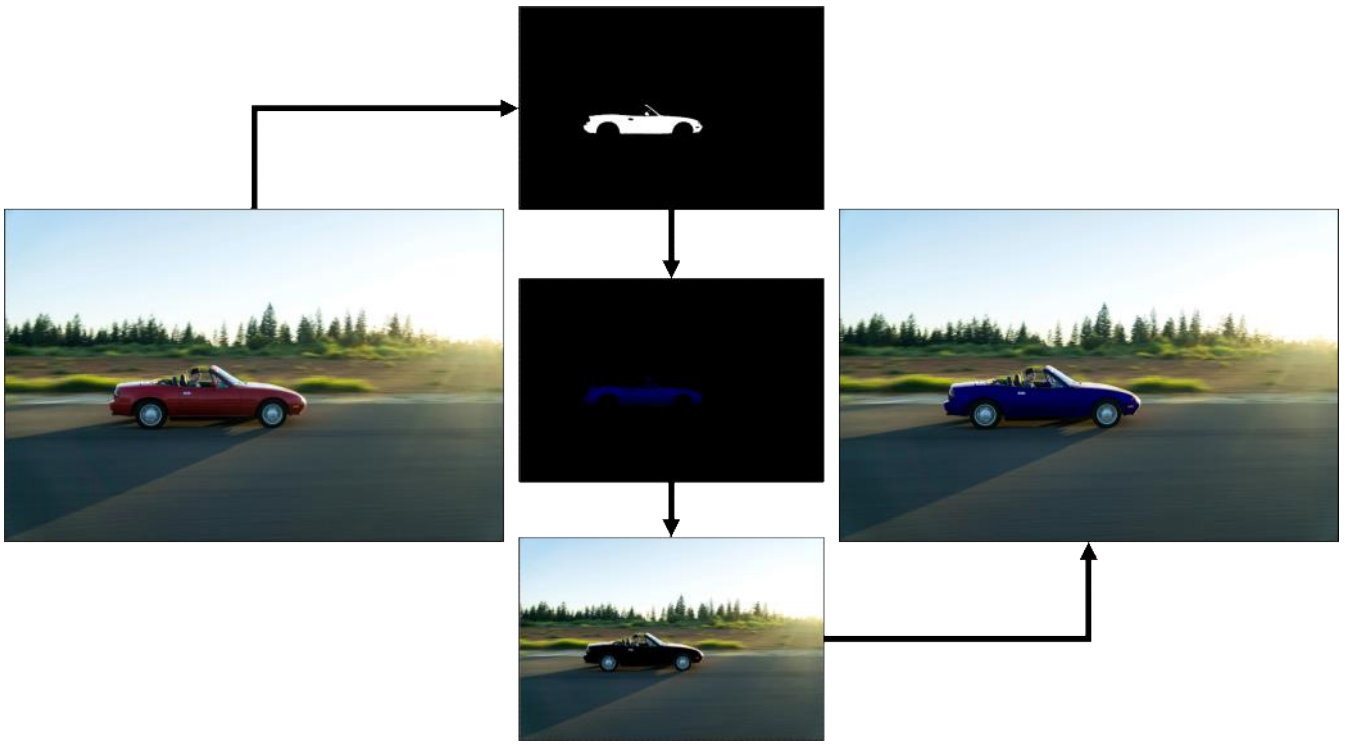
- Now, apply the inverse of the mask to the original image to create **imgNew_nocar** and add them together, just as we have done in the thresholding example.

imgNew_nocar = cv2.bitwise_and(img, img, mask=cv2.bitwise_not(mask))
imgNew = cv2.add(imgNew_car, imgNew_nocar)

- You have now changed the color of the car! Display your image:

cv2.imshow("new", imgNew)
cv2.waitKey(0)
cv2.destroyAllWindows

- Your final result should look like:
- 
- Our project's flow looked like:



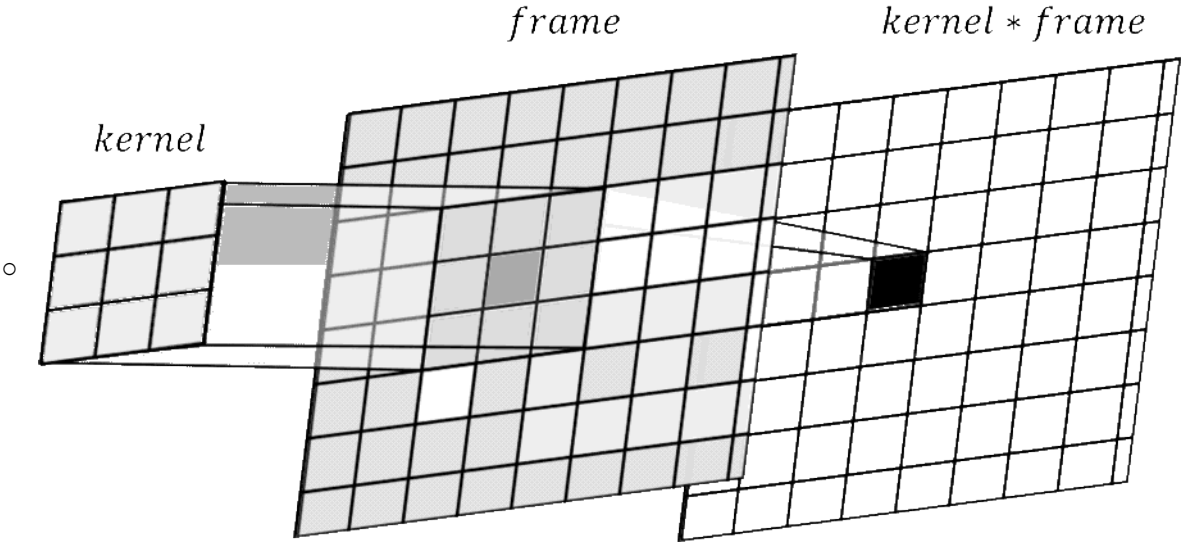
- Altogether, our code was:

```
import numpy as np
import cv2

img = cv2.imread('car.jpg')

imgHSV = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
mask_r1 = cv2.inRange(imgHSV, np.array((0, 100, 5)), np.array((10, 255, 230))) # reds
above 0
mask_r2 = cv2.inRange(imgHSV, np.array((161, 100, 20)), np.array((179, 255, 200))) # reds
below 0
mask = cv2.bitwise_or(mask_r1, mask_r2)

imgHSV[ …, 0 ] = 120
imgHSV[ …, 1 ] = 255

imgBlue = cv2.cvtColor(imgHSV, cv2.COLOR_HSV2BGR)
imgNew_car = cv2.bitwise_and(imgBlue, imgBlue, mask=mask)
imgNew_nocar = cv2.bitwise_and(img, img, mask=cv2.bitwise_not(mask))
imgNew = cv2.add(imgNew_car, imgNew_nocar)
cv2.imshow("new", imgNew)
cv2.waitKey(0)
cv2.destroyAllWindows
```

~~Transformations~~

- ~~resize~~
- ~~matrix multiplication~~
- ~~affine~~
- ~~rotation~~
- ~~perspective~~

Filtering

- Filtering can be performed on an image by sliding a kernel through every pixel in the image and performing correlation between the kernel and the region shaded by it. The result of this operation is stored in the respective value that the kernel is centered on. The kernel is a matrix of a specified size and values, which determines what operation the filter will perform.
  - 

- In correlation, pixels in the kernel are multiplied by their corresponding pixels in the image, and then all of these value products are added together. OpenCV provides the **cv2.filter2D** function for

performing this operation. For example, we can design a smoothing filter by using a kernel to calculate the average value of the pixels in the kernel's region, and then setting the center of the kernel to that value in the image:

$$K = \frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}$$

This kernel slides over the entire image, summing up pixels in a 3x3 region and divides the sum by 9 to find the average pixel value. We can implement this in OpenCV using:

kernel = np.ones((3,3), np.float32)/9
filt = cv2.filter2D(img, -1, kernel)

The -1 argument specifies that all layers of the image will be affected by the filter.

- Smoothing
  - We can use filters to apply blurs to an image. Smoothing serves as a low pass filter in images, allowing us to remove high-frequency noise that may skew our computations. In this section, we will apply the average, Gaussian, and median blurs onto this image:

import cv2
import numpy as np
img = cv2.imread("street.jpg")

  - Mean
    - As we have seen, filtering creates the foundation for image smoothing by making pixels closer resemble their adjacent counterparts. A general averaging kernel will look like:

$$kernel = \frac{1}{w_{kernel} * h_{kernel}}\begin{bmatrix}1 & \cdots & 1\\ \vdots & \ddots & \vdots\\ 1 & \cdots & 1\end{bmatrix}$$

OpenCV provides a function, **cv2.blur**, for applying the average blur without having to construct the kernel. This is done by using **cv2.blur**:

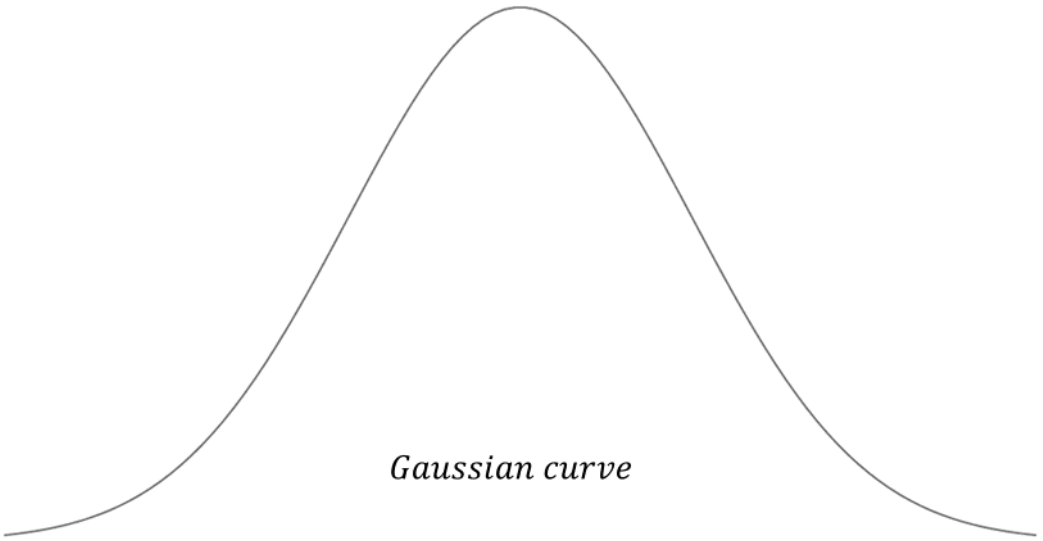
blur = cv2.blur(img, (3,3))

This method is the same as the one in the Filtering example; it is known as a box filter.

Average

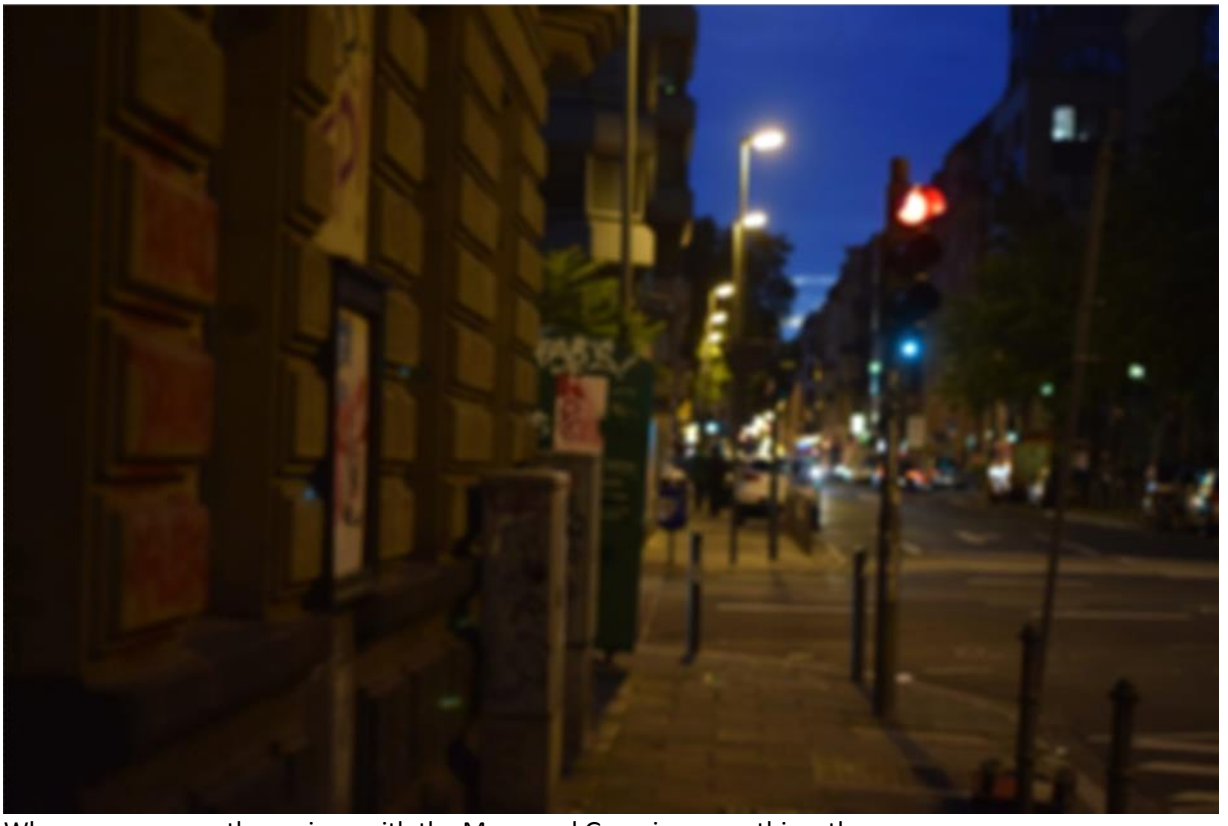
The problem with average blurring is that all pixels under the kernel are taken into account. Thus, the farther pixels may have a large impact on the mean value, therefore resulting in uneven jumps between pixels in the filtered image.

- Gaussian
  - A Gaussian kernel solves this problem by weighing values closer to the center of the kernel higher than those farther away. This allows for smoother transitions between pixels.

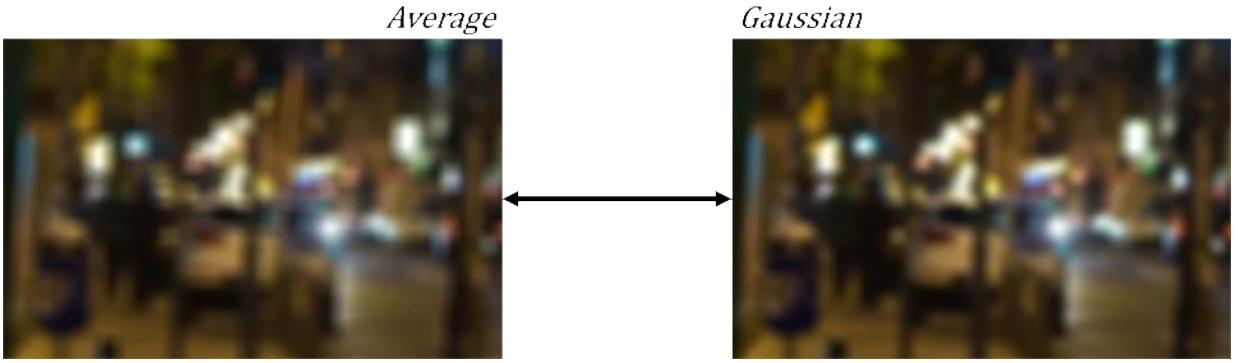


- Apply a Gaussian blur in OpenCV by using the **cv2.GaussianBlur** function, where you specify the kernel size and the standard deviations in the vertical and horizontal directions.

Gaussian



When we compare the regions with the Mean and Gaussian smoothing, the differences become apparent:



Average smoothing has a blockish texture to it, which the Gaussian blur does away with.

- Median
  - Another useful smoothing tool is the median blur, which takes the median of the pixel values within a given range around the pixel. This makes the blur be very useful for removing "salt-and-pepper" noise in an image. This blur carries the benefit of each pixel taking on a value that already exists within its proximity, and is therefore less susceptible to sudden outliers.
  - The downside of a median blur is that it cannot be achieved by mere correlation, and requires a sorting algorithm to run for every pixel. This makes this be a much more computationally intensive operation.
  - The median blur can be performed in OpenCV by using **cv2.medianBlur** and passing the desired size of pixels to be considered:

    median = cv2.medianBlur(img, 9)

- In summary, blurs allow us to remove noise from an image and make nearby pixels be more resembling of each other. This, in turn, makes images be easier to perform computations

on.

- Gradients
  - Another important use of filters comes from their ability to indicate gradients in the image. Using correlation, we can move a kernel that determines the vertical and horizontal changes between pixels in a grayscale image. Furthermore, we can use second derivatives to find the points containing the highest rates of change; these points are what will become our edge points.
  - We can find the first order partial derivatives in the vertical and horizontal directions using *Sobel* operators. These operators combine Gaussian blurring with the derivatives so that the operation is more resistant to noise. Commonly used Sobel kernels in the horizontal and vertical directions are:

$$K_{\partial x} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

□

$$K_{\partial y} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Sobel derivatives can be found in an image by calling the **cv2.Sobel** function:

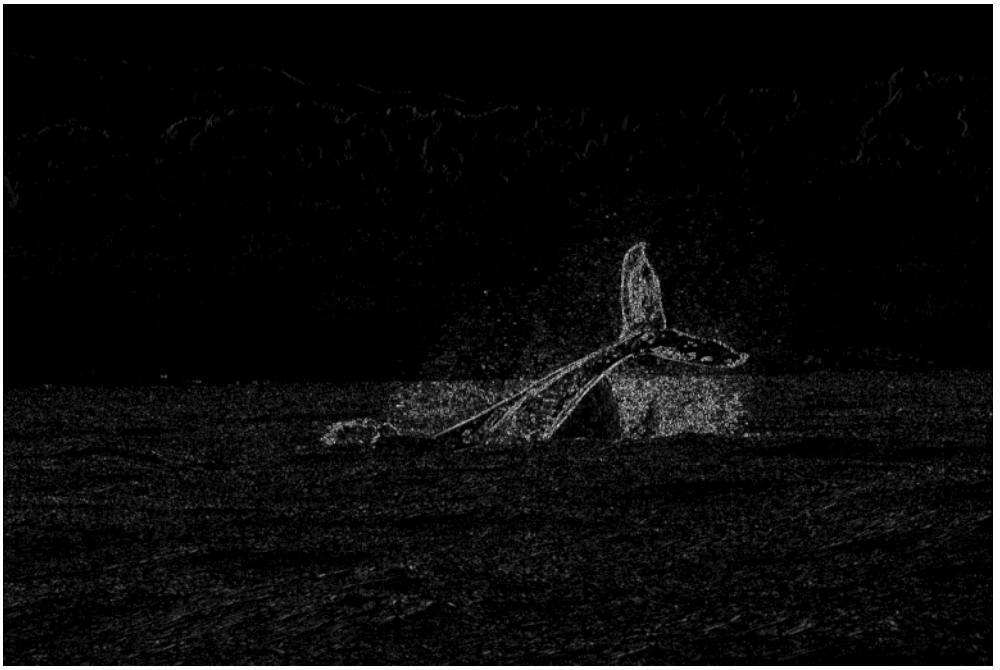
sobel_x = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=3)
sobel_y = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=3)

Here, the second argument determines the output datatype. Note that if the datatype is **uint8**, then the negative transitions will not be shown. In order to fix this, we need to take the absolute value of the derivatives and convert them back into **unit8**:

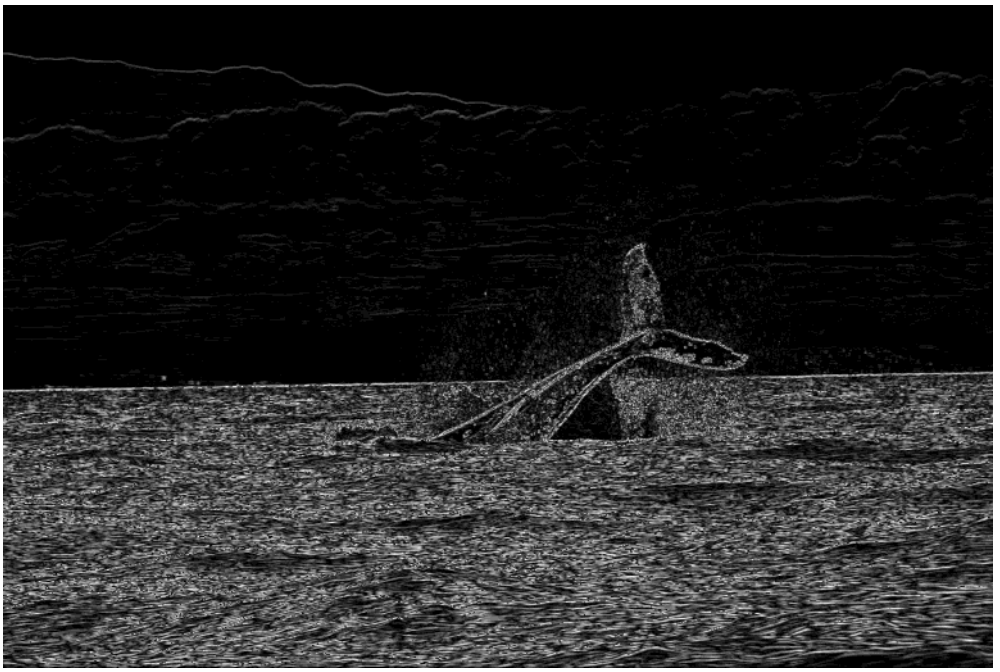
sobel_x = np.absolute(sobel_x)
sobel_y = np.absolute(sobel_y)
sobel_x = np.uint8(sobel_x)
sobel_y = np.uint8(sobel_y)

Thus, if our original image is:

- The horizontal Sobel derivative will be:



- And the vertical Sobel derivative is:



- We are interested in finding points that are edges in our image; for this purpose, it is best to look at the points where the rate of change is greatest. This is done by using the Laplacian operator, which is the sum of the second partial derivatives in both the *x* and *y* directions. The Laplacian kernel looks like:
  - $$K_{\nabla^2} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

  We can apply it to the image using **cv2.Laplacian**:

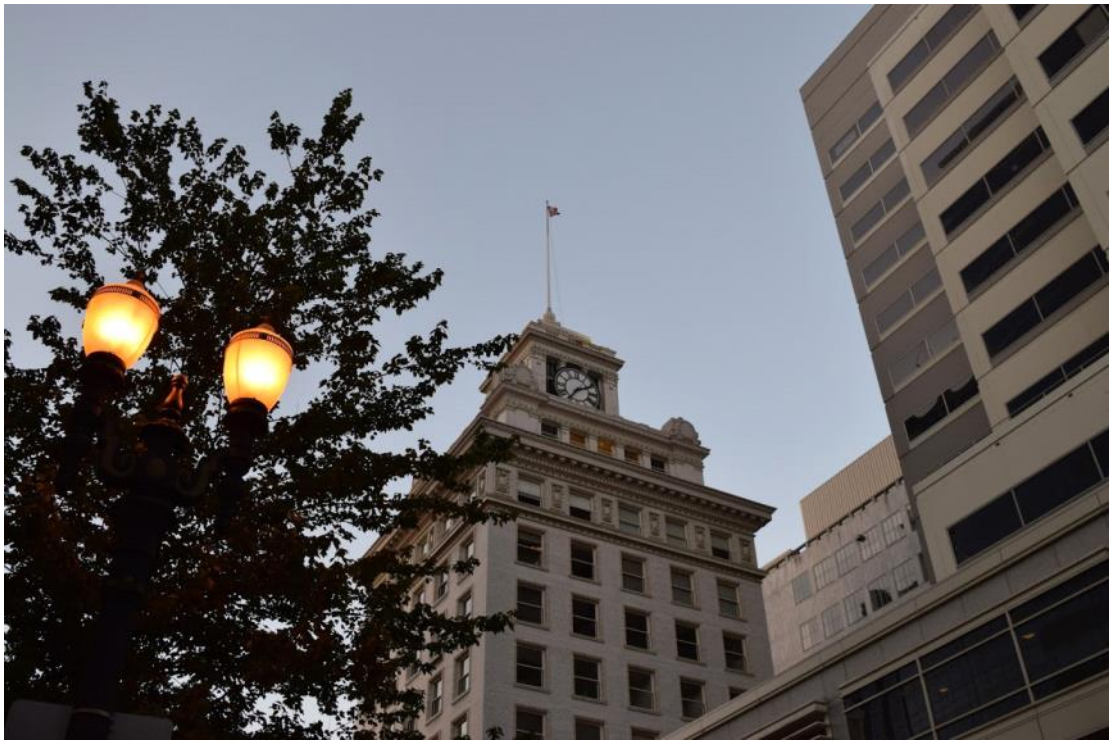
  laplacian = cv2.Laplacian(img, 0)

  the **0** signifies that the output is of type **uint8**

  The result we get is:

Canny Edge Detection

- The Laplacian provides us with the ability to detect edge points in an image. Unfortunately, the output will seldom provide us with a distinct edge; high frequency noise will often be let through the Laplacian, and curves signifying edges may have breaks in them. In order to detect edges in the optimal way, Canny edge detection is used.
- The Canny algorithm begins by applying a Gaussian filter to reduce noise in the image. It then takes *x* and *y* Sobel derivatives and uses them to determine the magnitude and direction of the gradient vector. Every pixel is checked to see whether it is a local maximum in its gradient direction; the pixels that are not are suppressed. We then apply minimum and maximum thresholds to the gradients. All values below the minimum threshold are discarded, and all values above the maximum threshold are recognized to be edges. The values between the two thresholds are then checked to see whether their pixels are connected to those above the threshold; if so, they are kept. Otherwise, the pixels are discarded. Ultimately, this algorithm outputs a binary mask with thin curves signifying edges.
- We will use Canny edge detection to find the edges of the scene below:



First, import the necessary packages and read the image as grayscale:

```
import cv2
import numpy as np
clock = cv2.imread( 'clock.jpg', 0 )
```

Smooth the image to reduce noise; due to the fine horizontal details, such as the flagpole, we select a Gaussian kernel that is shorter horizontally.

```
clock = cv2.GaussianBlur( clock, (3, 5), 0.525, .9)
```

We can now apply Canny edge detection:

```
edges = cv2.Canny( clock, 53, 160 )

cv2.imshow( 'clock', clock )
cv2.imshow( 'canny', edges )
cv2.waitKey( 0 )
cv2.destroyAllWindows()
```

-

Morphological Transformations

- As we have shown in previous examples, masking is a powerful tool that allows us to extract

image elements with certain properties. Nonetheless, masks are often subject to noise that can detract from our results. The smoothing and optimal thresholding techniques we have previously learned can help remove some of this noise, but certain amounts pass through nevertheless.

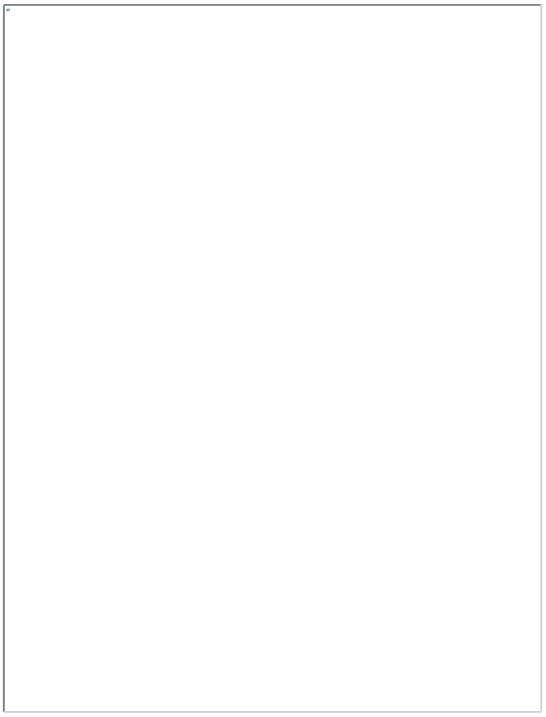
- Morphological transformations are operations that allow us to perform noise reduction on a mask itself, and proper combinations of these transformations greatly expand our ability to remove salt-and-pepper noise from a binary region.
- In OpenCV, these operations are performed on a region by a kernel, which is in itself defined as a binary mask. Similarly to correlation, the Kernel slides over a region and performs the morphological operation on the region shaded by its *on* values. Unlike correlation, however, the kernel morphological operations are binary in nature.
    - The kernel can either be created manually from a rectangular *numpy* array, as in
      kernel = np.ones((3, 3), np.uint8)
      or by using the **cv2.getStructuringElement** function, which generates the kernel. For example, a circular kernel can be generated by using:
      cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5,5))

- These fundamental morphological transformations are:
    - Erosion - the pixels in the kernel's region are kept alive if every pixel is on; otherwise, all of the pixels are turned off.

    -

      Erosion is implemented using the **cv2.erode** function:
      mask_eroded = cv2.erode(mask, kernel, iterations=1)

    - Dilation - if any pixels in the region are on, then the entire kernel is turned on; if no pixels are on in the region, then these pixels remain off.



Implement dilation using the **cv2.dilate** function:

mask_dilated = cv2.dilate(mask, kernel, iterations=1)

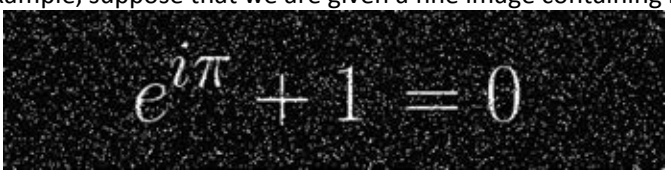
- Combinations of these two operations allow us to create more sophisticated filters for masks. OpenCV offers some preset morphological transformations, which can be applied using the **cv2.morphologyEx** command:
  - Closing - applying dilation followed by erosion. This allows us to remove negative noise from the mask while preserving the shape of the mask.
    - Close the image by using:

      closed_mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel)
  - Opening - applies erosion followed by dilation; removes positive noise while retaining the shape of the mask.
    - Apply opening by using:

      opened_mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)
  - Morphological Gradient - takes the difference between the image's dilation and erosion; returns an outline of the mask.
    - Apply the morphological gradient with:

      outline = cv2.morphologyEx(mask, cv2.MORPH_GRADIENT, kernel)

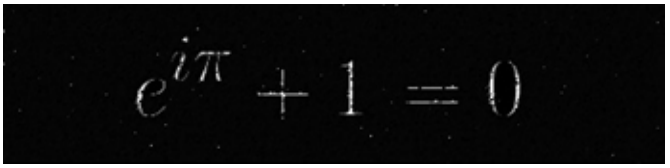
- For example, suppose that we are given a fine image containing noise:



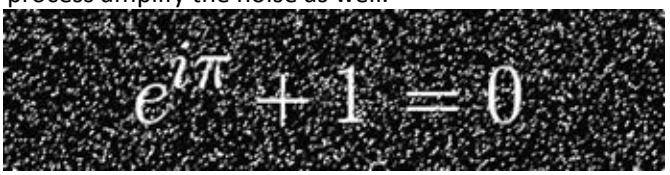
  - Using morphological transformation, we can reduce the noise levels while maintaining the desired information.
  - First, we create kernel for the transformation:

    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (2,2))
  - Then, we create a binary mask by thresholding the image:

    _, img = cv2.threshold(img, 30, 255, cv2.THRESH_BINARY)
  - If we erode the image, we remove the noise, but in the process we also destroy our desired equation.



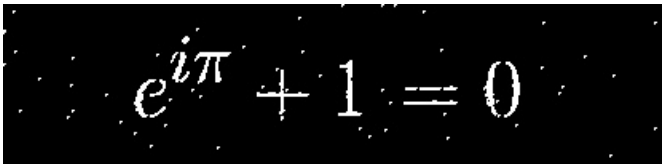
- If, on the other hand, we dilate the image, we are able to amplify the desired equation, but in the process amplify the noise as well.



- Thus, we need to use a combination of these two. Because we have positive noise, we need to perform an **opening** operation on the image:

  img = cv2.morphologyEx(img, cv2.MORPH_OPEN, kernel)

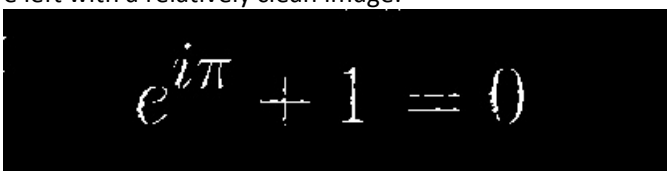
  This operation removes a majority of the noise in the image, although small areas are still left:



- In order to fix this, we apply another **erode**:

  img = cv2.erode(img, kernel)

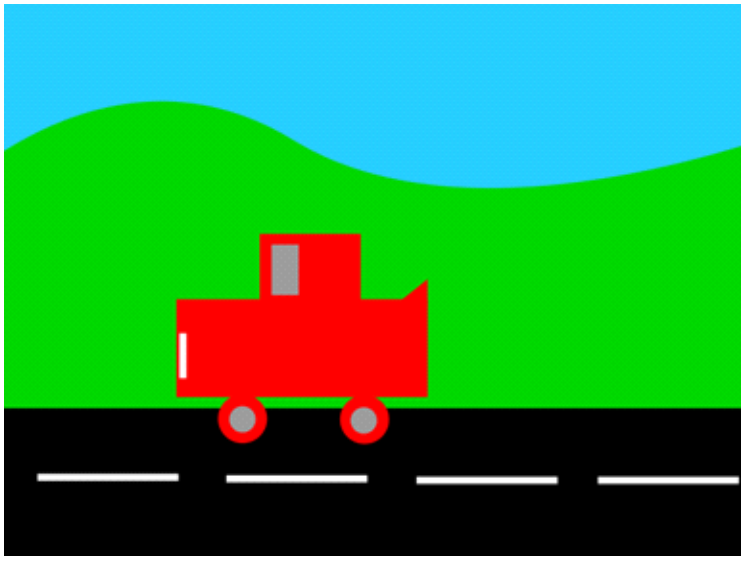
- We are left with a relatively clean image:



Contours

- The examples leading up to this point have dealt with the image as a whole; we have learned how to select specific regions based on ROI, thresholding, color, and making. We have learned to modify images and their regions by using color spaces, and we learned how to use filtering to remove noise from an image and detect edges within it. All of these algorithms involved us operating on an array of pixels, and all of the modifications we have done were on these pixels. There comes a point, however, when it is more useful to think of elements in the image as objects rather than values in an array.
- Suppose, for example, that we wish to find the center of a blob, or if we want to see whether a point is inside the area encompassed by a shape. In this case, it can be very inefficient to look at every single point inside the shape to provide us with answers, as we care only about the outer border of the shape. Once we have found where the shape itself exists in the image and need to perform mathematics with respect to the object's shape and position, we need *contours*.
- In OpenCV, **contours** are sets of connected points that contain information about the boundary of an object and its hierarchal relationship to the other contours in the image.
- Suppose we have an illustration of a red car, which we load in using

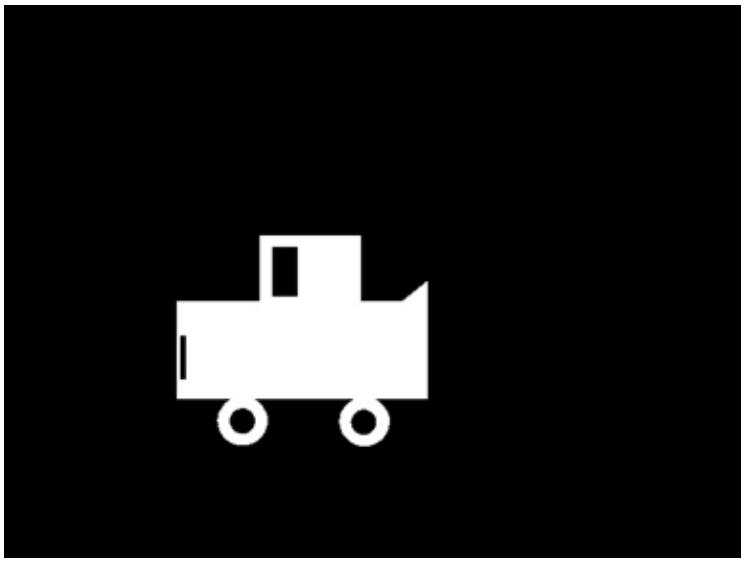
  import numpy as np
  import cv2
  img = cv2.imread('car.png')



- We detect the red body of the car using:

  mask = cv2.inRange(imgHSV, np.array((0, 10, 10)), np.array((15, 255, 255)))

  Our mask looks like:



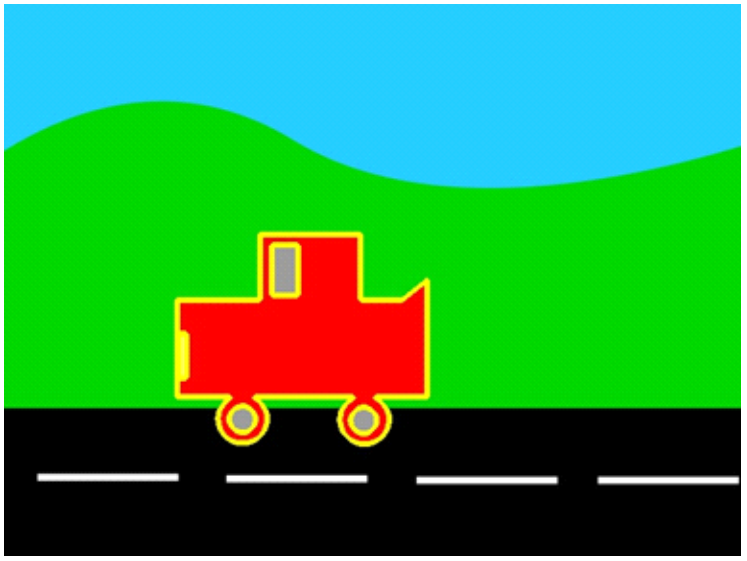
- Now that we have a solid mask, we can extract a list of contours from it. This is done by using the **cv2.drawContours** function:

  _, contours, hierarchy = cv2.findContours(mask, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)

  This function returns a list of contours, where each entry is an array with points contained within the contour, and a hierarchy, which describes the parent-child relationships of contours. the **cv2.RETR_TREE** makes **hierarchy** return the parent-child relationship for every contour, and the **cv2.CHAIN_APPROX_SIMPLE** makes **contours** contain every point that is on the contour.

  We can draw these contours in a yellow color by using:

  img = cv2.drawContours(img, contours, -1, (0, 255, 255), 3)



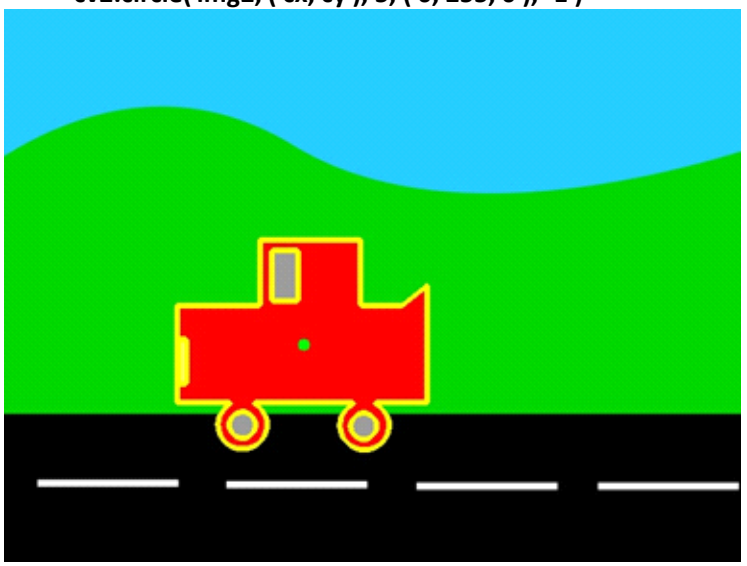
- Now that we have found the contours in the image, we are able to perform computations on them. Notice that in the image above, contours are drawn that both encapsulate the vehicle and those that outline the regions that are holes in the mask, such as in the wheels and window of the car. In our case, because all of the car contains a red region outside, we are specifically interested in this largest contour. Because of the **tree** hierarchy we specified, the main contour will be the first in our list. Thus, we can choose to perform calculations specifically on it:

  cnt = contours[0]
- We can determine the location of the centroid by using the **cv2.moments** function. Mathematically, the centroid is determined by dividing the horizontal and vertical moments by the sum of the density distribution. Similarly, we can find the centroid by dividing the moments in the *x* and *y* directions by the area of the contour:

  M = cv2.moments(cnt)
  cx = int(M['m10'] / M['m00'])
  cy = int(M['m01'] / M['m00'])

  Thus, we find the weighted center of the contour. We draw it with

  cv2.circle(img2, (cx, cy), 5, (0, 255, 0), -1)



- Another useful thing to know is the location of the points that make up the convex hull of the contour. These are the most external points of the contour, and knowing them allows us to determine the farthest regions that the contour spans to. These points can be found using the **cv2.convexHull** function:

  hull = cv2.convexHull(cnt)

  hull will contain a list of all of the outermost points of the car. We draw them with:

  for i in hull:

cv2.circle(img3, tuple(i[0]), 5, (255, 255, 255), -1)

- It is also possible to fit shapes to the contour. For example, we can find the bounding upright rectangle by using:

  x, y, w, h = cv2.boundingRect(cnt)

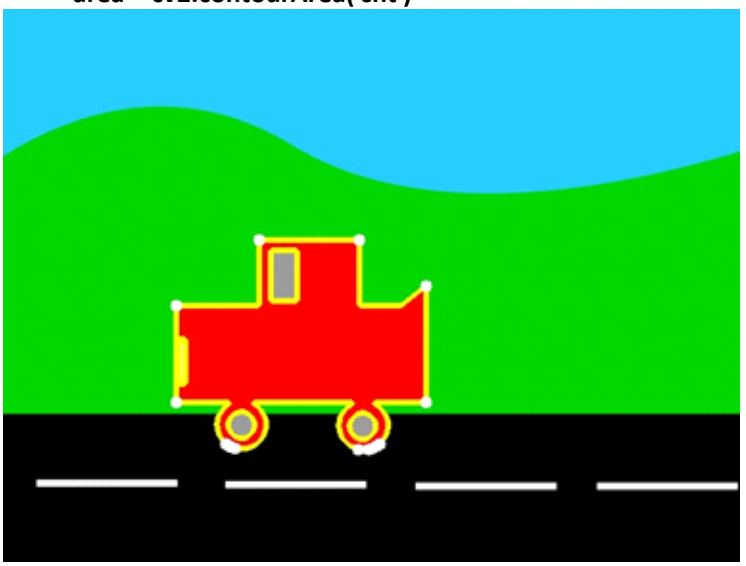
  this rectangle can be drawn with:

  cv2.rectangle(img4, (x, y), (x+w, y+h), (255, 0, 255), 2)
- Note that OpenCV contains a large set of functions for gathering information from contours. For example, we can find the perimeter of the contour by using:

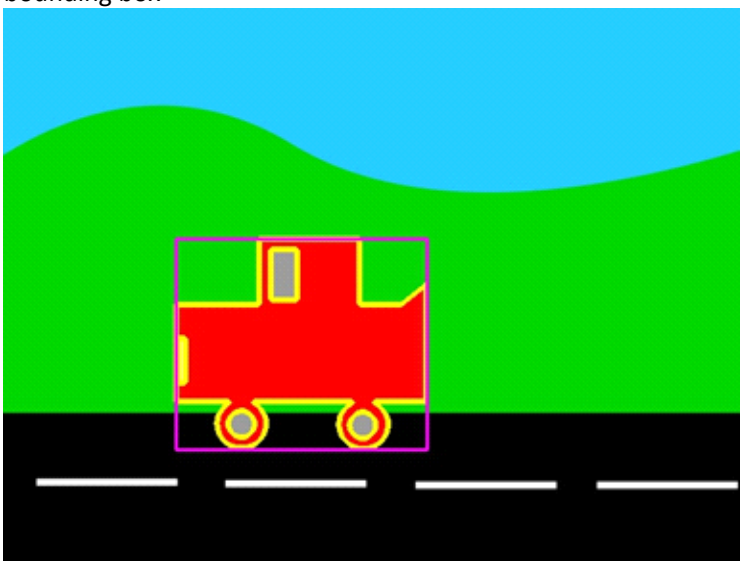
  perimeter = cv2.arcLength(cnt, True)

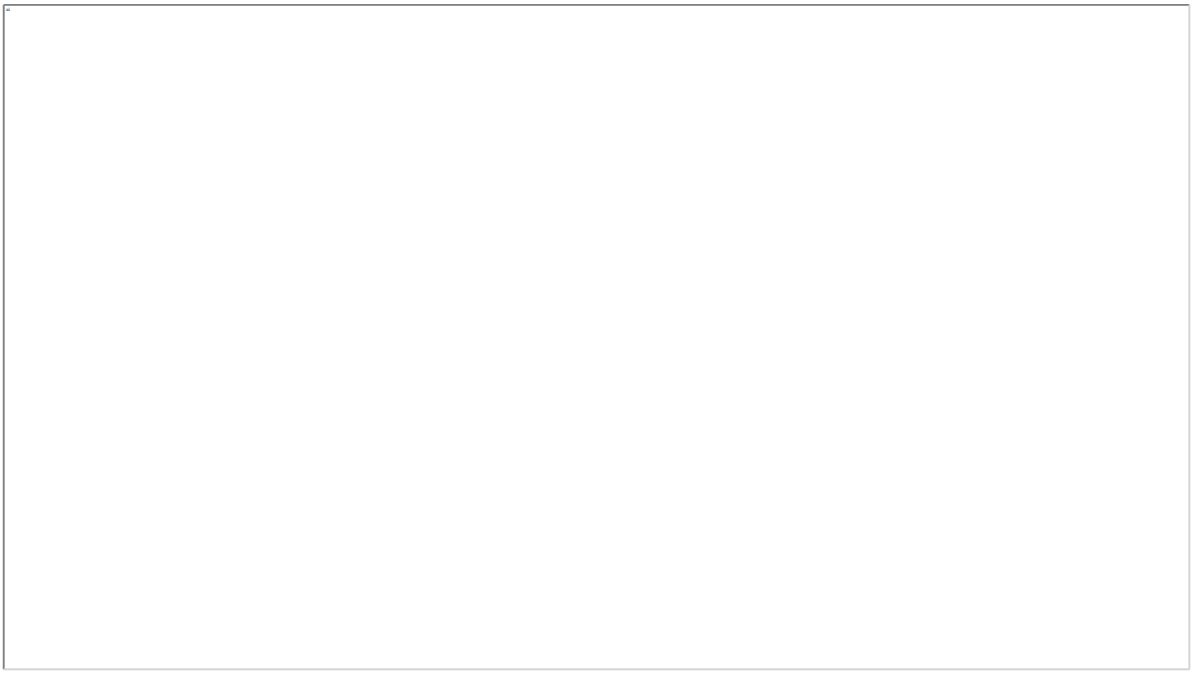
  and the pixel area is determined by:

  area = cv2.contourArea(cnt)



- bounding box

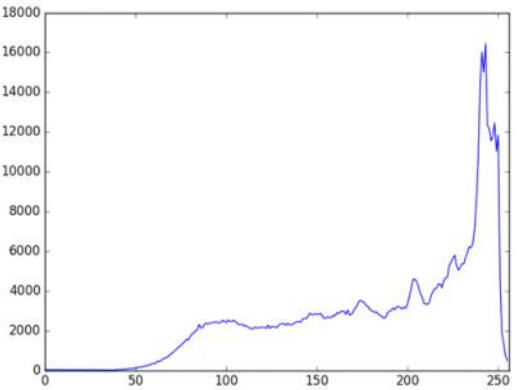


- 

Histograms

- At other times, it is useful to compare all of the values in an image or region to each other. In this case, histograms provide very valuable tools for analyzing and modifying large sections of images.
- One-dimensional Histograms
  - All a histogram does is plot the number of times a pixel value appears to the value (or, usually, the a small range of values). Single-dimensional histograms plot these intensity frequencies against intensities for a single channel, although multiple channels' histograms can be plotted on a single graph.
  - In OpenCV, finding the histogram for a single channel image is as simple as using the **cv2.calcHist** function, and this histogram can be plotted using *matplotlib*'s **pyplot** library:

    ```
    import cv2
    from matplotlib import pyplot as plt
    img = cv2.imread( 'winter.jpg', 0 )
    hist = cv2.calcHist( [img], [0], None, [256], [0, 256] )
    plt.plot( hist )
    plt.xlim( [0, 256] )
    plt.show(  )
    ```

  - The second argument **to cv2.calcHist** specifies the channel used to calculate the histogram; since we imported the image as grayscale, we only care about the first channel. The third is an optional mask to use before taking the histogram. The fourth and fifth arguments are the total size of the histogram and the domain intervals to include in the histogram.
  - 
    

- 2-dimensional Histograms
  - It is also possible to create a 2-dimensional histogram, which outputs the frequency of a

color's hue and saturation. This is done by adding an extra dimension to the vectors in **cv2.calcHist**.

- First, we import the image and convert it to **HSV**:

  img = cv2.imread('mountain.jpg')
  hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

- Then, calculate the 2D histogram:

  hist = cv2.calcHist([hsv], [0, 1], None, [180, 256], [0, 180, 0, 256])

- This returns **hist**, which is basically a 180x256 grayscale image. We can display the histogram by using either OpenCV:

  cv2.imshow('histogram', hist)

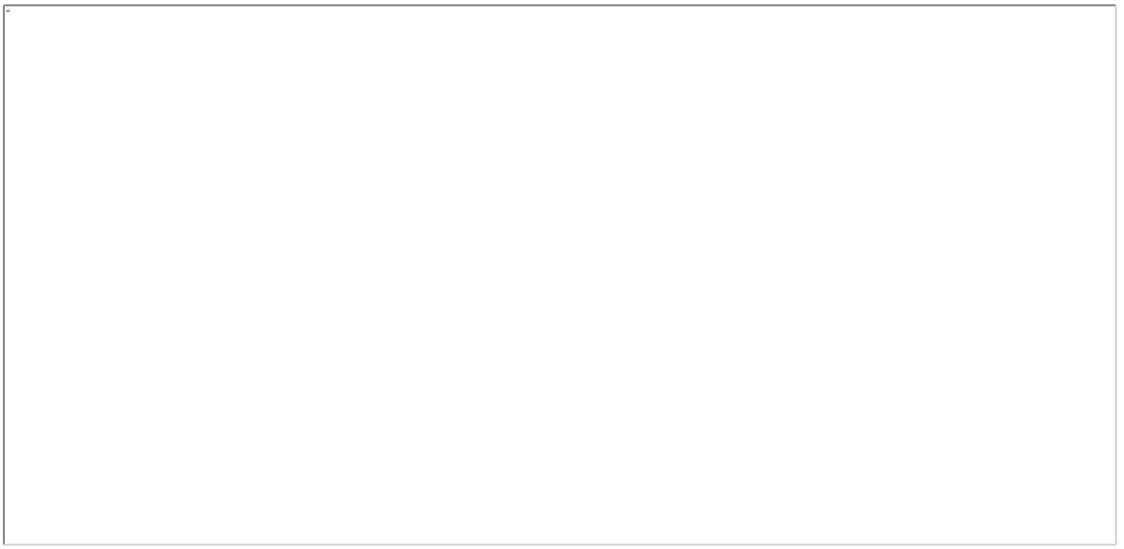
- or with matplotlib's *pyplot* module, assuming that

  from matplotlib import pyplot as plt

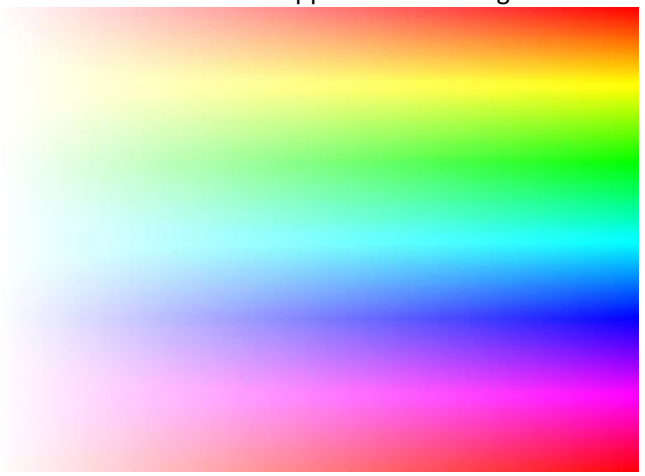
- has been imported at the top of the program:

  plt.imshow(hist,interpolation = 'nearest')
  plt.show()

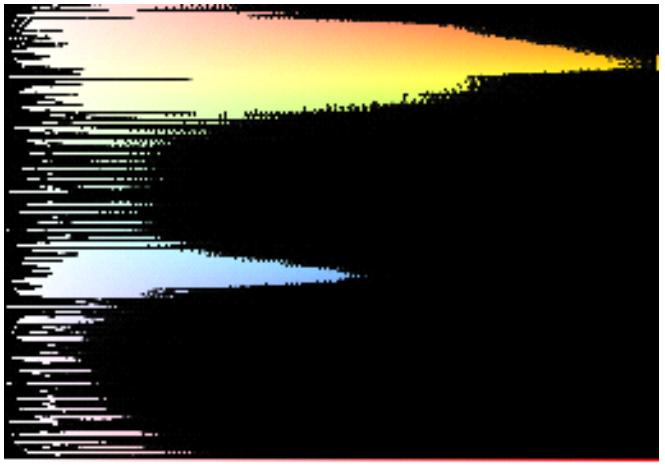
-



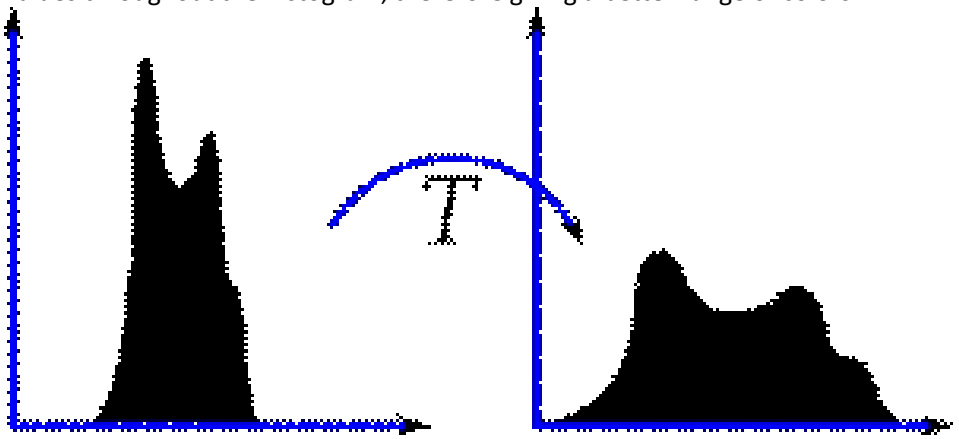
- Note how the colors are mapped on the histogram:

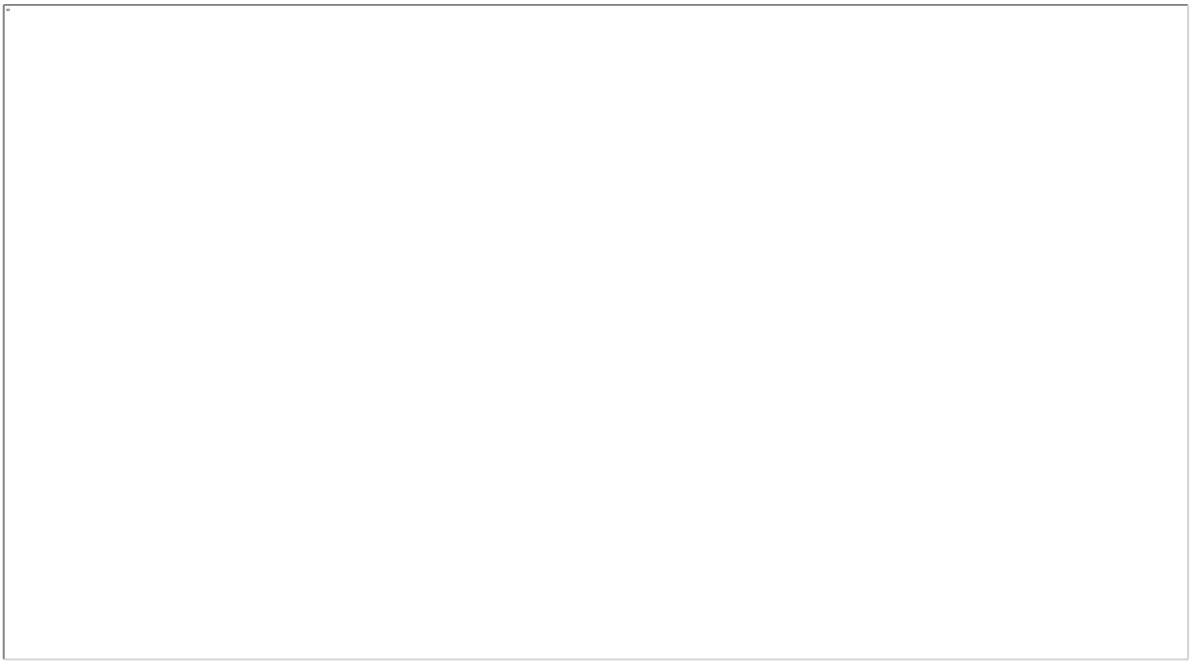


- Thus, there is a large number of pixels in the green, orange-yellow, and blue regions of the histogram.

- 

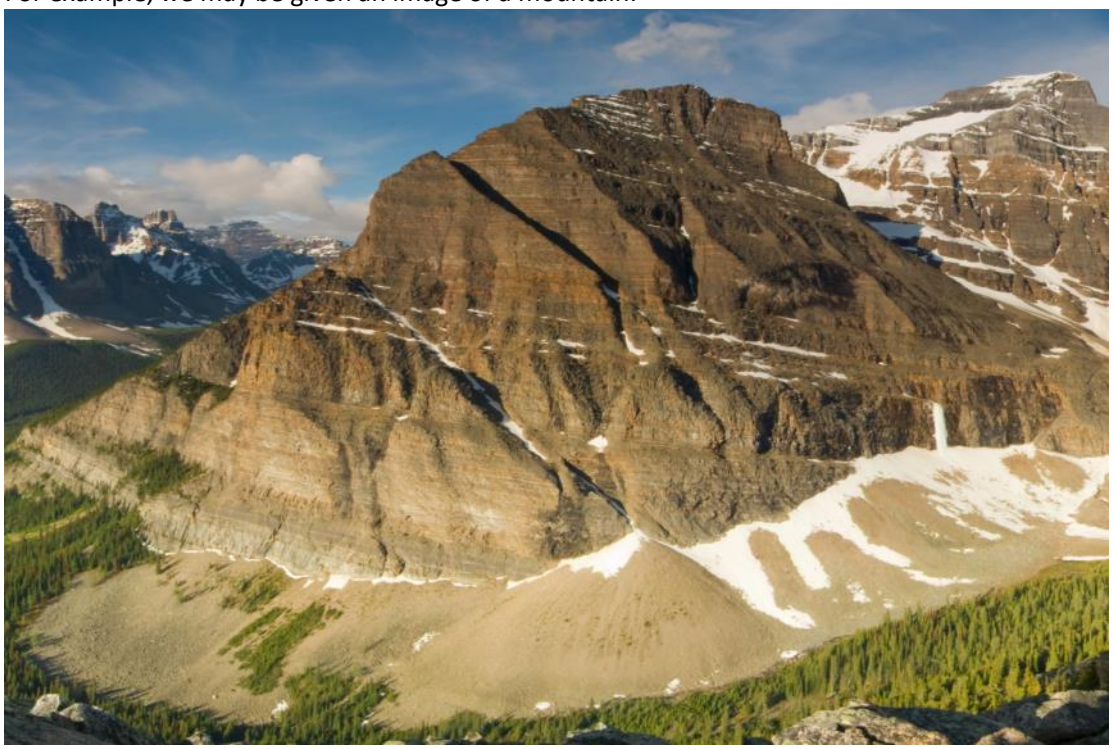
Equalization

- A once we calculate the histogram of an object, we are able to perform operations that relate the pixel values to all of the other pixels in the region. An important use of this is histogram equalization, which is used to improve the contrast of an image.
- If the majority of the histogram is clustered to one side, histogram equalization will spread these values throughout the histogram, therefore giving a better range of colors.
- 
- This is done by normalizing the pixels' frequencies by the total number of pixels in the image, distributing these frequencies throughout the whole color range, scaling these pixels back to the uint8 range, and then applying the factors that result to the pixels' intensities.
- Fortunately, OpenCV makes it easy to perform equalization by providing the **cv2.equalizeHist** function.
- To equalize an image, simply import it with
  img = cv2.imread('winter.jpg', 0)
  and equalize the image with:
  equ = cv2.equalizeHist(img)
- That's it! The histogram of the equalized image can then be found by using:
  histEq = cv2.calcHist([equ], [0], None, [256], [0, 256])
- Below are histograms from before and after the histogram equalization:

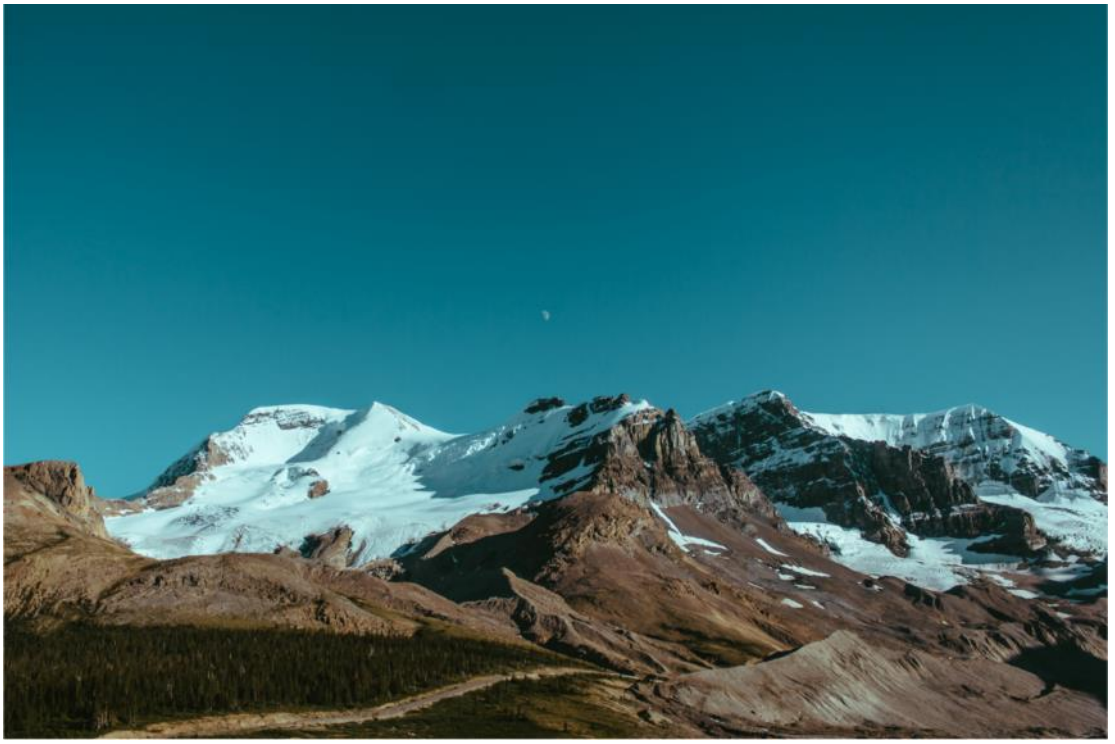
- 

Backprojection

- Histograms provide us with a powerful tool for finding objects containing certain colors sets. The **threshold** and **inRange** functions are useful for detecting objects of a certain intensity in a frame; the question becomes, however, how do we detect an object containing a variety of colors in an image? What if we want to detect objects with similar but different sets of these colors? Using 2D histograms, we can scan a region and compare it to the histogram of the object that we wish to find. We can give a probability that the two correspond to each other as the value to the pixel's location. We can then threshold this single channel of values to determine the locations where the object is most likely to be found.
- The this method is known as **backprojection**, and OpenCV provides us with the **cv2.calcBackProject** function
- For example, we may be given an image of a mountain:



- and may wish to use the texture on its surface to find a mountain in another image:



- We begin by importing the necessary packages, reading in both images, and converting them into HSV space so that we can calculate their histograms:

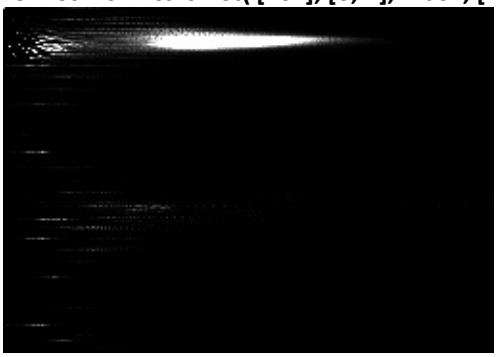
```
import cv2
import numpy as np

roi = cv2.imread( 'mtn1.jpg' )
hsv = cv2.cvtColor( roi, cv2.COLOR_BGR2HSV )

target = cv2.imread( 'mtn2.jpg' )
hsvt = cv2.cvtColor(  target, cv2.COLOR_BGR2HSV )
```

- We find a region in the first image where the colors and texture are a good representation of the mountain:

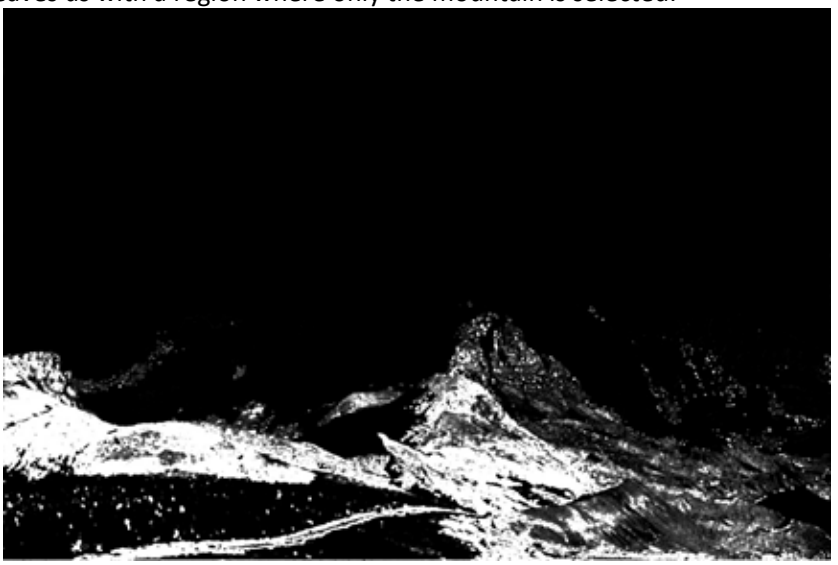
set a mask for this region of interest. This will be used in calculating the histogram for the ROI:

mask = np.zeros(roi.shape[0:2], np.uint8)
mask[302:1000, 766:1617] = 255

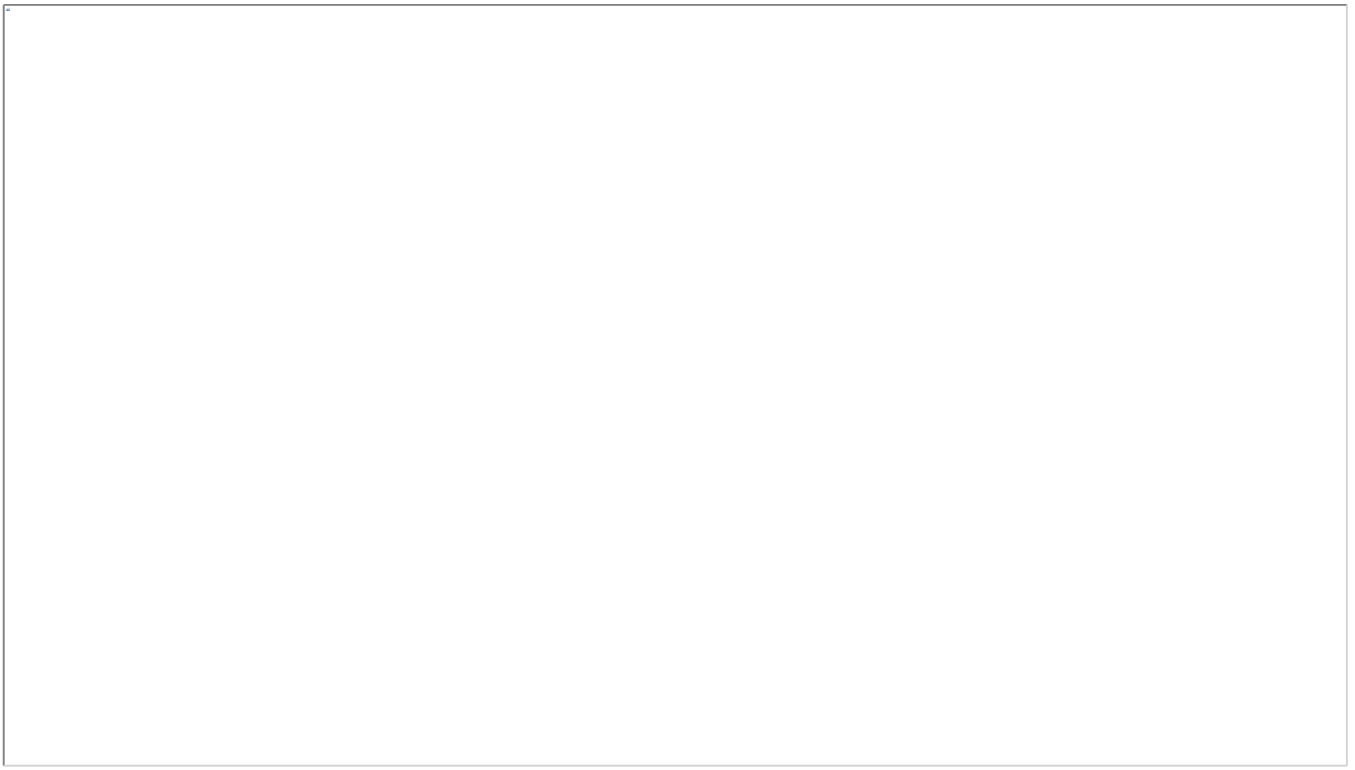
- We then calculate this region's 2D histogram:
  roihist = cv2.calcHist([hsv], [0, 1], mask, [180, 256], [0, 180, 0, 256])



- We normalize this histogram so that we have the frequencies of the colors be related to the frequencies of the other colors:
  cv2.normalize(roihist, roihist, 0, 255, cv2.NORM_MINMAX)
- We're now ready to apply the backprojection of this histogram onto the second image:
  prob = cv2.calcBackProject([hsvt], [0, 1], roihist, [0, 180, 0, 256], 1)
- We can now pass a filter through **prob** to accumulate regions of high probability:
  disk = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5,5))
  prob = cv2.filter2D(prob, -1, disk)
- This leaves us with a region where only the mountain is selected:



- Congratulations, you have completed the histogram backprojection!

- 

Color Tracker
Backprojection Example

Example: Color Tracker

Friday, August 19, 2016 10:38 AM

In this exercise, we track the center of an object of a unique color throughout the screen.

We will load video from the computer's webcam into the application. The user will click on an object of a specific color, and the program will draw a crosshair at the object.
The crosshair will then follow after the color on the screen. We will insert a slider into the user interface to help adjust the HSV ranges in which the color is accepted.

Our program structure will be as follows:

import packages
define crosshair drawing function
define callback functions
set up user interface
enter video loop
    define color ranges
    find colors in color range
    filter the mask
    create contours for the mask
    sort contours to find one with the largest area
    find its center and draw a crosshair

- Begin by importing the necessary packages
  import cv2
  import numpy as np
- Then, set a **begin** flag. This flag will be activated when the user clicks for the first time to select the color to follow
  begin = False
- We will define the function **drawXHair**, which will be used to create the crosshair at the center of the desired color
  def drawXHair(img, x, y):
- As we have learned in the OpenCV Basics lab, we will draw a red circle with two intersecting lines centered at the point we pass in, which will be the centroid of the color blob:

```
color = ( 0, 0, 255 )
radius = 20
thickn = 2
cv2.circle( img, ( int( x ), int( y ) ), 20, color, thickn )
cv2.line( img, ( x-radius, y ), ( x+radius, y ), color, thickn )
cv2.line( img, ( x, y-radius ), ( x, y+radius ), color, thickn )
```

- We can now create our callback functions. The first will be **colorSelect**, which will be called upon a mouse action in the window. When this callback is triggered, it will set the current color to the one of the pixel that was clicked.
  def colorSelect(event, x, y, flags, param):
- For this callback, **color** and the **begin** flag will be global variables.
  global color, begin
- The event on which our color selection routine executes will be when the left button is lifted:
  if event == cv2.EVENT_LBUTTONUP:
- The **begin** flag will be set to **True** if it isn't already
  if not begin:

begin = True

- And then the new color value to be detected is changed:

  color_bgr = frame[y, x, :]
  color = cv2.cvtColor(np.uint8([[color_bgr]]), cv2.COLOR_BGR2HSV)

- Our next callback is triggered when **cv2.createTrackbar** changes, and is designed to do nothing when it is triggered. The function is named accordingly.

  def doNothing(x):
  pass

- We now create the elements for our user interface. We create a window **'Track Color Object'**, in which all of the application will be contained, and set a mouse callback on it so that we can click inside the window to select our color:

  cv2.namedWindow('Track Color Object')
  cv2.setMouseCallback('Track Color Object', colorSelect)
- We then create three track bars, in which the user can enter the maximum deviation from the set HSV value in order for the color to be considered. We set the limit to this deviation as 50, and give each range a default value of **10**.

  cv2.createTrackbar('dH', 'Track Color Object', 10, 50, doNothing)
  cv2.createTrackbar('dS', 'Track Color Object', 10, 50, doNothing)
  cv2.createTrackbar('dV', 'Track Color Object', 10, 50, doNothing)

- Now that these elements are set up, we can create our video capture object and begin the video loop. We pass **cv2.VideoCapture** a value of **0**, in order to gather the video from the default webcam.

  cap = cv2.VideoCapture(0)
  while(1):

  - Inside of the loop, the first thing we do is read the HSV deviation values from the track bars.

    dh = cv2.getTrackbarPos('dH', 'Track Color Object')
    ds = cv2.getTrackbarPos('dS', 'Track Color Object')
    dv = cv2.getTrackbarPos('dV', 'Track Color Object')
  - We then place these values into an array, which can easily be added to and subtracted from the selected color.

    cRanArr = np.array([dh, ds, dv])
  - We now read the frame from the webcam. If the **begin** flag is set, we continue to perform computations on the frame; if not, then we simply show the frame:

    _, frame = cap.read( )
    if begin:

    - If **begin** is on and we can continue with our computations, we begin by finding the lower and upper HSV ranges that will select the color we have specified. We then use **inRange** to create a mask of our image.

      hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
      lower_color = color - cRanArr
      upper_color = color + cRanArr
      mask = cv2.inRange(hsv, lower_color, upper_color)
    - Now that we have a mask, we can proceed to filter it. We first use morphological closing to get rid of negative noise, and then we correlate the mask with a large, elliptical kernel in order to connect any disjoint sections of objects:

      kernel = np.ones((20,20), np.uint8)
      mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel)
      disk = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (35,35))

```
mask = cv2.filter2D(  mask, -1, disk )
```

- After modifying the mask, we are ready to find its contours:

```
im, contours, hierarchy = cv2.findContours( mask, cv2.RETR_EXTERNAL, \
cv2.CHAIN_APPROX_SIMPLE )
```

- Granted that some contours exist, we find the one with the largest area, which presumably spans our object. We do so by looping through each contour and calculating the area

```
if len( contours )>0:
    area = 0
    for i in contours:
        cnt_ar = cv2.contourArea( i )
        if cv2.contourArea( i )>area:
            largest_contour = i
            area = cnt_ar
```

- Now, we simply calculate the centroid of the largest contour and draw a crosshair at its position in the frame.

```
M = cv2.moments( largest_contour )
cx = int( M['m10']/( M['m00'] ) )
cy = int( M['m01']/( M['m00'] ) )
drawXHair( frame, cx, cy )
```

  - We show the contents of the frame, regardless of whether they were modified. If the user presses the ESC key when the program is running, it exits.

```
cv2.imshow( 'Track Color Object', frame )
if cv2.waitKey( 1 ) == 27:
    break
```

- End the program by releasing the video capture object and exiting the window:

```
cap.release(  )
cv2.destroyAllWindows(  )
```

Example: Backprojection

Friday, August 19, 2016 10:38 AM

In this exercise, we will create an interface that allows the user to select a region in the frame of a live video, which will then be backprojected onto the new video frames. This provides us with a more sophisticated model for object tracking.

Our program will be set up as follows:

Import packages
set flags and points
define region selection callbacks
begin video capture
set up window
enter video loop
convert color to HSV
create histogram mask
calculate 2D histograms
calculate backprojection
apply filter
apply thresholding
apply morphological transformations
apply colormap
show image

- Begin by importing the required packages,
  import numpy as np
  import cv2
- And set the ROI coordinates and the **selecting**, **newList**, and **begin** flags:
  xi, yi, xf, yf = 0, 0, 0, 0
  selecting = False
  newHist = False
  begin = False

- We now define the **regionSelect** mouse callback. It will contain the variables we have just defined as globals:
  def regionSelect(event, x, y, flags, param):
  global xi, yi, xf, yf, selecting, newHist, begin
  - When the user left clicks, the program goes into **selecting** mode, which causes the video to freeze, and sets the initial *x* and *y* coordinates of the region to the point where the mouse was clicked:
    if event == cv2.EVENT_LBUTTONDOWN:
    selecting = True
    xi, yi = x, y
  - When the mouse click is released, the ending coordinates of the region are set to the location where the left button was let up. **selecting** is turned off, allowing the screen to unfreeze, and **newHist** and **begin** are set to True. **newHist** tells the program that there is a new region selected whose histogram needs to be calculated, and **begin** tells the program that it can begin performing computations on the frames, as a region is now selected.
    elif event == cv2.EVENT_LBUTTONUP:
    xf, yf = x, y
    selecting = False
    newHist = True
    begin = True

- Now that the callback is defined, we can create a **VideoCapture** object that will receive video from the default webcam and create a new window, **"frame"**, in which our region selection interface will occur.

- We can now enter the video loop. first, we check that **selecting** is not on; if it is, we jump directly to displaying the

current frame, without grabbing a new one. Otherwise, we read a new frame:

```
cap = cv2.VideoCapture(0)

while(True):
    if not selecting:
        _, frame = cap.read()
```

- If **begin** is set, we continue with our computations. Otherwise, we simply display whatever the webcam sees:

```
if begin:
```

  - We now start performing computer vision on the frame. First, we convert the frame to HSV

```
hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
```

  - We then create a mask for our ROI, from which we will create our histogram:

```
mask = np.zeros( frame.shape[:2], np.uint8 )
mask[min( yi, yf ) : max( yi, yf ), min( xi, xf ):max( xi, xf )] = 255
```

  - Next, we check the **newHist** flag to see if a new ROI has been created; if so, we calculate a new ROI histogram:

```
if newHist:
    roiHist = cv2.calcHist( [hsv], [0, 1], mask, [180, 256], [0, 180, 0, 256] )
    roiHist = cv2.normalize( roiHist, roiHist, 0, 255, cv2.NORM_MINMAX )
    newHist = False
```

  - We now calculate the histogram of the current frame, **targetHist**, and perform the backprojection of **roiHist** onto it:

```
targetHist = cv2.calcHist( [hsv], [0, 1], None, [180, 256], [0, 180, 0, 256] )
dst = cv2.calcBackProject( [hsv], [0, 1], roiHist, [0, 180, 0, 256], 1 )
```

  - Now that we have a map of the backprojection, we apply a filter to it, such that closely spaced points can combine into regions:

```
disk = cv2.getStructuringElement( cv2.MORPH_ELLIPSE, ( 15,15 ) )
dst = cv2.filter2D( dst, -1, disk )
prox = np.copy( dst )
```

  - Note that this also amplifies some noise in the image. In order to counter that, we apply a threshold, and then do a morphological erosion and dilation to remove some of the extra pixels. Then, we perform a **bitwise and** between this thresholded and filtered mask with the previous proximity map in order to mask out the pieces with increased noise:

```
_, thresh = cv2.threshold( dst, 250, 255, 0 )
kernel = cv2.getStructuringElement( cv2.MORPH_ELLIPSE, ( 5,5 ) )
thresh = cv2.erode( thresh, kernel, iterations = 3 )
thresh = cv2.dilate( thresh, kernel, iterations= 3 )
masked_dots = cv2.bitwise_and( prox, prox, mask = thresh )
```

  - We now apply a color map, in order to make our display more visually appealing, and show the backprojection:

```
masked_dots = cv2.applyColorMap( masked_dots, cv2.COLORMAP_JET )
cv2.imshow( 'distance', masked_dots )
```

- We now display the current frame, regardless of whether **selecting** is enabled:

```
cv2.imshow( 'frame', frame )
```

- If the user presses **ESC**, we exit the video loop:

```
if cv2.waitKey(1) & 0xFF == 27:
    break
```

- We now release the video capture and close all windows:

```
cap.release()
cv2.destroyAllWindows()
```

Building a Motion Detector

Thursday, August 4, 2016 3:58 PM

We can now begin using OpenCV to create some pretty useful tools. These tools can then be enabled to trigger MQTT events, where data triggered from the video will be sent over a network. We can use OpenCV to transform a camera connected to the gateway into a sensor, whose data will trigger events, just as a physical sensor would.

In this example, we will build an network-connected motion detector, which will trigger an MQTT event and save video when motion is found in the frame.

Our motion detector will observe the difference between consecutive frames; when this difference is high, we can assume that motion was found. In order to prevent false positives, we will observe the standard deviation of the frame. When motion of a suitably sized object is detected, the standard deviation will rise, allowing us to trigger a *motion* event.

Our program will be laid out as follows:

set up **utils** package
    arguments
    tools
    trigger
import OpenCV and its dependencies
initialize values
create stdev average select callback
set up video recorder (if option is set)
start video loop
    calculate distance between frames
    shift frames
    apply Gaussian blur to distance mapping
    apply thresholding
    calculate st dev
    append for avg stdev calculation
    if motion is detected:
        send off trigger
        record (for specific # of frames)
    display st dev
release captures

In this example, we create an application that serves a practical purpose--it detects motion while filtering out small variations, a functionality that can be very useful for an intelligent security camera. In order to make this application be more robust, however, we give the users greater control over the program. Therefore, in this and the proceeding projects we will include a **utils** package, which will consist of useful modules as well as tasks like argument handling.

utils

- Begin by creating a **utils** directory and create a file inside of it called **__init__.py**. This file will combine all of the modules that we have put together.
- include the following imports in the file:
    from trigger import *
    from dtinfo import currDate, currTime
    from args import *
    from tools import *

args

- Next, create a module named **args.py** inside the **utils** directory. This is where we will place all of our command line arguments, which will control things like pre-setting the standard deviation, specifying whether video will be recorded when motion is detected, if the former then how much video to record, the camera or address of the video source, etc.
- in order to handle arguments, we import the **argparse** module:
    import argparse
- we then instantiate an argument parser:
    parser = argparse.ArgumentParser()
- and list every argument we would every expect the user to need to set…
    parser.add_argument("-l", "--length", help="number of frames to record upon detection", type=int, default=1000)
    parser.add_argument("-f", "--fps", help="framerate of video recording", default=20, type=int)
    parser.add_argument("-u", "--url", help="IP camera location url", type=str)

```
parser.add_argument("-c", "--camera", help="location of USB camera", default=0, type=int)
parser.add_argument("-ht", "--height", help="video recording height", default=480, type=int)
parser.add_argument("-w", "--width", help="video recording width", default=640, type=int)
parser.add_argument("-t", "--threshold", help="set threshold value", default=2.0, type=float)
parser.add_argument("-v", "--visual", help="turn on visual mode", action="store_true")
parser.add_argument("-n", "--noise", help="set noise multiplier", default=.005, type=float)
parser.add_argument("-d", "--debug", help="turn on debug mode", action="store_true")
parser.add_argument("-s", "--stdev", help="set average st dev", default=0, type=float)
parser.add_argument("-r", "--record", help="enable recording", action="store_true")
parser.add_argument("-m", "--message", help="enable MQTT transmission", action="store_true")
```

- When the program loads, we parse the command line options for these arguments.

```
args = parser.parse_args()
```

- We then give variables these argument values, which will be the exports of this module.

```
message = args.message
visual = args.visual
record = args.record

sdAvg = args.stdev
sdThresh = args.threshold
noise = args.noise # scaling of noise; 0-1
debug = args.debug
recordLength = args.length
# frame dimensions
frameHeight = args.height
frameWidth = args.width
# recording frame rate
frameRate = args.fps
```

- For the camera, we check whether a URL is given for the source; if not, we assign the video source with the camera value:

```
if args.url:
    dest = args.url
else:
    dest = args.camera
```

dtinfo

- in the same directory, create a module **dtinfo.py**. This module includes macros that return the current date and time as strings, which can then be sent with the trigger information

```
import time

# current date
def currDate(): # day-month-year
    return time.strftime("%d-%m-%Y")
# current time
def currTime(): # hours-minutes-seconds
    return time.strftime("%H-%M-%S")
```

tools

- We next create a **tools.py** module, which will contain useful functions for our project.
- Begin by importing **numpy**:

```
import numpy as np
```

- The first function, **distMap**, gives us the Pythagorean distance between the three BGR layers of 2 frames.

```
def distMap(frame1, frame2):
    """outputs pythagorean distance between two frames"""
    frame1_32 = np.float32(frame1)
    frame2_32 = np.float32(frame2)
    diff32 = frame1_32 - frame2_32
    norm32 = np.sqrt(diff32[:,:,0]**2 + diff32[:,:,1]**2 + diff32[:,:,2]**2)/np.sqrt(255**2 + 255**2 + 255**2)
    dist = np.uint8(norm32*255)
    return dist
```

- the second, **applyNoise**, creates noise that we can use for debugging purposes:

```
def applyNoise(frame, scale):
    noises_p = np.random.rand(*np.shape(frame))
    noises_n = -np.random.rand(*np.shape(frame))
    noises = (noises_p + noises_n) * 255 * scale
    noiseFrame = frame + np.uint8(noises)
    return noiseFrame
```

trigger

- the last module in our **utils** directory will be **trigger.py**, which will emit an MQTT request whenever we call the **trigger** function.
- we will use the **paho** library to publish MQTT requests, and the **JSON** format to carry this data. Import these modules by using
  ```
  import paho.mqtt.publish as mqtt
  import json
  ```
- Also, import he **message** variable from **args**, which decides whether an MQTT request is sent in the first place:
  ```
  from args import message
  ```
- We now define the **trigger** function, which accepts a **info** dictionary:
  ```
  def trigger(info):
  ```
  - we now use **json.dumps** to create a JSON object from the **info** dict:
    ```
    infoJSON = json.dumps(info)
    ```
  - If **message** was enabled from the command line arguments, we try to send off an MQTT request:
    ```
    if message:
        try
            mqtt.single("sensors/video/motion", infoJSON, hostname="localhost")
        except:
            print "No MQTT connection found"
            pass
    ```
  - Regardless of whether messaging is enabled, print the event
    ```
    print "Event triggered:", info["event"], "!"
    return 1
    ```

Now that we have created our supporting **utils** package, we are ready to build our OpenCV motion detector.

- Begin by importing the necessary dependencies, as well as the package we have just created:
  ```
  import numpy as np
  import cv2
  import utils
  ```

- Throughout this program, we will be keeping track of the **utils.visual** flag. This will allow us to display video if we are using system with a visual interface; if the system is running headless, however, leaving the flag unset will allow us to run motion detection autonomously.
  ```
  if utils.visual:
      cv2.namedWindow('frame')
      cv2.namedWindow('dist')
  ```
- We can now begin our video capture:
  - cap = cv2.VideoCapture(utils.dest)
- We set a **DISP_STDE**V literal in the program. This can be implemented as an argument, but is mostly usable when we are building and testing our program. If this flag is set, we will open a separate window, displaying the standard deviation.
  ```
  DISP_STDEV = True
  ```
- Next, we set the **DEBUG** flag. If **DEBUG** is true, we will forcefully apply noise to our program, to ensure that our motion detection model is robust enough for positive noise.
  ```
  DEBUG = utils.debug
  ```
- We use the **calcSDAvg** flag to state whether we are calculating the standard deviation of the scene. This will be used with our **stAvgSelect** callback.
  ```
  calcSDAvg = False
  ```
- Lastly, we set a **triggered** flag to **False**. This flag will be turned on whenever motion is detected and off when motion stops. This allows us to fire MQTT requests only when motion begins.
  ```
  triggered = False
  ```
- We now create our **stAvgSelect** callback, which begins averaging the standard deviation values on the first mouse click and ends the averaging on the second.
  ```
  def stAvgSelect(event, x, y, flags, param):
      global calcSDAvg, sdList, sdAvg
      if event == cv2.EVENT_LBUTTONDOWN:
          if not calcSDAvg:
              sdList = np.array([])
              calcSDAvg = True
              print "Click again to set average standard deviation"
          else:
              calcSDAvg = False
              sdAvg = np.mean(sdList)
              print "The mean standard deviation is {}".format(sdAvg)
  ```

- Next, we begin the main part of our script by specifying what the standard deviation will be. We check whether the standard deviation was input as a command line option, and if t was not we set it to a typical value, **2.5**, and offer the user to click on the screen to set a new average standard deviation for the current environment.
  ```
  if utils.sdAvg::
  ```

```
            sdAvg = utils.sdAvg
        else:
            sdAvg = 2.5
            cv2.setMouseCallback('frame', stAvgSelect)
            print "Click to begin capturing average standard deviation"
```

- Next, we set up our video recording. If the **record** argument was set when the program started, we will set up a video writer object and will write to it when our program is running. If this flag was not set, however, we will merely skip over this.

```
        if utils.record:
            # create the VideoWriter object
            fourcc = cv2.VideoWriter_fourcc(*'MJPG') # MJPG is encoding supported by Windows
            # create output video file name
            fname = utils.currDate() + "_" + utils.currTime() + ".avi"
            # configure output video settings
            out = cv2.VideoWriter(fname, fourcc, utils.frameRate, (utils.frameWidth, utils.frameHeight))
        else:
            fname = ''
```

- We now begin reading video frames. Because the motion detector requires at least two frames to function (we use three; this amplifies motion), we make two frame readings before beginning the video loop

```
        _, frame1 = cap.read()
        _, frame2 = cap.read()
```

- Due to the security applications of this project, when an MQTT trigger is sent out, it can be useful to state where the frames taken at the time that the event occured can be found. We provide this option by saving all the video that our program records to a single file, and then by pointing to the frame number in the video. In order to do this, however, we must keep track of the frames in our video; thus, we keep a frame counter and a **lastStart** variable, which tells us of the last time the video was started.

```
        frameCount = 1
        lastStart = -1
```

- We are now able to begin our video loop.

```
        while (True):
            _, frame3 = cap.read()
            rows, cols, _ = np.shape(frame3)
```

  - if debug mode is enabled, as we have previously said, we can apply noise to the frame:

```
            if DEBUG:
                frame3 = utils.applyNoise(frame3, utils.noise)
```

  - next, the crucial step in our program is to take the difference between two frames; we do this by using the **distMap** function we have created:

```
            dist = utils.distMap(frame1, frame3)
```

  - Now that this is done, we can shift our frames:

```
            frame1 = frame2
            frame2 = frame3
```

  - Next, we apply Gaussian smoothing to even out our distance mapping:

```
            mod = cv2.GaussianBlur(dist, (9,9), 0)
```

  - And threshold this result to retrieve a binary mapping of where motion is taking place.

```
            _, thresh = cv2.threshold(mod, 100, 255, 0)
```

  - At this point, we have a binary array that indicates where motion has occurred and where it has not. Now, we will use standard deviation to calculate where the motion is significant enough to trigger an alarm.
  - Calculate the standard deviation using:

```
            _, stDev = cv2.meanStdDev(mod)
```

  - Next, check the status of the **calcSDAvg** flag. If it is set, then the program is in the process of finding the average standard deviation for the scene, and we simply have to append the current standard deviation to the list of previous deviations.

```
            if calcSDAvg:
                sdList = np.append(sdList, stDev)
```

  - If this is not occuring, however, we proceed to check whether the current standard deviation exceeds the average plus the allowable threshold, and that it was not active in the previous iteration. If these conditions are met, we fire off a trigger.

```
            else:
                if stDev > sdAvg + utils.sdThresh:
                    if not triggered:
                        lastStart = frameCount
```

    - In this trigger, we incorporate the name of the event, a uri of the server on which the video will be saved (if video recording is enabled), the time at which the video took place, and the frame of the video at which the event is fired. This is written as a Python dictionary, which will later be converted into a JSON object before being sent by MQTT

```
                        triggerInfo = {
                            "event": "MotionDetect",
                            "uri": "http://gateway" + "/" + fname,
                            "timestamp": utils.currDate() + "--" + utils.currTime(),
```

"offsetframe": lastStart
}

- We now send off a trigger and turn on **triggered** if the frame has transmitted:
  triggered = True

- If the standard deviation does not exceed the average plus threshold values, we assume that no motion is detected
  else:
  triggered = False

- lastly, if the **visual** and **DISP_STDEV** flags are enabled, we create a window that displays the current frame's standard deviation on the screen.

```
if utils.visual and DISP_STDEV:
    blank = np.ones(np.shape(frame3))
    font = cv2.FONT_HERSHEY_SIMPLEX
    cv2.putText(blank, "{}".format(stDev[0][0]), (cols/8, rows/2), font, 5, (0, 0, 255), 3, cv2.LINE_AA)
    cv2.imshow("stDev", blank)
```

- We exit the video loop when the ESC key is pressed:

```
if cv2.waitKey(1) & 0xFF == 27:
    break
```

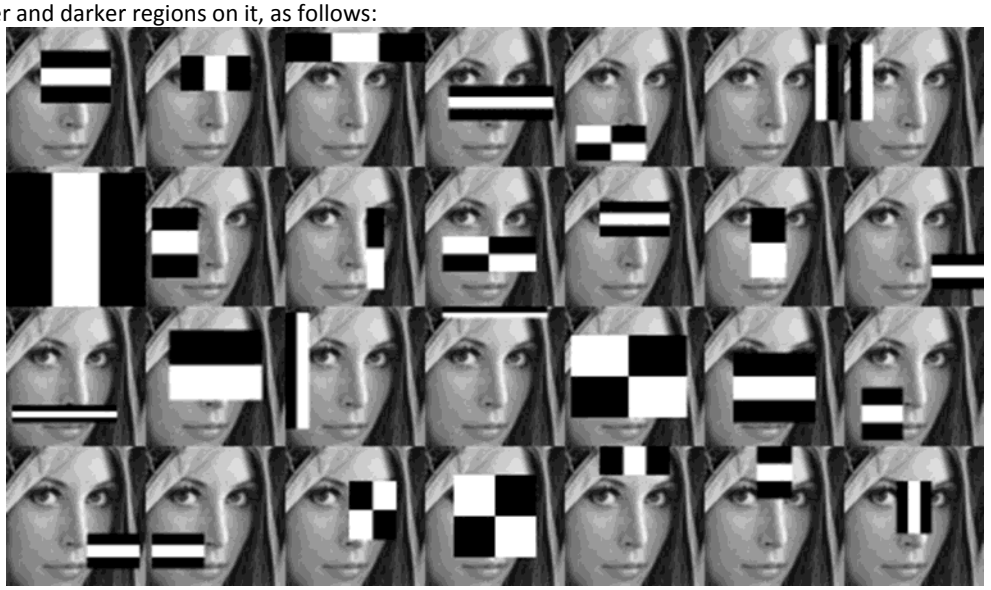
- Once the loop was exited, we release the video capture and recording. We then close the window.

```
if utils.record:
    out.release()
cap.release()
cv2.destroyAllWindows()
```

Building a Face Detector

Thursday, August 4, 2016 3:58 PM

The fundamentals we have previously shown can be combined to construct algorithms for detecting objects. One popular algorithm is the Viola-Jones detector, which uses binary classifiers to detect features in an image. For example, a face can have a broad feature set, indicated by lighter and darker regions on it, as follows:



We can train a classifier set using a large number of positive and negative images to detect these Haar features, as they are called. The Adaboost algorithm is applied to find the features that are more important, which are given heavier weights. After we run this process, we can detect objects that fulfill these profiles in an image by sending them through a *Haar cascade*. In this process, regions are checked for subsets of these classifiers, and the checking only continues if these classifiers are detected. This allows for very rapid detection of objects Haar features in different layers of an image.

In this example, we will create a system that detects faces in a video stream. Upon detection, we will write video to a file and trigger an MQTT request.

The flow of our program will be as follows:

import dependencies
create a buffer mask
instantiate a cascade classifier for the face
create a video capture from a given source
create video recording object
set frame counter
begin video loop:
    read frame
    convert to grayscale
    detect faces in image using Haar cascade
    update current buffer
    draw rectangle around faces
    transmit MQTT request if first face is found
    record video if desired
    display frame with faces indicated by rectangles

First, we set up our **utils** package, just as we have done in the motion detection example.

__init__.py

- Set up the initialization file, and include the following modules:

```
from trigger import *
from sys import argv
from buff import *
from dtinfo import currDate, currTime
from args import *
```

- Note our inclusion of the **buff** module, which is used to stabilize the face detection

args.py

- Here, we set up the command line arguments for the application. This part strongly resembles that of the motion detection example, except for the options for **scale** and **cascade**, which are specific to Haar object detection, and **buffer**, which is used to add stability to the

video recording.

```
import argparse
parser = argparse.ArgumentParser()

parser.add_argument("-l", "--length", help="number of frames to record upon face detect", type=int, default=1000)
parser.add_argument("-f", "--fps", help="framerate of video recording", default=20, type=int)
parser.add_argument("-b", "--buffer", help="calibrate face detection buffer length", default=30, type=int)
parser.add_argument("-u", "--url", help="IP camera location url", type=str)
parser.add_argument("-c", "--camera", help="location of USB camera", default=0, type=int)
parser.add_argument("-ht", "--height", help="video recording height", default=480, type=int)
parser.add_argument("-w", "--width", help="video recording width", default=640, type=int)
parser.add_argument("-k", "--cascade", help="Haar cascade classifier filename", type=str, default="haar_face.xml")
parser.add_argument("-s", "--scale", help="adjust the scale factor", default=1.25, type=float)
parser.add_argument("-d", "--debug", help="turn on debug mode", action="store_true")
parser.add_argument("-v", "--visual", help="turn on visual mode", action="store_true")
parser.add_argument("-r", "--record", help="enable recording", action="store_true")
parser.add_argument("-m", "--message", help="enable MQTT transmission", action="store_true")
args = parser.parse_args()
```

- The recording length tell the program how many frames to record after a face has been discovered. BUFF_LEN becomes a global parameter that specifies the number of frames after facial detection to count the face as being active on the screen, even if it temporarily isn't.
- We now convert these parsed arguments into variables exported by the **args** module:

```
BUFF_LEN = args.buffer
recordLength = args.length

if args.url:
    dest = args.url
else:
    dest = args.camera
cascPath = args.cascade
scaleFactor = args.scale
frameHeight = args.height
frameWidth = args.width
frameRate = args.fps

debug = args.debug
visual = args.visual
record = args.record

message = args.message
```

buff.py

- We now create the **buff** module, which will contain the **genBuffMask** macro. This generates a mask consisting of a specific number of ones. This mask will be bitwise and-ed with the buffer.

```
def genBuffMask(bufferFrames):
    'create bitwise mask for buffer length'
    # buffmask = 0...01...(n)...1
    buffMask = (2**bufferFrames) - 1
    return buffMask
```

dtinfo.py

- We create the date and time string macros, just as we have in the motion example:

```
import time

# current date
def currDate(): # day-month-year
    return time.strftime("%d-%m-%Y")
# current time
def currTime(): # hours-minutes-seconds
    return time.strftime("%H-%M-%S")
```

trigger.py

- Similarly, we use the same module as in the motion detection example to send MQTT commands:

```
import paho.mqtt.publish as mqtt
import json
```

```
from args import message
def trigger(info):
    infoJSON = json.dumps(info)
    if message:
        try
            mqtt.single("sensors/video/face", infoJSON, hostname="localhost")
        except:
            print "No MQTT connection found"
            pass
    print "Event triggered:", info["event"], "!"
    return 1
```

face_record.py

- We now proceed to the main part of the application.
- begin by importing **cv2** and **utils** dependencies:

```
import cv2
import utils
```

- Next, create a buffer mask with our **utils.genBuffMask** function. This returns a sequence of ones the length of BUFF_LEN

```
buffMask = utils.genBuffMask(utils.BUFF_LEN)
```

- We now instantiate the cascade classifier object, which will be used to detect faces in the image:

```
faceCascade = cv2.CascadeClassifier(utils.cascPath)
```

- Next, begin a **cv2.VideoCapture** instance and a video writer object, if **utils.record** is selected:

```
cap = cv2.VideoCapture(utils.dest)

if utils.record:
    fourcc = cv2.VideoWriter_fourcc(*'MJPG') # MJPG is encoding supported by Windows
    fname = utils.currDate() + "_" + utils.currTime() + ".avi"
    out = cv2.VideoWriter(fname, fourcc, utils.frameRate, (utils.frameWidth, utils.frameHeight))
```

- Create our buffer, **currBuff**, and set it to zero

```
currBuff = 0
```

- As we have in the Motion example, create our frame counter variable:

```
frameCount = 0
lastStart = -1
```

- Now, we enter the video loop

```
while True:
```

  - begin by reading the capture frame and converting it to grayscale

```
_, frame = cap.read()
gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

  - Now that we have the grayscale frame, we can apply a the Haar cascade to it. This is done by using the **detectMultiScale** method of the **faceCascade** object:

```
faces = faceCascade.detectMultiScale(
```

    - in the first parameter, we specify the frame on which we apply the cascade:

```
gray,
```

    - in the second, we indicate the **scaleFactor**. This is used for detecting faces of different sizes; for example, the camera may have multiple people in sight, some of which are closer and some are farther from its lens, therefore causing faces to appear in different sizes. Due to the speed of applying a Haar cascade, we are able to observe an image at different scales. The **scaleFactor** value determined by how much we zoom out each time we apply the cascade from a different scale. Thus, if the value is very low, such as 1.01, our program will run very slowly and may give an increased quantity of false positives. If, on the other hand, it is large like 1.9, we may miss a large number of faces that would otherwise be detected. We set this value to **utils.scaleFactor**, which has a default of 1.3.

```
scaleFactor=utils.scaleFactor,
```

    - Lastly, we specify the **minSize** parameter, where we state the minimum size of a rectangle that we would consider to be a face.

```
minSize=(20, 20)
)
```

  - Now, **faces** contains a list of locations of faces and their sizes in the image. We can visualize this list by drawing a rectangle over these faces with:

```
for (x, y, w, h) in faces:
    cv2.rectangle(frame, (x, y), (x+w, y+h), (0, 255, 0), 1)
```

  - Next, we update our buffer by shifting the current buffer to the left, OR-ing with the a byte stating whether any faces were detected in the image, and AND-ing this entire expression with buffMask. Essentially, we are left with a sequence of bits the length of BUFF_LEN, where each bit represents whether there were faces detected in this frame. Because this sequence is updated with every frame, we can tell whether the frames over the past BUFF_LEN number of frames had faces in them or not. This allows us to

reduce flickering in the face detection and to therefore send off a more reasonable number of MQTT requests.

```
currBuff = ( (currBuff << 1) | (len(faces)>0) ) & buffMask
```

- We then check whether the updated **currBuff** is equal to one; if so, this means that a face is recognized after a long period of not being recognized. If this is the case, we send off an MQTT trigger:

```
if(currBuff == 1):
    triggerInfo = {
        "event": "FaceDetect",
        "facenum": len(faces),
        "timestamp": utils.currDate() + "--" + utils.currTime()
    }
```

  - if recording is enabled, we include frame and file location information, so that this event can be found in the video that is saved:

```
if utils.record:
    triggerInfo['offsetframe'] = frameCount
    triggerInfo['uri'] = "http://gateway" + "/" + fname
    # start counting frames since seeing face
    lastStart = frameCount
```

  - lastly, we emit the MQTT request:

```
utils.trigger(triggerInfo)
```

- Next, we check whether the current buffer has had frames containing faces in the past; if so, we check whether the number of frames elapsed since then is less than the preset **utils.recordLength** option, and if it is so we write the frame. We then increment the **frameCount**:

```
if(currBuff > 0):
    if ( (frameCount - lastStart) < utils.recordLength ) and utils.record:
        out.write(frame)
    frameCount += 1
```

- We now show the frame and wait for the ESC key to be pressed to exit.

```
cv2.imshow('Video', frame)
if cv2.waitKey(1) & 0xFF == 27:
    break
```

- Once we exit the loop, we release the video capture and recording and destroy the windows.

```
if utils.record:
    out.release()
cap.release()
cv2.destroyAllWindows()
```

Building a Person Detector

Thursday, August 4, 2016 3:58 PM

In this lab, we will create a person detector, which will alert us when a person's body is seen on the screen. In the last example, we have used Haar cascades for facial detection. While the Viola-Jones detector can be decent for detecting a faces, which have rather consistent shading patterns, as the objects get more complex and variable in color, this algorithm does not provide optimal results. Another method, the Histogram of Oriented Gradients, was created to tackle these sorts of problems. The HOG operates by dividing the frame's gradient into sections, computing the magnitude and orientation of the gradients in these sections, calculating and normalizing the histograms to create vectors representing the section, and then applying a support vector machine (SVM) for determining whether or not an object is found. The gradient-based nature of HOG makes it be resistant to changes in color and lighting, and therefore useful for person detection. Even as the background, clothing, and lighting on the person may vary, the person's body shape can be found. We will be using these tools to build our person detector.

utils

__init__.py

- As always, begin by creating the initialization file in our **utils** directory:

```
from trigger import *
from dtinfo import currDate, currTime
from args import *
```

args.py

- The arguments in our HOG person detector will look very similar to those in the previous face detection example:

```
import argparse

# Initialize argument parser
parser = argparse.ArgumentParser()

# command line options
parser.add_argument("-l", "--length", help="number of frames to record upon person detection", type=int, default=1000)
parser.add_argument("-f", "--fps", help="framerate of video recording", default=20, type=int)
parser.add_argument("-u", "--url", help="IP camera location url", type=str)
parser.add_argument("-c", "--camera", help="location of USB camera", default=0, type=int)
parser.add_argument("-ht", "--height", help="video recording height", default=480, type=int)
parser.add_argument("-w", "--width", help="video recording width", default=640, type=int)
parser.add_argument("-k", "--cascade", help="HOG cascade classifier filename", type=str, default="hog.xml")
parser.add_argument("-s", "--scale", help="adjust the scale factor", default=1.05, type=float)
parser.add_argument("-d", "--debug", help="turn on debug mode", action="store_true")
parser.add_argument("-v", "--visual", help="turn on visual mode", action="store_true")
parser.add_argument("-r", "--record", help="enable recording", action="store_true")
parser.add_argument("-m", "--message", help="enable MQTT transmission", action="store_true")


args = parser.parse_args()
```

- We then set export variables from these arguments:

```
args = parser.parse_args()

message = args.message
scale = args.scale
# set variable values from options
recordLength = args.length
display = args.visual
if args.url:
    dest = args.url
else:
    dest = args.camera
cascPath = args.cascade
scaleFactor = args.scale
frameHeight = args.height
frameWidth = args.width
frameRate = args.fps
debug = args.debug
visual = args.visual
record = args.record
```

trigger.py

- The **trigger** module remains the same as before:

```
import paho.mqtt.publish as mqtt
import json
from args import message

def trigger(info):
    infoJSON = json.dumps(info)
    if message:
        try:
            mqtt.single("sensors/temperature/data", infoJSON, hostname="localhost") # blocks up application
        except:
            print "no MQTT connection found"
            pass
    print "Event triggered:", info["event"]
    return 1
```

dtinfo.py

- so does the **dtinfo** module:

```
# current date
def currDate(): # day-month-year
    return time.strftime("%d-%m-%Y")
# current time
def currTime(): # hours-minutes-seconds
    return time.strftime("%H-%M-%S")
```

pedestrian_detect.py

- We're now ready to create our main script. Import the necessary modules:

```
import numpy as np
import cv2
import utils
```

- We define some helper functions that will be useful to us later on. The first, **inside**, checks whether a rectangle is inside of another. We will use this to make sure that only the greatest encompassing person detector is kept:

```
def inside(r, q):
    rx, ry, rw, rh = r
    qx, qy, qw, qh = q
    return rx > qx and ry > qy and rx + rw < qx + qw and ry + rh < qy + qh
```

- The next helper function, **drawDetected**, will simply draw a rectangle for every person's body detected (this just helps keep our code cleaner):

```
def drawDetected(frame, rects):
    for x, y, w, h in rects:
        cv2.rectangle(frame, (x, y), (x+w, y+h), (0, 0, 255), 2)
```

- We now instantiate a hog descriptor and create an SVM detector. We tell the SVM to be detecting people's bodies using the default people detector included with OpenCV:

```
hog = cv2.HOGDescriptor()
hog.setSVMDetector( cv2.HOGDescriptor_getDefaultPeopleDetector() )
```

- If recording is enabled, we create the VideoWriter object:

```
if utils.record:
    fourcc = cv2.VideoWriter_fourcc(*'MJPG')
    fname = utils.currDate() + "_" + utils.currTime() + ".avi"
    out = cv2.VideoWriter(fname, fourcc, utils.frameRate, (utils.frameWidth, utils.frameHeight))
```

- We create a **personCount** variable, which keeps track of the number of people detected on the screen. The idea is that when the number of people changes, the program can emit an alert:

```
personCount = 0
```

- For recording purposes, as done in the previous labs, we include the frame counters:

```
frameCount = 0
lastStart = -1
```

- To complete the setup, we tell the user whether **debug** and **visual** modes have been enabled:

```
if utils.debug:
    print "debug mode turned on\n"
if utils.visual:
    print "visual mode turned on\n"
```

- Let's start capturing video!

```
cap = cv2.VideoCapture(utils.dest)
```

- We enter the while loop. Note that we call on the **cap.isOpened** method, which tells us whether frames are capable being returned from the video source.

```
while(cap.isOpened()):
```

_, frame = cap.read()

- Now, all that we have to do to detect bodies in the frame is to call the **hog.detectMultiScale** method:

  found, _ = hog.detectMultiScale(frame, winStride=(8,8), padding=(32,32), scale=utils.scale, hitThreshold = .71)

- Next, we apply filtering to our rectangles. For every rectangle, we check whether one is inside of the others. If it is not, we add it to the **found_filtered** list.

```
found_filtered = []
for ri, r in enumerate(found):
    for qi, q in enumerate(found):
```

  - As you can see, we loop the set of rectangles over itself. If a rectangle is inside any rectangle other than itself, we skip over to the next rectangle; otherwise, we add it to our filtered rectangle list:

```
if ri != qi and inside(r, q):
    break
else:
    found_filtered.append®
```

- We now move over to checking whether the number of people in the frame has changed since the last frame. If so, we create a dict with the necessary info to be sent with the MQTT rquest.

```
if len(found_filtered) != personCount:
    personCount = len(found_filtered)
    print "number of people changed"
    triggerInfo = {
        "event": "PersonDetect",
        "facenum": len(found_filtered),
        "timestamp": utils.currDate() + "--" + utils.currTime()
        }
    if utils.record:
        triggerInfo['uri'] = "http://gateway" + "/" + fname
        triggerInfo['offsetframe'] = frameCount
```

- We then trigger our MQTT request and update our counter:

```
utils.trigger(triggerInfo)
lastStart = frameCount
```

- Now, if visual mode is enabled, we draw rectangles around the body shapes, display the image, and see whether the ESC key is pressed, in which case we exit.

```
if utils.visual:
    drawDetected(frame, found_filtered)
    cv2.imshow('video', frame)
    k = cv2.waitKey(1)
    if k == 27:
        break
```

- If recording is enabled and the number of frames elapsed since last recording has not exceeded the set length, we record the video frame:

```
if utils.record and (frameCount - lastStart) < utils.recordLength:
    out.write(frame)
    frameCount += 1
```

- If debug mode is enabled, we simply print the number of bodies found in this iteration:

```
if utils.debug:
    print('%d (%d) found' % (len(found_filtered), len(found)))
```

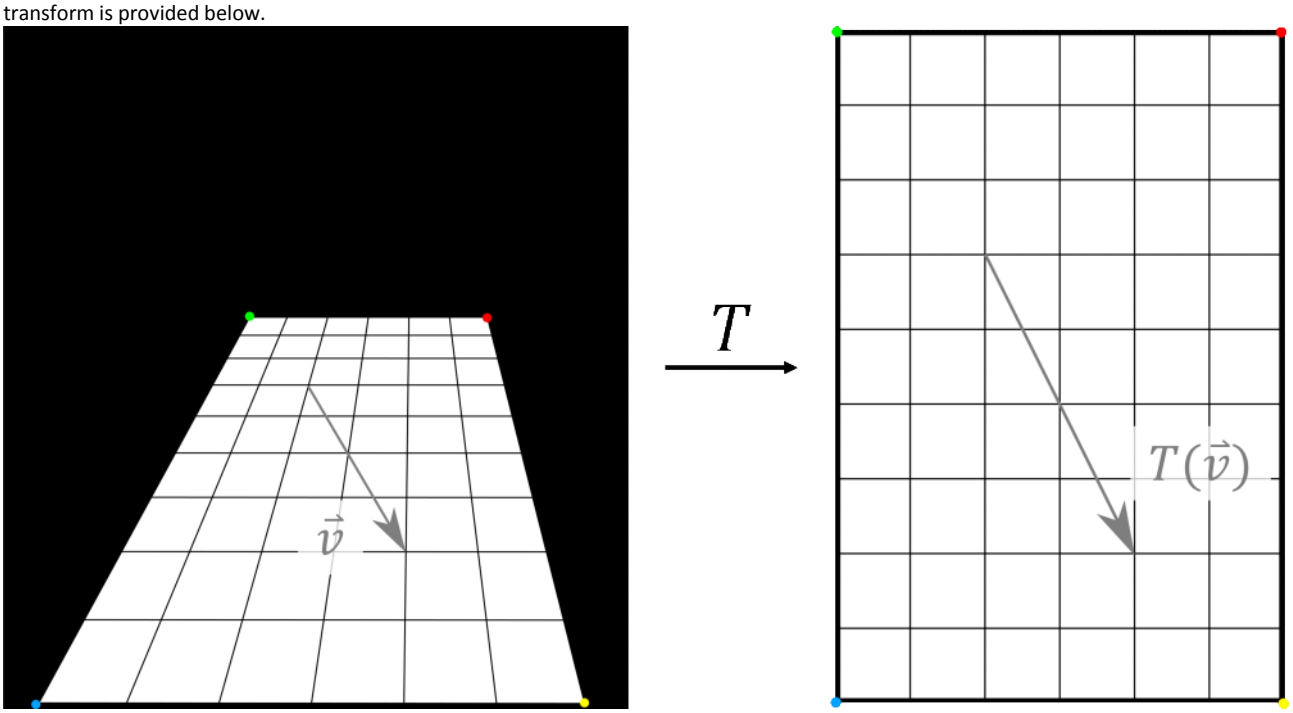
- Once the loop is broken out of, release the capture. If visual mode was enabled, close the windows. If recording was enabled, release it.

```
cap.release()
if utils.visual:
    cv2.destroyAllWindows()
if utils.record:
    out.release()
```

Building a Speed Monitor

Thursday, August 4, 2016 3:59 PM

In this lab, we will build a system that detects speeding on a highway and triggers an alert or records video when the car velocities in the frame surpass a speed limit. In order to do so, we track the motion of cars in a video, and convert the displacement of points between frames into real-world speed for cars that are passing by.
We calculate displacement vectors by using the Farneback optical flow algorithm, which uses polynomial expansion to calculate the dense optical flow in the image. The algorithm gives us an array of the same height and width as our frame, but where every pixel is replaced with a 2-dimensional displacement vector.
We convert these vectors into velocity readings by selecting a trapezoidal region on the road and calculating a perspective transformation matrix from the trapezoid's points, which allows us to convert the trapezoidal region into rectangular space.
We then select a line segment in the trapezoidal region along a road marker, which provides us with a scale for the image, as road markers are a constant length.
We then apply the perspective transformation to this segment and all of the displacement vectors in our video frames; an illustration of the perspective transform is provided below.



Once we find our displacement vectors, we observe only the ones contained in our trapezoidal ROI. We scale them into real-world dimensions by multiplying by the ratio of scaling for the marker segments. We then divide by the time elapsed between frames in order to calculate the car speeds. If the average of the car speeds exceeds a value that we have set, we can trigger an MQTT request or record the video.

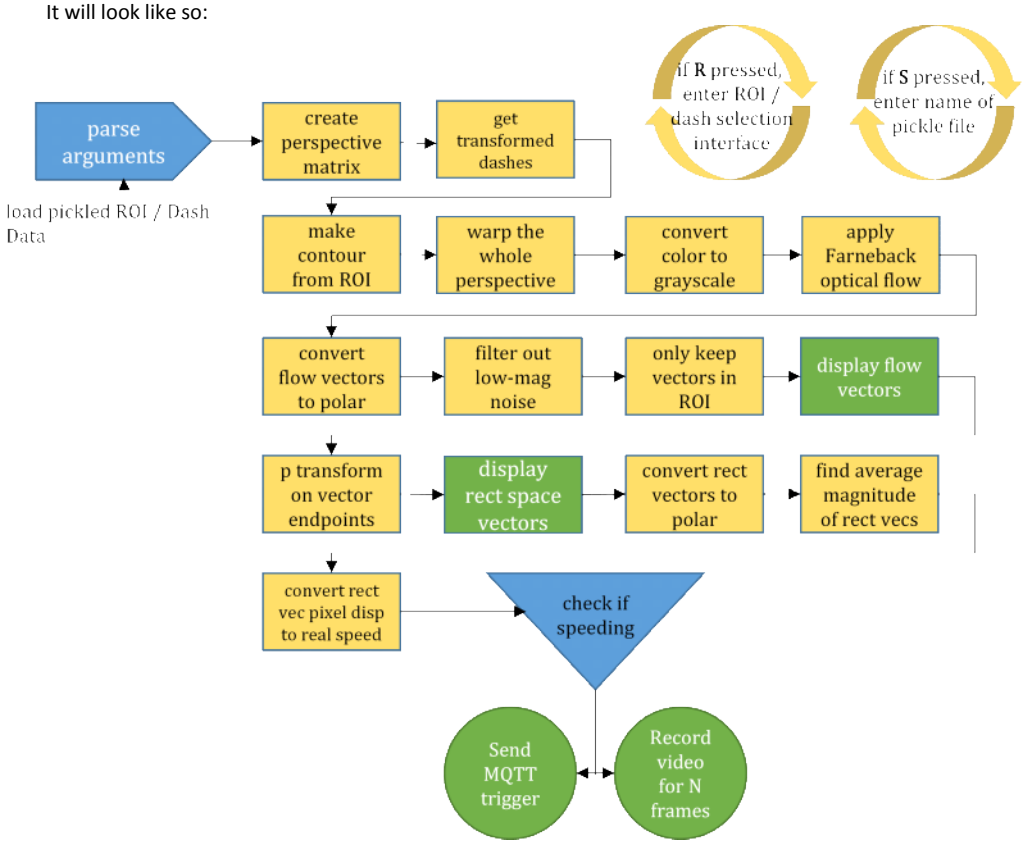
Note that in this application, it is vital to have proper points for the trapezoid and marker segment. We therefore provide the user with an interface for selecting these points. Once selected, we will provide the ability to save and load these points during other times using Python's **pickle** container.

We create our program as follows:

- create utils
  - in args, find project directory path (directory containing pickled files)
- create road selection tools
  - update elements
  - region selection callback
  - interface for selecting points for the road ROI
- create transform tools
  - functions for getting and applying perspective transform
- import necessary modules
- open pickled filed or default
- create useful functions
- import points from pickled file
- create contour from road ROI
- define transformed space dimensions
- apply transformation on ROI contour, road markers
- set real road marker length
- read frame and determine its dimensions
- convert color to gray for optical flow
- set flag and counter
- get fps, find time interval between frames
- adjust number of displacement vectors computed on

set speed limit
enter video loop:
    read video frame
    if this is the first iteration, get a new frame and start new iteration
    warp the perspective from the ROI to rectangular space
    compute optical flow
    determine magnitude and angle of optical flow vectors
    mark dashes on highway
    find magnitude above certain threshold, to filter small noise vectors
    determine vectors inside of ROI
    draw an arrow at these vectors and append them to array of starting and ending points
    convert arrays to floating point Numpy arrays
    apply perspective transform on displacement vectors
    draw arrows at transformed displacement vectors
    find displacement vector magnitudes
    calculate real-world displacement from these vectors
    calculate speed from displacement vectors
    convert speed to MPH
    if speeding,
        set up video recording
        send off MQTT trigger
        if recording enabled check that recording is still allowed, then write video frame
    draw contours for ROI and a line for dashes in rectangular space
    show images
    parse keypresses
        reset road ROI
        save current ROI
release recording and capture

It will look like so:



utils

- In our **__init__.py** file, include the typical imports:
    from trigger import *
    from dtinfo import currDate, currTime
    from args import *

args.py

- We begin by creating our argument parser, as usual:

    import argparse

```
import os.path as path
import platform
# Initialize argument parser
parser = argparse.ArgumentParser()

# command line options
parser.add_argument("-i", "--input", help="pickle file containing ROI and dash parameters", type=str, default='points.pickle')
parser.add_argument("-l", "--length", help="number of frames to record upon speeding detect", type=int, default=30)
parser.add_argument("-f", "--fps", help="framerate of video recording", default=20, type=int)
parser.add_argument("-n", "--number", help="number to divide vec quantity by", default=20, type=int)
parser.add_argument("-u", "--url", help="IP camera location url", type=str, default="carsCrop.avi")
parser.add_argument("-c", "--camera", help="location of USB camera", type=int)
parser.add_argument("-d", "--debug", help="turn on debug mode", action="store_true")
parser.add_argument("-v", "--visual", help="turn on visual mode", action="store_true")
parser.add_argument("-r", "--record", help="enable recording", action="store_true")
parser.add_argument("-x", "--displacement", help="set minimum displacement", type=int, default=2)
parser.add_argument("--inf", help="Loop video infinitely", action="store_true")
parser.add_argument("-m", "--message", help="enable MQTT transmission", action="store_true")


args = parser.parse_args()
```

- Note that there are several changes to what we have seen previously: we incorporate program-specific flags like **displacement**, which removes short-range displacement vectors occurring because of noise, and **number**, which allows us to specify the frequency at which the vectors are sampled. Because the setting of this workshop will often not be held above a highway, we will use a video in our project directly, **cars.avi**, to test our application. We can specify the name of the video in the **url** argument, and can set it to loop infinitely by using the **--inf** flag.
- Because we will be using data stored in our project directory, we find its location by using the **platform** and **os.path** modules. First, we detect the user's system. Then, we split the file location into a list of directories, and piece these together to return the project directory. The process looks as follows:

```
sep = "\\" if platform.system()=='Windows' else '/'
projDirList = path.realpath(__file__).split(sep)[0:-2]
projDir = sep.join(projDirList)
```

- Then, we define all of our program parameters:

```
message = args.message

# number to divide number of vectors by
vecDiv = args.number

# Minimum displacement threshold
x_thresh = args.displacement

# loop over video infinitely
inf = args.inf

# ROI and dash input parameters
inputParams = args.input
```

- Users can provide a pickle file by inputting it into **inputParams**. If the provided location is just the filename, then the project path is prepended to the name.

```
if inputParams.count(sep) == 0:
    inputParams = projDir + sep + inputParams
recordLength = args.length
display = args.visual

if args.camera == None:
    dest = args.url
    # if just a filename without path, add to project path
    if dest.count(sep) == 0:
        dest = projDir + sep + dest
else:
    dest = args.camera
frameRate = args.fps
debug = args.debug
visual = args.visual
record = args.record
```

dtinfo.py

- Create the file containing the date and time macros as previously:

```
import time

# current date
def currDate(): # day-month-year
    return time.strftime("%d-%m-%Y")
# current time
```

```
def currTime(): # hours-minutes-seconds
    return time.strftime("%H-%M-%S")
```

trigger.py

- Similarly, we include our **trigger** module from the previous projects. In order remove extra computation, we will check whether the **message** flag is set in our main script, before we put together our message rather than after:

```
import paho.mqtt.publish as mqtt
import json
def trigger(info):
    infoJSON = json.dumps(info)
    try
        mqtt.single("sensors/video/speed", infoJSON, hostname="localhost")
    except:
        print "No MQTT connection found"
        pass
    print "Event triggered:", info["event"], "!"
    return 1
```

Outside of **utils**, we create two more modules to help keep our code cleaner in this project. The first, **selectionTools.py**, provides the user with an interface for selecting the trapezoidal ROI. The second, **transformTools.py**, calls the necessary OpenCV functions and performs the datatype conversions for both generating and applying transformations to vectors.

selectionTools.py

- Using this library, the user can define the ROI for the road. This module provides functionality for entering the region selection mode, where the user clicks to select the four points on the ROI and then the two endpoints of the marker segment. Keystroke commands allow the user to move back and forth between points. Whenever the user presses **r** while in the video loop, he is taken to this function, where a window with a stationary image of the frame pops up and allows the user to select these points.
- begin by importing **cv2** and **numpy**:

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

- We create helper function **setElement**, which either sets a value in an array if a specified index exists, or appends the given value to the end of the array if the index exceeds the given array's size.

```
def setElement(arr, idx, val):
    high_id = len(arr) - 1
    if idx <= high_id:
        arr[idx] = val
    else:
        arr.append(val)
    return arr
```

- Next, we create the **regionSelect** callback function, which sets the trapezoid or marker points when a mouse is clicked. The point is specified by the state of the point that the program is in:

```
def regionSelect(event, x, y, flags, param):
    global mode, point, max_pt, roi_pt, dash_end
```

  - in the main region selection function, we will specify a **mode** flag; if it isn't set, we are dealing with a trapezoid, whose last point's index is 3. If it is, on the other hand, we are selecting the marker segment, which has 2 endpoints and therefore whose last index is 1.

```
max_pt = 1 if mode else 3
```

  - We can now register our mouse click events and set the important points that represent the marker and the road.

```
if event == cv2.EVENT_LBUTTONDOWN:
```

    - we check if **mode** is on or off. In our point selection routine, we will have the user first select the ROI points, starting with the point corresponding to the upper left corner of the rectangular space where cars are moving downwards, and then going clockwise. Essentially, after the transformation occurs we will warp the ROI into a rectangle, where the first point selected will be the upper left corner of the rectangle, the second will be the upper right, the third the bottom right, and the fourth point selected will be the bottom left corner of the rectangle. The next two mouse-clicks, the user specifies the marker segment. Thus, we allow the user to set the ROI corner points, and once they are set we make **mode** be True and the next callback, set the marker segment endpoints as well.

```
if not mode:
    if point <= max_pt:
        roi_pt = setElement(roi_pt, point, (x, y))
        point += 1
    else:
        mode = not mode
        point = 0
else:
    if point <= max_pt:
        dash_end = setElement(dash_end, point, (x, y))
        point += 1
    else:
        print "Press ENTER to save new points or ESC to cancel"
```

- We now come to our main region selection function, the **regionSelectionMode**. Whenever the user presses **r** in the main script, the original program comes to a halt, and this function is run. A new window named "Selection" is created, and a loop is run, displaying the current frame at the time that **r** was pressed. This loop displays the number of the point number and mode for which points are being selected by putting text into this window, and displays the points that are drawn upon mouse clicks. Thus, we write the beginning of this function as follows:

```
def regionSelectionMode(frame):
    global mode, point, max_pt, roi_pt, dash_end, selecting
    roi_pt = []
    dash_end = []
    mode = 0 # roi -> 0; dash -> 1
    point = 0
    font = cv2.FONT_HERSHEY_SIMPLEX
    cv2.namedWindow("Selection")
    cv2.setMouseCallback("Selection", regionSelect, None)
    while(True):
        showFrame = np.copy(frame)
        max_pt = 1 if mode else 3
        modeText = "Dash" if mode else "ROI"
        statString = modeText + (":{}".format(point))

        cv2.putText(showFrame, statString, (20,30), font, 1, (255, 255, 255), 1, cv2.LINE_AA)
        for i, pt in enumerate(roi_pt):
            cv2.putText(showFrame, str(i), pt, font, 1, (255, 0, 0), 1, cv2.LINE_AA)
        for i, pt in enumerate(dash_end):
            cv2.putText(showFrame, str(i), pt, font, 1, (0, 0, 255), 1, cv2.LINE_AA)
        cv2.imshow("Selection", showFrame)
```

- With every iteration of the loop, we see whether a key is pressed. This allows us to move between the points that we select by using **q** to move to the previous selection point and **e** to move to the next, and if we wish to choose another point, we are given the ability to do so. Furthermore, we look for the ENTER keypress, which will allow us to exit the selection interface once the points are selected and make these points be the ones used for the ROI and marker segments in the main program, and ESC, which makes us discard all changes.

```
        k = cv2.waitKey(1) & 0xff


        if k == 13: # ENTER
            if len(roi_pt)==4 and len(dash_end) == 2:
                print "done selecting"
            break
        elif k == 27: # ESC
            print "canceled"
            roi_pt = []
            dash_end = []
            break
        elif k == ord('e'):
            if point < max_pt:
                print "moving to next point"
                point += 1
            elif not mode:
                print "moving to dash"
                mode = 1
                point = 0
        elif k == ord('q'):
            if point > 0:
                print "moving to previous point"
                point -= 1
            elif mode:
                print "moving to roi"
                mode = 0
                point = 4
```

- lastly, when the user exits the loop, destroy the window and return the ROI and marker points:

```
    cv2.destroyWindow("Selection")
    return roi_pt, dash_end
```

- Note that in the main program, we can save the points that we have just selected in a pickle file and retrieve them later, such as on a headless gateway.

transformTools.py

- Next, we create our **transformTools** module. Here, we use the points we have just learned to select to create a perspective transform matrix, and then use it to transform our marker endpoints into points in the rectangular space. We can then calculate the length of these markers in order to determine the amount of displacement in the rectangular space.
- begin by importing **numpy** and **cv2**

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

- Next, we create the **getPerspective** function, which performs some datatype conversions and returns the perspective transform matrix. The inputs to this function are the points $\mathbf{p}_{ij}$ and $\mathbf{q}_{ij}$, where the **p**'s re points of the ROI and **q**'s are points that specify the corners of the rectangular space that we map onto. The subscripts, **i** and **j**, specify which corner of the quadrilateral is selected, in the following order:

00	01
10	11

Our function looks as follows:

```
def getPerspective(p00, p10, p11, p01, q00, q10, q11, q01):
```

- we begin by casting both sets of points into **np.float32**:

```
pts1 = np.float32([p00, p10, p11, p01])
pts2 = np.float32([q00, q10, q11, q01])
```

- Next we apply the perspective transform using **cv2.getPerspectiveTransform**,

```
M = cv2.getPerspectiveTransform(pts1, pts2)
return M
```

- The main transform function is the **transformedParams**, for which **getPerspective** merely serves as a helper function.

```
def transformedParams(p00, p10, p11, p01, m1, m2, q00, q10, q11, q01):
```

- first, we retrieve the perspective matrix by running **getPerspective**:

```
M = getPerspective(p00, p10, p11, p01, q00, q10, q11, q01)
```

- Next, we create a contour from p_{ij}. This will be useful in the main script, where we can easily determine whether vectors are inside the ROI or not, thus reducing on unnecessary computation in our program:

```
contour = np.array([p00, p01, p11, p10], dtype = np.int32).reshape((-1, 1, 2))
```

- The inputs m1 and m2 are the endpoints of our marker segments. We find their coordinates in the rectangular space by applying a perspective transform on the points:

```
markers = np.float32([[m1, m2]])
markers_rect = cv2.perspectiveTransform(markers, M)
markers_rect = markers_rect[0]
```

- Lastly, we calculate the length of the line segment between these points in rectangular space:

```
marker_len = np.linalg.norm(markers_rect[0] - markers_rect[1])
```

- We end by returning all of these values:

```
return M, markers_rect, marker_len, contour
```

car_flow.py

- Now that we have created the supporting modules for our program, we are ready to create our speed detection application.
- Begin our script by importing the necessary dependencies:

```
import cv2
import numpy as np
import pickle
import selectionTools
import transformTools
import utils
```

- Next, we unpickle a file containing the points of the ROI and markers. If no such file is specified, the program will import the default file for the test video, **points.pickle**.

```
fname_l = utils.inputParams
with open(fname_l, 'rb') as handle:
    pointContainer = pickle.load(handle)
```

- We write a macro, **ftps2mph**, for converting feet per second to miles per hour:

```
def ftps2mph(ftps):
    return ftps * 3600 / 5280.
```

- Next, we create the **"road"** and **"transformed"** windows:

```
cv2.namedWindow("road")
cv2.namedWindow("transformed")
```

- The next step is to retrieve the points from the **pointContainer**, a dict of the points that we loaded from the pickle file:

```
p00 = pointContainer["p00"]
p01 = pointContainer["p01"]
p10 = pointContainer["p10"]
p11 = pointContainer["p11"]
m1 = pointContainer["m1"]
m2 = pointContainer["m2"]
```

- We now begin our video capture

```
cap = cv2.VideoCapture(utils.dest)
```

- Next, we proceed to set some parameters for the program to use. We select an awesome color for our displacement vectors, and choose the height and width of our rectangular space:

```
cols_t = 350
rows_t = 700
```

We use these parameters to create the endpoints of our rectangular space and then enter all of this information into **transformTools.transformedParams** our transformation matrix, transformed marker information, and contour:

```
q00 = (0, 0)
q01 = (cols_t, 0)
q10 = (0, rows_t)
q11 = (cols_t, rows_t)
M, markers_rect, marker_len, contour = transformTools.transformedParams(p00, p10, p11, p01, m1, m2, q00, q10, q11, q01)
```

- Next, we specify the real-world length of the road markers. In the United States, these dashes are required to be 10 feet in length. Because our default sample video is not filmed in the US, however, we set the marker length to 5 feet:
  real_marker_len = 5.0 # ft
- We create a **begin** flag that we set to **True** to signify in the main loop that the program has just begun:
  begin = True
- We also create the **first_video_frame** flag. When speeding is first spotted in the frame, this flag will tell the program to create a new file to which it will write video.
  first_video_frame = True
- We now create our **frameCount** and **lastStart** variables, which, like in the other projects, will be used to tell us in which frame of the video the speeding is spotted.
  frameCount = 0
  lastStart = 0
- We calculate the time interval between frames by taking the inverse of the number of frames per second. We will use this when we will compute speeds from the displacement vectors.
  fps = cap.get(cv2.CAP_PROP_FPS)
  dt = 1./fps # seconds
- The variable SHOW_EVERY specifies one in how many vectors will be included in our computation. This is useful because we will calculate dense optical flow, where every pixel is assigned a flow vector. Because the cars in the frame will generally take up a large number of pixels, however, it is unnecessary to perform computation on every one of them. Calculating and displaying every SHOW_EVERY-eth vector allows us to both reduce the size of what needs to be computed and makes the set of vectors in the frame be more sparse and therefore more pleasant to look at.
  SHOW_EVERY = utils.vecDiv
- Lastly in our setup, we set the max speed, in miles per hour. Whenever the speed of the cars in the frame exceeds this speed, we trigger MQTT and record the video frames:
  max_speed = 65 # mph
- Now, we are ready to enter our main video loop.
  while(1):
  - We read a new frame
    ret, frame = cap.read()
  - Now, we check whether there is a new frame returned by the video source. If there is, we continue with the loop. If there is not, we will either loop back to the beginning of the video if our source is a video and the **inf** flag is set, or otherwise we will end the program.
    if ret:
    - Next, we check the **begin** flag to see whether this is the first frame ready by the program. Because optical flow requires two consequent frames, if this is the first frame we read, we set this frame's value to the **prev** frame and loop around to get a second one.
      if begin:
        prev = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        begin = False
        continue
    - We will now use the **cv2.warpPerspective** function to warp the area of the image in the ROI into a rectangular space representation
      transformed = cv2.warpPerspective(frame, M, (cols_t, rows_t))
    - Convert the last frame that we have retrieved into grayscale:
      nxt = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    - We're now ready to calculate the optical flow in the video. We will be using the Farneback optical flow algorithm, which uses polynomial expansion to calculate a dense representation of each frame's motion. This algorithm requires a set of fine-tuned parameters to function correctly; some that yield good results for our purposes are:
      flowParams = (0.5, 3, 15, 3, 5, 1.2, 0) # (pyr_scale, levels, winsize, iterations, poly_n, poly_sigma, flags))
    - Unzip these parameters into **cv2.calcOpticalFlowFarneback** to calculate our 2-channel array of displacement vectors:
      flow = cv2.calcOpticalFlowFarneback(prev,nxt, None, *flowParams)
    - In **flow**, the first channel represents the displacement of pixels in the y-direction, and the second in the x-direction.
    - We calculate the magnitudes and angles of these displacement vectors, which we will use to filter out noise:
      mag, ang = cv2.cartToPolar(flow[...,0], flow[...,1])
    - Create a copy of **frame**. This way, we have one image to draw the motion vectors on, and another that remains clean of annotation:
      viewFrame = np.copy(frame)
    - We can now use **viewFrame** to draw things on top of the frame. In this case, we draw a line segment along the road marker:
      cv2.line(viewFrame, m1, m2, (255, 0, 0), 2)
    - We proceed to detect **goodMags**, which are the vectors with their displacement magnitude above a certain threshold:
      goodMags = np.where(mag > utils.x_thresh)
    - Now, we initialize arrays for the start and end points of the vectors that we select to perform computations on:
      disp_arr_i = []
      disp_arr_f = []
    - We will iterate through every SHOW_EVERY-eth vector in **goodMags**:
      for i, _ in enumerate(goodMags[0]):
        if (i % SHOW_EVERY == 0):
      - Here, we look to see whether a vector's start point is inside of the ROI; we do so by using the **cv2.pointPolygonTest** function, which returns True if a point is inside of a contour, 0 if the point is on a contour, and -1 if it is outside.
        x, y = (goodMags[1][i], goodMags[0][i])
        if cv2.pointPolygonTest(contour,(x,y),False)==1:
        - If the point vector begins on the inside, we extract it vertical and horizontal components, **u** and **v**, draw an arrow on **viewFrame** indicating the vector's direction, and append the start and end points to our initialized displacement arrays.
          u, v = flow[y][x]
          cv2.arrowedLine(viewFrame, (x, y), (x+int(round(u)), y+int(round(v))), color_IntelBlue, 2)

```
                    disp_arr_i.append((x,y))
                    disp_arr_f.append((x+u, y+v))
```

- We now cast these lists to numpy floating point arrays that we can perform the perspective transform on.

```
disp_i = np.float32([disp_arr_i])
disp_f = np.float32([disp_arr_f])
```

- Great! If there are displacement vectors available, we proceed to perform the transform on the displacement vectors' beginning and end points:

```
if disp_i.any():
    disp_rect_i = cv2.perspectiveTransform(disp_i, M)
    disp_rect_f = cv2.perspectiveTransform(disp_f, M)
```

  - In order to draw them on the frame, convert these vectors to integers:

```
    disp_rect_i_int = np.int32(disp_rect_i[0])
    disp_rect_f_int = np.int32(disp_rect_f[0])
```

  - Draw an arrow beginning at the transformed starting point and ending on the corresponding endpoint in the **transformed** frame:

```
    for i, _ in enumerate(disp_rect_i_int):
        cv2.arrowedLine(transformed, tuple(disp_rect_i_int[i,...]), tuple(disp_rect_f_int[i,...]), color_IntelBlue, 2)
```

  - Now, we use extract the vertical and horizontal components of the vectors' endpoints in the rectangular space and calculate the vectors' magnitudes in this space:

```
    disp_rect_i_x = disp_rect_i[:, :, 0]
    disp_rect_i_y = disp_rect_i[:, :, 1]
    disp_rect_f_x = disp_rect_f[:, :, 0]
    disp_rect_f_y = disp_rect_f[:, :, 1]
    dispMag, velAng = cv2.cartToPolar(disp_rect_f_x - disp_rect_i_x,\
        disp_rect_f_y - disp_rect_i_y)
```

- If, on the other hand, no displacement vectors are available, we set **dispMag** to zero:

```
else:
    dispMag = 0
```

- It can be useful to visualize the speeds of the cars relative to each other. We do so by normalizing the displacement vectors by the maximal one, converting to an 8-bit depth, and applying a colormap:

```
normer = float(np.max(mag))
if normer: # checks that normer is not 0
    mag_norm = np.uint8(mag * 255 / normer)
    mag_norm = cv2.applyColorMap(mag_norm, cv2.COLORMAP_JET)
```

- In the effort to avoid detecting individual cars in the lab, we find the average distance that the cars are displaced between frames and calculate the car speeds off of this average.

```
avgMag = np.average(dispMag)
```

- We calculate the real displacement magnitude by multiplying **avgMag** by a ratio of the real marker length (which was given to us as a constant) to the marker length in the frame that we see. Essentially, what we are doing here is asking how many marker lengths in the video fit into the displacement vector's magnitude, and then multiplying by whatever this distance corresponds to in real life.

```
realDisp = avgMag * real_marker_len / marker_len
```

- We can now calculate the real speed by dividing by the time interval between frames and converting feet/second to miles per hour.

```
realSpeed = realDisp / dt
realSpeed_mph = ftps2mph(realSpeed)
```

- Now that the real speed is provided to us, we can do what we have been doing all along: providing options to the user for interfacing with the virtual sensor.

```
if realSpeed:
```

  - if the **debug** flag is on, we simply print the speed with every loop

```
    if utils.debug:
        print realSpeed_mph
```

  - Next, we check whether speeding is happening in the frame.

```
    if realSpeed_mph > max_speed:
```

    - if it is, we print at what speed this is occurring:

```
        print "speeding at:", realSpeed_mph, "mph"
```

    - We then update our **lastStart** variable and check whether this is the first frame of the video. If so, we set up a video recorder.

```
        lastStart = frameCount
        if utils.record and first_video_frame:
            fourcc = cv2.VideoWriter_fourcc(*'MJPG')
            fname_vo = "{}_{}_speeding.avi".format(utils.currDate(), utils.currTime())
            vidParams = (fname_vo, fourcc, utils.frameRate, (frame.shape[1], frame.shape[0]))
            out = cv2.VideoWriter(*vidParams)
            first_video_frame = False
```

    - If transmitting MQTT messages is enabled, transmit a message, just as we have done before:

```
        if utils.message:
            triggerInfo = {
            "event": "VehicleSpeed",
            "speed": realSpeed_mph,
            "timestamp": utils.currDate() + "--" + utils.currTime(),
            "speeding": True
            }
```

```
                    if utils.record:
                        triggerInfo['uri'] = "http://gateway" + "/" + fname_vo
                        triggerInfo['offsetframe'] = frameCount
                    utils.trigger(triggerInfo)
```

- Similarly to previous projects, if recording is enabled and the maximum number of frames for the past speeding event wasn't surpassed, write the frame and increment the frameCount.

```
                    if utils.record and (frameCount - lastStart) < utils.recordLength and not first_video_frame:
                        out.write(frame)
                        print "wrote video frame"
                        frameCount += 1
```

- We now draw the contours on our **viewFrame** and a line segment on the dashes in our rectangular space, in order to show that our mechanism is functioning:

```
cv2.drawContours(viewFrame, [contour], 0, (0, 255, 255), 1)
cv2.line(transformed, tuple(markers_rect[0]), tuple(markers_rect[1]), (255, 0, 0), 2)
```

- We then show our annotated and transformed frames:

```
cv2.imshow("road", viewFrame)
cv2.imshow("transformed", transformed)
```

- If **normer** has a value, we display it as well:

```
if normer and utils.visual:
    cv2.imshow("normalized magnitudes", mag_norm)
```

- Lastly, we supply several commands activated on keypress:

```
k = cv2.waitKey(1)
```

- If the ESC key is pressed, we exit the program:

```
if k == 27:
    break
```

- If **r** is pressed, we open the region selection interface and allow for the user to select new points for the ROI and markers:

```
elif k == ord('r'):
    roi_pts, dash_pts = selectionTools.regionSelectionMode(frame)
    if len(roi_pts) == 4 and len(dash_pts) == 2:
        p00, p01, p11, p10 = roi_pts
        m1, m2 = dash_pts
        M, markers_rect, marker_len, contour = transformTools.transformedParams(p00, p10, p11, p01, m1, m2, q00, q10, q11, q01)
```

- If **s** is pressed, we create a dictionary containing all of these points and ask the user where it is intended to save the pickle file containing these points:

```
elif k == ord('s'):
    pointContainer = {
        "p00" : p00,
        "p01" : p01,
        "p10" : p10,
        "p11" : p11,
        "m1" : m1,
        "m2" : m2
    }
    fname_s = raw_input("Save to file:\t")
    with open(fname_s, 'wb') as handle:
        pickle.dump(pointContainer, handle)
    print "Saved to file:\t{}".format(fname_s)
```

- Lastly, at the end of the iteration of the main video loop, we set the new value of **prev** to what was just **nxt**

```
prev = nxt
```

- Recall that at the beginning of the loop we check whether a frame is returned or not. If not, we jump to this part of the loop. Now, if the **utils.inf** flag is set we are able to restart the video; if, however, the flag is **False**, then we break out of the program.

```
else:
    if utils.inf:
        cap = cv2.VideoCapture(utils.dest)
    else:
        break
print "done"
```

- We then release the capture and the video recording, if video was indeed being recorded. We then close all windows from the program.

```
cap.release()
if utils.record and not first_video_frame:
    out.release()
cv2.destroyAllWindows()
```

- Congratulations, you have created your own speed detection system!