Getting started with OpenCV 2.4

Blake Anderson

June, 2015

Abstract

OpenCV is an open source computer vision package written for C++ and ported to python. It allows user to manipulate and create images and videos, and includes advanced functionality such as machine learning algorithms for detection of objects. The following guide walks through the basics necessary to read and write images and video, draw shapes, interact with windows using mouse or track bar input, and generate a simplistic motion tracking script using background subtraction

1 Getting Started

The first thing to do when starting is to import necessary packages, generally I include the OpenCV library, as well as the OS package so you can easily change the working directory, and manipulate files that are in subdirectories

```
import cv2
import os
#Linux
os.chdir("/home/blake/Documents")
#Windows XP -> 8.1
os.chdir("C:\Users\Blake\Documents")
#Windows 10 (preview builds)
os.chdir("C:/Users/Blake/Documents")
```

OpenCV is represents images as arrays of numbers, typically ranging from 0-255, which represent intensity in the respective spectrum. For example in RGB (Red,Green,Blue) color space, (255,0,0) would represent a solid red pixel, (255,255,255) would represent a white pixel, (0,0,0) would appear black, and (255,0,255) would be pink (red+blue). To easily manipulate these we will use the NumPy (Numerical Python) package, and access it through an alternate name space to save time down the line

```
import numpy as np
```

2 Reading images and videos

To read a picture stored in your working directory, you first want to store it to a variable using the imread() function, and then display it using the imshow() function. The image will only be visible instantaneously, to actually view it you must tell python to stop processing information using the waitKey() function, here we will display the image for 5 seconds (5000 ms), or until keyboard input is detected.

```
image = cv2.imread('Partygrade.jpg',FLAG)
cv2.imshow('Preview',image)
cv2.waitKey(5000)
```

In the imread() function you have to option to designate an additional flag, by default it passes a value of 1, pertaining to a BGR color image (RGB in reverse order), though you can pass 0 to read grayscale, or -1 to read it in the format it is stored (this allows you to include an alpha (transparency) channel).

If you want to resize the window you must designate that property of the window, `'Preview'`, manually - as it will create a window of the same size as the image by default. Here we will pass a 0 (or a negative integer) to the waitKey() function, telling it to wait indefinitely for keyboard input.

```
cv2.namedWindow('Preview',cv2.WINDOW_NORMAL)
cv2.imshow('Preview',image)
cv2.waitKey(0)
```

You will notice if you try to resize the image after keyboard input is passed through to the window that the image fails to redraw itself. If you want to close the window without terminating the interpreter you can use one of the destroy functions

```
cv2.destroyWindow('Preview')
cv2.destroyAllWindows()
```

These have different degrees of functionality between operating systems. They work as intended on windows, however on linux systems they only work when running the entire program, sometimes only when followed by a waitKey() command.

To read a video a slightly different syntax is used, rather than the imread() function we will initialize the video using the VideoCapture() function, and then sequentially read frames. The easiest way to work with a single video is to store the VideoCapture() object in a variable ('cap')

```
cap = cv2.VideoCapture('ExampleVideo.avi')

while (1):
        image = cap.read()[1]
        cv2.imshow('Preview',image)
        k = cv2.waitKey(100)
```

The read() function is a combination of two other functions, grab(), which pulls the next frame from the video, and retrieve(), which decodes a grabbed frame so it can be displayed/manipulated. In theory these can be used separately. It's worth noting that the read() command advances the video by a single frame each time it is used.

You will also notice that the read() function is followed by a [1], this is because it outputs two variables, the first is a boolean operator which reports whether the function successfully grabbed and decoded a frame, the second is the contents of the

frame. The [1] tells the function to only save the 2nd variable (remember python counts from 0, so [0] would save the TRUE/FALSE output to the 'image' variable).

If you let the video play the entire way through you likely noticed an error, this occurs because when the video reaches its last frame, this script will attempt to read a subsequent frame (which does not exist). This can be remedied by using the isOpened() function, which returns a boolean.

cv2 functions will print their output to the console, to suppress this you can simply save them to variables. Here the waitKey() is stored in the variable 'k'. This is because a -1 is returned each time no keyboard input is detected during the designated wait time. To view what keys you are pressing while watching a video you can include a simple if() statement

```
while(cap.isOpened()):
        image = cap.read()[1]
        cv2.imshow('Preview',image)
        k = cv2.waitKey(100)
        if not k==-1:
                print k
```

Note that keyboard input is only detected in the time between processing frames, and this delay does not count the time your computer spends processing frames. If you set the waitKey to (1) the ratio of time spent computing vs waiting for input will be very skewed, and you will notice that a 100 frame video takes much longer than 100 ms to play back.

When you finish interacting with a video capture object it is always good practice to release it from use so it can be accessed by other processes. You can also use the reported values by the previous loop to interact with the video in a number of ways:

```
while(cap.isOpened()):
        image = cap.read()[1]
        cv2.imshow('Preview',image)
        k = cv2.waitKey(100)
        if k==10: #ENTER key (13 on windows)
                print 'Hello world!'
        if k==ord('a'):
                print 'Letters or key codes work'
cap.release()
```

3 Video Properties

To interact with videos in a more advanced way, you often require more information about the videos, or need to manipulate them. The get(PROPERTY) and set(PROPERTY, VALUE) functions allow such functionality, where the property identifier can be any of the following, passed as either a number, or a cv2.cv.CV_CAP_PROP_ suffix (from the documentation; 11-16 only apply to cameras; 18-22 only supported by DC1394 v 2.x backend):

0. POS_MSEC — Timestamp/Current position (in milliseconds)
1. POS_FRAMES — 0-based index of next frame to be decoded/captured.
2. POS_AVI_RATIO — Relative position of the video file: 0 - start of the film, 1 - end of the film.
3. FRAME_WIDTH — Width of the frames in the video stream.
4. FRAME_HEIGHT — Height of the frames in the video stream.
5. FPS — Frame rate.
6. FOURCC — 4-character code of codec.
7. FRAME_COUNT — Number of frames in the video file.
8. FORMAT — Format of the Mat objects returned by retrieve() .
9. MODE — Backend-specific value indicating the current capture mode.
10. BRIGHTNESS — Brightness of the image.
11. CONTRAST — Contrast of the image.
12. SATURATION — Saturation of the image.
13. HUE — Hue of the image.
14. GAIN — Gain of the image.
15. EXPOSURE — Exposure.
16. CONVERT_RGB — Boolean flags indicating whether images should be converted to RGB.
17. WHITE_BALANCE_U — The U value of the whitebalance setting.
18. WHITE_BALANCE_V — The V value of the whitebalance setting.
19. RECTIFICATION — Rectification flag for stereo cameras.
20. ISO_SPEED — The ISO speed of the camera.
21. BUFFERSIZE — Amount of frames stored in internal buffer memory.

```
cap = cv2.VideoCapture('ExampleVideo.avi')
#cap.get(PROPERTY)
#cap.set(PROPERTY,VALUE)

# Pass number or text
framerate = cap.get(6)
framecount = cap.get(cv2.cv.CV_CAP_PROP_FRAME_COUNT)
cap.get(0) # Original position (in ms)
cap.set(cv2.cv.CV_CAP_PROP_POS_AVI_RATIO,0.50)
cap.get(0) # Updated position - 50% through the video
```

In addition to these properties, you can decide whether to represent variables in the grayscale, RGB, or HSV color space. While RGB is the default color space, it is important to note the bits are reversed in cv2, that is, the channels are Blue, Green, and Red, respectively (hence why many cv2 commands use the notation BGR). Since all images are stored in numpy array objects, you can isolate a single channel of the tensor (matrix with more than 2 dimensions) easily

```
image=cv2.imread("Partygrade.jpg")
blueChannel=image[:,:,0]
greenChannel=image[:,:,1]
redChannel=image[:,:,2]
cv2.imshow('Blue',blueChannel)
```

```
cv2.imshow('Green',greenChannel)
cv2.imshow('Red',redChannel)
cv2.imshow('Original',image)
```

To convert between color spaces you can use the cvtColor() function. You can either output the conversion to a new variable or overwrite the original variable

```
#cv2.cvtColor(SOURCE,TRANSFORMATION,DESTINATION)
imageBGR=image # Original image in BGR space
# Convert variable to grayscale
cv2.cvtColor(image,cv2.cv.CV_BGR2GRAY,image)
# Duplicate original image to HSV space
cv2.cvtColor(imageBGR,cv2.cv.CV_BGR2HSV,imageHSV)
# Alternative syntax, copies HSV image to RGB space
imageRGB=cv2.cvtColor(imageHSV,cv2.cv.CV_HSV2RGB)
```

if you use the imshow() function with the RGB and BGR images, they will appear identical, however you will notice if you look at the first slice of the variables they are being represented differently in numerical form

```
sum(imageRGB[:,:,0]-imageBGR[:,:,0])
```

cv2 will automatically create new variables of the proper format for the destination files, however you can also do this yourself:

```
imageRGB=np.zeros(np.shape(image),dtype=np.uint8)
imageGray=np.zeros(np.shape(image)[:2],dtype=np.uint8)
```

The destination variables will appear black until you write data to them, alternatively you could add 255 to them to make them appear white. When creating the grayscale variable the [:2] denotes that only the first 2 dimensions of the tensor are used to create the empty variable (horizontal and vertical resolution).

Perhaps most noteworthy is the last argument, which specifies the data format, uint8, which stands for 'unsigned 8 bit integer'. What this means is the numbers stored in this variable must be positive (unsigned, so no +/- sign allowed), whole numbers (as integers are; to store decimals you must use floating point), and 8 bit (ranges from 0-255).

> As a brief refresher in binary:
>
> Each column (bit) from right to left represents a power of 2, ranging from 2^0, to 2^7 (128,64,32,16,8,4,2,1)
>
> 0 0 0 0 0 0 0 0 = 0 (0+0+0+0+0+0+0+0)
>
> 0 0 0 0 0 0 0 1 = 1 (0+0+0+0+0+0+0+1)
>
> 0 0 0 0 0 0 1 0 = 2 (0+0+0+0+0+0+2+0)
>
> 0 0 0 0 0 0 1 1 = 3 (0+0+0+0+0+0+2+1)
>
> 0 0 0 1 0 1 0 1 = 21 (0+0+0+16+0+4+0+1)
>
> 1 1 1 1 1 1 1 1 = 255 (128+64+32+16+8+4+2+1)
>
> A signed integer would require 1 bit of data to store the sign (+ or -), and thus would only have 7 bits to store numbers. As a result, signed 8 bit integers range from -128 to +127

Numerical formats are important when doing image additions, and it is worth noting that mathematical operations using uint8 arrays differ between cv2 and NumPy. In NumPy, if a mathematical operation would result in a number > 255, it is rounded off to 255 and displayed as white. In cv2, the remainder of mathematical operations resulting in numbers > 255 are carried over. If we imagine image1 and image2, two uniformly colored grayscale images. image1 has an intensity of 200, and image2 of 155.

```
cv2.add(image1,image2) #cv2 operation
image1+image2 #NumPy operation
```

We would end up with two very different outputs. The resulting cv2 image would be a dark gray (200+155=355, since $355 > 255$, the remainder of 355-255 would be the returned value from this operation). The resulting image from the NumPy operation would be white (since $355 > 255$, the remainder would be truncated and 255 would be returned).

When performing intermediate operations on arrays it may be necessary to convert the variables to different number system (uint32, uint64, int32, int64; or int, which defaults to either 32 or 64 bit) before performing operations, though the array will need to be converted back to the proper system before it will be handled properly by cv2.

4 Video Recording

This next section will address writing a video using a webcam device and defining regions of interest to record. We will start by telling a video capture object to read from the webcam, to do this we will replace the file name of a video with a device identifier (0,1,2 etc). The first webcam attached to your computer will be device number 0, the second will be 1, etc. We will then read and display frames as long as the video device remains connected, or until the user presses escape.

```
cap = cv2.VideoCapture(0)
while cap.isOpened():
        frame=cap.read()[1]
        cv2.imshow('Preview',frame)
        K=cv2.waitKey(25)
        if K == 27:
                break
```

To save webcam feeds to video, we must first choose a video codec to use (using the fourCC code by which it is referred), and then create a video writer object in cv2.

```
FCC = cv2.cv.CV_FOURCC(*'XVID')
rec = cv2.VideoWriter('output.avi',FCC,fps,(hRes,vRes))
```

Where FPS is the target frame rate, and hRes/vRes is the horizontal and vertical dimensions of the video you want to output. These need to match the frames you will be passing through to the video writer using the `.write()` function. If you put these together into a loop however, the frames will be captured as fast as the processor can store them in the video, and regardless of the number of frames captured, they will play back at the set FPS. To get around this, you either a) pass a delay in the `cv2.waitKey()` command that is equal to the difference between the desired frame rate and the average time needed to process a frame, or b) Implement a delay between each frame manually. We will use the 'time' package to do the latter when we put this altogether:

```
import time
fps=10
FCC = cv2.cv.CV_FOURCC(*'XVID')
rec = cv2.VideoWriter('vid.avi',FCC,fps,(1920,1080))
delay=1.0/FPS #include decimal to specify float
while cap.isOpened():
        sTime=time.time() #Start time
        frame=cap.read()[1]
        rec.write(frame) #Write frame to video
        cv2.imshow('Preview',frame)
        wTime=(delay-(time.time()-sTime))*1000
        K=cv2.waitKey(max(wTime,1))
        if K==27: #ESC key to exit
                break
cap.release() #Release video capture object
rec.release() #Release video writer object
```

In the above script we denote `sTime` as the time at which we begin processing each frame, pull a frame, and then `.write()` the frame to our video capture object. We then calculate how long to wait before pulling the next frame, at 10 fps, this is 0.1 seconds, minus by the time it took to read and write the frame, multiplied by 1000 (seconds -> milliseconds). If for some reason the computer cannot keep up with the specified frame rate, a negative values will be returned, causing `cv2.waitKey()` to wait indefinitely. To circumvent this we include the `max(X,1)` as an alternative.

Note: that because this method rounds to the nearest millisecond, it will lose time over long recordings (the degree is proportional to the fps). If you are recording at high frame rates or for long durations, consider specifying your initial start time outside the loop, and using this as a frame of reference for when to record each frame (not covered here).

5 Trackbars and Shapes

Track bars are useful ways of dynamically specifying variables, and previewing the effect they will have on your video. In the following section we will build on our previous code and implement a track bar to specify a region of interest with which to view.

We will start by creating a window containing 4 track bars to specify 4 edges:

```
cv2.namedWindow('Preview',cv2.WINDOW_NORMAL)
cv2.createTrackbar('XL', 'Preview',100,1920,nothing)
cv2.createTrackbar('XR', 'Preview',1820,1920,nothing)
cv2.createTrackbar('YT', 'Preview',100,1080,nothing)
cv2.createTrackbar('YB', 'Preview',980,1080,nothing)
#cv2.createTrackbar(bar name, containing window, ...
#    starting value, max value, function run on update)
```

By default every time a slider is dragged a function is called, we have no use for this feature here though, so we will define an empty function called 'nothing'.

```
def nothing(x):
        pass
```

We will then read from webcam indefinitely, and use the position of the four track bars to draw a square and circle on the displayed frames.

```
while cap.isOpened():
  frame=cap.read()[1]
  XL = cv2.getTrackbarPos('XL','Preview')
  XR = cv2.getTrackbarPos('XR','Preview')
  YT = cv2.getTrackbarPos('YT','Preview')
  YB = cv2.getTrackbarPos('YB','Preview')
  #cv2.ellipse(image, Xcenter, Ycenter, Xradius, Yradius, ...
  #rotation, start angle, end angle, color (BGR), line weight)
  cv2.ellipse(frame,(int((XR-XL)/2+XL),int((YB-YT)/2+YT)),
        (int((XR-XL)/2),int((YB-YT)/2)),0,0,360,(255,0,0),-1)
  #Negative thickness specifies to fill the object
  #cv2.rectangle(image, corner 1, corner 2, color,thickness)
  cv2.rectangle(frame,(XR,YB),(XL,YT),(0,255,0),2)
  cv2.imshow('Preview',frame)
  K=cv2.waitKey(25)
  if K==27:
    break
```

You can draw text too:

```
cv2.putText(frame,'Example Text',(100,100), ...
        FONT_HERSHEY_SIMPLEX,1,255)
#(frame, text, bottom left origin, font face, scale, color)
```

(See documentation for full font faces and options)

Upon exiting using the escape key, we can reinitialize the webcam feed, this time using only the region contained within the green rectangle (since the boundaries were saved to the variables XL, XR, YT, and YB). Append the following code to your script, immediately following the previous **while()** loop.

```
while cap.isOpened():
        frame=cap.read()[1]
        croppedFrame=frame[YT:YB,XL:XR]
        cv2.imshow('Cropped',croppedFrame)
        cv2.waitKey(25)
        if K==27: #ESC key to exit
                break
```

6 Mouse integration

By defining functions you can use the mouse to interact with images, we will start by initializing a webcam feed and telling python to wait for mouse input

```
cv2.namedWindow('Preview',cv2.WINDOW_NORMAL)
while cap.isOpened():
        frame = cap.read()[1]
        #SetMouseCallback(Window, Function, (ignore) )
        cv2.cv.SetMouseCallback('Preview', on_mouse, 0)
        cv2.imshow('Preview',frame)
        cv2.waitKey(25)
```

Now we can define a function at the start of our script that prints the X and Y coordinates of the mouse (as it pertains to the interacting window) to the python console on left click

```
def on_mouse(event, x, y, flags, params):
        if event == cv2.cv.CV_EVENT_LBUTTONUP:
                print ('Mouse Position: ', x, y)
```

We can modify this further by adding additional conditional event statements:

```
if event == cv2.cv.CV_EVENT_RBUTTONDOWN:
        cv2.circle(frame,(x,y),5,(255,0,255),-1)
```

Note that if you want drawn objects to persist you need to save them to a variable outside of the function, and draw them on each loop. Here we will continuously print the mouse coordinates to console, and the last circle drawn will persist:

```
XYmat=[0,0]
def on_mouse(event, x, y, flags, params):
        global XYmat
        if event == cv2.cv.CV_EVENT_MOUSEMOVE:
                print ('Mouse Position: ', x, y)
        if event == cv2.cv.CV_EVENT_LBUTTONDOWN:
                XYmat=[x,y]
cap=cv2.VideoCapture(0)
while cap.isOpened():
        frame = cap.read()[1]
        cv2.circle(frame,(XYmat[0],XYmat[1]),5,(255,0,255),-1)
        cv2.cv.SetMouseCallback('Preview', on_mouse, 0)
        cv2.imshow('Preview',frame)
        K=cv2.waitKey(25)
        if K==27:
                break
```

7 Background Subtraction and Contours

In this next section we will write a basic program to detect and track moving objects on a background. While there are a number of more advance methods, we will stick to using a background subtraction which is sufficient to detect uniformly colored objects provided the background image remains unchanged throughout the duration of the video. As a note: cv2 includes functionality for a number of more advanced approaches, many of which incorporate aspects of machine learning to optimize accuracy, these are not touched on here but can be found in the online documentation.

We will start by initializing a video and generating the average background image. For 30fps videos this can be computationally demanding, and, provided the variation in still images is consistent across the video, sampling a small number of frames (100-200) will yield comparable accuracy at a fraction of the computing time.

```
import numpy as np
import cv2

cap = cv2.VideoCapture("ExampleVideo.avi")
frameCount = cap.get(7) #Get total frames in the video
samplingFrames = 100 #Sample 200 distributed frames
hRes=cap.get(3)
vRes=cap.get(4)

backgroundImage=np.zeros((vRes,hRes,3,samplingFrames))
for frames in range(samplingFrames):
        cap.set(1,frames*(frameCount*samplingFrames))
        backgroundImage[:,:,:,frames]=cap.retrieve()[1]

backgroundImage=np.average(backgroundImage,axis=3).astype(np.uint8)
cv2.imshow("Background",backgroundImage)
cv2.waitKey(0)
```

In the above code, 100 frames are read and put in a 4 dimensional matrix, the matrix is then averaged along the 4th dimension, leaving only the X/Y pixel values for the 3 color channels, before converting to an integer (which also rounds down all the decimals; you can use the np.round function first if you want to round all values >0.5 to 1 instead of 0) You will notice that there is still substantial ghosting, which is an issue inherent to averaging, an alternative solution is to take the median, or even the 25th or 75th percentile (or more extreme if sufficient frames are sampled), depending on whether you are tracking objects lighter, or darker than the background, respectively.

```
backgroundImage=np.percentile(backgroundImage,75,axis=3).astype(np.uint8)
```

We will then play through the video, only this time we will perform a background subtraction, so only aspects of the current frame that are darker will be shown (reversing the order of operations will show lighter objects)

```
cap.set(1,0)
while cap.isOpened():
        frame = cap.read()[1]
        subtraction = cv2.subtract(backgroundImage,frame)
        cv2.imshow('Subtraction',subtraction)
        K=cv2.waitKey(1)
        if K==27:
                break
```

The subtraction appears gray here, but still retains color differences, and these can be isolated if need be. To detect the objects we will use the cv2.findContours() function, which highlights the boundaries where color or intensity change. Note that this only works on single channel images, and works better on binary channels, so we will convert our image to grayscale and apply a binary threshold, though you could also use other included cv2 functions such as the Canny edge detection algorithm (not touched on here)

```
cap.set(1,0)
frame = cap.read()[1]
subtraction = cv2.subtract(backgroundImage,frame)
subtraction = cv2.cvtColor(subtraction,cv2.cv.CV_BGR2GRAY)
thresholdImage = cv2.threshold(subtraction,50,255, ...
        cv2.THRESH_BINARY)[1]
cv2.imshow('Threshold',thresholdImage)
contours, hierarchy = cv2.findContours(thresholdImage, ...
        cv2.RETR_TREE,cv2.CHAIN_APPROX_NONE)
cv2.drawContours(frame,contours,-1,100,1)
cv2.imshow('Subtraction',subtraction)
cv2.imshow('Contours',thresholdImage)
cv2.imshow('Frame',frame)
K=cv2.waitKey(0)
```

Note that `cv2.findContours()` overwrites whatever image is passed to it, so if you want to keep the original image it must be saved to an untouched variable.

Within the `cv2.threshold()` function, the second and third arguments refer to the pixel value at which the threshold is applied, 255 refers to the max value which is given to objects meeting the threshold value.There are several options for the latter term, which include THRESH_BINARY_INV, THRESH_TRUNC, which leaves values below the threshold untouched, and THRESH_TOZERO/THRESH_TOZERO_INV.

The `cv2.findContours()` outputs two items, the first, contours, is a numpy array with XY coordinates of all pixels meeting its criterion for 'contour', the second, hierarchy, tells you which contours are nested within another contour. The second argument changes how hierarchical information is stored. RETR_LIST stores all contours on the same hierarchical level. RETR_EXTERNAL only returns outer contours, so all contours nested within another are ignored. RETR_CCOMP alternates between the two levels, so the outermost object is stored in level 1, and every contour within that object is level 2, if any of those objects have contours within them they are placed within hierarchy level 1 again, and so on and so forth. RETR_TREE creates the full list, storing all hierarchical information as deep as it may go. The last term in the `cv2.findContours()` denotes how much contour information to store. CHAIN_APPROX_NONE stores all contour pixels in the output matrix, while CHAIN_APPROX_SIMPLE only stores the minimum pixels necessary to draw lines around the entire object. This approximation can greatly reduce memory if you are 1) saving all contours throughout the course of a video and 2) dealing with regular shapes. Here we are doing neither so we will settle on using the non-approximated method.

Within `cv2.drawContours()`, the first argument defines the frame upon which the contours are drawn, the second refers to the variable within which the contours are stored. The third argument states which contour to draw (-1 draws all contours), while contours 4 and 5 denote line color (single number or [B,G,R]) and thickness, respectively.

There are a number of useful ways to gain meaningful information from contours. Here we will, from a single frames contours, find all contours drawn around objects with an area greater than 30 pixels, highlight them in red, and then draw a circle within them using the contour moments (center of mass). You can `print [mx, my]` to console during each pass, or save them to a matrix/.csv if you want to subsequently view the coordinates of all the objects. You will note that not all objects meet the criteria in the example video provided, this is because objects that remain stationary for the duration of the video will not be picked up by the frame averaging method. You will need to either apply a blur to the background and hope they are removed, or, using the `cv2.SetMouseCallback()` function, code a function to manually remove blemishes. One approach would be to sample the color of the pixels surrounding the mouse cursor on right click (`sample = cv2.mean(frame[y-2:y+2,x-2:x+2])`; note that frame will need to be set as a global variable), and drawing a circle on the image upon a left click.

```
for cont in range(len(contours)):
        if len(contours[cont])>4 and cv2.contourArea(contours[cont])>30:
                cv2.drawContours(frame,contours,cont,[0,0,255],1)
                Moment = cv2.moments(contours[cont])
                mx = int(Moment['m10']/Moment['m00'])
```

```
                    my = int(Moment['m01']/Moment['m00'])
                    cv2.circle(frame,(mx,my),2,[0,255,0],-1)
cv2.imshow('Frame',frame)
K=cv2.waitKey(0)
```

Before finding the contour area, we must first verify that the contour has at least 4 points in it, which is the minimum number of points required to calculate the area of the contour. You will icewi

Using contours you can perform a number of addition operations, including: calculating perimeter, aspect ratio, or extreme points, approximating the best fitting inner or outer shape, calculating convexity, bounding/best fitting shapes/lines, calculating the degree of similarity to another object, and generating masks to apply further thresholds, or to calculate the color of enclosing objects.