Basic Hazmat Detection Guide

By SART 2019

Introduction

This guide will cover a basic method of hazmat detection which was used by SART in the RMRC at RoboCup 2019. The instructions in this guide aim to reconstruct, from scratch, the hazmat detector used by SART in the 2019 competition, where we successfully won the point for hazmat detection.

This guide assumes you understand python fundamentals such as lists, tuples, how to define functions and classes, as well as the concept of a python module.

If you have any questions, feel free to email me at anthony.gambale@sfxrescue.com.

Contents

Part 0 – Preparation

Part 1 – Helper Classes

Part 2 – HOGUtils Helper Functions

Part 3 – Classification

Part 4 – Bringing it all Together

Part 0 – Preparation

To prepare for the guide, set up your project files and folders like this:

- hazmat.py
- **modules**
    - HOGUtils.py
    - colorlabeler.py
    - shapedetector.py
    - **classify**
        - classify_abstracted.py
        - **templates**
            - 1.jpg
            - 1.png
            - 2.jpg
            - 2.png
            - …
            - 26.jpg
            - 26.png

This file tree is constructed exactly as the SIGHTSVision repository was during RoboCup 2019. The images in the **templates** folder will be found somewhere in the SIGHTSVision repository at https://github.com/SFXRescue/SIGHTSVision.

You will need to have python 3 installed, as well as the packages *imutils, argparse* and *numpy* which can be installed via pip.

One more package, opencv, can be installed from https://opencv.org/. The code in this guide was run, functionally, on OpenCV version 3.4.2.

Part 1 – Helper Classes

First, we will first write two helper classes, ShapeDetector and ColorLabeler (defined in shapedetector.py and colorlabeler.py respectively), which will aid us by detecting shapes and labelling colours. The ShapeDetector and ColorLabeler objects we used in the competition were borrowed from Adrian Rosebrook's tutorials, which you can find here:

ShapeDetector:

https://www.pyimagesearch.com/2016/02/08/opencv-shape-detection/

ColorLabeler:

https://www.pyimagesearch.com/2016/02/15/determining-object-color-with-opencv/

Note that the definitions of these classes are only a part of both of these tutorials. Instead of following the entire tutorials, you should skip to the part where Adrian defines the class and copy that definition.

Part 2 – HOGUtils Helper Functions

Most of the HOGUtils helper functions, originally written by Alex Cavalli, will be described in the following part for you to define in HOGUtils.py in your own project.

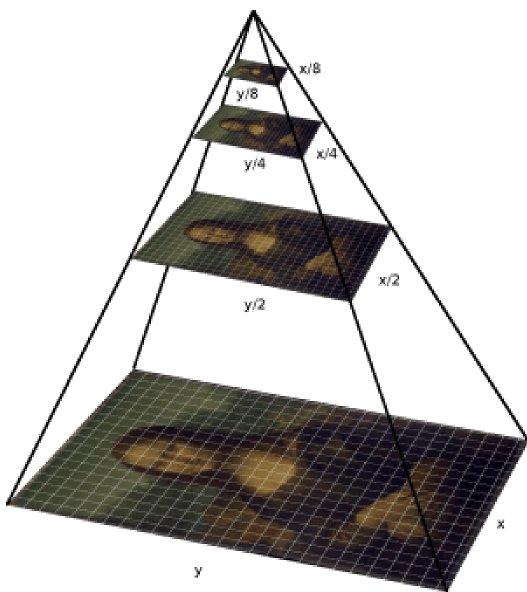
First are the modules to import:

```
1    import imutils
2    import cv2
3    import argparse
4    import numpy as np
5    import math
6    from modules.colorlabeler import ColorLabeler
7    from modules.shapedetector import ShapeDetector
```

As you can see, ShapeDetector and ColorLabeler are both imported from our project files.

Image Pyramids

First we will write a function that generates an image pyramid. An image pyramid is simply a series of images. Each image is a copy of the original image (also called the "base" image, as it is the pyramid's base) with each successive image having its length divided by some scale factor, s. The following diagram depicts an image pyramid with a scale factor of 2:



Each layer has a width and height which is equal to the width and height of the previous layer, divided by s. The base layer has width and height w and h. The next layer has width and height w/s and h/s, and the next layer will have w/s^2 and h/s^2, and so on.

The following diagram depicts a general image pyramid with a scale factor of s:

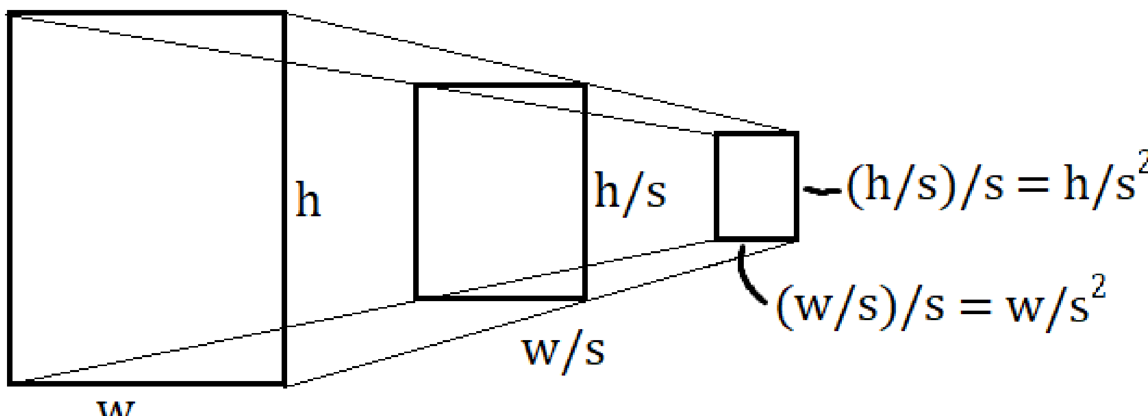


Image pyramids are useful for increasing the accuracy of image analysis tasks, like contour detection.

The pyramid() function will take three inputs: an image, a value for s, and a tuple for the minimum dimensions. The function ends when the current layer is smaller than the minimum size (so the current layer will become the final layer).

Define the function as follows:

```
21    def pyramid(image, scale=1.5, minSize=(30, 30)):
22        base_width = image.shape[1]
23        # yield the original image
24        yield (image, 1)
```

After the definition, we store the width of the base image in *base_width* and yield the base of the pyramid to be the first entry in the output. The *(image, 1)* tuple will make more sense when you see what the rest of the outputs look like.

Next, we open a while True loop. This loop will yield successive levels of the pyramid, until we reach an image which is smaller than the *minSize* tuple, when the loop will break.

```
27    while True:
28        # compute the new dimensions of the image and resize it
29        new_width = int(image.shape[1] / scale)
30        image = imutils.resize(image, width=new_width)
31        ratio = base_width/float(image.shape[1])
```

First, we divide by s to get the width of the next level. Then, we resize the current image to have this new width, to be the image for the next level.

Now, we want to know the ratio between the width of the base level and the width of the current level. Unlike "s" (scale), which gives us a connection between the current level and the level beforehand (level-beforehand-width / s = current-level-width) we want this number to connect the base level to the current level (base-width / ratio = current-level-width).

We can consider each level of the pyramid having a number – the base is level 0, then the next are 1, 2, 3, 4… we will call this number n. As seen in the diagrams above, the width of the nth pyramid level will be equal to w/s^n (w/s/s/s/s n times). So, this number that we want is s^n, as I described in the paragraph above (base-width / ratio = current-level-width).

We find s^n by dividing the width of the base of the pyramid (base_width) by the width of the current level of the pyramid (image.shape[1]). This can be shown by:

$$ratio = \frac{w}{w_{current}}$$

And:

$$w_{current} = \frac{w}{s^n}$$

So:

$$\begin{aligned} ratio &= \frac{w}{w_{current}} \\ &= \frac{w}{w/s^n} \\ &= \frac{w}{w/s^n} \cdot \frac{s^n}{s^n} \\ &= \frac{w \cdot s^n}{w} \\ &= s^n \end{aligned}$$

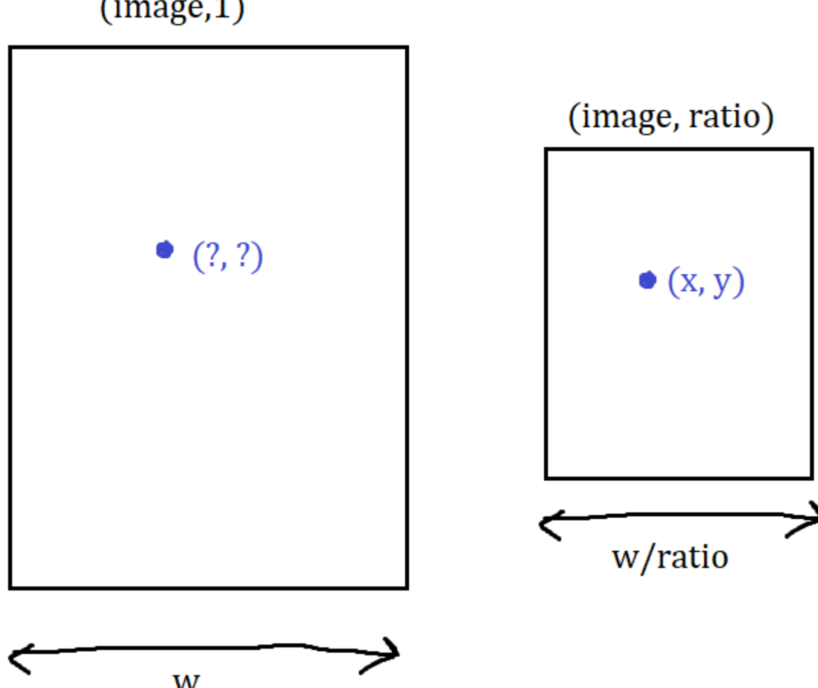
Now, with these things we've computed, we finish by checking if the current image is under the minimum width or under the minimum height. If so, we break. If not, we yield the current image and the current ratio.

```
34        if image.shape[0] < minSize[1] or image.shape[1] < minSize[0]:
35            break
36
37        # yield the next image in the pyramid
38        yield (image, ratio)
```

The output of this function will look something like:

pyramid(img, s, minSize) = [(img, 1), (img, s), (img, s^2), (img, s^3) … (img, s^n)]

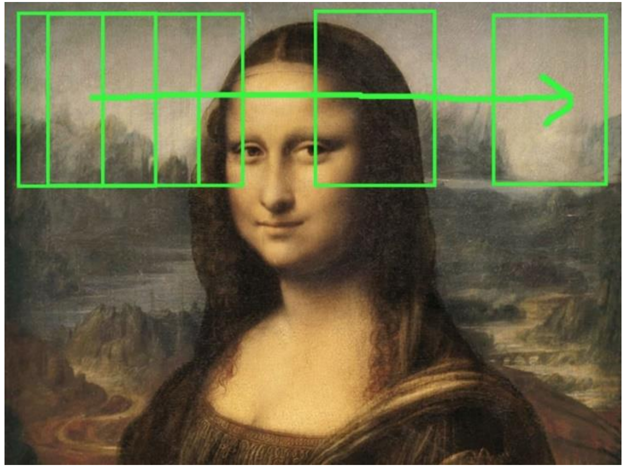
When searching through the image, any coordinates you find on a higher level of the pyramid will need to be scaled back up to the base layer. Suppose the following:



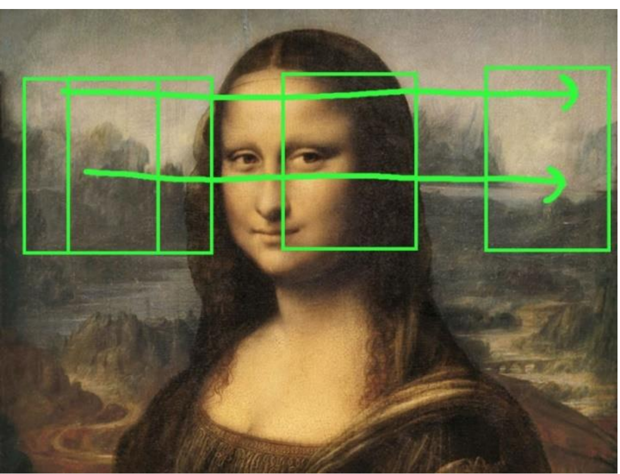
As the width and height of the current layer are equal to the width and height of the base layer divided by the ratio, you can imagine that the x and y coordinates we have were also divided by the ratio to be mapped onto the smaller image. So, to bring it back to the base image we need to multiply them by the ratio. Note that this "ratio" I am referring to is the one given in the output of the pyramid() function. I give more detail on how to do this in the *Improving Contour Detection* section.

Sliding Windows

The next HOGUtils function is similar to image pyramid, in that it turns a single image into a series of images so your image analysis can be more effective. The sliding window slices the image up into sections, as though you were looking over it with a flashlight.



As seen in the image above, in a similar motion to moving a flashlight, the sliding window takes slices out moving across. When it reaches the edge it lowers the window slightly and begins again from the left:



This process should be paired with an image pyramid. You should create an image pyramid and put a sliding window over *each* level. I discuss this further in the *Improving Contour Detection* section.

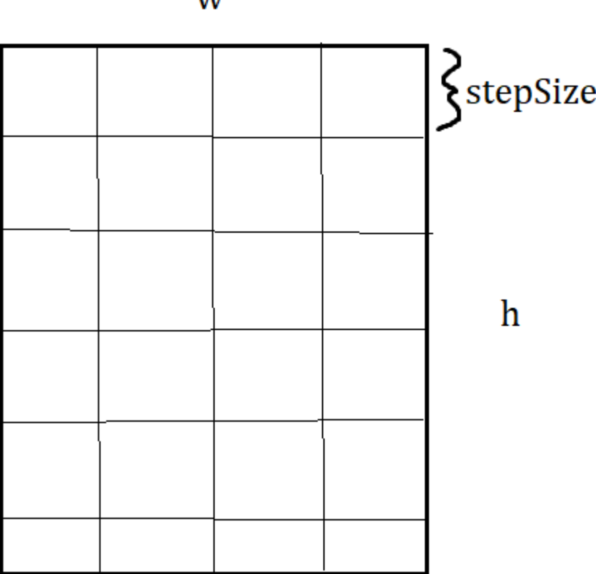
The sliding window function takes an image, a step distance and a window size tuple as inputs. We define it as follows:

```
42  def sliding_window(image, stepSize = 32, windowSize = (128, 128)):
43      # slide a window across the image
44      for y in range(0, image.shape[0], stepSize):
```

First, we begin looping over every possible y height for the sliding window using the range function. Underneath this for loop, we also loop over every possible x value:

```
44      for y in range(0, image.shape[0], stepSize):
45          for x in range(0, image.shape[1], stepSize):
```

You can imagine the x and y values generated as being in these positions:

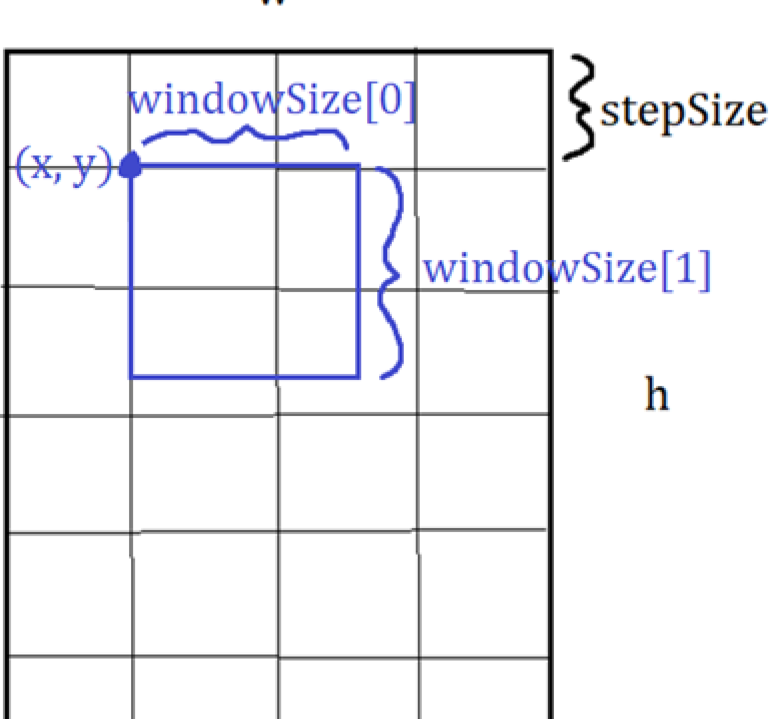


The double for loop pairs every possible y height with every possible x distance. You can imagine these pairs as the spots on the image above where the lines cross over. With every possible combination of the x and y values, we can cut evenly spaced slices out to use as our “sliding window”, moving from one to the next like a flashlight.

```
47                  yield (x, y, image[y:y + windowSize[1], x:x + windowSize[0]])
```

Now, we simply yield a new tuple. The tuple is of form (x, y, image). The x and y are the coordinates of the top left hand corner of the image, which is calculated by slicing a region out of the whole image, starting at x and y and going to the right and down by our sliding window dimensions from the function inputs.

Calling image[1:2, 6:7], for example, will cut out a slice between the x values 6 and 7 and between the y values 1 and 2. We are doing the same, except with y:y+windowSize[1], and x:x+windowSize[0], as shown in the image below.



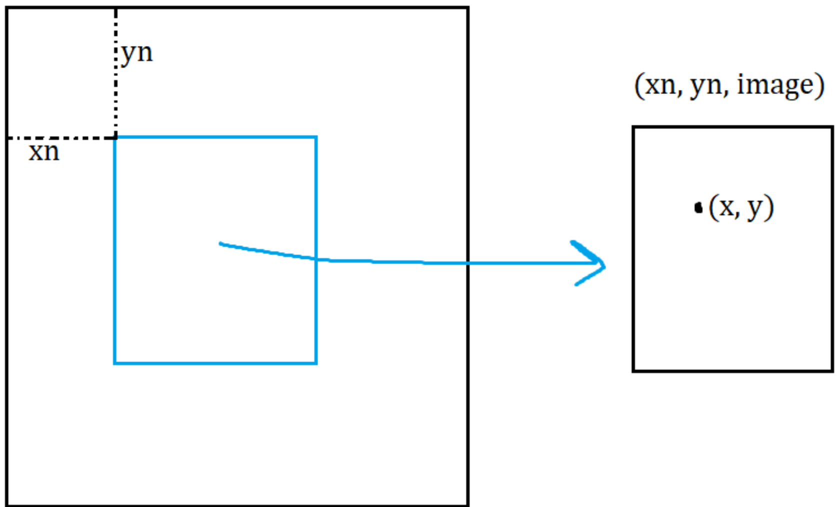
The output of the function should look like this:

sliding_window(img, stepSize, windowSize) = [(x1, y1, image), (x2, y1, image), (x3, y1, image) … (x1, y2, image), (x2, y2, image), (x3, y2, image) …]

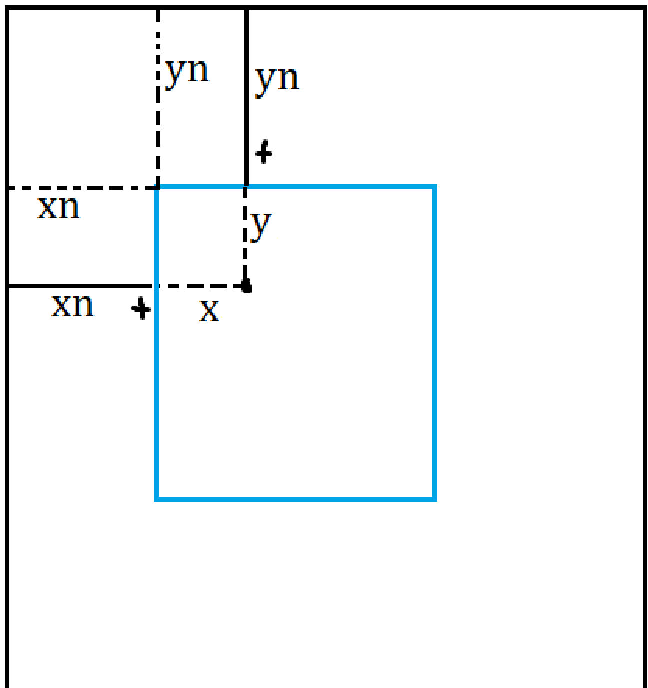
(the y values stay constant because they are on the outside of the double for loop).

This output can be iterated over, and each image operated on individually.

Similar to image pyramids, if you find a coordinate inside a sliding window that you want to keep, you need to do some maths to map it back to the original image. Suppose the following scenario:



To map our (x,y) point that we found in the sliding window back to the base image, we simply add xn to x and yn to y to get this:



Contour Detection

Contour detection locates areas in the image which seem to be a boundary between two colours. We did not utilize sliding windows or image pyramids in the final version of our contour detection; however they are very important concepts and you can see *Improving Contour Detection* to see how you would apply them.

We will make a function called colorShape that will take 4 inputs: an image, a boolean stating whether or not to include colour detection, another for shape detection, and a threshold value for binary thresholding. Define it as follows:

```
195  def colorShape(image, shapes = True, colors = True, thresh = 127):
196          # blur the resized image slightly, then convert it to both
197          # grayscale and the L*a*b* color spaces
198          blurred = cv2.GaussianBlur(image, (5, 5), 0)
199          gray = cv2.cvtColor(blurred, cv2.COLOR_BGR2GRAY)
```

And, after the definition, create some variations on the image. *blurred* is the image with a slight blur applied, and *gray* is the image in greyscale. This is done with some OpenCV helper functions.

Next, we perform binary thresholding on the greyscale variation of the image using the OpenCV function. The function returns an array, but we only want the image, so we take it from the array's second entry (hence the [1] at the end of each function call).

```
208          thresholded = cv2.threshold(gray, thresh, 255, cv2.THRESH_BINARY)[1]
209          inverted = cv2.threshold(gray, thresh, 255, cv2.THRESH_BINARY_INV)[1]
```

Binary thresholding is a very simple procedure, so much so that we could implement it ourselves (As OpenCV images are simply an array of pixel values, we could iterate over the array and manipulate each value manually). It simply looks at the brightness value of each pixel (255 being bright white, 0 being dark black). If it is below a certain threshold, force the pixel to be black, and if it is above that threshold force it to be white. We also create an inverted image, which switches the black pixels to white and the white ones to black. This is because the edge detector gives slightly different results for each.

Now, we find the contours in each of these images (taking the second output only) using the OpenCV functions.

```
211          # find contours in the thresholded image
212          _, cnts_thresh, _ = cv2.findContours(thresholded.copy(),
213          cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
214          _, cnts_invert, _ = cv2.findContours(inverted.copy(),
215          cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
```

Now, we initialise a shape detector and a colour labeller to help analyse the contours we found. We also make an empty list called *desc* where we will put our completed descriptions of each contour. On the last line, we start the for loop which iterates over every contour. We start with *cnts_thresh*, and we will repeat the exact same process for *cnts_invert* afterwards.

```
219          sd = ShapeDetector()
220          cl = ColorLabeler()
221          desc = []
222          # loop over the contours
223          for c in cnts_thresh:
```

In the following image, there is old code for computing the centre of the contour. We don't do this anymore, however the centre point is still included in the output. It would be too difficult to remove it from the output, because we would then have to find each place that the function is called and change it to expect a different number of outputs. It is far easier just to set it to an arbitrary pair of numbers.

```
224                 # compute the center of the contour
225                 # M = cv2.moments(c)
226                 # if M["m00"] != 0:
227                 #         cX = int((M["m10"] / M["m00"]))
228                 #         cY = int((M["m01"] / M["m00"]))
229                 # else:
230                 #         cX = int((M["m10"] / (M["m00"]+1)))
231                 #         cY = int((M["m01"] / (M["m00"]+1)))
232                 cX, cY = 0, 0
```

You do not have to include any of the commented out code, unless you want to utilize the centre point of your contours in your project.

Next, we ask the colour labeller and the shape detector to determine the shape and colour of whatever is inside the current contour (line) we are analysing.

```
235                 if shapes:
236                         shape = sd.detect(c)
237                 else:
238                         shape = ""
239                 if colors:
240                         color = cl.label(lab, c)
241                 else:
242                         color = ""
243
```

Next, we format the colour and the shape into some text:

```
247                 text = "{} {}".format(color, shape)
```

Add them to the array of descriptions:

```
252                 desc.append((c, (cX,cY), text))
```

And continue looping until we run out of contours. After this, we rewrite the same code but for *cnts_invert*, as seen below:

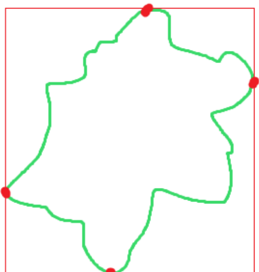
```
253         # loop over more contours
254         for c in cnts_invert:
255
256                 cX, cY = 0, 0
257
258                 if shapes:
259                         shape = sd.detect(c)
260                 else:
261                         shape = ""
262                 if colors:
263                         color = cl.label(lab, c)
264                 else:
265                         color = ""
266
267                 text = "{} {}".format(color, shape)
268
269                 desc.append((c, (cX,cY), text))
```

Now, outside of both of these for loops, we can return our now full array of descriptions:

```
284         return desc
```

As you can see, the level of indentation of the *return desc* statement should be the same as both for loops.

Finally, we want to create a very simple function that turns a contour into a bounding box. It simply finds the highest, lowest, leftmost and rightmost reaches of the contour and fits a box around it, as sketched below:



Passing in a contour “c” to our function, we can read the min and max values for X and Y as so:

```
95  def boundingBox(c):
96          extLeft = c[c[:, :, 0].argmin()][0][0]
97          extRight = c[c[:, :, 0].argmax()][0][0]
98          extTop = c[c[:, :, 1].argmin()][0][1]
99          extBot = c[c[:, :, 1].argmax()][0][1]
100         angle  = 0
101         return np.array([[extLeft,extTop,extRight,extBot,angle]])
```

The angle feature is from an older version and is no longer used, but we’ve left it there so we don’t have to change the parts of the code that call this function to expect a different number of outputs.

That is all for HOGUtils. However, if you would like to see how to combine contour detection with the *pyramid()* and *sliding_window()* functions, I have included a demonstration at the end of the guide called *Improving Contour Detection*.

Part 3 – Classification

Classification, in this context, will be the process of taking a region of an image which we’ve detected and assigning it with the type of hazmat sign we think it’s most likely to be. For this, we use feature matching algorithms that come with OpenCV to make a decision about which of our reference images looks the most like the photo we took. This will go in *classify_abstracted.py*.

The following images, taken from my blog on the SART website on classification, should help to give you an intuitive understanding of the algorithm. Features are mapped on an image, represented by the rings drawn on the following hazmat sign template:

The algorithm can also compare features on two different images, to see if they are in similar positions. The algorithm does account for rotation.

First, the packages to import:

```
2    # imports
3    import numpy as np
4    import argparse
5    import glob
6    import cv2
```

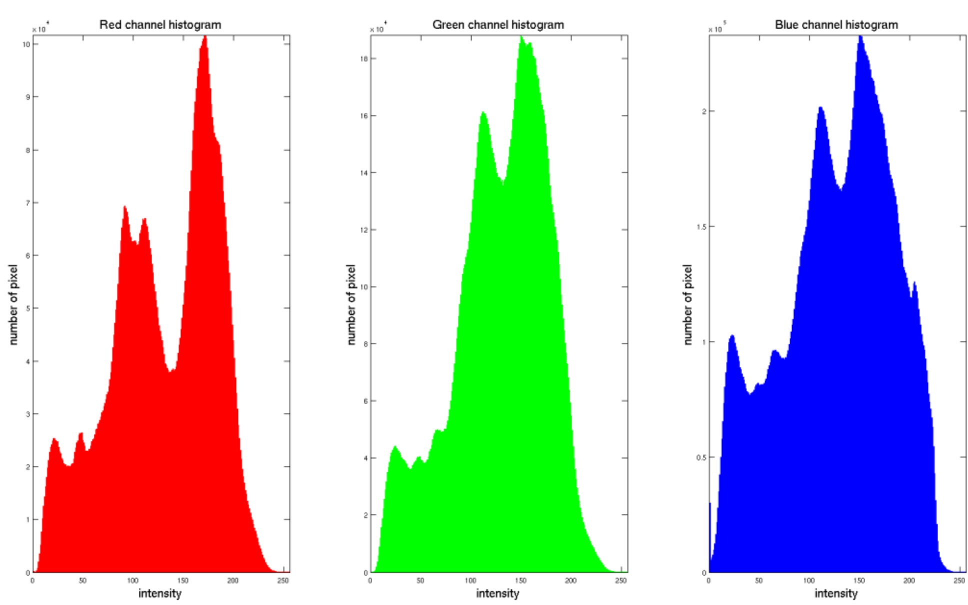
Now, we define the “min match rating” parameter for the brute force feature matching. The number is associated with how similar the positions of features in one image need to be to the features in another image to be considered a good match. We use it to filter out bad matches a list of possibilities. Later, when we define the bff_match() function, I will explain why lowering the number makes the filtering process more strict, and increasing it makes is less strict.

```
8    # Brute Force Feature matching
9    MIN_MATCH_RATING = 0.7
```

Now, some parameters for the colour based comparison:

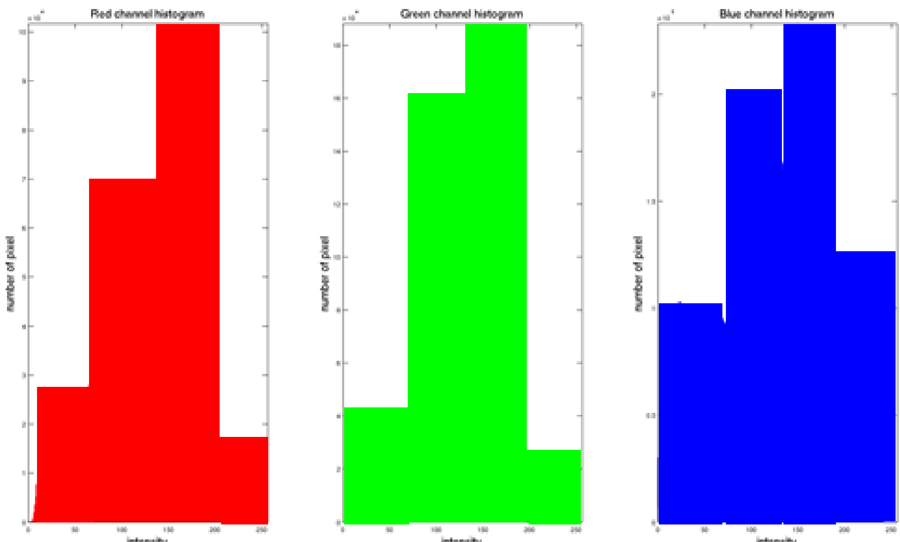
```
11    # Colour histogram comparisons
12    BINS = 4
13    COL_COMP_METHOD = cv2.HISTCMP_INTERSECT
```

When the colours in two images are compared, they are first made into histograms, showing the frequency of each colour in the image. An image can be considered a long list of 3 dimensional points, with a red value, blue value and green value. These points can be reduced to a frequency histogram which would look something like this:



These can be compared to see which two images have more similar colours, and which have less similar. The only issue is it completely disregards the *position* of things in an image – you could flip an image upside down and it would have an identical colour histogram. This is why we also do feature matching.

The “BINS” variable defines the number of bins to which numbers will be allocated. In a histogram, bins can be though of as columns. With 4 bins, the histograms above would each only have four columns, as shown below:



This is less accurate, but would probably be faster to compute.

The “COL_COMP_METHOD” variable chooses an option for which method to use to compare the similarity of two histograms. The chosen one is called “HISTCMP_INTERSECT” and probably has something to do with where the lines of the histograms intersect.

Next, we create a template for a sign object:

```
16  # Create a class to store sign image, name and match data
17  class Sign(object):
18
19      def __init__(self, image, name):
20          self.name = name
21          self.image = image
22          self.col = 0
23          self.bff = 0
24          self.bff_data = []
```

Signs will have multiple properties, including a name, an image (file location to save memory), their colour histogram, a list of all the features which have been detected on them that match features detected on some other image (stored in self.bff_data) and the *number* of those features (in self.bff). This will make more sense later.

Next, create a helper function which gets the histogram of an image using our parameters:

```
33  # Simply convert to a histogram
34  def get_hist(image):
35      hist = cv2.calcHist([image], [0, 1, 2], None, [BINS, BINS, BINS],
36          [0, 256, 0, 256, 0, 256])
37      return cv2.normalize(hist,hist).flatten()
```

I can only assume that the “cv2.normalize” function called on the histogram stretches it to fit in a certain size. It would be problematic trying to compare two images of different sizes – even if one of the two images is very red, it may still have less red pixels than the other, simply because it is smaller. Next, we make a helper function which compares two histograms:

```
41    def color_match(image, template):
42        return cv2.compareHist(get_hist(image), get_hist(template), COL_COMP_METHOD)
```

Depending on which method you use for colour comparison, it will in most cases return a single number as a similarity score. This is meaningless on its own, but when compared with scores from other images, you can tell which are more similar. This is not actually used in our feature matching, as it is not entirely necessary, but it could be added in future.

Now, we start defining the feature matching function. Start with two parameters, one for the image we think may be a sign (image) and one for our template (template).

```
78    def bff_match(image, template):
79        sift = cv2.xfeatures2d.SIFT_create()
80
81        kp_detection, des_detection = sift.detectAndCompute(image, None)
82        kp_template, des_template = sift.detectAndCompute(template, None)
```

Next, we make a SIFT object which will detect the features in both images. The detectAndCompute function that SIFT performs for us will give us a list of keypoints (kp_detection and kp_template) and descriptors (des_detection and des_template). Keypoints and descriptors are the raw data behind the abstract idea of a "feature" in an image.

Now, we set some parameters and make a "flann-based" KNN matcher:

```
84        FLANN_INDEX_KDTREE = 0
85        index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
86        search_params = dict(checks = 50)
87        flann = cv2.FlannBasedMatcher(index_params, search_params)
```

The flann matcher will use the KNN algorithm to match the features from our detected sign and our sign template, as follows:

```
89        matches = flann.knnMatch(des_detection, des_template, k=2)
```

The "matches" variable will be a list of pairs. Each pair will be a descriptor from one image and a descriptor from the other, which the KNN algorithm deemed to be a good match.

Now, we perform our own filtering on these matches. This is where the MIN_MATCH_RATING variable from earlier comes in, to let fewer or more matches through.

```
91        good = []
92        for m,n in matches:
```

First, we create an empty array for all "good" matches. Then, we search through every pair m and n in matches. The m variable will be a descriptor from one of the two images, and the n variable will be its counterpart descriptor from the other image. Then, we perform the following check:

```
93            if m.distance < MIN_MATCH_RATING*n.distance:
94                good.append(m)
```

As we aren't too sure what the "descriptors" of a feature really are, we can't be sure what their "distance" property is. However, if we use simple logic, we can find out what happens when we make MIN_MATCH_RATING bigger or smaller, without understanding these concepts. We are comparing these two "distances", but we don't know what they mean, so let's just call them A and B.

A < MIN_MATCH_RATING * B

As we can see:

A < 0.0001 * B

is less likely to be true than

A < 9999 * B

If the statement is true, the match is added to the “good” group. Therefore, we can deduce that having a smaller value for MIN_MATCH_RATING makes this check more “strict” and lets less matches into the “good” group, because the statement will be true less often.

Finally, we return the list of good matches:

```
95        return good
```

And finally, the function should look like this (make sure your whitespace is correct):

```
78    def bff_match(image, template):
79        sift = cv2.xfeatures2d.SIFT_create()
80
81        kp_detection, des_detection = sift.detectAndCompute(image, None)
82        kp_template, des_template = sift.detectAndCompute(template, None)
83
84        FLANN_INDEX_KDTREE = 0
85        index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
86        search_params = dict(checks = 50)
87        flann = cv2.FlannBasedMatcher(index_params, search_params)
88
89        matches = flann.knnMatch(des_detection, des_template, k=2)
90
91        good = []
92        for m,n in matches:
93            if m.distance < MIN_MATCH_RATING*n.distance:
94                good.append(m)
95        return good
```

The final function takes all these and compares the region we have found with *every* sign template to see which it best matches.

```
77    # classify the input image
78    def classify(image, sign_list):
79
80        for sign in sign_list:
81            template = cv2.imread(sign.image)
82            sign.bff_data = bff_match(image, template)
83            sign.bff = len(sign.bff_data)
84            sign.col = color_match(image, template)
```

First, we iterate over every sign in the sign list and calculate their features, including the image, feature data, colour histogram data, etc. This could probably be done at the start of the program instead of running it every single time we try to classify a sign, which is probably what causes the pause between when a sign is detected and when its name is displayed on screen. This should be the first thing you change when you complete this guide.

As you can see, bff_data is used to store the actual features themselves (which match with the image we’re trying to classify), and bff is used to store the *number* of those features.

```
89        sign_list.sort(key=lambda x: x.bff, reverse=True)
90
91        best = sign_list[0]
92
93        return best.name
94
```

This last part is very simple. We sort the list of sign templates by how many matches they have, but in reverse so that the sign with the most matches comes first. Then, we take it out and store it in “best” and return its name. This is the name that is displayed on screen.

Part 4 – Bringing it all Together

Now, we use all of the abstraction we just made to create a simple program that does the actual hazmat reading for us. This goes in *hazmat.py*.

We start by importing all the packages we need, including HOGUtils which we wrote. We also create a variable to store the ip address of our robot on our local network.

```
2     import modules.HOGUtils as HU
3     import imutils
4     import argparse
5     import cv2
6     import numpy as np
7     from math import pi, degrees
8     from modules.classify.classify_abstracted import *
9     from imutils.video import VideoStream
10    import time
11
12    # ip
13    ip = "10.0.0.3"
```

Now for the arguments:

```
15    ap = argparse.ArgumentParser()
16    ap.add_argument("-vs", "--videosource", default='r', help="-vs r for robot
17    ap.add_argument("-t","--threshVal", default=127, type = int, help="Thresh
18    ap.add_argument("-m", "--minimum", default=200, type=int, help="The minim
19    ap.add_argument("-v", "--video", default=None, help="The path to the input
20    args = vars(ap.parse_args())
```

The “help” argument is the last for each, and you can put whatever you like in it.

Now, we need to create a list of signs for each template sign, as in our modules/classify/templates folder. As shown below, create a list and manually append each sign into it with their file path and name.

```
22  # Init templates
23  sign_list = []
24  templates_dir = "modules/classify/"
25  FILETYPE = ".png"
26  sign_list.append(Sign(templates_dir+"templates/1" + FILETYPE, "Explosives 1.1 1"))
27  sign_list.append(Sign(templates_dir+"templates/2" + FILETYPE, "Explosives 1.2 1"))
28  sign_list.append(Sign(templates_dir+"templates/3" + FILETYPE, "Explosives 1.3 1"))
29  sign_list.append(Sign(templates_dir+"templates/4" + FILETYPE, "Explosives 1.4 1"))
30  sign_list.append(Sign(templates_dir+"templates/5" + FILETYPE, "Blasting Agents 1.5 1"))
31  sign_list.append(Sign(templates_dir+"templates/6" + FILETYPE, "Explosives 1.6 1"))
32  sign_list.append(Sign(templates_dir+"templates/7" + FILETYPE, "Flammable Gas 2"))
33  sign_list.append(Sign(templates_dir+"templates/8" + FILETYPE, "Non-Flammable Gas 2"))
34  sign_list.append(Sign(templates_dir+"templates/9" + FILETYPE, "Oxygen 2"))
35  sign_list.append(Sign(templates_dir+"templates/10" + FILETYPE, "Inhalation Hazard"))
36  sign_list.append(Sign(templates_dir+"templates/11" + FILETYPE, "Flammable 3"))
37  sign_list.append(Sign(templates_dir+"templates/12" + FILETYPE, "Gasoline 3"))
38  sign_list.append(Sign(templates_dir+"templates/13" + FILETYPE, "Combustible 3"))
39  sign_list.append(Sign(templates_dir+"templates/14" + FILETYPE, "Fuel Oil 3"))
40  sign_list.append(Sign(templates_dir+"templates/15" + FILETYPE, "Dangerous When Wet 4"))
41  sign_list.append(Sign(templates_dir+"templates/16" + FILETYPE, "Flammable Solid 4"))
42  sign_list.append(Sign(templates_dir+"templates/17" + FILETYPE, "Spontaneously Combustible 4"))
43  sign_list.append(Sign(templates_dir+"templates/18" + FILETYPE, "Oxidizer 5.1"))
44  sign_list.append(Sign(templates_dir+"templates/19" + FILETYPE, "Organic Peroxide 5.2"))
45  sign_list.append(Sign(templates_dir+"templates/20" + FILETYPE, "Inhalation Hazard 6"))
46  sign_list.append(Sign(templates_dir+"templates/21" + FILETYPE, "Poison 6"))
47  sign_list.append(Sign(templates_dir+"templates/22" + FILETYPE, "Toxic 6"))
48  sign_list.append(Sign(templates_dir+"templates/23" + FILETYPE, "Radioactive 7"))
49  sign_list.append(Sign(templates_dir+"templates/24" + FILETYPE, "Corrosive 8"))
50  sign_list.append(Sign(templates_dir+"templates/25" + FILETYPE, "Oher Dangerous Goods 9"))
51  sign_list.append(Sign(templates_dir+"templates/26" + FILETYPE, "Dangerous"))
52
```

Now, we want to figure out where the video is coming from based on the arguments from argparse.

```
53  if args['videosource'] == "r":
54      # robot
55      vs = VideoStream(src="http://"+ip+":8081").start()
56  else:
57      # webcam
58      if args["video"] == None:
59          vs = VideoStream(src=0).start()
60          time.sleep(2.0)
61      # video file
62      else:
63          vs = cv2.VideoCapture(args["video"])
64
```

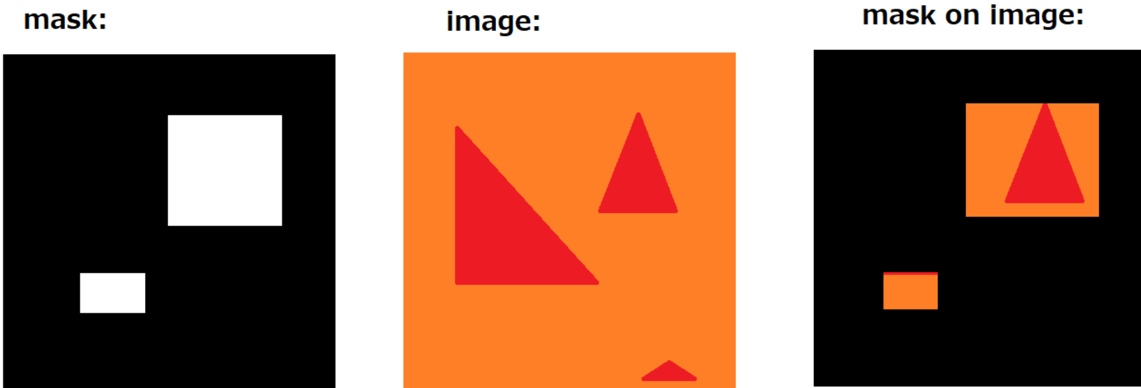
Now for the main loop. The first thing in the main loop will be to read the most recent frame.

```
66  # loop over the frames of the video
67  while True:
68      frame = vs.read()
69      frame = frame if args["video"] == None else frame[1]
70
71      # if the frame could not be grabbed, then we have reached the end
72      # of the video
73      if frame is None:
74          break
```

If the frame is being read from a video file then we need to read it slightly differently, as can be seen in line 69. If no frame is being read we break the loop. Next, operate on the frame:

```
76      # operate
77      frame = imutils.resize(frame, width=500) # the frame
78      descs = HU.colorShape(frame, colors=False, thresh=int(args['threshVal'])) # info about the frame
79      mask = np.zeros(frame.shape[:2], dtype="uint8") # a new black image the same size as the frame
80      sqrs = np.zeros([0,5]) # a new array to store the bounding boxes
```

First, we resize the frame to a width of 500 pixels. Then we read the colorShape output for the frame. Then we create a black image with the same dimensions as the frame which we will use as a mask. A mask is a black and white image which the original image can “wear” like a mask e.g.:



Finally, we create an array for all bounding boxes of the contours we detected. A bounding box has 5 elements – a min and max X and Y value (4) plus the angle (5) (see part 2). So, the new array is 5 wide, and starts as 0 deep – this will increase as we add boxes to it.

Now, we want to loop over all the contours we detected:

```
81      for (contour, centre, desc) in descs: # for each contour in the info about the frame
82          # if it is a square and is big enough
83          if "square" in desc.lower() and cv2.contourArea(contour) > int(args['minimum']):
84              sqrs = np.append(sqrs, HU.boundingBox(contour),0) # append it to the bounding boxe
85              cv2.drawContours(mask, [contour], -1, 255, -1) # draw it onto the black image in a
```

If the shape “square” was detected, and the contour area (number of pixels within the contour) is larger than our minimum (just stick with the default), we want to add the bounding box around that contour to the sqrs array. We also want to draw the contour onto the black image we created to be our mask. We draw it in white (hence the 255) and fill it in (hence the -1 for thickness) to create the mask effect as shown above.

Next, we want to apply the mask to the image to create a “masked” version, and we also want to choose an image that we want to draw on and then display (I chose the frame).

```
87      # this combines them into one image
88      masked = cv2.bitwise_and(frame, frame, mask=mask)
89      draw_image = frame
```

Now, we iterate over all the bounding boxes. We take the current bounding box out as a region, which we will operate on. Even though we include the space around the sign in our feature mapping, it shouldn’t affect it as we take it from the masked image (so it’s black).

```
91      # calculate
92      for bounding_box in sqrs:
93
94          # box
95          x1, y1, x2, y2 = int(bounding_box[0]), int(bounding_box[1]), int(bounding_box[2]), int(bounding_box[3])
96          region = masked[y1:y2, x1:x2]
```

Now, we want to draw this result onto the screen. We use our classify function to predict what it is, and come up with various other things like coordinates, colours, font settings etc. The text x and y is at the centre of the bounding box.

```
98      # constants
99      text = classify(region, sign_list)
100     text_x = int(x1 + (x2-x1)/2)
101     text_y = int(y1 + (y2-y1)/2)
102     colour = (255, 255, 255)
103     black = (0, 0, 0)
104     font_size = 0.5
105     font_thickness = 1
106     buff = 0
107
```

Now, we draw the text as well as a box behind it onto the screen:

```
108     # get text position etc
109     (text_width_1, text_height_1) = cv2.getTextSize(text, cv2.FONT_HERSHEY_SIMPLEX,
110      fontScale=font_size, thickness=font_thickness)[0]
111
112     # get coords of rectangle behind text
113     box_coords_1 = ((text_x + buff, text_y + buff),
114      (text_x + text_width_1 - 2*buff, text_y - text_height_1 - 2*buff))
115
116     # draw rectangle behind text
117     cv2.rectangle(draw_image, box_coords_1[0],
118      box_coords_1[1], black, cv2.FILLED)
119
120     # draw text
121     cv2.putText(draw_image, text, (text_x, text_y),
122      cv2.FONT_HERSHEY_SIMPLEX, font_size, colour, font_thickness)
```

After the loop is complete, and this process is finished for all contours we detected, we draw the frame to the screen and check to see if the user exits the program (make sure this happens outside the for loop).

```
120     cv2.imshow("Camera Feed", draw_image) # i
121
122     # if the "q" key is pressed, break from t
123     key = cv2.waitKey(500) & 0xFF
124     if key == ord("q"):
125         break
```

Then, outside the while loop, we add some code to close the window (which will only be run if the while loop has been broken):

```
127 # cleanup the camera and close any open windows
128 vs.stop() if args["video"] == None else vs.release()
129 cv2.destroyAllWindows()
```

And that’s it for hazmat.py. To run it, type in console “python hazmat.py -vs w” to make sure it uses the webcam. The rest of the arguments can be left to their defaults.

Improving Contour Detection

We can improve on contour detection by turning an image into a pyramid and applying a sliding window on every single level, and finding as many contours as we can.

Before I get into it, I want to discuss the way contours are structured in OpenCV. When you run cv2.findContours() on an image, you get an array of contours. Each contour is a line that follows an edge somewhere on your image. So, you could say the contours are structured as **[line1, line2, line3 …]** inside the array. So, the lines can be iterated over.

A line is an array of pixel points on your image. However, it is structured strangely. Here is an example of a line array:

```
[[[ 98  74]]

 [[ 97  75]]

 [[ 97 110]]

 [[ 96 111]]

 [[ 96 127]]

 [[127 127]]

 [[127  75]]

 [[114  75]]

 [[113  74]]]
```

Each point is contained in two sets of square brackets, like [[x, y]] (as opposed to [x, y]). You can imagine that [x, y] is the first element in a separate array, [[x, y]].

If I iterate over a line in some dummy code:

```
19    for point in line1:
20        # point will be [[x, y]]
21        print(point[0])
22        # point[0] will be [x, y]
```

I have to take point[0], as [[x, y]][0] = [x, y]. After I have stripped off the outside square brackets I can reference the x and y entry of the point separately. For example:

[[x, y]][0][0] = [x, y][0] = x

and

[[x, y]][0][1] = [x, y][1] = y

I haven't included the following file in the project files list from above. This file is purely a dummy for testing. From here, you would want to apply this to colorShape in HOGUtils (once you have it functioning).

First, we import everything and define some variables.

```
2     import cv2
3     import modules.HOGUtils
4     import numpy
5
6     image = cv2.imread("shapes_and_colors.jpg")
7     blurred = cv2.GaussianBlur(image, (5, 5), 0)
8     gray = cv2.cvtColor(blurred, cv2.COLOR_BGR2GRAY)
9     thresholded = cv2.threshold(gray, 127, 255, cv2.THRESH_BINARY)[1]
10
11    all_contours = []
```

I read a random image called *shapes_and_colors.jpg*. I blur the image, greyscale it and threshold it. Typically I would want to threshold it *and* invert it, but this is only for testing.

I also create an array called *all_contours*. This array will store all our contours (the individual lines themselves, not the arrays of these lines).

```
13  for pyramid_level in modules.HOGUtils.pyramid(thresholded.copy()):
14  
15      pyramid_image = pyramid_level[0]
16      scale = pyramid_level[1]
```

First, I turn the image into an image pyramid and loop over all the levels. Whenever we come to a new level of the image pyramid, I do two things. The image pyramid function outputs not only an image, but a tuple pair with the image and the *ratio* of the level. See the *Image Pyramids* section for more details on the output. I save both values in *pyramid_image* and *scale* respectively.

Then, I iterate over the list of sliding windows for this level of the pyramid. See *Sliding Windows* for more details on the function.

```
18      for window in modules.HOGUtils.sliding_window(pyramid_image):
19  
20          window_x_position = window[0]
21          window_y_position = window[1]
22          window_image = window[2]
```

I also store the x and y coordinates of the *top left corner* of the current sliding window in two variables, and the image in its own variable last.

Now, I find all the contours within this specific sliding window of this specific level of the pyramid:

```
24          _, contours, _ = cv2.findContours(window_image.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
```

Iterate on them:

```
26          for line in contours:
27  
28              for point in line:
```

Translate and then scale the points (method as discussed in the *Image Pyramids* and *Sliding Windows* sections):

```
30                  (point[0])[0] = scale * ((point[0])[0] + window_x_position)
31                  (point[0])[1] = scale * ((point[0])[1] + window_y_position)
```

Note that I use point[0] to get the point as I described earlier. Add the newly translated and scaled contour (line) to the *all_contours* array:

```
33              all_contours.append(line)
```

Note that this statement should be *outside* the “for point in line” loop, as you only want to add this line to *all_contours* once, after it has finished being modified.

It should look like:

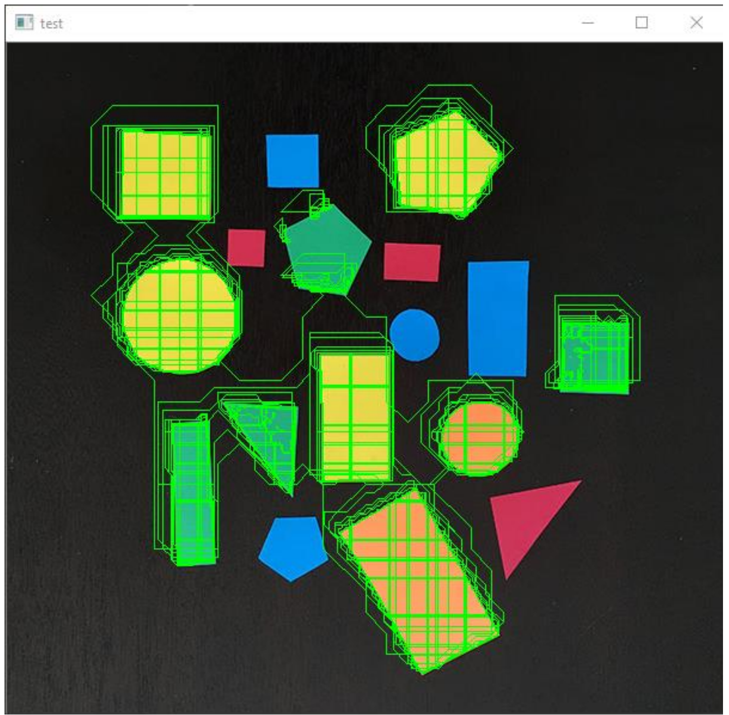
```python
for pyramid_level in modules.HOGUtils.pyramid(thresholded.copy()):

    pyramid_image = pyramid_level[0]
    scale = pyramid_level[1]

    for window in modules.HOGUtils.sliding_window(pyramid_image):

        window_x_position = window[0]
        window_y_position = window[1]
        window_image = window[2]

        _, contours, _ = cv2.findContours(window_image.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

        for line in contours:

            for point in line:

                (point[0])[0] = scale * ((point[0])[0] + window_x_position)
                (point[0])[1] = scale * ((point[0])[1] + window_y_position)

            all_contours.append(line)
```

Now we have an array, all_contours, filled with lines on our main image that were found in different image pyramid levels and different sliding windows.

Now, I have the following code:

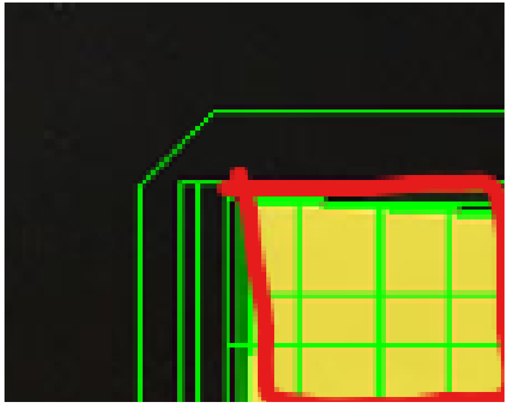
```python
print(all_contours[0])

for contour in all_contours:
    cv2.drawContours(image, [contour], 0, (0,255,0), 1)

cv2.imshow("test", image)
cv2.waitKey(0)
```

Which prints out all the points in the first contour (line) in *all_contours* which is useful for debugging, draws each line onto the image and displays it, then waits for a keypress. It looks like:



This is very ugly, but actually very accurate. The reason why some shapes were not detected is because we did not use the inverse of the binary thresholded image. Were we to use both, all shapes should be detected.

The overlapping contours are caused by the sliding window covering only part of a shape. For example, if the sliding window were at this position, only covering part of this rectangle:



It would draw a contour akin to the red line, covering the edge of the screen. This can be dealt with using nonmax suppression. An implementation of NMS can be found in SART's version of HOGUtils.py, if you'd like to look into it further.