Barcodes Detection and Description from Video Images

Bernardo Leite
up201404464@fe.up.pt
José Freixo
up201504988@fe.up.pt
Vítor Magalhães
up201503447@fe.up.pt

Faculdade de Engenharia
Universidade Do Porto
Porto, PT

Abstract

The necessity of access to information motivates the need for visual representations of said information. Detecting these representations, named barcodes, is important to quickly use the obtained data.

Using computer vision techniques, it is possible to rapidly detect barcodes from images and describe their contents. Tested images include simple barcodes and barcodes not in an almost frontal view or not occupying a significant portion of the image. The team presents a complete solution for detecting and isolating a barcode given an image and then characterizing the barcode bars and spaces.

1 Introduction

As people need faster ways to identify data, barcodes are furtherly used to represent even more information in a visual format in which machines can read from them. A barcode is a method of representing data in a visual, machine-readable form. When introduced, barcodes represented data by varying the widths and spacings of parallel lines. These barcodes, now commonly referred to as linear or one-dimensional (1D), can be scanned by special optical scanners, called barcode readers. Later, two-dimensional (2D) variants were developed, using rectangles, dots, hexagons and other geometric patterns. In this work our focus is only on 1D barcodes [3].

The objective of this project is to develop an application to detect barcodes and describe their contents, such as the width of each bar and space, using computer vision techniques. This project proposes an effective method for isolating the barcode from an image and then analyzing and characterizing each bar. In addition, the project also presents a visual solution so that it is possible to show the exact width of the black bars and white spaces between them.

Initially, the image will be segmented in order to identify any possible area where the barcode could be. Afterwards, the barcode is detected by calculating its contours and the area of the barcode is rotated and cropped in order to obtain the barcode in a cleaner image. Secondly, the program detects the bars and spaces of the barcode, as well as indicating the width of each and plotting a colored line over the code. Finally, the original image is presented with the barcode detected and the barcode is showed with the line in a separate code, along with the identification of the width of each bar and space.

Section 2 demonstrates related work. Section 3 presents the methodology adopted and, in section 4, results and discussion is presented. Finally, in section 5, the team concludes and discusses challenges and future work.

2 Related Work

Chunhui Zhang, Jian Wang, Shi Han, Mo Yi, and Zhengyou Zhang [5] have proposed an automatic and real-time barcode localization algorithm. The authors had proposed a method with a two-stage processing. They were able to detect barcodes through a region based analysis of a low-resolution image and then reading them in its original resolution. Extensive experiments have been conducted in complex 3-D scenes under various conditions.

Lichao Xu, Vineet Kamat, and Carol Menassa developed algorithms that are able to automatically extract barcodes from video data, and verifying their feasibility and promise for inventory management in warehousing applications [4]. The algorithm proposed by E. Ohbuchi, H. Hanaizumi, and L.A. Hock's algorithm is based on the code area found by four corners detection for 2D barcode and spiral scanning for 1D barcode using the embedded DSP.[2]

Finally, Rubén Muñiz, Luis Junco and Adolfo Otero presented a method based on the Hough transform which solves the problems related with noise like signatures and marks. It can be easily adapted to read any 1D barcode. [1]

3 Development

3.1 Solution

In this section, there will be a description of the solution of the problem, including Segmentation, Barcode Detection and Bar and Spaces Detection.

3.1.1 Segmentation

The Sobel operator is used particularly within edge detection algorithms where it creates an image emphasising edges. In this project, it is used to remove any horizontal lines in order to highlight vertical lines, which correspond to the barcode, by subtracting the x and y gradient of the operator.

Through blurring and applying a threshold, the barcode area is further highlighted through de-noising and removal of zones of the image that do not correspond to barcodes.

To finish work on the image, a kernel is applied to close any vertical gaps in order to connect loose white rectangles into one singular chunk. Afterwards, a series of erosions and dilations are performed to reduce more noise. [Fig. 1]

Figure 1: Progression of the Segmentation process

3.1.2 Barcode Detection

By using the resulting image, the objective is to find the contours of any white chunk in the image. With the function *findContours*, all contours are found. By calculating the extent, the correct contours are filtered and a bounding rectangle is created surrounding the barcode area to be applied in the original image. Extent is the ratio of contour area to bounding rectangle area.

$$Extent = \frac{ObjectArea}{BoundingRectangleArea}$$

A process of cropping is used to crop the barcode itself to a new image, whether the barcode is rotated or not. By using the bounding rectangle of the barcode area as a baseline for the cropping, a large and irrelevant portion of the image is removed, however it is not enough since the barcode can be slightly tilted [Fig. 3(a)].

Figure 6 is a scheme that represents how the rotation angle was calculated with angle α.

$$\alpha = \arccos\left(\frac{|A_x - B_x|}{\sqrt{(A_x - B_x)^2 + (A_y - B_y)^2}}\right)$$

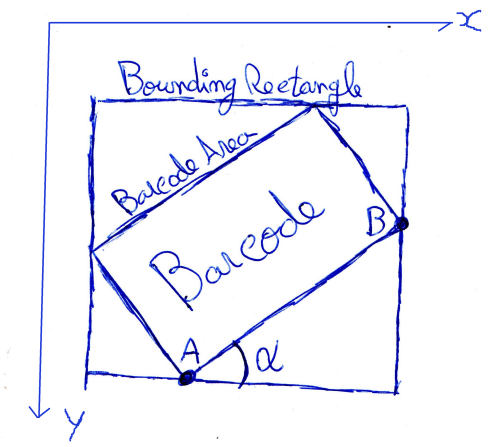

Figure 2: Scheme example of detected barcode

To rotate the barcode to a horizontal position, the angle of the barcode area is extracted through simple trigonometry and the image is therefore rotated to the side of the barcode area's largest side: this is necessary to avoid a vertical alignment instead of a horizontal one. This allows a horizontal view of the barcode. [Fig. 3(b)]

(a) Initial Crop (b) Rotated image (c) Final Crop

Figure 3: Crop progression

Lastly, to remove the black bars which originated from the rotation [Fig. 3(c)], the extra width and height need to be extracted. They can both be easily calculated through trigonometry but, their relation to the angle αis different when αis greater or lower than 45º. [Fig. 4]

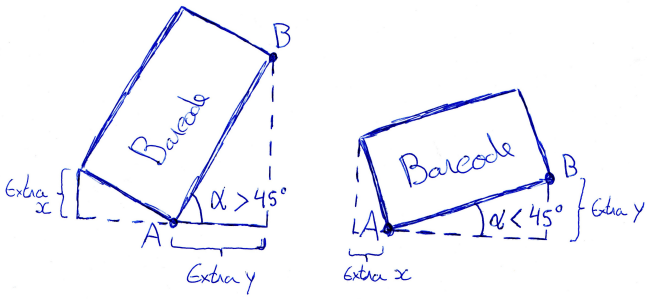

Figure 4: Scheme example of α's influence on the extra width and height

3.1.3 Bar and Spaces detection

After slightly blurring the image and thresholding it with a simple threshold algorithm, the program tries to find the contours that represent the bars and the spaces.

Firstly, the image containing the barcode is converted to gray scale so that a threshold is calculated using Otsu's Algorithm.[Fig. 5(a)]

This threshold is used to convert the image to black and white and define which pixels will be black or white. The pixels which have the RGB white color (255,255,255) are the white bars and the pixels that have the RGB black color (0,0,0) are the spaces. From here, it is was possible to distinguish black bars and white spaces.[Fig. 5(b)] All that remains is to calculate the center of the image so the program can draw a horizontal line across the bar code.

To draw the line, the following criteria is used: Through each position corresponding to the current pixel, its RGB component is analised:

- If the RGB component is white, then append to an array with red value.
- If the RGB component is black, then append to an array with blue value.

This way, the program obtains an array with all the pixels that should be drawn in the original image (final image cropped with the original colors), *i.e.* when the line crosses the black bars, it will turn red and when the line crosses the white spaces, it will turn blue. [Fig. 5(c)]

Finally, to calculate the width of each bar, the program knows the percentage of pixels of the red and blue colors. The team guarantees that the total number of bars detected will be thirty (standard barcode). This way, the barcode is fully characterized. Note that for exception situations where no thirty bars are detected the line is represented anyway only with the bars and spaces that were actually detected.

(a) Inverted barcode (b) Black and white barcode (c) Barcode with plotted line

Figure 5: Bar detection process

Finally, the output console shows the detailed information regarding the barcode. Firstly, it displays the dimensions of the image and then informational output about each bar as a percentage of the whole width of the barcode. [Fig. 6]

```
Black Bar detected. NR:  24
Number of Pixels:  11
Percentage of pixels: 3.09 %

White Bar detected.
Number of Pixels:  8
Percentage of pixels: 2.247 %

Black Bar detected. NR:  25
Number of Pixels:  11
Percentage of pixels: 3.09 %

White Bar detected.
Number of Pixels:  3
Percentage of pixels: 0.843 %

Black Bar detected. NR:  26
Number of Pixels:  4
Percentage of pixels: 1.124 %

White Bar detected.
Number of Pixels:  7
Percentage of pixels: 1.966 %
```

Figure 6: Console output showing the width of each bar and space

4 Results

4.1 Current Status

Despite the fact that some barcodes of some test images were not correctly detected (arguably the hardest ones to detect), the team was able to successfully obtain barcodes and identify information regarding the width of the bars and spaces as well as drawing a line throughout the code.

Overall, the project is able to identify barcodes that aren't in an almost frontal view and that don't occupy a significant portion of the image.

4.2 Performance

The following table presents the tests made and the overall sucess of finding the barcode (Segmentation):

Passed	Almost Passed	Didn't pass
10	9	6

The following table presents the tests made and the overall sucess of detecting the bars and spaces:

Passed	Almost Passed	Didn't pass
6	8	10

4.3 Main Problems and solutions

There were some problems regarding the segmentation of the barcode. Initally, the team started using Sobel's operator and tried switching to Laplacian's operator, but the outcome worsened, so there was some time spent on testing and ultimately rollbacking to the original operator.

Furthermore, the calculation of the extent of the contours allowed the program to identify the best area where the barcode could be.

There was some discussion between detecting contours as geometric shapes or detecting lines, in which the former was decided as the best approach.

5 Conclusions

This paper presents an effective solution for detecting and characterizing barcodes in different contexts. The techniques used allowed the team to isolate a barcode from a given image and then characterize each bar and space according to its width.

The results showed that the final program can be an alternative solution to existing methods depending on the needs in which it can be applied. For example, in a scenario where it is customary to have slanted barcodes, this solution might be a good approach. There are, of course, several paths that can be followed with the goal of improving the implemented system.

The main features to improve are specific situations such as barcodes hidden by some object, unwanted light/clarity, position and depth of the barcode within the image and curved surfaces. These situations are considered as possible improvements to be made in the future. In addition, a possible extension to the system would be to include detection of QR Codes.

References

[1] Rubén Muñiz, Luis Junco, and Adolfo Otero. A robust software barcode reader using the hough transform. pages 313–319, 02 1999. ISBN 0-7695-0446-9. doi: 10.1109/ICIIS.1999.810282.

[2] E. Ohbuchi, H. Hanaizumi, and L.A. Hock. Barcode readers using the camera device in mobile phones. pages 260 – 265, 12 2004. ISBN 0-7695-2140-1. doi: 10.1109/CW.2004.23.

[3] Wikipedia contributors. Barcode — Wikipedia, the free encyclopedia, 2019. URL `https://en.wikipedia.org/w/index.php?title=Barcode&oldid=924222989`. [Online; accessed 2-November-2019].

[4] Lichao Xu, Vineet Kamat, and Carol Menassa. Automatic extraction of 1d barcodes from video scans for drone-assisted inventory management in warehousing applications. *International Journal of Logistics Research and Applications*, 21:1–16, 10 2017. doi: 10.1080/13675567.2017.1393505.

[5] Chunhui Zhang, Jian Wang, Shi Han, Mo Yi, and Zhengyou Zhang. Automatic real-time barcode localization in complex scenes. pages 497–500, 01 2006. doi: 10.1109/ICIP.2006.312435.

```
# USAGE
# python detect_barcode.py --image images/barcode_01.jpg

# import the necessary packages
import numpy as np
import argparse
import imutils
import cv2
import math
import detect_shapes
from PIL import Image, ImageStat

# Detects if image is black and white or not
def detect_color_image(file, thumb_size=40, MSE_cutoff=22, adjust_color_bias=True):
 thresh = 225
 pil_img = Image.open(file)
 bands = pil_img.getbands()
 if bands == ('R','G','B') or bands== ('R','G','B','A'):
  thumb = pil_img.resize((thumb_size,thumb_size))
  SSE, bias = 0, [0,0,0]
  if adjust_color_bias:
   bias = ImageStat.Stat(thumb).mean[:3]
   bias = [b - sum(bias)/3 for b in bias ]
  for pixel in thumb.getdata():
   mu = sum(pixel)/3
   SSE += sum((pixel[i] - mu - bias[i])*(pixel[i] - mu - bias[i]) for i in [0,1,2])
  MSE = float(SSE)/(thumb_size*thumb_size)
  if MSE <= MSE_cutoff:
   thresh = 50
 elif len(bands) == 1:
  thresh = 50

 return thresh



# Parses argument and returns image and image in grayscale
def loadImage():
 ap = argparse.ArgumentParser()
 ap.add_argument("-i", "--image", required = True,
  help = "path to the image file")
 args = vars(ap.parse_args())
 thresh = detect_color_image(args["image"])

 # load the image and convert it to grayscale
 image = cv2.imread(args["image"])
 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
 return image, gray, thresh


def MainAlgorithm(gray, calc_thresh):
 # compute the Scharr gradient magnitude representation of the images
 # in both the x and y direction using OpenCV 2.4
 ddepth = cv2.CV_32F
 gradX = cv2.Sobel(gray, ddepth=ddepth, dx=1, dy=0, ksize=-1)
 gradY = cv2.Sobel(gray, ddepth=ddepth, dx=0, dy=1, ksize=-1)

 # subtract the y-gradient from the x-gradient
 gradient = cv2.subtract(gradX, gradY)
 gradient = cv2.convertScaleAbs(gradient)

 # blur and threshold the image
 blurred = cv2.blur(gradient, (9,9))
 (_, thresh) = cv2.threshold(blurred, calc_thresh, 255, cv2.THRESH_BINARY)

```

```
 # construct a closing kernel and apply it to the thresholded image
 kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (40, 7))
 aux = cv2.morphologyEx(thresh, cv2.MORPH_CLOSE, kernel)

 concat1 = np.hstack((blurred, aux))

 # perform a series of erosions and dilations
 closed = cv2.erode(aux, None, iterations = 6)
 closed = cv2.dilate(closed, None, iterations = 6)

 concat2 = np.hstack((concat1, closed))
 return closed, concat2

# shows concatenated images in a window
def showProgressInWindow(image):
 screen_res = 1440, 1080
 scale_width = screen_res[0] / image.shape[1]
 scale_height = screen_res[1] / image.shape[0]
 scale = min(scale_width, scale_height)
 window_width = int(image.shape[1] * scale)
 window_height = int(image.shape[0] * scale)
 cv2.namedWindow('Progress', cv2.WINDOW_NORMAL)
 cv2.resizeWindow('Progress', window_width, window_height)
 cv2.imshow('Progress', image)
 cv2.waitKey(0)

# Expands the box by 5% from it's center
def expandBox(box):

 expansion_rate = 0.03

 box[0,0] = box[0,0] + (box[0,0] - box[2,0]) * expansion_rate
 box[0,1] = box[0,1] + (box[0,1] - box[2,1]) * expansion_rate

 box[1,0] = box[1,0] + (box[1,0] - box[3,0]) * expansion_rate
 box[1,1] = box[1,1] + (box[1,1] - box[3,1]) * expansion_rate

 box[2,0] = box[2,0] + (box[2,0] - box[0,0]) * expansion_rate
 box[2,1] = box[2,1] + (box[2,1] - box[0,1]) * expansion_rate

 box[3,0] = box[3,0] + (box[3,0] - box[1,0]) * expansion_rate
 box[3,1] = box[3,1] + (box[3,1] - box[1,1]) * expansion_rate


 return box

# find the contours in the thresholded image
def findCountors(closed):
 cnts = cv2.findContours(closed.copy(), cv2.RETR_EXTERNAL,
  cv2.CHAIN_APPROX_SIMPLE)
 cnts = imutils.grab_contours(cnts)
 cnts = sorted(cnts, key = cv2.contourArea, reverse = True)

 for c in cnts:
  area = cv2.contourArea(c)
  x,y,w,h = cv2.boundingRect(c)
  rect_area = w*h
  extent = float(area)/rect_area
  if extent > 0.6:
   break

 rect = cv2.minAreaRect(c)
 box = cv2.cv.BoxPoints(rect) if imutils.is_cv2() else cv2.boxPoints(rect)
 box = np.int0(box)
 box = expandBox(box)
 x, y, w, h = cv2.boundingRect(box)
```

```
 return box, x, y, w, h

# Returns the rotated rectangle's angle in relation to the X axis
def getRotationAngle(box):
 dist1 = math.sqrt((box[0,0] - box[1,0])**2 + (box[0,1] - box[1,1])**2)
 dist2 = math.sqrt((box[1,0] - box[2,0])**2 + (box[1,1] - box[2,1])**2)

 angle = 0
 if (dist1 > dist2):
  angle = -math.acos( (box[0,0] - box[1,0]) / dist1)
 else:
  angle = math.asin( (box[0,0] - box[1,0]) / dist1)
 return angle * 180 / math.pi, dist1, dist2

# Crops ROI of image
def crop(img, box, rectx, recty, w, h):

 cropped = np.zeros((h, w, img.shape[2]), img.dtype)
 for y in range(img.shape[0]):
  for x in range(img.shape[1]):
   if (cv2.pointPolygonTest(box, (x,y), False) >= 0):
    cropped[y - recty, x - rectx] = img[y,x]

 angle, dist0_1, dist1_2 = getRotationAngle(box)
 cropped = imutils.rotate_bound(cropped,angle)

 if dist0_1 > dist1_2:
  if angle > -45:
   extra_y = abs(box[0,1] - box[1,1])
   extra_x = abs(box[0,0] - box[3,0])
  else:
   extra_y = abs(box[0,0] - box[1,0])
   extra_x = abs(box[0,1] - box[3,1])
  cropped = cropped[extra_y:extra_y + int(dist1_2), extra_x:extra_x + int(dist0_1)]
 else:
  if angle < 45:
   extra_y = abs(box[0,1] - box[3,1])
   extra_x = abs(box[0,0] - box[1,0])
  else:
   extra_y = abs(box[0,0] - box[3,0])
   extra_x = abs(box[0,1] - box[1,1])
  cropped = cropped[extra_y:extra_y + int(dist0_1), extra_x:extra_x + int(dist1_2)]

 return cropped

# Prints line in image
def printLineInImage(inverted_image, pixelsColorLine, cropped, offSet):

 imageWidth = inverted_image.shape[1]
 imageHeight = inverted_image.shape[0]

 yPos = int(imageHeight/2)
 xPos = 0

 barCount = 0
 blackCount = 0
 whiteCount = 0

 print("\n")
 print("--- Image: ", imageWidth, "px :", imageHeight,"px --- \n")

 while xPos < len(pixelsColorLine): #Loop through collumns

  if set(pixelsColorLine[xPos]) == set([255, 153, 51]):
   whiteCount = whiteCount + 1
   if xPos > 0:
```

```
    if set(pixelsColorLine[xPos-1]) == set([0, 0, 255]):
     barCount = barCount + 1
     cv2.imshow("Building Line", cropped)
     print("Black Bar detected. NR: ", barCount)
     print("Number of Pixels: ", blackCount)
     print("Percentage of pixels:", str(round(blackCount * 100.0/len(pixelsColorLine), 3)), "% \n")
     blackCount = 0
     cv2.waitKey(0)

   if set(pixelsColorLine[xPos]) == set([0, 0, 255]):
    blackCount = blackCount + 1
    if xPos > 0:
     if set(pixelsColorLine[xPos-1]) == set([255, 153, 51]):
      print("White Bar detected.")
      print("Number of Pixels: ", whiteCount)
      print("Percentage of pixels:", str(round(whiteCount * 100.0/len(pixelsColorLine), 3)), "% \n")
      whiteCount = 0

   if barCount >= 1 and barCount < 30:
    cropped.itemset((yPos, xPos + offSet, 0), pixelsColorLine[xPos][0]) #Set B
    cropped.itemset((yPos, xPos + offSet, 1), pixelsColorLine[xPos][1]) #Set G
    cropped.itemset((yPos, xPos + offSet, 2), pixelsColorLine[xPos][2]) #Set R

   if barCount == 30 or barCount == 0:
    if set(pixelsColorLine[xPos]) == set([0, 0, 255]):
     cropped.itemset((yPos, xPos + offSet, 0), pixelsColorLine[xPos][0]) #Set B
     cropped.itemset((yPos, xPos + offSet, 1), pixelsColorLine[xPos][1]) #Set G
     cropped.itemset((yPos, xPos + offSet, 2), pixelsColorLine[xPos][2]) #Set R

    xPos = xPos + 1

   xPos = 0
   cv2.destroyAllWindows()

   cv2.imshow("Final cropped with Line", cropped)
   cv2.waitKey(0)


# Main program execution
def Main():
   (image, gray, thresh) = loadImage()

   (closed, concatImages) = MainAlgorithm(gray, thresh)
   #showProgressInWindow(concatImages) #for debug only
   box, rectx, recty, w, h  = findCountors(closed)

   cropped = crop(image, box, rectx, recty, w, h)


   cv2.drawContours(image, [box], -1, (0, 255, 0), 1)
   cv2.imshow("Image", image)
   cv2.waitKey(0)
   cv2.destroyWindow("Image")

   inverted_image = cv2.bitwise_not(cropped)

   pixelsColorLine, offSet = detect_shapes.shape_detection(inverted_image)

   printLineInImage(inverted_image, pixelsColorLine, cropped, offSet)


Main()
```

```
# import the necessary packages
import imutils
import cv2
import numpy as np
import math
import matplotlib.pyplot as plt
from PIL import Image, ImageStat

# Augments image's threshold to use in otsu's algorithm and returns calculated threshold
def findThresh(gray):

 hi_contrast = np.zeros(gray.shape, gray.dtype)
 for line in range(gray.shape[0]):
  for col in range(gray.shape[1]):
   hi_contrast[line, col] = np.clip(1.0 * gray[line, col], 0, 255)
 thresh, otsu = cv2.threshold(hi_contrast, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)

 return int(thresh)

# Detects the shapes of the bars and spaces
def shape_detection(inverted_image):

 # Convert to HSV colourspace
 hsv = cv2.cvtColor(inverted_image, cv2.COLOR_BGR2HSV)

 # Convert to grayscale
 gray = cv2.cvtColor(inverted_image, cv2.COLOR_BGR2GRAY)

 thresh = findThresh(gray)

 # Define the limits of the "black" colour
 black_lo = np.array([0, 0, 0])
 black_hi = np.array([360, 255, thresh])

 # Create mask to select "blacks"
 mask = cv2.inRange(hsv, black_lo, black_hi)

 # Change "blacks" to pure black and "whites" to pure white
 inverted_image[mask > 0] = (0, 0, 0)
 inverted_image[mask <= 0] = (255, 255, 255)

 resized = imutils.resize(inverted_image, width=1000)
 ratio = inverted_image.shape[0] / float(resized.shape[0])


 gray = cv2.cvtColor(resized, cv2.COLOR_BGR2GRAY)
 blurred = cv2.GaussianBlur(gray, (5, 5), 0)
 thresh = cv2.threshold(blurred, 60, 255, cv2.THRESH_BINARY)[1]

 # find contours in the thresholded inverted_image
 cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
 cnts = imutils.grab_contours(cnts)
 allCenterx = []
 allCentery = []

 # Set Pixel Colors according Black and White regions
 imageWidth = inverted_image.shape[1]
 imageHeight = inverted_image.shape[0]

 yPos = int(imageHeight/2)
 xPos = 0

 barCount = 0
 foundFirstBlack = False
 offSet = 0
```

```
66
67  pixelsColorLine = []
68
69  while xPos < imageWidth and barCount < 30:
70   if set(inverted_image[yPos, xPos]) == set([255,255,255]):
71    if not foundFirstBlack:
72     foundFirstBlack = True
73     offSet = xPos
74    pixelsColorLine.append([0, 0, 255])
75   elif foundFirstBlack:
76    pixelsColorLine.append([255, 153, 51])
77    if set(inverted_image[yPos, xPos - 1]) == set([255,255,255]):
78     barCount = barCount + 1
79
80   xPos = xPos + 1
81
82  xPos = 0
83
84  for c in cnts:
85
86   c = c.astype("float")
87   c *= ratio
88   c = c.astype("int")
89
90   rect = cv2.minAreaRect(c)
91   box = cv2.boxPoints(rect)
92   box = np.int0(box)
93
94   (x, y, w, h) = cv2.boundingRect(c)
95
96   dist1 = math.sqrt((box[0,0] - box[1,0])**2 + (box[0,1] - box[1,1])**2)
97   dist2 = math.sqrt((box[1,0] - box[2,0])**2 + (box[1,1] - box[2,1])**2)
98
99   if (dist1 == 0 or dist2 == 0):
100   continue
101
102  if (dist1/dist2 >= 8) or (dist2/dist1 >= 8):
103   allCenterx.append(x + int(w/2))
104   allCentery.append(y + int(h/2))
105
106   cv2.circle(inverted_image, (x + int(w/2), y + int(h/2)), 1, (255, 0, 0), 2)
107
108   cv2.drawContours(inverted_image, [box], -1, (0, 255, 0), 1)
109
110
111 return pixelsColorLine, offSet
```