Contents

Experiment 1: Getting to grips with OpenCV

1 Introduction

We will first download an unpack a a zip-file which contains the relevant software and test images:

mkdir vision-expt-01

cd vision-expt-01

unzip ~/Downloads/expt01.zip

2 Displaying images

We will be using the tool 'xv' to display a series of images:

*xv *.jpg*

This will display the first image and wait for user input:

- **Spacebar:** move to the next image
- **Delete:** move to the previous image

If we click the right mouse button, a dialogue box will appear that lets the user:

- choose other images
- perform simple operations such as rotating
- crop a section using the "c" key
- open a color editor with the "e" key

3 Summarizing images

The zip file for this lab contains a Python program called *summarize*. This program can be run like this:

*./summarize *.jpg*

This will then display some of the properties of the selected image.

2006-10-23-001.jpg

2006-10-23-002.jpg

2006-10-23-003.jpg

After we have run *summarize* on these images, we can see that a histogram is computed, and the results are output to a file in the same directory.

4 Plotting the histograms

We will be using *Gnuplot*, which is a general-purpose graph-plotting package. GNuplot expects the data it is to plot to be stored with an *(x, y)* pair on each line, with white space separating the values – this is the format that we have used for our values.

To run Gnuplot, we can use the command:

gnuplot

We will start by defining the annotation of the plot, and then tell Gnuplot to plot the data:

unset key

set grid

set title "Histogram of 2006-10-23-001.jpg"

set xlabel "grey level"

set ylabel "frequency"

plot "2006-10-23-001.jpg.dat"

The resulting plot will then appear as a series pf +.shaped points.

This is acceptable but a histogram is normally drawn with vertical bars, the appropriate incantation is:

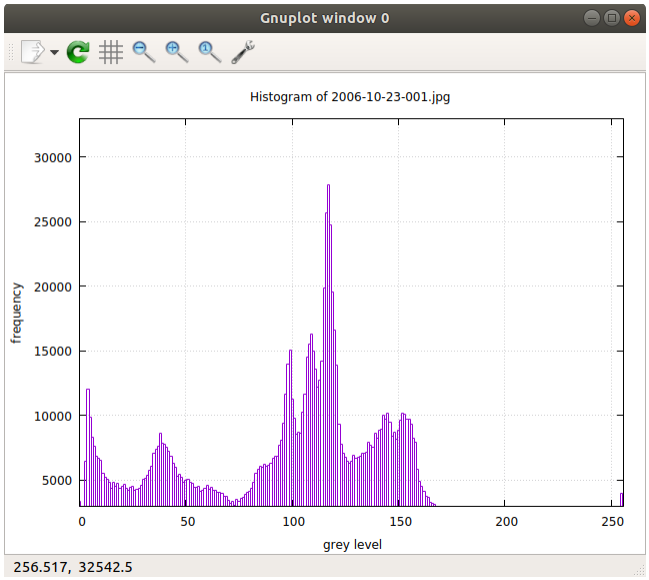
Plot "2006-10-23-001.jpg dat" with boxes

To restrict the range of the x-axis to 0–256, type

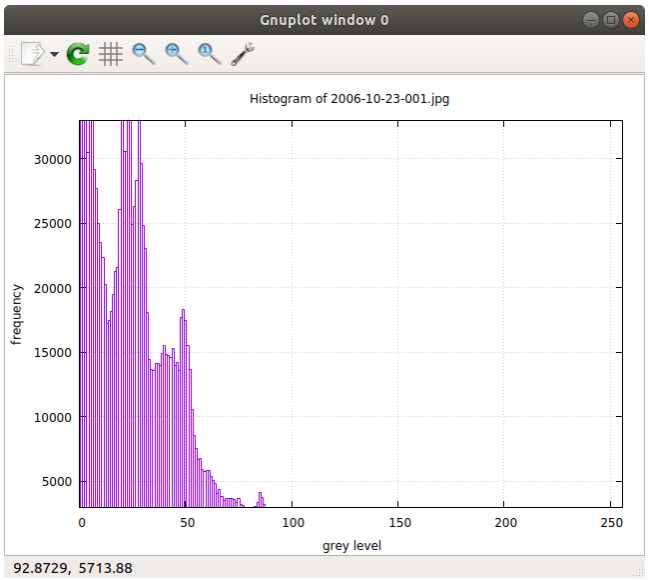
set xrange [0:256]

replot

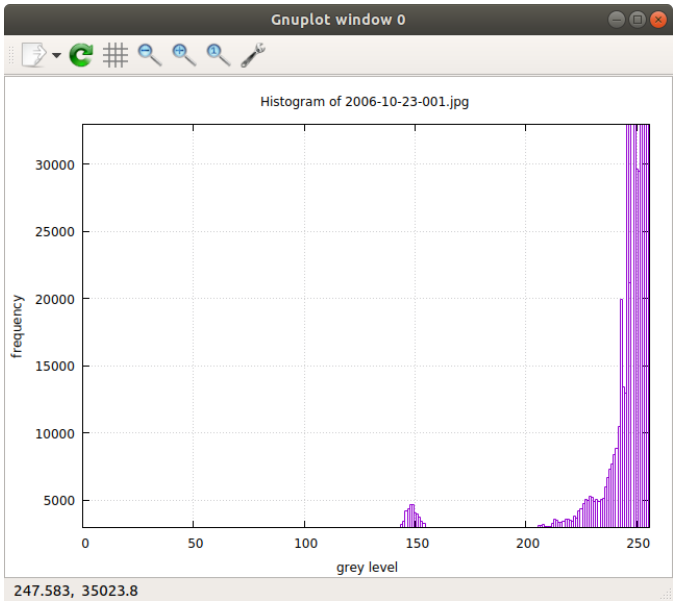
2006-10-23-001.jpg (well-exposed)



2006-10-23-001.jpg (under exposed)



2006-10-23-001.jpg (over exposed)



5 Improving the histogram routine

We will now display the image 'm51.tif' using the xv tool.

If we use the colour editor and select 'random, we can see that some structure appears.

When we run the image through *'summarize'*, we can see the maximum is: 39.00

The problem is that the image has 16 bits per pixel (typing "i" when xv is displaying the image will tell you this). We can overcome this problem by:

- setting MAXGREY to 216 = 63336 (too many bins to plot)
- divide each pixel by 256

A much better approach is to find the minimum and maximum values of the image and "contrast stretch" a value in the image into one of 256 bins:

$$256 \times \frac{v - I_l}{I_h - I_l}$$

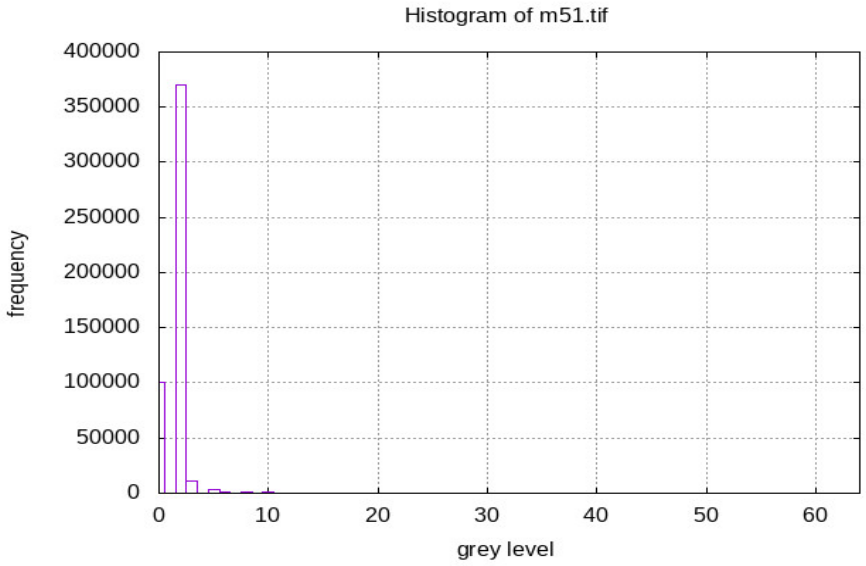
The summarize program already shows how to calculate Il and Ih;

With these, we need make only a one-line change to the histogram routine. Plot the resulting histogram, taking care to produce sensible xaxis values. In fact, 256 bins in a histogram is rather a large number; with this contrast-stretching in place, we can reduce MAXGREY to 64, which is a more useful value.

```python
def histogram (im, fn):
    "Determine the histogram of an image -- simple version."
    global MAXGREY

    # We shall fill the array hist with the histogram.
    hist = numpy.zeros (MAXGREY)

    # Get the image sizes.
    sizes = im.shape
    if len (sizes) == 2:
        # it's monochrome
        ny = sizes (0)
        nx = sizes (1)
        nc = 1
    else:
        # it has several channels
        ny, nx, nc = sizes

    # Work through the image, accumulating the histogram.
    for y in range (0, ny):
        for x in range (0, nx):
            for c in range (0, nc):
                v = int (im[y,x,c])
                hist[v] += 1

    # Output the histogram values to a file.
    with open (fn, "w") as f:
        for i in range (0, MAXGREY):
            print >>f, i, hist[i]
```

```python
def histogram (im, fn):
    "Determine the histogram of an image -- simple version."
    global MAXGREY

    # We shall fill the array hist with the histogram.
    hist = numpy.zeros ((MAXGREY, 3))

    # Get the image sizes.
    sizes = im.shape
    if len (sizes) == 2:
        # it's monochrome
        ny = sizes (0)
        nx = sizes (1)
        nc = 1
    else:
        # it has several channels
        ny, nx, nc = sizes
    min = im.min ()
    max = im.max ()
    # Work through the image, accumulating the histogram.
    for y in range (0, ny):
        for x in range (0, nx):
            for c in range (0, nc):
                v = int (im[y,x,c])
                v = int((MAXGREY-1)*((v-min)/(max-min))+0.5)
                hist[v][c] += 1

    # Output the histogram values to a file.
    with open (fn+".c0", "w") as f:
        for i in range (0,(MAXGREY-1)):
            print >>f, i, hist[i][0]
    with open (fn+".c1", "w") as f:
        for i in range (0,(MAXGREY-1)):
            print >>f, i, hist[i][1]
    with open (fn+".c2", "w") as f:
        for i in range (0,(MAXGREY-1)):
            print >>f, i, hist[i][2]
```

m51.tif (plotted a 16-bit image, very under exposed – fully black to human eye)



6 Per-channel colour histograms

The histograms we have looked at contain the red, green and blue channels of an image. However, it is more useful for us to histogram each of them separately.

We will now modify the histogram routine so that it generates separate output files for each 'channel' of an image and plot the result of processing the file sx.jpg. We can tell Gnuplot to plot multiple plots on the same axes by separating the files with commas:

plot "sx.jpg.c0" with boxes, "sx.jpg.c1" with boxes, "sx.jpg.c2" with boxes

We will return to these per-channel histograms in a later experiment.

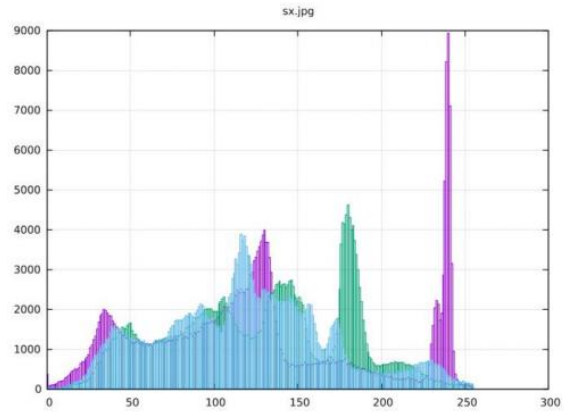
Separating red and green channels and displaying on the histogram:

```
def histogram (im, fn):
    "Determine the histogram of an image -- simple version."
    global MAXGREY

    # We shall fill the array hist with the histogram.
    hist = numpy.zeros (MAXGREY)

    # Get the image sizes.
    sizes = im.shape
    if len (sizes) == 2:
        # it's monochrome
        ny = sizes (0)
        nx = sizes (1)
        nc = 1
    else:
        # it has several channels
        ny, nx, nc = sizes

    # Work through the image, accumulating the histogram.
    for y in range (0, ny):
        for x in range (0, nx):
            for c in range (0, nc):
                v = int (im[y,x,c])
                hist[v] += 1

    # Output the histogram values to a file.
    with open (fn, "w") as f:
        for i in range (0, MAXGREY):
            print >>f, i, hist[i]
```

```
def histogram (im, fn):
    "Determine the histogram of an image -- simple version."
    global MAXGREY

    # We shall fill the array hist with the histogram.
    hist = numpy.zeros ((MAXGREY, 3))

    # Get the image sizes.
    sizes = im.shape
    if len (sizes) == 2:
        # it's monochrome
        ny = sizes (0)
        nx = sizes (1)
        nc = 1
    else:
        # it has several channels
        ny, nx, nc = sizes
    min = im.min ()
    max = im.max ()
    # Work through the image, accumulating the histogram.
    for y in range (0, ny):
        for x in range (0, nx):
            for c in range (0, nc):
                v = int (im[y,x,c])
                v = int ((MAXGREY-1)*((v-min)/(max-min))+0.5)
                hist[v][c] += 1

    # Output the histogram values to a file.
    with open (fn+".c0", "w") as f:
        for i in range (0,(MAXGREY-1)):
            print >>f, i, hist[i][0]
    with open (fn+".c1", "w") as f:
        for i in range (0,(MAXGREY-1)):
            print >>f, i, hist[i][1]
    with open (fn+".c2", "w") as f:
        for i in range (0,(MAXGREY-1)):
            print >>f, i, hist[i][2]
```

This code will display 3 channels after entering the command:

plot "sx.jpg.c0" with boxes, "sx.jpg.c1" with boxes, "sx.jpg.c2" with boxes



Experiment 2: Content-Based Image Retrieval

1 Introduction

In this lab we will be using a technique called content-based image retrieval (CBIR).

Instead of running a single CBIR program on a dataset and obtaining a few numbers, we will be using a 'benchmark' or 'strawman' program, and then in turn attempt to improve it, and compare whether the improvement actually leads to better performance

2 The strawman CBIR software

The CBIR program in the experiment's zip-file calculates the histogram of each image. The CBIR is not a particularly good algorithm because it does not distinguish colours, so an image with many bright blue pixels can be confused with one with many bright green ones.

Included in the zip file for this lab is a set of images and a test harness, a program that runs a series of tests and keeps track of the number of successes and failures.

We will firstly assess the performance of the CBIR program, ensure it is executable:

chmod +x cbir

Then we will run FACT, the test harness, by issuing the following commands:

chmod +x fact

./fact --interface=cbir execute fruit

Note: The file fruit.fact contains the tests that are to be executed. The execute on the second line tells FACT to run CBIR on the test script fruit.fact and output a "transcript".

When we execute the above command, FACT will write output to the terminal window.

Note: The first line contains some identification information, used for checking in the analysis stages, followed by a single line per test.

To save this transcript in a file, we can issue the command:

./fact --interface=cbir execute fruit > cbir.res

Having generated the transcript, the next stage is to analyze it:

./fact analyse cbir.res

Note: FACT will output more detail by appending *-detail=2* to the command.

Terminal output after running command "*./fact analyse cbir.res*"

```
result ass1-073 tomato S tomato
result ass1-074 tomato S tomato
result ass1-075 tomato S tomato
result ass1-076 tomato S tomato
result ass1-077 tomato S chili
result ass1-078 tomato S tomato
result ass1-079 tomato S tomato
result ass1-080 tomato S tomato
transcript_end 0.048721
al17846@cseeproj05:~/expt02/expt02$ ./fact --interface=cbir execute fruit > cbir.res
al17846@cseeproj05:~/expt02/expt02$ ./fact analyse cbir.res
Error rates calculated from cbir.res
#  tests      TP      TN      FP      FN accuracy   recall precision specificity class
      10      10       0       0       0     1.00     1.00      1.00        0.00 banana
      10       7       0       3       0     0.70     1.00      0.70        0.00 chili
      10       6       0       4       0     0.60     1.00      0.60        0.00 gapple
      10      10       0       0       0     1.00     1.00      1.00        0.00 gfruit
      10       8       0       2       0     0.80     1.00      0.80        0.00 orange
      10       9       0       1       0     0.90     1.00      0.90        0.00 pear
      10       5       0       5       0     0.50     1.00      0.50        0.00 rapple
      10       9       0       1       0     0.90     1.00      0.90        0.00 tomato
      80      64       0      16       0     0.80     1.00      0.80        0.00 overall

Confusion matrix calculated from cbir.res
                                expected
   actual   banana    chili   gapple   gfruit   orange     pear   rapple   tomato
   banana       10        0        0        0        0        0        0        0
    chili        0        7        0        0        0        0        0        1
   gapple        0        0        6        0        0        1        4        0
   gfruit        0        0        0       10        2        0        0        0
   orange        0        0        0        0        8        0        0        0
     pear        0        0        0        0        0        9        0        0
   rapple        0        0        3        0        0        0        5        0
   tomato        0        3        1        0        0        0        1        9

al17846@cseeproj05:~/expt02/expt02$
```

3 Refining the CBIR software

The histogram that CBIR calculates is not sophisticated enough to perform the task effectively. We will now improve the algorithm and ascertain whether it actually delivers better performance, we will make a copy of cbir into cbir3 (for three histograms).

The new cbir3 program will now need to:

- calculate separate histograms for red, green and blue;
- join together these three histograms end-to-end into one long one

We will now through the same stages as before but with a different interface file:

chmod +x cbir3

./fact --interface=cbir3 execute fruit > cbir3.res

./fact analyse cbir3.res

We can also add an additional step, comparing the two transcripts:

./fact compare cbir.res cbir3.res

Note: FACT will output more detail by appending -detail=2 to the command.

cbir3 program

```
def hist (im):
    "Return the grey-level histogram of an image."
    global NBINS

    # Determine the data range.
    lo = im.min ()
    hi = im.max ()

    # Compute the histogram and return it.
    h, bins = numpy.histogram (im.ravel(), NBINS, [lo, hi])
    return h
```

```
def hist (im):
    "Return the grey-level histogram of an image."
    global NBINS

    # Determine the data range.
    lo = im.min ()
    hi = im.max ()

    # Compute the histograms for each chanel B G R, then hstack (combine) and return.
    hB, binsB = numpy.histogram (im[:,:,0].ravel(), NBINS, [lo, hi])
    hG, binsG = numpy.histogram (im[:,:,1].ravel(), NBINS, [lo, hi])
    hR, binsR = numpy.histogram (im[:,:,2].ravel(), NBINS, [lo, hi])
    h = numpy.hstack((hB,hG,hR))
    return h
```

./fact analyse cbir3.res

```
Terminal
File  Edit  View  Search  Terminal  Help
al17846@cseeproj05:~/expt02/expt02$ ./fact --interface=cbir3 execute fruit > cbir3.res
al17846@cseeproj05:~/expt02/expt02$ ./fact analyse cbir3.res
Error rates calculated from cbir3.res
#  tests      TP      TN      FP      FN accuracy   recall precision specificity class
      10      10       0       0       0     1.00     1.00      1.00        0.00 banana
      10       7       0       3       0     0.70     1.00      0.70        0.00 chili
      10       6       0       4       0     0.60     1.00      0.60        0.00 gapple
      10      10       0       0       0     1.00     1.00      1.00        0.00 gfruit
      10       8       0       2       0     0.80     1.00      0.80        0.00 orange
      10       9       0       1       0     0.90     1.00      0.90        0.00 pear
      10       5       0       5       0     0.50     1.00      0.50        0.00 rapple
      10       9       0       1       0     0.90     1.00      0.90        0.00 tomato
      80      64       0      16       0     0.80     1.00      0.80        0.00 overall

Confusion matrix calculated from cbir3.res
                                expected
  actual   banana    chili   gapple   gfruit   orange     pear   rapple   tomato
  banana       10        0        0        0        0        0        0        0
   chili        0        7        0        0        0        0        0        1
  gapple        0        0        6        0        0        1        4        0
  gfruit        0        0        0       10        2        0        0        0
  orange        0        0        0        0        8        0        0        0
    pear        0        0        1        0        0        9        0        0
  rapple        0        0        3        0        0        0        5        0
  tomato        0        3        0        0        0        0        1        9

al17846@cseeproj05:~/expt02/expt02$
```

./fact compare cbir.res cbir3.res

```
Terminal
File  Edit  View  Search  Terminal  Help
      80      64       0      16       0     0.80     1.00      0.80        0.00 overall

Confusion matrix calculated from cbir3.res
                                expected
  actual   banana    chili   gapple   gfruit   orange     pear   rapple   tomato
  banana       10        0        0        0        0        0        0        0
   chili        0        7        0        0        0        0        0        1
  gapple        0        0        6        0        0        1        4        0
  gfruit        0        0        0       10        2        0        0        0
  orange        0        0        0        0        8        0        0        0
    pear        0        0        1        0        0        9        0        0
  rapple        0        0        3        0        0        0        5        0
  tomato        0        3        0        0        0        0        1        9

al17846@cseeproj05:~/expt02/expt02$ ./fact compare cbir.res cbir3.res
Comparison of cbir.res and cbir3.res
Z-score  class
   0.00  banana
   0.00  chili
   0.00  gapple
   0.00  gfruit
   0.00  orange
   0.00  pear
   0.00  rapple
   0.00  tomato

al17846@cseeproj05:~/expt02/expt02$ ./fact compare cbir.res cbir3.res -detail=2
```

4 Segmenting out the object

The fact that the fruit differ in size means that the number of background pixels will vary.

It is possible to overcome this by segmenting out the object from the background.

We will first use xv to look at several of the images. When we press the middle mouse button, it displays on the image the RGB and HSV values of the pixel under the cursor.

We will then make a copy of your cbir3 in cbir3s. This program will convert the image to HSV and use the hue limits that we have identified to identify background pixels.

We must also check that the HSV values of a pixel from xv are consistent with those computed by OpenCV, which we can do by printing out the HSV values of the top-left pixel of the image. Then make cbir3s convert all of the background pixels to white

We will calculate the histograms of these images with the whitened background; but when we combine the histograms, we will omit the very highest one (white).

cbir3s - converts image to HSV, sets lower and upper boundary, generates a mask for the background, applies it to the coloured image and returns that final image.

```
def backgroundToWhite(im):
    hsv = cv2.cvtColor(im, cv2.COLOR_BGR2HSV)
    lowerHSV = numpy.array([10,0,0])
    upperHSV = numpy.array([150,255,255])
    mask = cv2.inRange(hsv, lowerHSV, upperHSV)
    res = cv2.bitwise_and(im,im, mask= mask)
    coloured = res.copy()
    coloured[mask == 0] = 255

    #cv2.imshow ('c', coloured)
    #cv2.waitKey ()
    return coloured
```

cbir3s - definition is called when the image is processed.

```
# Read in the probe image and find its histogram.
im = cv2.imread (probe_file)
im = backgroundToWhite (im)
probe = hist (im)

# We now enter the main loop.  The basic idea is to load an image, find its
# histogram, then compare that with the histogram of the probe image.  We are
# careful to skip the case when the test image is the same as the probe.
for file in sys.argv[2:]:
    if file != probe_file:
        im = cv2.imread (file)
    im = backgroundToWhite (im)
        h = hist (im)
        v = compare (probe, h)
        if v > v_best:
            v_best = v
            f_best = file
```

./fact analyse cbir3.res

```
Terminal
File  Edit  View  Search  Terminal  Help
as16815@cseelab413:~/YEAR 3/ce316/lab2/expt02$ chmod +x cbir3
as16815@cseelab413:~/YEAR 3/ce316/lab2/expt02$ ./fact --interface=cbir3 execute fruit > cbir3s.res
as16815@cseelab413:~/YEAR 3/ce316/lab2/expt02$ ./fact analyse cbir3s.res
Error rates calculated from cbir3s.res
#  tests      TP      TN      FP      FN accuracy   recall precision specificity class
      10      10       0       0       0     1.00     1.00      1.00        0.00 banana
      10      10       0       0       0     1.00     1.00      1.00        0.00 chili
      10      10       0       0       0     1.00     1.00      1.00        0.00 gapple
      10      10       0       0       0     1.00     1.00      1.00        0.00 gfruit
      10      10       0       0       0     1.00     1.00      1.00        0.00 orange
      10      10       0       0       0     1.00     1.00      1.00        0.00 pear
      10       7       0       3       0     0.70     1.00      0.70        0.00 rapple
      10      10       0       0       0     1.00     1.00      1.00        0.00 tomato
      80      77       0       3       0     0.96     1.00      0.96        0.00 overall

Confusion matrix calculated from cbir3s.res
                                expected
  actual   banana    chili   gapple   gfruit   orange     pear   rapple   tomato
  banana       10        0        0        0        0        0        2        0
   chili        0       10        0        0        0        0        1        0
  gapple        0        0       10        0        0        0        0        0
  gfruit        0        0        0       10        0        0        0        0
  orange        0        0        0        0       10        0        0        0
    pear        0        0        0        0        0       10        0        0
  rapple        0        0        0        0        0        0        7        0
  tomato        0        0        0        0        0        0        0       10

as16815@cseelab413:~/YEAR 3/ce316/lab2/expt02$
```

Note: The results are almost perfect, however the rapple is not recognised completely.

Experiment 3: Thresholding and contours

1 Introduction

This experiment is based on a tutorial on contours using OpenCV, which touches on thresholding and then its contour functionality. Contours are more powerful than region labelling; but the code to implement it is significantly more complicated.

The aim of this experiment is to isolate each of the visible die faces in the example image for this experiment by thresholding, and then count the number of dots on each die's face.

2 The Program

The starting point to the assignment is the program:

```
#!/usr/bin/env python
"contours -- demo of OpenCV's contour-processing capabilities"
from __future__ import division
import sys, cv2

# Handle the command line.
if len (sys.argv) < 3:
    print >>sys.stderr, "Usage:", sys.argv[0], "<image> <threshold>"
    sys.exit (1)
img = cv2.imread (sys.argv[1])
t = int (sys.argv[2])

# Produce a binary image.
gray = cv2.cvtColor (img, cv2.COLOR_BGR2GRAY)
blur = cv2.GaussianBlur (gray, (5, 5), 0)
(t, binary) = cv2.threshold (blur, t, 255, cv2.THRESH_BINARY)

# Find contours.
(_, contours, _) = cv2.findContours (binary, cv2.RETR_EXTERNAL,
    cv2.CHAIN_APPROX_SIMPLE)

# Print a table of the contours and their sizes.
print "Found %d objects." % len(contours)
for (i, c) in enumerate(contours):
    print "\tSize of contour %d: %d" % (i, len(c))

# Draw contours over original image and display the result.
cv2.drawContours (img, contours, -1, (0, 0, 255), 5)
cv2.namedWindow (sys.argv[0], cv2.WINDOW_NORMAL)
ny, nx, nc = img.shape
cv2.resizeWindow (sys.argv[0], nx//2, ny//2)
cv2.imshow (sys.argv[0], img)
cv2.waitKey (0)
```

Like most programs that work purely by calling OpenCV routines, it is fairly short.

It works by reading in an image, and blurring it slightly with a Gaussian-shaped mask.

It then thresholds the image using an OpenCV function.

The resulting binary image is passed to OpenCV's contour-finding function, which returns a list of the contours found. These are printed out and then drawn onto the image for display.

We can run the program on the supplied image and see how well it works. We can invoke it with two arguments, the name of the file containing the image to be processed and the value of the threshold (for example with a value of 200):

python contours 08-dice.jpg 200 200 determines the threshold

3 Thresholding

The routine cv2.threshold accepts several arguments. One characteristic of the image is that its background it not constant but appears lighter towards the upper right corner. This is not a problem in this case because the upper surfaces of the dice appear so white; but in many tasks, this variation in the background may become problematic.

OpenCV provides an adaptive thresholder, where the threshold relates to regions of the image rather than being global. This routine is used in the following way:

binary = cv.adaptiveThreshold (blur, 255, cv.ADAPTIVE_THRESH_MEAN_C,

cv.THRESH_BINARY, 11, 2)

Note: the last two arguments can be tuned to make the routine perform well, the value of the threshold has to be provided by the user on the command line.

Lectures described the use of Otsu's approach, which works well when the image is bimodal, you can use Otsu's method with cv2.threshold too, using a call like:

thresh, binary = cv.threshold (blur, 0, 255,

cv.THRESH_BINARY + cv.THRESH_OTSU)

Note: This has an additional option in the last argument and sets the threshold argument to zero; these tell cv2.threshold to compute the threshold value using Otsu's method. The first argument returned from the call, stored in thresh here, is the threshold value computed.

```
(t, binary) = cv2.threshold (blur, t, 255, cv2.THRESH_BINARY)
```

```
# Adaptive Thresholding
binary = cv2.adaptiveThreshold (blur, 255, cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 11, 2)
```

```
#Otsu's method
(t, binary) = cv2.threshold (blur, t, 255,cv2.THRESH_BINARY + cv2.THRESH_OTSU)
```

Note: The ideal threshold that returns the best results is the first one provided in the original code (the first image). It distinguishes the dots from the die most effectively

4 Processing contours

We will now try to find and process the contours in the image. The call to **cv2.findContours** passes in three arguments and receives three outputs, two of which are currently ignored:

- The first parameter in the call relates to the image in which the contours are to be found. This image should be binary, with the objects for which contours are to be found in white and a black background.
- The second argument is a constant indicating what kind of contours are to be found. As we are currently interested in the main objects in the image, we want only contours around the outermost edges of objects and so we pass in **cv2.RETR_EXTERNAL**.
- The last parameter tells cv2.findContours whether or not it should simplify the contours; we use **cv2.CHAIN_APPROX_SIMPLE**, which tells it to simplify by using line segments when it can — that saves memory and computation time.

Note: If we required more information, such as the contours of the dots marked on a die's face, then we would use another parameter such as **cv2.RETR_TREE** or **cv2.RETR_CCOMP**.

cv2.findContours returns a tuple of three values:

- The first is an intermediate image produced during the contour-finding process; we are not interested in in it here so we store it in a junk variable.
- The second return value is a list of numpy arrays, each holding the points for a single contour in the image.
- The final return value is a numpy array that contains hierarchy information about the contours, again not of interest to us here.

Note:

finding and processing contours in the image

```
# Find contours (EXTERIOR ONLY).
#(_, contours, _) = cv2.findContours (binary, cv2.RETR_EXTERNAL,
#    cv2.CHAIN_APPROX_SIMPLE)

# Interior also
(junk, contours, hierarchy) = cv2.findContours (binary, cv2.RETR_TREE,
    cv2.CHAIN_APPROX_SIMPLE)
```

- **RETR_EXTERNAL:** only selects contours around the outermost edges of objects.
- **RETRE_TREE**: allows for information such as contours of dots marked on a die's face.

5 Counting the dots

The program initially finds only the external contours of objects. If we change the third argument from **cv2.RETR_EXTERNAL** to **cv2.RETR_TREE**, it will also return internal contours, and we can use this to count the number of dots on each die's face.

When using **cv2.RETR_TREE**, the contours are arranged in a hierarchy, with the outermost contours for each object at the top. Moving down the hierarchy, each new level of contours represents the next innermost contour for each object

We obtain that information about the contour hierarchies via the third return value from the **cv2.findContours** call:

(junk, contours, hierarchy) = cv2.findContours(binary, cv2.RETR_TREE,

cv2.CHAIN_APPROX_SIMPLE)

The third return value, saved in hierarchy here, is a three-dimensional numpy array, with one row, 36 columns, and a "depth" of 4 from the test image. The 36 columns correspond to the contours found by the method.

Note: There are 36 contours rather than seven because **cv2.RETR_TREE** tells the routine to return internal contours as well as external ones. Column zero corresponds to the first contour, column one the second, and so on.

Each of the columns has a four-element array of integers, representing indices of other contours, according to the scheme:

[next, previous, firstChild, parent]

The next index refers to the next contour in this contour's hierarchy level, while the previous index refers to the previous contour in this contour's hierarchy level.

The **firstChild** index refers to the first contour that is contained inside this contour. The parent index refers to the contour containing this contour. In all cases, a value of −1 indicates that there is no linked contour.

Note: To count the dots a threshold of 190 is used.

- **R:** top of hierarchy
- **G:** next outermost contours (dots)
- **B:** innermost contours, (lost paint in one of the dots in central die)

Output image:

Full code:

```
#!/usr/bin/env python
"contours -- demo of OpenCV's contour-processing capabilities"
from __future__ import division
import sys, cv2

# Handle the command line.
if len (sys.argv) < 3:
    print >>sys.stderr, "Usage:", sys.argv[0], "<image> <threshold>"
    sys.exit (1)
img = cv2.imread (sys.argv[1])
t = int (sys.argv[2])

# Produce a binary image.
gray = cv2.cvtColor (img, cv2.COLOR_BGR2GRAY)
blur = cv2.GaussianBlur (gray, (5, 5), 0)
(t, binary) = cv2.threshold (blur, t, 255, cv2.THRESH_BINARY)

# Adaptive Thresholding
#binary = cv2.adaptiveThreshold (blur, 255, cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 11, 2)

#Otsu's method
#(t, binary) = cv2.threshold (blur, t, 255,cv2.THRESH_BINARY + cv2.THRESH_OTSU)
# Viewing processed images for testing

#cv2.imshow ('binary', binary)
#cv2.waitKey ()

# Find contours (EXTERIOR ONLY).
#(_, contours, _) = cv2.findContours (binary, cv2.RETR_EXTERNAL,
#    cv2.CHAIN_APPROX_SIMPLE)

# Interior also
(junk, contours, hierarchy) = cv2.findContours (binary, cv2.RETR_TREE,
    cv2.CHAIN_APPROX_SIMPLE)

# Count the number of dots on the dice faces.  We do this by iterating over
# hierarchy[0], first to find the indices of the dice contours, then again
# to find the dot contours.
dice = []   # list of dice contours
dots = []   # list of dot contours

# Find the dice contours.
for (i, c) in enumerate (hierarchy[0]):
    if c[3] == -1:
        dice.append (i)

# Now find the dot contours, and output how 190 many we find.
for (i, c) in enumerate (hierarchy[0]):
    if c[3] in dice:
        dots.append (i)
print "Total number of dots:", len (dots)

# Print a table of the contours and their sizes.
print "Found %d objects." % len(contours)
for (i, c) in enumerate(contours):
    print "\tSize of contour %d: %d" % (i, len(c))
```

```
 # Print array
for (i, c) in enumerate (hierarchy[0]):
    print(c)

 # Draw contours for external in RED
for (i, c) in enumerate (hierarchy[0]):
    if c[3] != -1:
        cv2.drawContours (img, contours, c[3], (0, 0, 255), 7)


 # Draw contours for dots in GREEN
for (i, c) in enumerate (hierarchy[0]):
    if c[0] > -1 and c[3] != -1:
        cv2.drawContours (img, contours, c[0], (0, 255, 0), 5)

 # Draw innermost contours in BLUE
for (i, c) in enumerate (hierarchy[0]):
    if c[2] != -1:
        cv2.drawContours (img, contours, c[2], (255, 0, 0), 5)

cv2.namedWindow (sys.argv[0], cv2.WINDOW_NORMAL)
ny, nx, nc = img.shape
cv2.resizeWindow (sys.argv[0], nx//2, ny//2)
cv2.imshow (sys.argv[0], img)
cv2.waitKey (0)
```

Experiment 4: Faces

1 Introduction

To do this experiment, we will need to download the accompanying zipfile.

2 Viola-Jones

The zip-file contains a program, face-detect, which uses the OpenCV implementation of Viola's and Jones's algorithm to locate faces.

We can run it on a test image by typing:

```
./face-detect -d ese-small.jpg
```

Note: We need to add the –eve qualifier for small displays.

We can also run the program to locate eyes within faces:

```
./face-detect -h
```

3 Face recognition

The zip-file also contains a program that uses the Eigenfaces algorithm described in lectures to perform face recognition. This program has two phases, training and testing. To train, you run it with a set of images which it uses to create its "knowledge base":

```
./eigenrec train t6.kb train/s*/*
```

In the train directory tree, there are six examples of each face.

When trained, you can test it using the command:

```
./eigenrec test t6.kb test/s*/*
```

For each test image, the program reports the nearest training image and a score. From the directory names, we can figure out whether the result was correct or not.

We will now delete one of the training images for each subject:

```
rm train/s*/*6.pgm
```

And then repeat the tests with the remaining images:

```
./eigenrec train t5.kb train/s*/*
```

```
./eigenrec test t5.kb test/s*/*
```

4 Putting the parts together

For the last part of this experiment, we will merge the functionalities of the two programs.

The **eigenrec** program now needs to use Viola-Jones to cut out the face from each training image as it goes, and then cycle through each face found in each test image, comparing it in turn with its knowledge base.

We have to convert an image from the floating-point representation used by EVE to the unsigned byte one required for the Viola-Jones subsystem of OpenCV. The relevant conversion is done by a line such as:

```
bim = im.astype (dtype=numpy.uint8)
```

Note: im is the image read in by EVE and bim is the unsigned byte equivalent needed for OpenCV routine detectMultiScale.

insert screenshot of code here

Experiment 5: Stereo

1 Introduction

For this lab we will be using the following equation:

$$Z = \frac{f\ B}{D}$$

- Z - the distance to the object
- f - the focal length
- B - the baseline and
- D - the disparity (parallax) of a feature between the left and right images.

2 Estimating the focal length

The virtual cameras are at(x, y, z) locations (±75, 300, 800), which means that B = 150 mm. Vertex 5 of Candide, the tip of her nose, is at location (0, 289, 280); hence, the distance from the cameras to it, Z, is approximately 800 − 280 = 520 mm.

We will now measure the locations of that point in the images from the left and right cameras: use xv to display each image and press the middle mouse button on the tip of her nose to read the position. Estimate your accuracy in reading the position and, using all this information, work out using a calculator the focal length and its uncertainty (“error”) as described in the lecture notes.

3 Camera calibration to determine the focal length

The calculation above gives a rough estimate of the camera’s focal length. We will now use camera calibration to get a more accurate result there are routines in OpenCV to do this.

A series of images has been rendered using POV-ray with the same cameras as for Candide, but with a model of a calibration target, these are included the zip-file along with the program that we need to use for this experiment. We can run this program as follows:

python calibrate.py

It should output the computed focal length and an estimate of its error (uncertainty).

4 Distance calculation from calibration data

Using the focal length output by the calibration program, we will now calculate the distance to the tip of Candide's nose and its uncertainty.

Experiment 6: Evaluation and Machine Learning

1 Introduction

In this experiment, we will train up a number of vision techniques and evaluate the trained systems on the popular MNIST example, which comprises 60,000 training images and 10,000 test ones. To carry out this experiment, we need a zip-file of the software.

Note: This will not allow us to carry out all the training ourselves as some of it will take too long on the machines in CSEE's Software Labs.

2 Compiling and running WISARD

WISARD is a simple pattern recognition scheme which is described in detail in the lecture notes on machine learning. We will be using an implementation of WISARD which was originally devised for the EPSRC/BMVA Summer School on Computer Vision.

WISARD is written in C and stored in two files, wisard.c and rng.c, the former containing all the important code and the latter simply random number generators. We can compile the program on a Unix system by issuing the command:

```
gcc -o wisard wisard.c rng.c –lm
```

This will create an executable program called wisard. The program is used in two ways…

Training: To train a network you execute the program as follows:

```
./wisard train wiz1.net mnist/train/1-*.pgm
```

Note: train is a keyword that tells the program to run in training mode, wiz1.net is the file in which the trained network is saved, and the files on the remainder of the command line are used for training

Testing: To test a network you execute the program as follows:

```
./wisard test wiz1.net mnist/test/1-*.pgm
```

Note: test is a keyword that tells the program to run in testing mode, wiz1.net is the file from which a trained network is loaded, and the files on the remainder of the command line are used for testing.

A single WISARD network is able to recognize only one pattern. To be able to distinguish several patterns, it is necessary first to train a network for each of them; this is straightforward given the command above.

Each test pattern must then be run through all the trained networks and the one that yields gives the largest score ascertained. If no network yields a higher score than all the others, the test fails. This functionality is encapsulated in wisard-if.py, which interfaces between the wizard executable program and the FACT test harness that was used in an earlier lab.

To run all 10,000 test digits through the trained networks, we can issue the command:

./fact --interface=wisard-if run digits > wisard.res

We can also run the following command:

./fact analyse wisard.res

This will produce a table of error rates, and a class confusion matrix.

3 Using eigenrec

We will be using the **eigenrec** program that we had used in an earlier experiment on faces.

The zip-file includes a modified version of **eigenrec**: it is modified because the normal way of passing in training images on the command line exceeds the maximum allowable command line length, so the program has been hacked to read in the images explicitly.

Note: A transcript file from the program is provided in eigenrec.res

We can now train the program by issuing the command:

./eigenrec train mnist.kb x *(where the last argument is ignored)*

And then run it with the command:

./eigenrec test mnist.kb mnist/test/1-0001.pgm

And then run the FACT test script:

./fact --interface=eigenrec-if run digits.fact > eigenrec.res

Note: This command will crash the computer (takes 120 minutes).

4 Using a Support Vector Machine

The support vector machine (SVM) is another approach that's available for us to use. The standard implementation of the SVM is libSVM, and the easiest way of using it is via Python's Scikit-learn package, which is already installed on our machines. A program that uses this is included in the zip-file as svm-mnist.py; which can be run by typing:

python svm-mnist.py

It takes about 5 minutes to train and test the SVM, furthermore the FACT transcript has also been included in the zip-file for this lab.

Note: The program downloads the MNIST database onto your machine when it is first run, putting it into your directory: ~/scikit_learn_data.

This directory is over 50 Mbytes in size, we will delete the directory after we have finished this experiment. On subsequent runs, the program will use this cached copy of the data.

5 Using a Multi-Layer Perceptron

A multi-layer perceptron (MLP) is a 'traditional' artificial neural network that we will be using for this experiment. A simple network is implemented in the program mlp-mnist.py and we can run it in a similar way to the SVM:

python mlp-mnist.py

This runs in a matter of seconds on the author's laptop, and the transcript is in the zip-file.

6 Comparing the performances of algorithms

A good test harness should allow one to compare the performances of algorithms in a statistically-valid way. FACT does this by using McNemar's test and to use it, we can run it in compare mode with a pair of transcript files:

./fact compare wisard.res warlock.res

Note: warlock.res contains the results from a fictitious algorithm.

7 Comparing more than two algorithms

“As described above, all you have to do is compare each algorithm with every other algorithm using FACT in compare mode and identify which are significant. Well, yes... and no. You do perform this pairwise comparison but you need to adapt the critical value for significance of 1.96 which is described in the lecture notes, and the reason for this is quite subtle.

When you choose a pair of algorithms to compare, you are choosing from an ensemble (to use the correct statistical term) of all possible algorithms. If you keep doing this many times, you will eventually choose a pair of algorithms for which there appears to be a significant difference in performance just because of the arrangement of the data. Remember, the critical value of 1.96 given in the lecture notes corresponds to an expectation that the data will make one algorithm appear to be better than another, simply as a consequence of the data used, one time on twenty — so if you perform twenty pairwise comparisons, one of them might be expected to appear significant simple because of the data and not because of a genuine performance difference. (Confusing, isn’t it? Do talk to a demonstrator about this.)

What this means is that we need to increase the critical value that indicates significance so that a larger Z is needed from McNemar’s test. The most widely-used such correction is the Bonferroni correction and comes down to multiplying the critical value by the number of algorithms being tested. You need to do this when interpreting the result from factcompare.

When you have done this, you are in a position to judge which is the best algorithm to use on MNIST from all those considered in this experiment.”

Color Spaces in OpenCV | Python

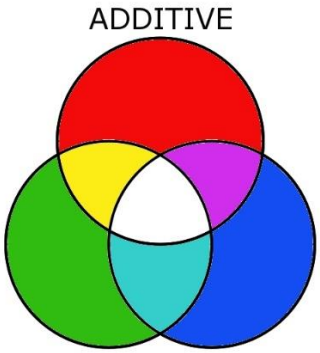
Color spaces are a way to represent the color channels present in the image that gives the image that particular hue. There are several different color spaces and each has its own significance.

Some of the popular color spaces are

- RGB (Red, Green, Blue)
- CMYK (Cyan, Magenta, Yellow, Black)
- HSV (Hue, Saturation, Value), etc.

BGR color space

OpenCV's default color space is RGB. However, it actually stores color in the BGR format. It is an additive color model where the different intensities of Blue, Green and Red give different shades of color.



Complementary (basically opposite)

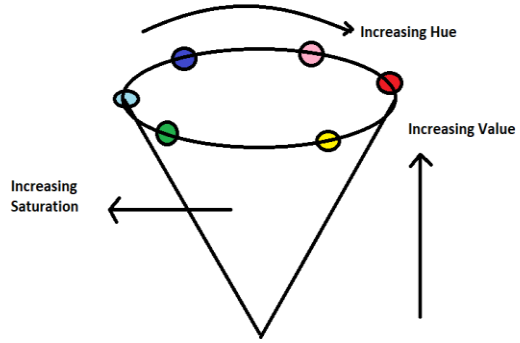
Greens complementary is magenta
Blues complementary is yellow etc.

Mixes:

HSV color space

It stores color information in a cylindrical representation of RGB color points. It attempts to depict the colors as perceived by the human eye.

- Hue value varies from 0-179
- Saturation value varies from 0-255
- Value value varies from 0-255. It is mostly used for color segmentation purpose.



CMYK color space

Unlike, RGB it is a subtractive color space. The CMYK model works by partially or entirely masking colors on a lighter, usually white, background. The ink reduces the light that would otherwise be reflected. Such a model is called subtractive because inks “subtract” the colors red, green and blue from white light.

- White light minus red leaves cyan
- white light minus green leaves magenta
- and white light minus blue leaves yellow.