1. COLOR REMAP AND IMAGE BLENDING

Goal was to try reduce impact of shadows and enhancing pear fruits, to create images that could perform better in subsequent methods (edge and contour detection)

|  |  |  |
| --- | --- | --- |
| USE | FUNCTION | NOTE |
| Inverse, negative image | negative = abs(255-img) |  |
| Remap in L\*a\*b color space | lab = cv.cvtColor(img, cv.COLOR\_BGR2LAB) | Used since allows to overcome various lighting condition problems |
| Addition of 2 images | addition = cv.add(img1, img2) |  |
| Blending of 2 images | blended = cv.addWeighted(img1, 0.7, img2, 0.9, 0.0) | Different weights to control for transparency for each image (0.7, 0.9), 0 is gamma so a static weight that will be added to all the pixels of the image |

 blending of original image and its binary

blending of original image (0.9) and a white image (0.7)

 image remapped in L\*a\*b color space

1. CONTOUR DETECTION

Contour are defined as abstract collections of points and segments corresponding to the shapes of the objects in the image.

As a preprocessing step to contour detection, the image is converted in gray scale. In addition, adjustment of contrast and brightness and blurring of the image could be applied with the following methods:

* gray = cv.cvtColor(img, cv.COLOR\_RGB2GRAY)
* gray = cv.blur(gray, (3,3))🡪 parameters: source image, tuple representing the blurring kernel size
* image = cv.bilateralFilter(gray, 20, 60, 60) 🡪 parameters:
  + source image
  + diameter of each pixel neighborhood
  + sigma color: value of sigma in the color space; the greater the value, the colors farther to each other will start to get mixed
  + sigma space: value of sigma in the coordinate space; the greater its value, the more further pixels will mix together, given that their colors lie within the sigma color range.
* adjusted = cv.convertScaleAbs(img, alpha=1, beta=20)🡪 parameters:
  + source image
  + contrast value: to lower the contrast, use 0 < alpha < 1. And for higher contrast use alpha > 1.
  + brightness value: a good range for brightness value is [-127, 127]

While convolution such blur, median blur often result in a loss of important edge information, since they blur out everything, irrespective of it being noise or an edge, a bilateral filter is used for smoothening images and reducing noise, while **preserving edges**. It is a non-linear filter and helps reduce noise: at each pixel, its value is substituted by the average of the neighborhood pixels.

Examples images of, respectively: blurred image, bilateral filter on image, both blur and bilateral filter

To successfully detect contours in an image, OpenCv requires to convert the image to a binary image (which should be a result of a thresholded image – see example images at the end of the section- or [edge detection](https://www.thepythoncode.com/article/canny-edge-detection-opencv-python)). The code to do so could be one of the following:

* \_, binary = cv.threshold(gray, 240, 255, cv.THRESH\_BINARY) 🡪 threshold is 240: if less, pixel set to 0 otherwise set to 255
* retVal, binary = cv.threshold(gray,0,255,cv.THRESH\_BINARY+cv.THRESH\_OTSU) 🡪 otsu threshold where the value of the threshold is determined automatically (bimodal distribution)
* binary = cv.adaptiveThreshold(gray, 255, cv.ADAPTIVE\_THRESH\_GAUSSIAN\_C, cv.THRESH\_BINARY, 135, 2) 🡪 adaptive thresholding. Parameters:
  + source image
  + maximum value that can be assigned to a pixel
  + adaptive method to decide how threshold value is calculated, for example adaptive\_thresh\_gaussian\_c or adaptive\_thresh\_mean\_c;
  + type of thresholding to be applied.
  + 135 is blockSize of a pixel neighborhood to calculate a threshold value
  + 3 is a constant subtracted from the mean or weighted sum of the neighborhood pixels

With global thresholding method, the same value of *T* is used to test *all pixels* in the input image but due to variations in lighting conditions, shadowing, etc., it may be that one value of *T* will work for a certain part of the input image but fail on a different segments. To overcome this problem, we can use adaptive (local) thresholding, which considers small neighbors of pixels and then finds an optimal threshold value *T* for each neighbor, implying that local regions of an image will have similar lighting.

Once the binary has been calculated, it can be used with the findContours() OpenCV function:

contours, hierarchy = cv.findContours(binary, cv.RETR\_EXTERNAL, cv.CHAIN\_APPROX\_SIMPLE) 🡪 parameters:

* binary image
* contour-retrieval mode, for example retr\_external = retrieves all of the contours and reconstructs a full hierarchy of nested contours; retr\_tree = retrieves only the extreme outer contours
* contour-approximation method, for example chain\_approx\_none = all the boundary points are stored; chain\_approx\_simple = removes all redundant points

Finally the contours can be drawn with:

cv.drawContours(img, contours, -1, (0, 0, 255), 2)🡪 parameters: image to draw contours on, contours, -1 = draw all contours, contours color, thickness

To draw, among all the one identified, the contours that appears to be more rounded, approxPolyDP () from OpenCV can be used since approximates a contour shape to another shape with less number of vertices:

for cnt in contours:

approx = cv.approxPolyDP(cnt,0.01\*cv.arcLength(cnt,True),True)

if len(approx) > 12:

cv.drawContours(img,[cnt],0,(0,0,255),2)

approx = cv.approxPolyDP(cnt,0.01\*cv.arcLength(cnt,True),True)🡪 parameters:

* cnt = array of the contour points
* 0.01\*cv.arcLength(cnt,True) is the epsilon, the precision = maximum distance from contour to approximated contour. Cv.arcLength is used to calculate contour perimeter, where the second argument specify whether shape is a closed contour (if passed True), or just a curve.
* Third argument specifies whether curve is closed or not.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Contour | Approximated contour |
| Manual threshold | Blurring | Threshold = 130 |  |
| Adjusting contrast and brightness, bilateral filter, blurring | Threshold = 210  Contrast = 1.5  Brightness = 70 |  |
| Adaptive threshold | Blurring | blocksize = 135 |  |
| Adjusting contrast and brightness, bilateral filter, blurring | contrast = 1  brightness = 20 |  |
| Otsu threshold | Blurring |  |  |
| Adjusting contrast and brightness, bilateral filter, blurring | contrast = 1.5  brightness = 50 |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Contour | Approximated contour |
| Manual threshold | Blurring | Threshold = 140 |  |
| Adjusting contrast and brightness, bilateral filter, blurring | Threshold = 195  Contrast = 1.5  Brightness = 20 |  |
| Adaptive threshold | Blurring | blocksize = 135 |  |
| Adjusting contrast and brightness, bilateral filter, blurring | contrast = 1.2  brightness = 40 |  |
| Otsu threshold | Blurring |  |  |
| Adjusting contrast and brightness, bilateral filter, blurring | contrast = 1.2  brightness = 50 |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Contour | Approximated contour |
| Manual threshold | Blurring | Threshold = 80 |  |
| Adjusting contrast and brightness, bilateral filter, blurring | Threshold = 195  Contrast = 2  Brightness = 50 |  |
| Adaptive threshold | Blurring | blocksize = 135 |  |
| Adjusting contrast and brightness, bilateral filter, blurring | contrast = 2.5  brightness = 50 |  |
| Otsu threshold | Blurring |  |  |
| Adjusting contrast and brightness, bilateral filter, blurring | contrast = 2  brightness = 30 |  |

Examples of contours detected through the use of a thresholded image obtained with three different methods.

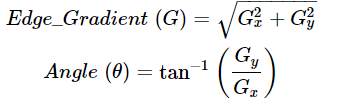
1. EDGE DETECTION

Canny Edge Detection is a multi-stage algorithm composed of different stages:

1. Noise Reduction

Since edge detection is susceptible to noise in the image, first step is to remove the noise in the image with a 5x5 Gaussian filter

1. Finding Intensity Gradient of the Image

Smoothened image is then filtered with a Sobel kernel in both horizontal and vertical direction to get first derivative in horizontal direction (Gx) and vertical direction (Gy). From these two images, we can find edge gradient and direction for each pixel as follows:

1. Non-maximum Suppression

After getting gradient magnitude and direction, a full scan of image is done to remove any unwanted pixels which may not constitute the edge. For this, at every pixel, pixel is checked if it is a local maximum in its neighborhood in the direction of gradient. In short, the result you get is a binary image with "thin edges".

1. Hysteresis Thresholding

This stage decides which among all edges are really edges and which are not. For this, we need two threshold values, minVal (T\_lower) and maxVal (T\_upper). Any edges with intensity gradient more than maxVal are sure to be edges and those below minVal are sure to be non-edges, so discarded. Those who lie between these two thresholds are classified edges or non-edges based on their connectivity. If they are connected to "sure-edge" pixels, they are considered to be part of edges. Otherwise, they are also discarded.

The goal is to find the edges, meaning the points where the intensity of colors changes significantly, and then simply turn those pixels on.

edges = cv.Canny(img, 80, 200, L2gradient= True) 🡪 parameters:

* source image,
* T\_lower: Lower threshold value in Hysteresis Thresholding.
* T\_upper: Upper threshold value in Hysteresis Thresholding.
* aperture\_size: optional parameter to specify the order of the Sobel filter used to calculate the gradient in the Canny algorithm. The default value is 3 and its value should be odd between 3 and 7. You can increase the Aperture size when you want to detect more detailed features
* L2Gradient: optional boolean parameter that specifies the equation for finding gradient magnitude.  If it is True, it uses the equation sqrt(gradient\_x\_square + gradient\_y\_square) which is more accurate, otherwise it uses the function : abs(gradient\_x) + abs(gradient\_y).

Example images with the canny edge algorithm applied, with the parameters set as edged = cv.Canny(img, 100, 200, L2gradient= True) for the first one and edged = cv.Canny(img, 200, 300, L2gradient= True) for the last two.

The binary images obtained can be used with the find contours OpenCv method:

* Edged = cv.Canny(img, 100, 200, L2gradient= True) , contours and approximated contours



* Edged = cv.Canny(img, 200, 300, L2gradient= True) , contours and approximated contours

* Edged = cv.Canny(img, 200, 300, L2gradient= True) , contours and approximated contours



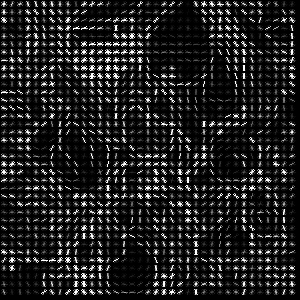
1. HISTOGRAM OF ORIENTED GRADIENTS

[The Histogram of Oriented Gradients (HOG)](https://en.wikipedia.org/wiki/Histogram_of_oriented_gradients" \t "_blank" \o "The Histogram of Oriented Gradients (HOG)) is a feature descriptor used in [computer vision](https://www.thepythoncode.com/topic/computer-vision) and image processing applications for the purpose of the [object detection](https://www.thepythoncode.com/article/yolo-object-detection-with-opencv-and-pytorch-in-python). It is a technique that counts events of gradient orientation in a specific portion of an image or region of interest.

The feature vector is not really useful for the purpose of viewing the image, but rather for tasks like image recognition and object detection. The feature vector produced by these algorithms when fed into an image classification algorithms like Support Vector Machine (SVM) produce good results.

fd, hog\_img = hog(img, orientations=9, pixels\_per\_cell=(8, 8), cells\_per\_block=(2, 2), block\_norm='L2-Hys', transform\_sqrt=True, visualize=True) 🡪 parameters:

* The original descriptor is fd; hog\_image is the descriptor image that we can visualize
* image: The target image you want to apply HOG feature extraction.
* orientations: Number of bins in the histogram we want to create, the original research paper used 9 bins so we will pass 9 as orientations.
* pixels\_per\_cell: Determines the size of the cell, here it is 8x8.
* cells\_per\_block: Number of cells per block, will be 2x2
* block normalization: the L2-Hys method is used to normalize the blocks and it consists of an L2-norm followed by clipping and a renormalization
* transform\_sqrt : square root normalization
* visualize: A boolean to return the image of the HOG, we set it to True so we can show the image.

Example histogram of oriented gradients images of the sample images



The steps of the HOG Feature Descriptor are:

* Preprocessing (Gamma/Color Normalization and Resizing).
* Computing the image gradients in both the *x* and *y* direction, final gradient magnitude, orientation of the gradient for each pixel in the input image
* Spatial / Orientation Binning: dividing the image into cells and blocks
* Block Normalization: to account for changes in illumination and contrast, we can normalize the gradient values *locally*. This requires grouping the “cells” together into larger, connecting “blocks”. It is common for these blocks to *overlap*, meaning that each cell contributes to the final feature vector more than once.
* Get the HOG Feature Vector.

1. HOUGH CIRCLES

Hough circle transform is a feature extraction method used to detect circles in an image. It works by transforming the image into a parameter space, where each point represents a possible circle center. The technique then searches for the best parameters to define a circle that fits the image.

The OpenCV library provides a function called HoughCircles that implements the Hough transform for circle detection.

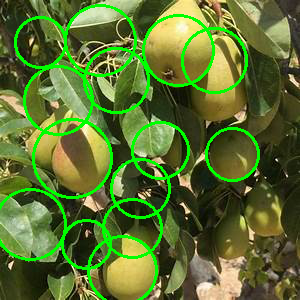
circles = cv.HoughCircles(gray, cv.HOUGH\_GRADIENT, 1, minDist=gray.shape[0]/8, param1=100, param2=20, minRadius=20, maxRadius=40)🡪 parameters:

* Grayscale input image
* cv2.HOUGH\_GRADIENT detection method (uses gradient of the edges instead of filling up the entire 3D accumulator matrix, speeding up the process)
* dp: inverse ratio of the accumulator resolution to the image resolution. Here is 1 so the accumulator has the same resolution as the input image. Essentially, the larger the dp gets, the smaller the accumulator array gets.
* minDist: minimum distance between the centers of the detected circles. All the candidates below this distance are neglected. If the minDist is too small, multiple circles in the same neighborhood as the original may be (falsely) detected. If it is too large, some circles may be missed.
* param1: the higher threshold of the two passed to the Canny edge detector
* param2: accumulator threshold for the circle centers at the detection stage. By increasing this threshold value, we can ensure that only the best circles, corresponding to larger accumulator values, are returned. The smaller the threshold is, the more circles will be detected (including false circles). The larger the threshold is, the more circles will potentially be returned.
* minRadius: minimum circle radius size, in pixels. If unknown, put zero as default.
* maxRadius: maximum circle radius size, in pixels. If unknown, put zero as default.

Then, to draw the detected circles we can use cv.circle()method. Main parameters are:

* image: the image on which the circle is to be drawn.
* center\_coordinates: It is the center coordinates of the circle. The coordinates are represented as tuples of two values i.e. (X coordinate value, Y coordinate value).
* radius: It is the radius of the circle.
* color: It is the color of the borderline of a circle to be drawn. For BGR, we pass a tuple. eg: (255, 0, 0) for blue color.
* thickness: It is the thickness of the circle border line in px. Thickness of -1 px will fill the circle shape by the specified color.

circles = cv.HoughCircles(gray, cv.HOUGH\_GRADIENT, 1, minDist = gray.shape[0]/8, param1=100, param2=20, minRadius=20, maxRadius=40)



circles = cv.HoughCircles(gray, cv.HOUGH\_GRADIENT, 1, minDist = gray.shape[0]/8, param1=100, param2=20, minRadius=0, maxRadius=20)



circles = cv.HoughCircles(gray, cv.HOUGH\_GRADIENT, 1, minDist = gray.shape[0]/8, param1=100, param2=10, minRadius=0, maxRadius=15)



To detect circles, we are required to find 3 parameters: a, b and r. Therefore, the problem is in a 3-dimensional search space. To find possible circles, the algorithm uses a 3-D matrix called the “Accumulator Matrix” to store potential a, b and r values. The value of a (x-coordinate of the center) may range from 1 to rows, b (y-coordinate of the center) may range from 1 to cols, and r may range from 1 to maxRadius =

Below are the steps of the algorithm.

* Initializing the Accumulator Matrix: Initialize a matrix of dimensions rows \* cols \* maxRadius with zeros.
* Pre-processing the image: Apply blurring, grayscale and an edge detector on the image. This is done to ensure the circles show as darkened image edges.
* Looping through the points: Pick a point  on the image.
* Fixing r and looping through a and b: Use a double nested loop to find a value of r, varying a and b in the given ranges.
* Voting: Pick the points in the accumulator matrix with the maximum value. These are strong points which indicate the existence of a circle with a, b and r parameters. This gives us the Hough space of circles.
* Finding Circles: Finally, using the above circles as candidate circles, vote according to the image. The maximum voted circle in the accumulator matrix gives us the circle.

1. COLOR HISTOGRAMS

A color histogram counts the number of times a given pixel intensity (or range of pixel intensities) occurs in an image. Using a color histogram we can express the actual distribution or “amount” of each color in an image. The counts for each color/color range can then be used as feature vector (i.e. a list of numbers used to quantify an image and compare it to other images) and color histograms can be clustered to (automatically) create groups of images that have similar color distributions: histograms that belong to a given cluster will be more similar in color distribution than histograms belonging to other cluster; comparing the “similarity” of color histograms can be done using a distance metric (Euclidean, correlation, Chi-squared, intersection, and Bhattacharyya).

Assumption: images with similar color distributions contain equally similar visual contents — this assumption may or may not hold considering different particular applications.

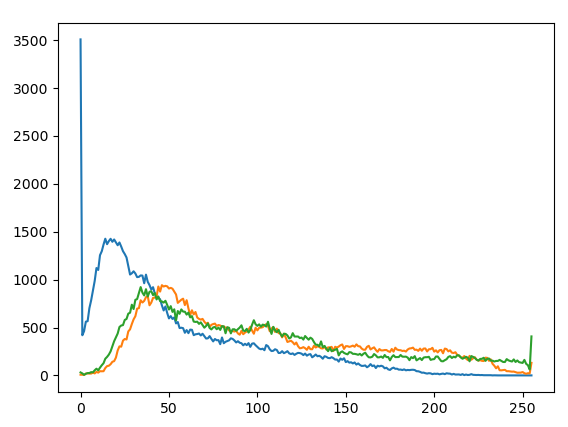
hist\_bw = (cv.calcHist([img\_bw], [0], None, [256], [0, 256])) 🡪 parameters:

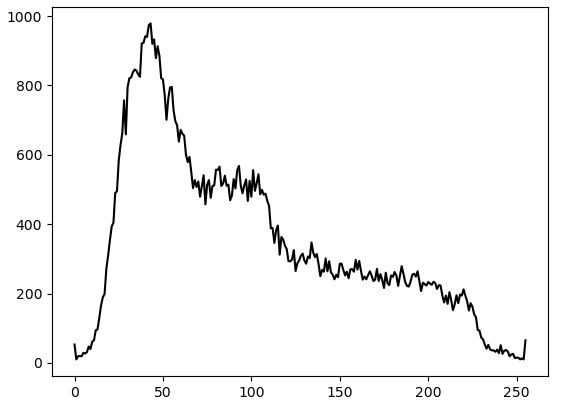
* image that we want to compute a histogram for. Wrap it as a list: [myImage].
* channels: A list of indexes, where we specify the index of the channel we want to compute a histogram for. To compute a histogram of a grayscale image, the list would be [0]. To compute a histogram for all three red, green, and blue channels, the channels list would be [0, 1, 2].
* mask: a uint8 image with the same shape as the original image, where pixels with a value of zero are ignored and pixels with a value greater than zero are included in the histogram computation. Using masks allow to only compute a histogram for a particular region of an image.
* histSize: This is the number of bins we want to use when computing a histogram. It is a list, one for each channel we are computing a histogram for. The bin sizes do not all have to be the same.
* ranges: The range of possible pixel values. Normally, this is [0, 256] for each channel, but if using a color space other than RGB (such as HSV), the ranges might be different.

To apply it on a colored image:

1. Split the image into its three channels: blue, green, and red. We then initialize a tuple of strings representing the colors.
2. Loop over the channels and compute a histogram for each. Then concatenate the color histogram of each channel into the features list: the histogram will result as a single list of pixel counts, defined as flattened histogram.

Color histograms of the sampled image in original color and gray scale.





1. FILTER COMBINATION

Blending: is a sort of image addition, where different weights are given to images so that it gives a feeling of blending or transparency (are blending weights that determine the contribution of each image to the final output). Remember, both images should be of equal size and depth.

img = cv2.addWeighted(img1, 0.3, img2, 0.7, 0)🡪 parameters:

* img1: first Input Image array (single-channel, 8-bit or floating-point)
* 0.3: weight of the first input image elements to be applied to the final image
* img2: second Input Image array (single-channel, 8-bit or floating-point)
* wt2: weight of the second input image elements to be applied to the final image
* gammaValue: optional, measurement of light