Transformations and Basic Computer Vision Tools

Susan Fox

Activity Goals

Through this activity, you will practice with transformations, filters, and simple feature-detection in images. When you're finished, you should be able to:

-
- Understand and use morphological filters to preprocess images
- Use thresholds and masking to process images
- Apply contour, edge, and corner detection algorithms to images

Overview

This activity covers tools for "preprocessing" images to make extracting information from them easier, and also basic tools for getting information from images. There are two major tasks: finding signs in an image with a person's help, and detecting coins or lines in a puzzle. Additional milestones practice with useful techniques. Be creative, and don't forget to try manipulating with blurring, morphing, thresholding, and masking images to make contour, edge, and corner detection easier.

Image Resizing and other Geometric Transformations

Image Resize

The most basic transformation is resizing an image. The `resize` function will scale a picture up or down, and can also be used to stretch an image. You must give the `resize` function a source image and an input for the dimensions of the new picture. However, you can alternately give factors to multiply the source's dimensions by. The examples below illustrate different ways of calling `resize`.

Examples	**Meaning**
`cv2.resize(src, (100, 100))`	Returns a new image that is 100 x 100 pixels, a stretched/squashed version of the original

`cv2.resize(src, (0, 0), fx = 2, fy = 2)`	Returns a new image that is twice the size of the original, same aspect ratio
`cv2.resize(src, (0, 0), fx = 0.5, fy = 1.0)`	Returns a new image whose columns have been squashed to half the original size

Try these examples on one of the images from `SampleImages` to see what they do.

Affine and Perspective Warping

We're not going to talk about these geometric transformations. With them you can translate, rotate, or warp the contents of a picture. If you need these tools for your project, look up the `warpAffine` function, and we can talk individually about how these work (one application can be for data augmentation in machine learning).

Filtering Images

A filter looks at every pixel value, and uses those values to determine new values. Filtering can be used to create a number of interesting effects, but is often used to manipulate an image that is then sent to a more complex computer vision algorithm.

Blurring an image

Blurring is probably the most basic filter of all. Images are often blurred as part of other operations, and to reduce the amount of random noise in an image.

You will find example code to illustrate how blurring works in the file `blurring.py`; get a copy of it and add it to your project. The program shows the original image, and then lets you choose between two different blur functions, and to choose the parameters of the blur.

When you blur a picture, you change each pixel so that its color is the average of the pixels around it. The basic `blur` function takes a rectangular neighborhood centered around the target pixel, and it performs a straight average, channel-by-channel (all the red values are averaged, all the green values are averaged, and all the blue values are averaged). The resulting picture is placed in the new image at the target pixel location. The process is repeated for every pixel in the image. The program in `blurring.py` allows you to change the size of the neighborhood using the w-a-s-d keys. You may increase or decrease the height of the region using w and s, and you may do the same to the width of the region using a and d. Try this out. The larger the region, the more blurred the result will be.

The second blur function is called `GaussianBlur`, and it performs a weighted average of the pixels in the neighborhood, with more weight given to pixels closer to the center. Gaussian, here, refers to Gaussian functions, more commonly called normal curves or bell curves. Experiment with how the plain blur compares to the Gaussian blur by switching between them: typing 1 will switch to the plain blur, and typing 2 will switch to Gaussian blur.

Morphological Filters

There are a set of “morphological” filters to experiment with. The program `morph.py` demonstrates each, and lets you experiment with their effects. When you run the program, you will see two sliders at the top of the screen. Text displayed on the picture tells you which effects are currently in place, and the value of the parameters for these filters. You can change between different effects using the 1 key, and you can change the geometric shape used to generate the effects using the 2 key. Play with the program, and also read the brief description of each effect below.

Dilation and Erosion

Much like blurring dilation and erosion determine the value for a pixel based on a neighborhood of pixels from the original image. Unlike the default in blurring, we can select the shape of the neighborhood, as well as its size, to be either rectangular, elliptical, or cross shaped. With dilation, the value of each channel of a pixel is the *maximum* value of that channel in any pixel in its neighborhood. Erosion is the opposite: the value of each channel of a pixel is the *minimum* value of that channel in any pixel in its neighborhood.

These can be used to emphasize and thicken edges or regions of color in a particular part of an image.

Opening and Closing

Sometimes we want to preserve both dark and light features of an image. Opening and closing combine dilation and erosion. Opening an image means first performing an erosion of the image, then a dilation, using the same size and shape of neighborhood. Closing first dilates the image, then erodes it.

Both these operations are good at removing noise and small details from images.

White and Black Top-Hat

The Top-Hat filters do the opposite of opening and closing. Instead of removing the fine details, these filters keep just the fine details. The white top-hat takes the difference between the original image and the opening of the image. The black top-hat takes the difference between the closing and the original image.

These filters may be used for feature extraction tasks, or image enhancement.

Morphological Gradient

The morphological gradient takes the difference between the dilation and erosion of an image. This emphasizes the places where there is a change in color, and can be useful to enhance edges.

Milestone 1: Blurring and morphing video

Start with a program that just displays the frames from the webcam.

Blurring the video gradually:
Step 1: Use the Gaussian blur function to blur the image to be displayed.
Step 2: Set a minimum neighborhood size and a maximum neighborhood size. Define a variable `blurDir` that holds 2 or -2, depending on whether we are increasing or decreasing the neighborhood size. Set up a variable to hold the current neighborhood size, initially set to minimum. Use Gaussian blur to blur the image with the current size, then update the current size by adding `blurDir` to it. If the resulting blur amount is greater than the maximum or less than the minimum, change `blurDir` to the other value.

Morphing the video, instead:
You may need to look up the documentation page for the morphological filters to help with this one. Choose at least 1 of the morphological filters described above and demonstrated in `morph.py`. Modify the blurring program so that it operates with a changing neighborhood size but applies the chosen morphological filter instead of blurring. Try changing the neighborhood shape to see what effect it has.

Thresholding and Masking

Thresholding and masking are common techniques to limit the data our computer vision programs work on. Neither of these are kernel-based manipulations. With thresholding, we specify a threshold value, and a maximum value, and the function converts values on one side or the other of the threshold. The exact effect varies depending on which threshold operation we choose. The table below lists the different operations:

Threshold operator	**Meaning**
`cv2.THRESH_BINARY`	Values above the threshold are set to the maximum value, values less than or equal to the threshold are set to zero
`cv2.THRESH_BINARY_INV`	Values above the threshold are set to zero, values less than or equal to the threshold are set to the maximum value
`cv2.THRESH_TRUNC`	Values above the threshold are set to the threshold value, values less than or equal to the threshold are unchanged
`cv2.THRESH_TOZERO`	Values above the threshold are left unchanged, values less than or equal to the threshold are set to zero
`cv2.THRESH_TOZERO_INV`	Values above the threshold are set to zero, values less than or equal to the threshold are left unchanged

The `threshold` function takes in four inputs: a source image, a threshold value, a maximum value, and the code for which operator to perform. The script below illustrates a use of the `threshold` function.

```
import cv2
img = cv2.imread("SampleImages/wildColumbine.jpg")
cv2.imshow("Original", img)
img2 = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
res, img3 = cv2.threshold(img2, 128, 255, cv2.THRESH_BINARY)
cv2.imshow("Thresholded", img3)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Experiment with this script, varying the threshold value and maximum value, and change the type of thresholding to see the effect.

Masking

Masking is similar to thresholding, except that we include or exclude pixels based on location. A mask is typically a binary image the same size as our main image, but only one channel, and containing values that are 255 or 0. We can combine this mask with our main image using boolean operations, `bitwise_and` and `bitwise_or`, provided by OpenCV. Some operations on images take an optional mask input, and automatically exclude any zero areas from the operation.

Experiment with changing the location of the mask. Note that you can also make masks that aren't regular shapes.

```
import cv2
import numpy as np
img = cv2.imread("SampleImages/wildColumbine.jpg")
gImg = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
(height, width, depth) = img.shape

# a rectangular mask that keeps the middle of a picture
rMask = np.zeros((height, width, 1), np.uint8)
rMask[50:height // 2, 50:width // 2] = 255

# applying a mask to an image (this is a bit weird)
maskedImg1 = cv2.bitwise_and(img, img, mask=rMask)
cv2.imshow("Original", img)
cv2.imshow("Rectangular Mask", maskedImg1)

# Using thresholding to create a mask
res, tMask = cv2.threshold(gImg, 128, 255, cv2.THRESH_BINARY)
maskedImg2 = cv2.bitwise_and(img, img, mask=tMask)
cv2.imshow("Threshold Mask", maskedImg2)
cv2.waitKey(0)
```

inRange

Another thresholding function is called `inRange`. This function takes color images, and two arrays that describe the minimum values for each color channel and the maximum values for each color channel. You can use it to select only bright or dark regions, or only pixels with values in a certain color range. Try the script below, then experiment with changing the ranges.

```
import cv2
import numpy as np
img = cv2.imread("SampleImages/wildColumbine.jpg")
cv2.imshow("Original", img)
img2 = cv2.inRange(img, np.array([0, 150, 150]), np.array([255, 255, 255]))
cv2.imshow("In Range", img2)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Milestone 2: Cleaning up card or puzzle pictures

Thresholding and masking can be used to simplify images so that we can find details only in the parts we care about. Pick one (or two) of the “card” pictures, and one (or two) of the puzzles (not the computer-generated ones). Then try combining morphological transformations with the thresholding (or inRange) and masking, to see if you can clean away irrelevant details like the surrounding room, or the clues in the crossword, and tidy up the pictures so that they are simple black and white images.
Don’t look for perfection: it is okay to tune your code to one picture, but do try it, if you can, on other similar pictures.

Contours

OpenCV has a very useful tool for extracting information from an image: finding contours. The `findContours` function takes in an image, preferably a “binary” image that is just black and white, and it finds contiguous contours in the image. There is a `drawContours` function that can draw the contours on an image. You pass `findContours` the image, and then two inputs that specify what method the function should use, and how it should approximate the contours. Below are two tables listing the different options

Mode (second input)	
cv2.RETR_LIST	A list of all contours
cv2.RETR_EXTERNAL	Only the outermost contours (it omits any that are completely contained inside others)
cv2.RETR_TREE	Returns a hierarchy of contours so you can tell which are inside which
cv2.RETR_CCOMP	Returns a two-level hierarchy, where outer contours and then the “hole” inside are organized, but further contours inside are made top-level.

Approximation method (third input)	
cv2.CHAIN_APPROX_NONE	Return all points found, no approximation

cv2.CHAIN_APPROX_SIMPLE	Approximates end points of horizontal, vertical, and diagonal lines

The function `findContours` returns three values: a copy of the image, the data structure holding the contours, and a data structure holding the hierarchy, if any. **Important Note: `findContours` destroys the image that is input to it!**

In order to find meaningful contours, you need to **preprocess** the image using thresholding and other image processing methods. Here is a simple demo program. Try it out to see how it works:

```
import numpy as np
import cv2

origIm = cv2.imread('SampleImages/coins5.jpg')
imgray = cv2.cvtColor(origIm,cv2.COLOR_BGR2GRAY)
ret,thresh = cv2.threshold(imgray,127,255,0)
res, contrs, hier = cv2.findContours(thresh,cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
cv2.drawContours(origIm, contrs, -1, (0,255,0), 3)
cv2.imshow('Contours', origIm)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Milestone 3: Finding coins

For this, work with the coin pictures in `SampleImages`. Suppose we want to locate the coins in an arbitrary image. The problem is that some parts of coins are brightly lit, others are more dimly lit. Try using `findContours` to find the outlines of the coins in one of the coin pictures. You **will** need to preprocess the image using thresholding, maybe masking out certain colors, etc. Blurring and morphological operations might also be useful. Try using `findContours` to encircle the coins.

How well can you do? Does your program work better on some coin images than on others? **Do not expect perfection here. If you tune your program to one image, you may find it fails to work on others.**

Edge Detection

Finding edges and lines in images is one of the classic computer vision tasks. Edges are defined to be locations in an image where the color or brightness changes dramatically. There are different tools for determining these edges; we'll look at the Sobel gradient operation, and the Canny edge detection algorithm.

For several of these tools, we treat the brightness across the pixels of the image as a surface, a two-dimensional function and apply some calculus tools to it. Fortunately, you don't have to know multivariate calculus to understand the gist of what the methods are doing.

Sobel Gradient-Finding

Sobel first uses Gaussian blurring on an image to smooth the surfaces in the image, and then it looks for peaks and valleys in the result. In mathematical terms, it computes the first derivative of the function represented by the smoothed image at each pixel location. Thus the result of Sobel is a floating-point number, which we must convert into a grayscale value for displaying it. It works better to compute the Sobel values in two directions separately, and then to combine the two. Note that, for precision, the type of data in the gradients is 32-bit floating-point numbers, which are then scaled into unsigned 8-bit numbers for display.

```
import cv2
img = cv2.imread("SampleImages/chicago.jpg")
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Compute gradient in horizontal direction (detects vertical edges)
sobelValsHorz = cv2.Sobel(gray, cv2.CV_32F, 1, 0)
horzImg = cv2.convertScaleAbs(sobelValsHorz)
cv2.imshow("horizontal gradient", horzImg)

# Compute gradient in vertical direction (Detects horizontal edges)
sobelValsVerts = cv2.Sobel(gray, cv2.CV_32F, 0, 1)
vertImg = cv2.convertScaleAbs(sobelValsVerts)
cv2.imshow("vertical gradient", vertImg)

# Combine the two gradients
sobelComb = cv2.addWeighted(sobelValsHorz, 0.5, sobelValsVerts, 0.5, 0)
# Convert back to uint8
sobelImg = cv2.convertScaleAbs(sobelComb)
cv2.imshow("Sobel", sobelImg)
```

Try the example above. Then try modifying things. What happens if you put -1 in for the `cv2.CV_32F`, which causes the resulting gradients to use the same type as the images themselves, unsigned 8-bit numbers? What if you change the weighting in the `addWeighted` function? What if you don't convert back to unsigned 8-bit values? What if you use the color image? Or the hue channel from an HSV image?

Canny Edge Detection

The Canny algorithm goes beyond what Sobel does. It performs a Sobel transformation in both horizontal and vertical directions as its first step. It then removes points that have a change in brightness, but that are not "local maxima" (whose neighbors in the direction of the change are not all smaller than it). Finally, it has two threshold values (supplied as inputs). Any edge value below the min threshold is discarded. Any edge value above the max threshold is kept. In between the two thresholds, edges are kept only if they are connected to some edge above the threshold.

The Canny algorithm takes a grayscale picture as input, and the two thresholds.

```
import cv2
img = cv2.imread("SampleImages/chicago.jpg")
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

cannyImg = cv2.Canny(gray, 100, 200)
cv2.imshow("Canny", cannyImg)
```

One of the difficulties with Canny edge detection is determining the proper threshold values. Write a program that loops through possible threshold values in increments of 5 or 10 (the threshold value must be between 0 and 255, and one value must be larger than the other), and displays the edges that are found. Decide what produced the "best" value. Be sure to try a variety of pictures in the `SampleImages` folder, to see how results vary across different kinds of images.

Create your own program to show the video image from your computer's camera, displaying the result of Canny edge detection.

There are line detection algorithms, Hough (pronounced "huff") lines and circles, which you may experiment with. We are going to skip them and focus on corners instead of edges.

Milestone 4: Edges of coins, signs, and puzzles

Part A: For this, work with the coin pictures in `SampleImages`. Try calling `Sobel` on some of the coin pictures, how well does it separate background from coins? Try `Canny`, and vary its parameters. Can you get edges good enough for `findContours` to find the coins? Make your best program using edge detection to find the coins.

Part B: Now, try a different kind of problem. Inside the `CardsAndSigns` folder are pictures of me holding cards. Can you generate a contour around the outside of the sign, using edge detection and `findContours`. Once you have a decent program, come borrow some cards from me and try your program on the webcam video from your computer.

Part C: Lastly, look at the `PuzzlesAndGames` folder. There are screenshots or photos of sudoku, tic-tac-toe, and crossword puzzles. Can you find a contour around the outside of the puzzle? For a challenge, can you find contours around the interior boxes of each puzzle. Which ones work best?

Feature Detection

Computer vision researchers have developed multiple algorithms for finding features of interest, often considered "corners", in images: a corner is a place where two edges meet. The most famous corner detectors are the Harris corner detector, and the Shi-Tomasi detector. OpenCV provides functions that implement both of these, but it also provides a function that combines them for you and is easier to use. We'll look at all three methods here.

Harris Detector

The Harris corner detector starts with the Sobel edge-detecting process, and then does some mathematics to determine changes in intensity for some distance in each direction. While the math is discussed in the OpenCV Python tutorial about this method, and probably in the Szeliski book, I have found this online tutorial to be very understandable: http://aishack.in/tutorials/harris-corner-detector/.

The parameters for the Harris detector include the neighborhood size, a parameter to control the Sobel step in the process, and a parameter from the formula that is being calculated. The input to the detector also needs to be a 32-bit floating-point array, based on the grayscale version of the image. One complication is that the Harris algorithm produces a grayscale image as its output, where the brightness indicates how strong a corner has been found. To isolate the best corners, we can dilate the resulting image, and then use a threshold to convert the biggest values to white and all others to black. Sample code is below.

```
import cv2
import numpy as np

img = cv2.imread("SampleImages/PuzzlesAndGames/puzzle4.jpg")
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
gray = np.float32(gray)
```

```
# Compute Harris
dst = cv2.cornerHarris(gray, 2, 3, 0.04)

# Isolate Harris corners from image
dilDst = cv2.dilate(dst, None)
thresh = 0.01 * dst.max()
ret, threshDst =  cv2.threshold(dilDst, thresh, 255, cv2.THRESH_BINARY)

# Display corners points
disp = np.uint8(threshDst)
cv2.imshow("Harris", disp)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Try out this code. Look at what happens when you don't dilate the Harris result before thresholding it. What happens if you change the three parameters of the call to the Harris detector? Make a version of this program that draws a circle on the original image at every location where a corner has been found.

Shi-Tomasi Detector: Good Features to Track

Shi and Tomasi wrote a paper entitled "Good Features to Track," which modified the approach of the Harris detector. It calculates the same matrix of intensity values as the Harris detector does, and uses the "eigenvalues" of that matrix (you don't have to get all this math), but then checks if the minimum of the two values is over a threshold.

The `goodFeaturesToTrack` function takes a grayscale image, and at least three other parameters. The first defines the maximum number of corners to return; the algorithm returns the strongest corners up to that number. The second parameter defines the quality level of corners to be found acceptable, and the third parameter specifies how many pixels must separate adjacent corners.

The script below shows how to call the function. The features returned are a Numpy array; Each row of the array contains a single feature point. There is only one column in the array, and the x and y values are stored in the third dimension. Examine the print statements below that show how to pick apart the values in the `goodFeats` array.

```
import cv2
img1 = cv2.imread("SampleImages/PuzzlesAndGames/puzzle4.png")
grayImg = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
goodFeats = cv2.goodFeaturesToTrack(grayImg, 100, 0.1, 5)
print(goodFeats[0], goodFeats[1])
print(goodFeats[0,0])
print(goodFeats[0,0,0], goodFeats[0,0,1])
```

After you understand what the print statements are telling you, remove them. In their place, add a `for` loop that loops over the rows of the `goodFeats` array. For each row, extract the x and y values, and draw a small circle on the original image at that location.

Feature Detection with FAST

Many computer vision applications need real-time processing of images. The FAST algorithm was developed to run faster than Harris or Shi-Tomasi detectors, while giving reasonably good performance. It does not give an orientation for the features it finds. Below is a small sample program showing how to use FAST. It returns "keypoints," which may have additional information attached to them.

```
import cv2

img = cv2.imread("SampleImages/chicago.jpg")
cv2.imshow("Original 1", img)

# create a FAST object, that can run the FAST algorithm.
fast = cv2.FastFeatureDetector_create()
# detect features
keypts = fast.detect(img, None)
img2 = cv2.drawKeypoints(img, keypts, None, (255, 0, 0), 4)
cv2.imshow("Keypoints 1", img2)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Milestone 5: Corners on signs and puzzles

Write a program to use corner features on the sign pictures. Try the Shi-Tomasi and FAST methods to finding corners. Try preprocessing, including thresholding, blurring, morphological changes. Can you find just the corners of the signs?

Write a sample program to run Shi-Tomasi or FAST (separately) on the video feed. Is Shi-Tomasi noticeably slower?

Think about the overall performance of the Shi-Tomasi and FAST detectors. Which one works better on any given individual picture? Which one works better over a set of pictures? We'll discuss this in our group session

Feature Detection with ORB

Beyond detecting corners, many algorithms also look for unique features within images. It is particularly important to find them quickly, and to find features at different scales, rotations, etc. These features are used to match parts of one image against parts of another image.

ORB is an open-source variant of methods SIFT and SURF (you can find published papers about them). For now, you don't need to understand exactly how ORB works; you just need to use them. If your research project ends up using ORB, then you will need to study how it and the related methods work. Here I will only give a very brief overview. ORB, like SIFT and SURF, makes a "pyramid" of copies of the original image resized to different scales so that it can find features that appear both tiny and large. It then uses a combination of the FAST feature detector with Harris corner detection to find features, which we call **keypoints**. Then ORB applies another technique called BRIEF to construct "descriptors" that tell you where and at what scale the features were found. ORB returns an array of the locations of its keypoints, and a separate array of "descriptors."

Before going further pull up the ORB tutorial and OpenCV documentation. This contains more details about how the algorithm works and what options you can use for it. Also this separate tutorial seems like a nice intro as well.

The script below shows how to use ORB to find features, called keypoints, and how OpenCV allows you to display them.

```
import cv2

# Some versions of OpenCV need this to fix a bug
# cv2.ocl.setUseOpenCL(False)

img = cv2.imread("SampleImages/PuzzlesAndGames/puzzle4.jpg")


# create an ORB object, that can run the ORB algorithm.
orb = cv2.ORB_create()    # some versions use cv2.ORB() instead

keypts, des = orb.detectAndCompute(img, None)

img2 = cv2.drawKeypoints(img, keypts, None, (255, 0, 0), 4)
cv2.imshow("Keypoints 1", img2)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Note that the features here are of different sizes, unlike the corner-detector. They also have orientations given by the lines drawn inside the circles. These features can be matched against features found in another image to locate an object in an image, or to look for ways to stitch images together.

Matching Features Across Images

We can use features like those found with ORB to identify the existence of an object inside a larger picture, or to match two pictures with each other. To do this, we use one of OpenCV's matcher algorithms. OpenCV provides several versatile feature-matching algorithms. We are going to use the "Brute Force" matcher, because it works better with ORB than the alternatives.

The brute-force matcher takes in the descriptors ORB generates for two different images. It compares every descriptor from the first image to every descriptor from the second image, and computes a measure of "distance" between the descriptors. If the descriptors are identical, the distance is zero, so smaller values are better than larger ones. It reports the matches it finds that are "stable," meaning that the first descriptor A finds descriptor B to be its closest match, and B finds A to be closest, too. The number and quality of matches from the matcher can tell us how well the two images match each other. The code below illustrates the brute-force matcher in action.

```
import cv2

img1 = cv2.imread("SampleImages/Coins/coins1.jpg")
img2 = cv2.imread("SampleImages/Coins/dollarCoin.jpg")

orb = cv2.ORB_create()
bfMatcher = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)

kp1, des1 = orb.detectAndCompute(img1, None)
kp2, des2 = orb.detectAndCompute(img2, None)

# Find all stable matches
matches = bfMatcher.match(des1, des2)

# Sort matches by distance (best matches come first in the list)
matches.sort(key = lambda x: x.distance)

# Find index where matches start to be over threshold
for i in range(len(matches)):
    if matches[i].distance > 50.0:
        break
```

```
# Draw good-quality matches up to the threshold index
img3 = cv2.drawMatches(img1, kp1, img2, kp2, matches[:i], None)
cv2.imshow("Matches", img3)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Try the code out on some pictures of your own choosing, both those that match well and those that don't. This code draws all matches where the distance is less than or equal to 50.0. Try varying that number to see how the "good" matches change.

Milestone 6: Finding Objects

For this milestone, I want you to take a picture of something that you have that ORB can find good features on. It could be a sign like the ones I have provided pictures of, or a coin. ORB likes shapes with lots of corners. Next, I want you to build a program that takes images from your webcam, and looks for the object you took a picture of in the webcam frame Use ORB and the Brute-Force matcher. See how well your program can find the object when it is present, and NOT match against other things. You might need to experiment to find an object that works reasonably well.