Image Processing Lab

In this lab, we will be exploring processing images. We will look how to write a color identification script using OpenCV, the importance of taking good pictures, and the benefits and detriments of different size/resolution images.

Note: If you are using Windows and have not set up X11 forwarding via PuTTY, it may be difficult to quickly look at images to debug. Be sure you either set this up, get familiar with SCP commands to copy images to your Windows machine, or ensure your GitHub is working properly.

If you haven't set up any of these options, new images created in the lab should be saved to the images folder and you can send any images you want to view to your personal computer to view there.

Set Up

Clone the ImageProcessing repo, found here:
https://github.mit.edu/BWSI-CubeSat2021/Labs/image_processing

After cloning the repository, make a copy of it so you can edit the code without having to deal with git. Inside the repo, there are some images you'll be playing with and skeleton code.

Activity 1: Color Identification Code

The first thing we're going to do is write code to ID colors. Be sure to

help each other out, and don't be afraid to ask for help.
Remember: for the sake of this lab, we want working code, not perfect code. Your code should successfully identify colors in test.jpg, but it does not need to perform perfectly for every image. You will have time to modify it for the final project.

Part 1: Working off unprocessed images

For the first part, we're going to be looking at images with no enhancement.

Some pseudocode:

- Load the image
- Figure out appropriate lower and upper bounds for each color (use color picker website or analyze pixel numbers in test.jpg)
- Use inRange() to create a threshold image for each color
- Figure out how to find the amount of pixels each threshold identifies for each color
- Store number of pixels identified as each color in the color_amount variable
- Return the percentage of each color

Other things you should do:

- Include a visualization for which pixels the code is identifying as each color. This will make it much easier to tune and debug your code. There is skeleton code on how to do this provided.

To run your code, type:
python3 color_id.py FILENAME True

Where FILENAME is, well, the file name, and True says that you want images of whatever transforms you did on the original image and the masks to show up (only works if you have VNC/X forwarding set up). When the images show up, press any key to get the images to go away. If you

don’t want these images to show up or don’t have that function set up on your pi, first change the folder_path variable at the top of the color_id function to the path to your Flat Sat Challenge folder with your name on it. Passing False instead of True will save these images to that folder. From there you can git push in the Flat Sat Challenge folder and view the images on your computer through Github.

The file test.jpg, when ran, should return equal percentages of each color. Use this to get your code working.

Then, run your code on good.jpg, blurry.jpg, and dark.jpg.

What do you notice about your results? How many false positives and false negatives are you getting in each?

Play around with changing the bounds of your color ranges. Does this help?

Hopefully, this helps you see that taking images when shaky or with bad lighting can make it much more difficult to process them.

Something to consider in the future (NOT for this lab): automating setting the ranges of RGB values by using a calibration image of your plastic in the environment you’ll be imaging in.

Part 2: Saturation, Contrast, Brightness

Sometimes, changing the ranges just isn’t enough. We’re going to try and get some better results.

You can experiment with both the HSV and BGR versions of the original image to change these characteristics of the image. Remember that these

images are basically just numpy arrays and can be edited like any other numpy array.

You can choose to do the following using for loops, but as we learned in the edx python course you can use vectorized operations to make these steps much faster. However, what's more important is that you finish the lab so if you can't figure out a vectorized way of doing these steps, just do a for loop.

Pseudocode:

- HSV Image:
    - To change the **saturation** of an image, you can add a constant to each pixel's S column in the HSV image
- BGR Image
    - To change **contrast**, multiply each pixel by a constant. Make it so the max value a pixel can have even after multiplying is 255. Multiplying by a number greater than 1 will increase contrast, multiplying by a number less than 1 will decrease contrast
    - To change **brightness**, add a constant to all pixel values. A negative constant will decrease brightness and a positive constant will increase brightness. Like for everything else, make it so the max value for a pixel's color channel is 255

Uncomment the part 2 code in the color_id function to actually run the work you did in the part_2 function.

Does each image need the same amount of enhancement?

Activity 2: Capturing Options

For this activity, we will be looking at how changing different capture options changes the size of image files.

You can look at image sizes using:
du –sh filename

Fire up your old FlatSat challenge code OR make a new script to take some images with. You can try taking pictures of the shapes on one of the posters if you want to see how these pictures work with your image processing code.

1. Quality
First, try using the quality feature of the capture command:
camera.capture(“Imagename.jpg”, quality=n)
Save images for n values for 1, 2, 3, 4, and 5.

2. Resolution
A full list of resolutions available on the pi camera can be found here:
https://picamera.readthedocs.io/en/release-1.12/fov.html
Set it using:
camera.resolution(resolution, resolution)
Save images for resolutions 1920x1080, 1280x720, and 640x480.

3. File Type
Take some images in file types other than jpg. Information on how to do that can be found here:
https://picamera.readthedocs.io/en/release1.8/api.html?highlight=capture#picamera.PiCamera.capture

Save an image in:

- jpg
- bmp
- rgb

Once you have all these files saved, we can analyze them. For each, use the spreadsheet you made to calculate downlink time to see how they

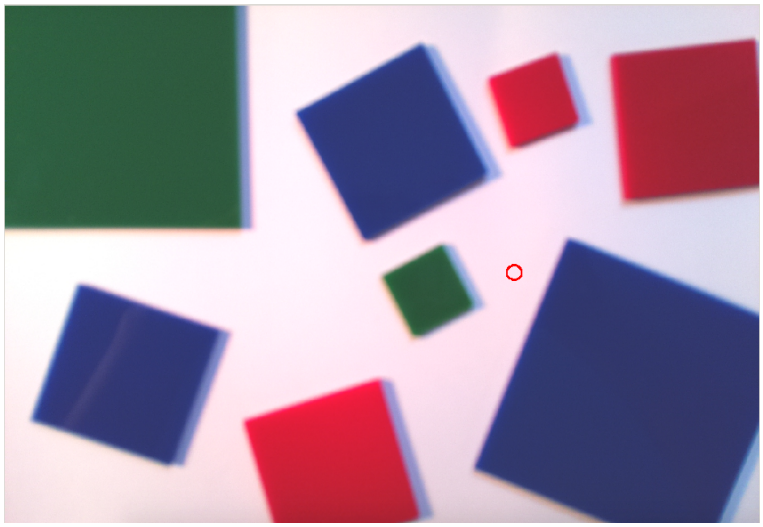
fare.

Now, try running your image processing code on each of these. Notice the tradeoffs between size and ease of image processing: a low quality image may be very small, but it probably doesn't work as well for processing.

We only looked at a portion of options for quality, resolution, and file type. When you have time, play around with more of them and see if you can find a "sweet spot."

Extra Credit (except you don't actually get any points)

Try to figure out how to find the average coordinates of a color in an image.



Example result of finding the average location: Red circle represents what my code identified as the average location of the red pixels in this image (good.jpg). This will be useful to know how to do for when you're identifying the location of penguin guano colonies in the final project.