**NASA Psyche Meteorite Image Analysis System**

**Blob Detection / Feature Recognition in Meteorite Images**

**Conor Yates-Koch**

Table of Contents:

**0.0.1: Drawing an image to the screen using OpenCV in Python3**

This step was fairly simple, following the tutorials laid out online [here](https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_gui/py_image_display/py_image_display.html). Reading the OpenCV documentation helped fill in any gaps.

The overall process goes as follows:

1) Create a window

2) Read an image in

3) Draw the image to the window

A basic menu class was implemented to handle primitive interaction with the system, but most of the functionality has yet to be implemented. So far the ability to re-print the menu and quit the program are present. Planned functions for the future will likely include toggling visibility of what the system sees as detected blobs, scanning through / selecting individual blobs, finalizing the extraction of blob features (to be passed on to the next step in the system).

**0.0.2: Detecting “blobs” in the image**

*As a potential alternative to blob detection, I’m considering looking into contour detection and edge detection.*

Originally, [blob](https://www.learnopencv.com/blob) [detection](https://www.learnopencv.com/blob-detection-using-opencv-python-c/) did not seem to be working properly, as none of the keypoints were being drawn to the modified image “Meteorites with Keypoints”. After some evaluation, I discovered that the issue was having a threshold max size that was too small. By using setThreshold(params,1,1,100) and setAreaRange(params,1,2500) began drawing the keypoints to the screen (recall that the parameters are ordered as ‘params, min, step\_size, max’). setThreshold(params,1,1,100) seems to work well for picking out keypoints along the edges of the desired mineral blobs, and does well to ignore the metalic ‘background’ of the meteorite. However, this is not the desired result. The individual keypoints are still incredibly small, each identifying only a small portion of the desired blob. They just happen to fall quite reliably along the edges of each blob. Simply messing with the minimum and maximum area do not solve this issue either. I believe a combination of color filtering, area filtering, and the proper threshold range and stepsize will be sufficient to identify most blobs.

In future, it may be valuable to identify only the substantial blobs (relatively large on the image), and then search nearby for smaller groups of pixels that are close in distance and close in the color space to the parent blob. These small blobs are often only a few pixels in area (often times less than 20 pixels) and would not be useful in our current training model (that is not to say that they cannot be useful in another model). By leaving these out, it would help accelerate the training of the AI. These blobs could then be labeled the same as their parent blob for the purposes of calculating the final mineral composition. It may also be possible / reasonable to leave tiny blobs out of the calculations entirely, as they may not affect the final composition calculation much anyways.

**0.0.2b: A note about image backgrounds**

So far, there does not seem to be any tangible difference in the performance of the blob detection on black, white, or clear backgrounds. All seem to produce identical results. As long as the background is completely consistent, and is either clear (in the .png format), or is high in contrast from the meteorite, the algorithm seems to do a fine job of ignoring it.

**0.0.3: Progress on Blob Detection**

After some more tinkering, I’ve discovered that importing the image as greyscale helps thresholding work in a much more familiar way. On a greyscale image, each pixel is assigned a value of 0 to 255 depending on how dark or bright it is (black being 0). In this way, thresholding functions as a brightness filter. Using setThreshold(params,1,1,110), the blob detector will reliably “fill” in the mineral areas with circles. This is still not the intended behavior, but is a step forward. There are a few issues, however. First, and most critical: by importing the images to greyscale, it is obvious that color information is lost. It may be the case that the classification AI needs (or would perform significantly better given) the color information. Second: using the threshold with absolute values in this way makes the detection process extremely sensitive to change. If the images are taken with a camera using a different white balance, or if the lighting is different, values on the edge of the threshold may not be reliable.

I can think of two ways to approach these problems currently. First, the issue with losing color information: it may be possible to simply detect the locations and boundaries of the blobs on a greyscale image, and then pass the location information to another window which holds a color version of the same image. Blobs would then be taken from and analyzed using the color image, so that color information may still be present further down the line.

Second, when dealing with sensitivity to absolute pixel values, it may be possible to perform some pre-processing function(s) on the images such that small variances can be controlled for. Alternatively, it may be possible to identify a blob and average the values of it’s pixels, setting this value as a preliminary base-line. The threshold would then be applied based on some pre-defined range (ex. min = base-line – 15, max = baseline + 15). Additional pixels would then be added to the blob based on whether or not they fit this threshold range, and the base-line would be updated for each new pixel added.

Ultimately, I would like to figure out some way to use relative values rather than absolute values, so that the system is not so sensitive to the specific values of the pixels, but rather to the value of a pixel as compared to its neighbors.

I have set up an appointment with Dr. Jorge Caviedas, a professor experienced in using OpenCV, to speak with him about other potential solutions, and to gauge my progress so far. I also intend to set up an appointment with Dr. Ryan Meuth, an assistant Psyche capstone advisor and computer vision expert, to discuss similar topics.

**0.0.4: Meeting with Dr. Caviedas**

Dr. Caviedas has suggested using Canny Edge Detection in order to identify mineral regions. He has also suggested looking into a smoothing filter, particularly median shift. This will make the image fairly uniform by replacing each pixel with the median of the pixels around it. This gets rid of small insignificant details that may throw the detectors off, and make blob / edge detection cleaner. An advantage median shift has over mean shift is that median shift will always produce color values that existed in the original image, where mean shift often creates very similar, but slightly different color values. In OpenCV, Canny Edge Detection has a built in 5x5 Gaussian blur that occurs before edge detection automatically.

He has also made me aware of Grouping and Merge functions that go along with SimpleBlobDetector, that could help create larger blobs out of the smaller individual ones.

**0.0.5: Meeting with Dr. Meuth: Dropping Blob Detection, Considering KNN**

After going over [the documentation](https://docs.opencv.org/3.3.1/d0/d7a/classcv_1_1SimpleBlobDetector.html) in more depth with Meuth, we’ve come to the conclusion that the OpenCV blob detection function may not be powerful enough for this given application, or at least that trying to make it work is more effort than reasonable. We tinkered with several of the additional parameters including merging and grouping parameters, but nothing seemed to take us a step in the right direction. I had imagined that the algorithm would behave in a similar way to the magic wand function in Photoshop, but this is not the case. For the time being, I will be shelving blob detection. Meuth, like Caviedas, suggested edge detection and / or contouring, and so I will be moving on to those in the next iteration.

Meuth also proposed another possible, very interesting solution in the way of KNN (K-Nearest Neighbors). KNN is an algorithm that functions as a classifier by identifying the k-nearest points (or centroids) to a given point (or centroid), and identifying the new point based on the “majority vote” of the k-nearest neighbors. For this application, that would involve placing each pixel in the meteorite’s image into a 3-dimensional color space (defined by it’s RGB values in place of XYZ values). The idea is that pixels from the same types of mineral inclusions will exist close to each other in this color space. Once a sufficient color-space analysis has been performed, where each color-space cluster has been assigned some preliminary label (even by hand), KNN can begin to identify the “class” of each pixel in a given image.

As Meuth and I discussed, this solution will likely get us 80% of the way there (assuming edge detection / contour detection is not satisfactory, or given that I have enough time this semester to implement it as a personal experiment). The remaining 20% will be attributed to finding the correct value of K for the algorithm, and preserving spatial information.

First, the consequences of a value of K that is too high would mean that two or more separate clusters’ centroids in color space could be bounded by the same region, leading to obvious inaccuracies. This is the easiest of the issues to spot and can likely be solved simply by just decreasing the K value.

However, the consequences of too low of a K value may be more difficult to identify. It is possible that KNN will attempt to identify centroids of clusters that shouldn’t (or don’t) exist. In essence, it’s trying to solve an equation for a variable doesn’t exist. This could cause problems by dividing clusters in color space between multiple centroids that should otherwise be bounded by one. This could lead to another problem, spatially convoluted clusters.

Pixels in a cluster in color-space are independent of their position on the image. This means that within a single cluster, there will be multiple completely separate mineral inclusions (separate in space in the 2-D image). Should KNN divide such a cluster amongst 2 or more centroids, that could mean that within a single mineral inclusion, some fraction of pixels are identified correctly while some fraction are identified incorrectly. This effectively leads to each mineral inclusion having “holes” of incorrect labels. It may not be enough to simply give all of the nearby centroids in color space the same label (though, ironically, k = 1 (the most centroids possible) provides the least possible error as a given data set approaches infinity. This is not generally computationally efficient, though it is noted that values of K do tend to be lower rather than higher (e.g. more rather than fewer centroids)).

However, this is where machine learning, in the form of Neural Networks, could provide a robust solution without necessarily having to find the “perfect” value of K for this given data set. Say for example that Silicates (dark gray) are split between two centroids. One centroid (which I will call A) identifies Silicates fairly well, but some of Silicate’s pixels are too dark and fall outside A’s bounding region into the second centroid’s region, B. B identifies Graphite (black) pixels very well, and tends to ignore the Silicate pixels that spilled over. In this instance, Graphite would be easily identified, but Silicate regions would often turn up with small regions of wrongly identified Graphite within their bounds. By training a neural network on arbitrary labels (such as A and B), the network would learn that Silicate pixels most often fall within A, but some percentage may be identified as B. Thus, in this case, Silicates would be easily identified by a neural net using the ratio of A pixels to B pixels that appear within it’s bounds on the 2-D image.

**0.1.0: Moving on to Edge and Contour Detection**

Currently I feel I have put enough effort into blob detection without significant results that it’s time to move on and explore other solutions.

In exploring [contour detection documentation](https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_contours/py_contours_begin/py_contours_begin.html), I have found that it works best on binary images (i.e. the resulting image after a threshold or edge detection has been applied). This means it is necessary to apply at at least one of OpenCV’s thresholding or edge detection functions, if not both. I have decided to go with adaptive Gaussian thresholding first. What is appealing about this thresholding function is that A) it works very well for images that have varied lighting across the image and B) it does not function on an absolute threshold value, but rather adapts (quite well, I might add) to each individual image.

**0.1.1: Meeting with Cassie**

We met as a team briefly to speak with Cassie Bowman. The primary goal of the meeting was to acquire as many training images as we could to test and experiment with, or to at least get a request for some into the project pipeline. As of right now I am using photos taken from my phone that had the backgrounds removed via Photoshop. Cassie pointed us to some online collections of images, which may help in the mean time.

Another meeting topic was to clarify our deliverables for the end of the semester. Given the nature of the Psyche mission, this project will be handed off to another capstone group in Fall 2018, and development on MIAS will continue in this fashion for the foreseeable future. Given that we are the first step in a many year development process, our sponsor is primarily interested in our documentation, rather than a functioning system. While a working prototype would be ideal, what will help subsequent groups most is not having to perform for a second time all of the research we conducted.

Finally, we simply updated each other and Cassie on our current progress with the project.

**0.1.2:** **Success of Adaptive Thresholding vs Edge Detection**

Initial testing with [Adaptive Gaussian Thresholding](https://www.tutorialspoint.com/opencv/opencv_adaptive_threshold.htm) has been extremely promising. Adaptive thresholding works on the assumption that smaller sub-regions of an image are more likely to have uniform lighting conditions, and are thus more suitable for thresholding. I’m not completely certain how the OpenCV function calculates the threshold value for each of these sub-regions, but some popular methodologies include evaluating the histogram produced by the pixels within the region, setting the threshold equal to the mean value, the median value, or the mean between the max and min values in the region.

Edge detection has proven much less promising, ironically. This is due to the fact that the edge detection functions I found still used absolute values, rather than relative values. Given the erratic and random nature of the color and lighting values in the meteorites, this poses a significant challenge to these edge detection functions. It is important to note that this does not mean that edge detection as a whole is unfit for this task. An adaptive edge detection function that works similarly to the adaptive thresholding may prove even more powerful and useful, but I have not found one. For the time being, adaptive thresholding seems to be the most robust option, able to handle all the variance in pictures taken from my cellphone, so I image it will perform just as well on professionally taken photos in a controlled lighting environment.

Given that both adaptive thresholding and Canny edge detection are thresholding functions (in that they both produce a binary image as output based on a value filter), I do not see any reason to attempt to mix adaptive thresholding and Canny edge detection.

**0.1.3: Contour Detection with Adaptive Thresholding**