Ian Hudis
ECE 448
12/19/19

Final Project: Feature Detection and Matching Using ORB

-Abstract

The goal of this project was to design a fast-Oriented Rotated Brief (ORB) feature detector in MATLAB that can be used to find features in an image. This process started with a FAST to find edges as well as a Harris Detector to locate corners. Then these points were oriented to find coordinate angle data to calculate the approximate angle of each corner. From here, a rotated brief was used to find the key feature points in an image. The process of finding features was used to make a matching algorithm to match orb points from two separate images. This process is efficient when it came to performing feature detection. The feature matching system also has multiple applications such as object detection as well as image threading. The results when the algorithm was tested for both applications showed a surprisingly small amount of error in the results.

-The Fast and Orientation

The ORB process started by taking the FAST (Features from Accelerated Segment Test) points of an image. This specific process was primarily for edge detection. The algorithm started by combing through an image pixel by pixel and pulling out which pixels should be center pixels (I_p) using intensity thresholds and spatial conditions ("Tyagi"). Fast worked by taking part of an image in a given radius to the center pixels (In this case the radius is nine pixels.) and placing the pixels into three separate categories. These categories are brighter (b), darker (d), or around the same as the central pixel I_p (s).

$$S_{p\to x} = \begin{cases} d, & I_{p\to x} \le I_p - t \quad \text{(darker)} \\ s, & I_p - t < I_{p\to x} < I_p + t \quad \text{(similar)} \\ b, & I_p + t \le I_{p\to x} \quad \text{(brighter)} \end{cases}$$

Figure 1: Fast Categories ("Mordvintsev")

By using a given threshold "T" (In this case T = 8.) and a decision tree based on which categories and location the pixel is in, the FAST algorithm can determine the edges by seeing if there are T number of darker pixels and T number of lighter pixels in a given radius (Figure 2).

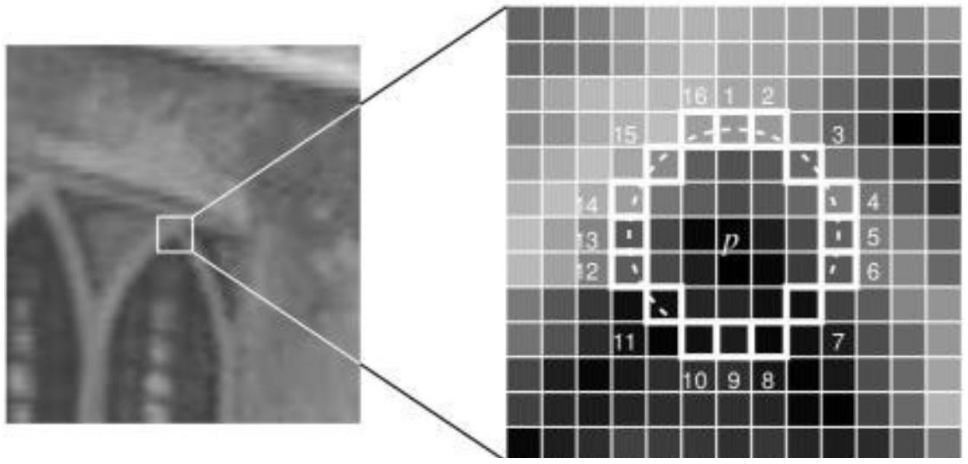


Figure 2: Fast Circle on a Corner (“Mordvintsev”)

The FAST process initially lacked multiscale features as well as orientation component. For multiscale, ORB can be thought of as using a multiscale image pyramid (Figure 3). The image pyramid was a visual representation of an image with images on top of it each with different resolutions. The higher a level was on a pyramid, the more down-sampled the image is from the original. Therefore, much larger images will take longer go through the FAST process and be oriented due to needing more pyramid layers. The FAST algorithm while it looked for points, did so from each level in order to effectively locate key points at a different scale and quantization. This is what allowed ORB to be partially scale invariant. This means that the feature detection was generally able to recognize more features between two images if the data was not too quantized.

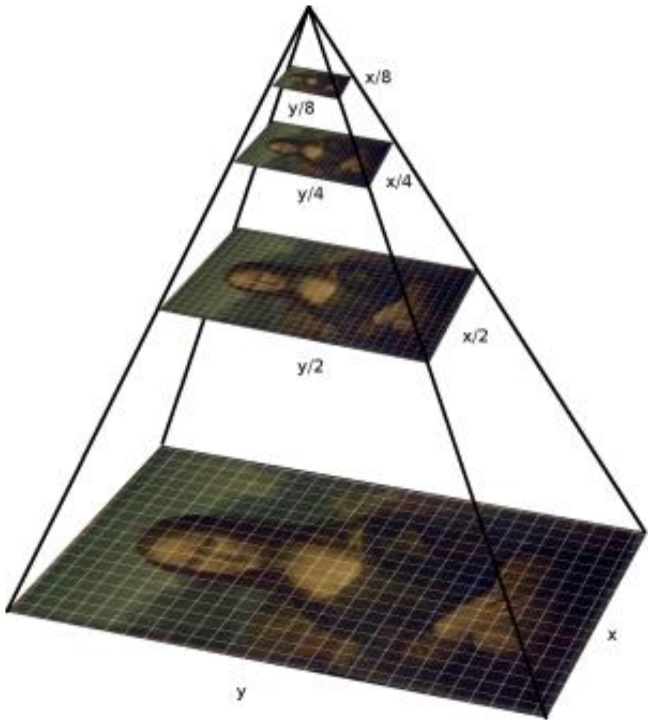


Figure 3: Multi-scaling Pyramid (Taken from 5)

```
function [NonMaxFastValues,fastScale] =
FAST(InputImage,corners,fscore)

% image size
[y,~]=size(InputImage);

% get vectoral indices for corners
pixel = zeros(size(corners,1),1);
for n = 1:size(corners,1)
    pixel(n) = (corners(n,1)-1)*y + corners(n,2);
end

% get indices for surrounding pixels
KernelSize = 5;
kernel = KernelSize*2+1;
KernelVector = zeros(kernel^2,1);
%redistribute kernel
for i = 1:kernel
    for j = 1:kernel
        KernelVector((i-1)*kernel + j,1) = j-6 + (i-6)*y;
    end
end

% create fast score map
ScoreMap = zeros(size(InputImage));
ScoreMap(pixel) = fscore;

% initiate non maximum suppression
CronerMax = zeros(size(corners,1),1);

for i = 1:size(corners,1)
    surround = pixel(i) + KernelVector;
    CronerMax(i) = (sum(fscore(i) >= ScoreMap(surround)) ==
kernel^2);
end

NonMaxFastValues = corners(logical(CronerMax),:);
fastScale = fscore(logical(CronerMax));
end
```

Figure 4: FAST MATLAB Algorithm

Besides the multi-scaling component, another key characteristic was being able to find angles so the ORB could recognize if the image has been rotated or altered. That is why an Orientation function was necessary. Orientation is the process of using corners to find angles between points. A problem with FAST by itself was that it did not reveal where the corners in an image were located. Therefore, a Harris corner detection algorithm was implemented to do this task ("Ethan"). Having several key corners points was useful since it made it the Fast points cherrypicked so that only the critical pixels were taken for further processing. Once the corners were found, they can be used to find the angles that were necessary for the rotated brief. Finding intensity centroids is the key to performing orientation. This assumed a corner's intensity is offset from its center, and the coordinates could be used to impute an orientation. Finding the moments needed can be shown as:

$$m_{pq} = \sum_{x,y} x^p \cdot y^q \cdot I(x,y)$$

This data can be used for a center of mass which in this case is used to find the centroid.

$$C = (\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}})$$

From here, these moments to calculate the angle of the corners:

$$\theta = \mathrm{atan2}(\mathrm{m}_{01}, \mathrm{m}_{10})$$

(“atan2” is the quadrant aware version of arc tangent.)

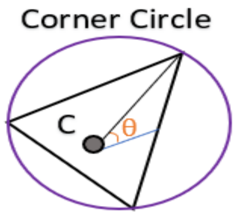


Figure 5: Visual Representation of Orientation

A MATLAB version of orientation can be seen below (Figure 6).

```
function angle = orientation(InputImage,coordinates)
% transpose to compute angle
radious = 3; % FAST corner detector's default radius
InputImage = double(InputImage);  % converts column major to row
major for efficiency.
m = strel('octagon',radious); mask = m.Neighborhood; % FAST search
mask
Ip = padarray(InputImage,[radious radious],0,'both'); % padding

r = size(mask,2); % find mask size
c = size(mask,1);

angle = zeros(size(coordinates,1),1);

for i = 1:size(coordinates,1)

    vert = 0;
    horz = 0;
…
```

```
    …
    Rinitial = coordinates(i,2); %initializes corners
    Cinitial = coordinates(i,1);

    for j= 1:r   %find the vertical and horizontal dist between 
Corner pixels
        for k = 1:c
            if mask(k,j)
               pixel = Ip(Cinitial + k-1, Rinitial + j-1);
               vert = vert + pixel * (-radious + k - 1);
               horz = horz + pixel * (-radious + j - 1);
            end
        end
    end
    angle(i) = atan2(vert,horz); %Uses vert and horz to find angle
end
end
```

Figure 6: MATLAB Orientation Algorithm

-The Rotated BRIEF

A BRIEF in image processing is a description composed of a string of binary values. In this case the string was a 256 by 256 bit-map. Briefs start by smoothing image using a Gaussian mask to prevent the descriptor from being confused by high-frequency noise. From there, the brief would choose a random pair of pixels in a patch* around key points of an image ("Mordvintsev"). These patches were computed by a Gaussian distribution in order to predict what the pixels of interest were base off a given set of input data.

However, the problem with normal briefs is that they are computationally expensive and slow in performance. They are also not rotationally invariant so it was better to use an algorithm called a Rotated Brief. A Rotated Brief takes the angles that were found in the orientation process from earlier and then used them to find the rotations at which the input images' corners were located. For each corner, two rotational brief estimations were found with the same input angle. From here, the Rotated Brief takes the pixel data found in between the inner and outer rotated briefs and outputs these pixels as ORB key points. This process can be seen in Figure 7. By using these angles, the rotational brief could pick out key points of an image without having to scan every single patch of pixels which made the process significantly more efficient.

* A patch is a predefined neighborhood of pixels usually in the shape of a square.

```
function features = rotatedBrief(InputImage,corners,patterns,angle)

% initialize features
features = zeros(size(corners,1),256);

for a = 1:size(corners,1)
% Compute rotation based on corresponding angles and patterns
    Rotation1 = round( [ cos(angle(a)), -sin(angle(a));
sin(angle(a)),cos(angle(a))] * patterns(:,1:2)')';   %inside rotation

    Rotation2 = round( [ cos(angle(a)), -sin(angle(a));
sin(angle(a)),cos(angle(a))] * patterns(:,3:4)')'; %Outside rotation

% Use Rotations to take the BRIEF
    for b = 1:256
        p1 = InputImage(corners(a,2) +  Rotation1 (a,2),corners(a,1) +
Rotation1 (b,1)); %inside rotated brief
        p2 = InputImage(corners(a,2) + Rotation2(b,2),corners(a,1) +
Rotation2(b,1)); %outside rotated brief
        features(a,b) = double(p1 < p2); %Grab the points between the inside
and outside briefs
    end
end
end
```

Figure 7: Rotated Brief MATLAB Algorithm

-The ORB Detection Algorithm

By using the FAST, Orientation, and Rotated BRIEF processes together, an ORB detection function was be made that can take just about any image and locate its key features. The FAST and Harris process finds the edge points and the scaling pyramid of the image which allowed for the process to be scale invariant. The orientation inputs data from the FAST points and the Harris Corners and used it to compute angles making the process rotationally invariant. Finally, the rotated Brief takes each corner's location and the angles and uses it to compute a Rotated Brief which finds the key features necessary to get the ORB points. The flowchart of this process (Figure 8) as well as a MATLAB code version (Figure 9) can be seen below.

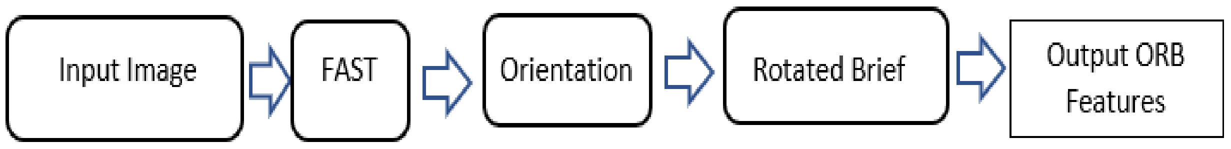


Figure 8: General Flowchart of ORB Feature Detection Process

```
function [Orbpoints, cornerID] = OrbSearcher(InputImage)
%This Looks searches for orb points using fast orientation and rotated
briefs.

%converts the image to greyscale
[~, ~, numberOfColorChannels] = size(InputImage); %checks to see if image
is greyscale
if numberOfColorChannels > 1
GreyInputImage = rgb2gray(InputImage);
else
    GreyInputImage=InputImage;
end

% extract FAST corners and its score
[corners, score] = FindFast(GreyInputImage,20, 1); %takes scores and
corners
  [corners,~] = FAST(GreyInputImage,corners,score); %uses scores to
perform fast


% compute Harris corner score
HarrisInput = harris(GreyInputImage);
harris1 = HarrisInput(sub2ind(size(HarrisInput),corners(:,2),
corners(:,1)));

% refine FAST corners with harris scores
[~,Index] = sort(harris1);
cornerID = corners(Index(1:size(Index)),:);

% get orientations (angles) for the selected points
angles = orientation(GreyInputImage,[cornerID(:,2),cornerID(:,1)]);

% compute rotational BRIEF
sample=briefpoints;
Orbpoints = rotatedBrief(GreyInputImage,cornerID,sample,angles);
end
```

Figure 9: MATLAB ORB Algorithm

This process was first tested on a 256 by 256-pixel test image in order to see if the ORB algorithm would be able to find key features of the image (Figure 10). The results appeared to be promising since the red dots that marked the Orb points managed to locate random corners from all over the image. For comparison purposes, this test image was also run through the ORB process on MATLAB's image toolbox to see if it would get the same results and found that most of the same orb points matched. The only major difference was that the toolbox version had an undocumented method of plotting the points with circles with radii where size was determined by the angle of the corner. However, the information from both processes outputted roughly the same information.

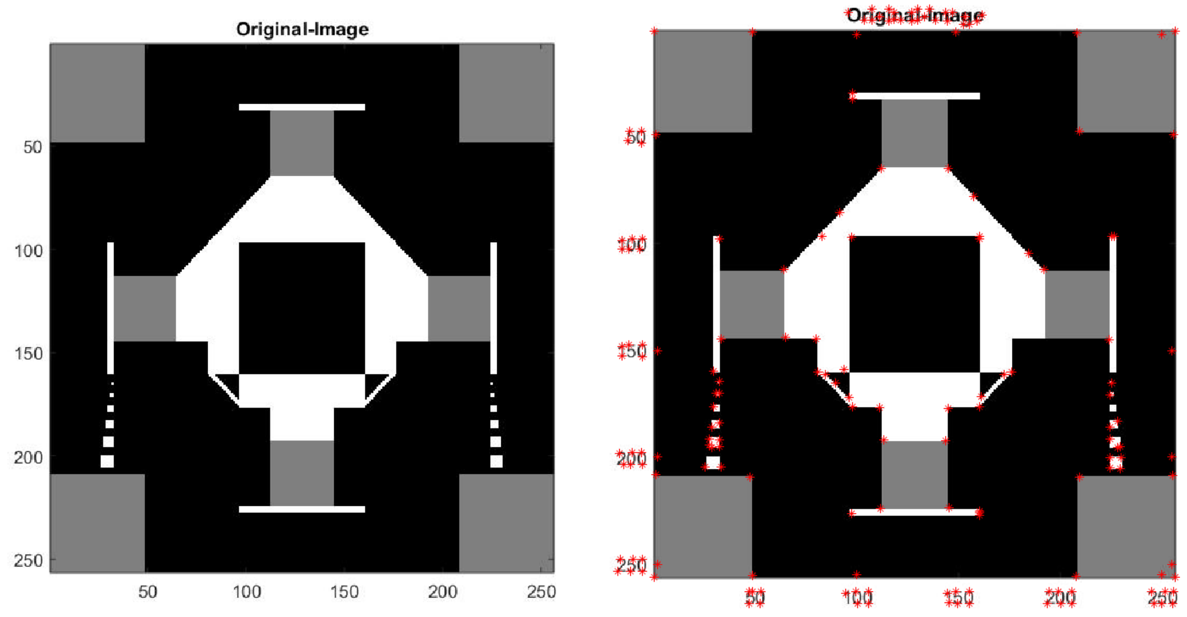


Figure 10: Applying ORB Detection onto a Test Image

Besides the test image, the ORB function was also tested on the Cameraman image to see how well it could find ORB points on a real image (Figure 11). The red dots once again can locate random corners of the image and find the angles of the corners. An observation made was that the process ran much fast when an image was quantized or has less pixels assuming the image pixel dimensions were always by a power of two.

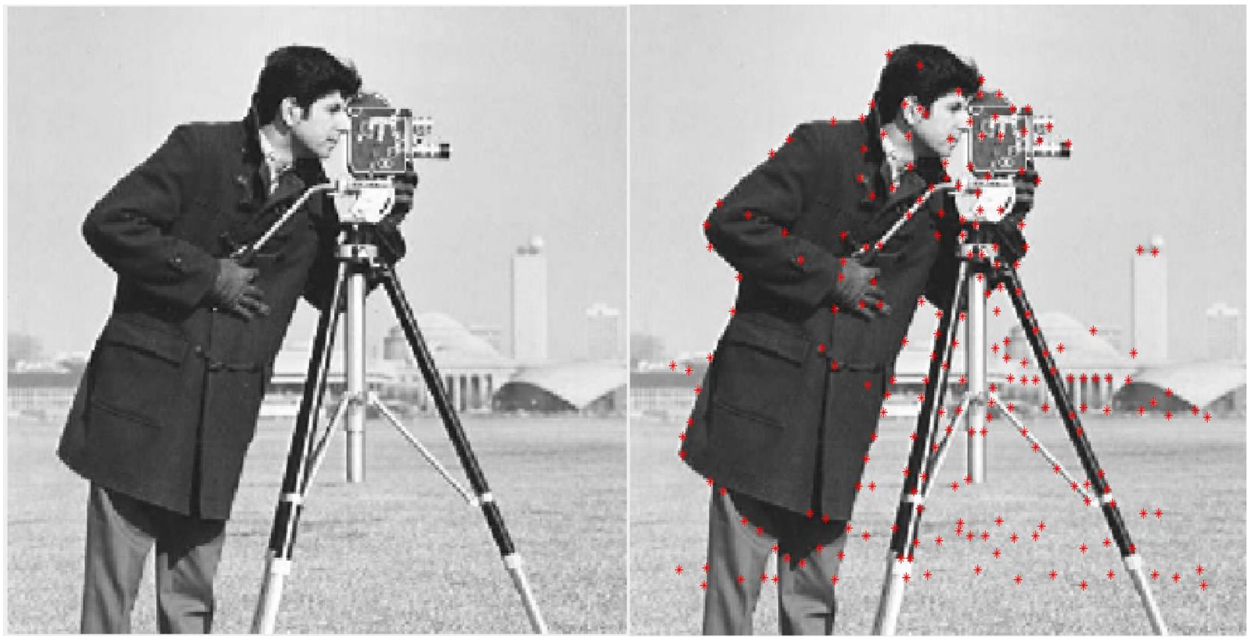


Figure 11: Applying the ORB Detection to the Cameraman Image

-The ORB Matching algorithm

In order to further test how well the ORB detection works, an ORB point matching function was made in order to be able to compare two different input images. The process was that two images go through the ORB Algorithm and then from there they are both entered into the function below (Figure 12). This was convenient for seeing which orb points from both images roughly match up. This shows an image display that ends up showing drawn lines from one image to another. Although a small portion of these lines were often off the mark, most of the matching lines pointed to the same features in the other image.

```
function [matches,warpedImage] =
OrbFeatureDetection(InputImage1,InputImage2,brief1, brief2,corner1 ,
corner2)
% Matlab operation to find matches
[index1,dist1]= knnsearch(brief2,brief1,'K',2,'Distance','hamming'); % knn
distance for descriptor 1
[index2,~]= knnsearch(brief1,brief2,'K',2,'Distance','hamming'); % knn
distance for descriptor 2

matches = zeros(size(brief1,1),4);
for i = 1:size(brief1,1)
    if (dist1(i,1) <= 64/256 && dist1(i,1)/dist1(i,2) <=0.98 && i ==
index2(index1(i),1))
        % hamming distance < 0.25 + ratio between smallest and second
        % smallest < 0.98  and cross minimum value check
        matches(i,:) = [corner1(i,:),corner2(index1(i,1),:)];
    end
end
% matches found
filter = find(matches(:,1));
matches = matches(filter,:);

% matched points (note that there are several outliers since not all
points will match.)
feature1 = matches(:,1:2);
feature2 = matches(:,3:4);
end
```

Figure 12: MATLAB ORB Point Matching Algorithm

Below is a MATLAB function built in order to efficiently show out the full results of the ORB Feature detection process (Figure 13).

```
function output =
showresult(image1,image2,feature1,feature2,corner1,corner2,matche
s,newimage)
%this was a program for printing test results quicker

figure(1);
image(image1) %show image1
set(gcf, 'Position',  [100, 100, 256, 256]) %controls image size
hold on;plot(corner1(:,1),corner1(:,2),'r*')  %show image 1 orb
points
title('What we Are Looking For');
figure(2);
image(image2) %show image2
set(gcf, 'Position',  [100, 100, 256, 256]) %controls image size
hold on;plot(corner2(:,1),corner2(:,2),'r*') %show image 2 orb
points
title('What we Are Looking Into');
figure(3)

newImg = cat(2,image2,image1);
imshow(newImg)
hold on
plot(feature2(:,1),feature2(:,2), 'g.')
plot(feature1(:,1)+size(image2,2),feature1(:,2), 'r.')
for i = 1:size(matches,1)
    plot([matches(i,3) matches(i,1)+size(image2,2)],[matches(i,4)
matches(i,2)],'b')
end
title('The Matched feature between the two images')

output = image1;

figure(4); % warped image
imshow(newimage);

end
```

Figure 13: MATLAB Function Used to Display Results

Once this code was written, the test image from earlier was used to make sure the matching function worked correctly. It was tested on itself to make sure every ORB keypoint would be matched up. As can be seen from Figure 14, this is exactly what happened. It was clear from these results that the algorithm was able to recognize if two images are the same.

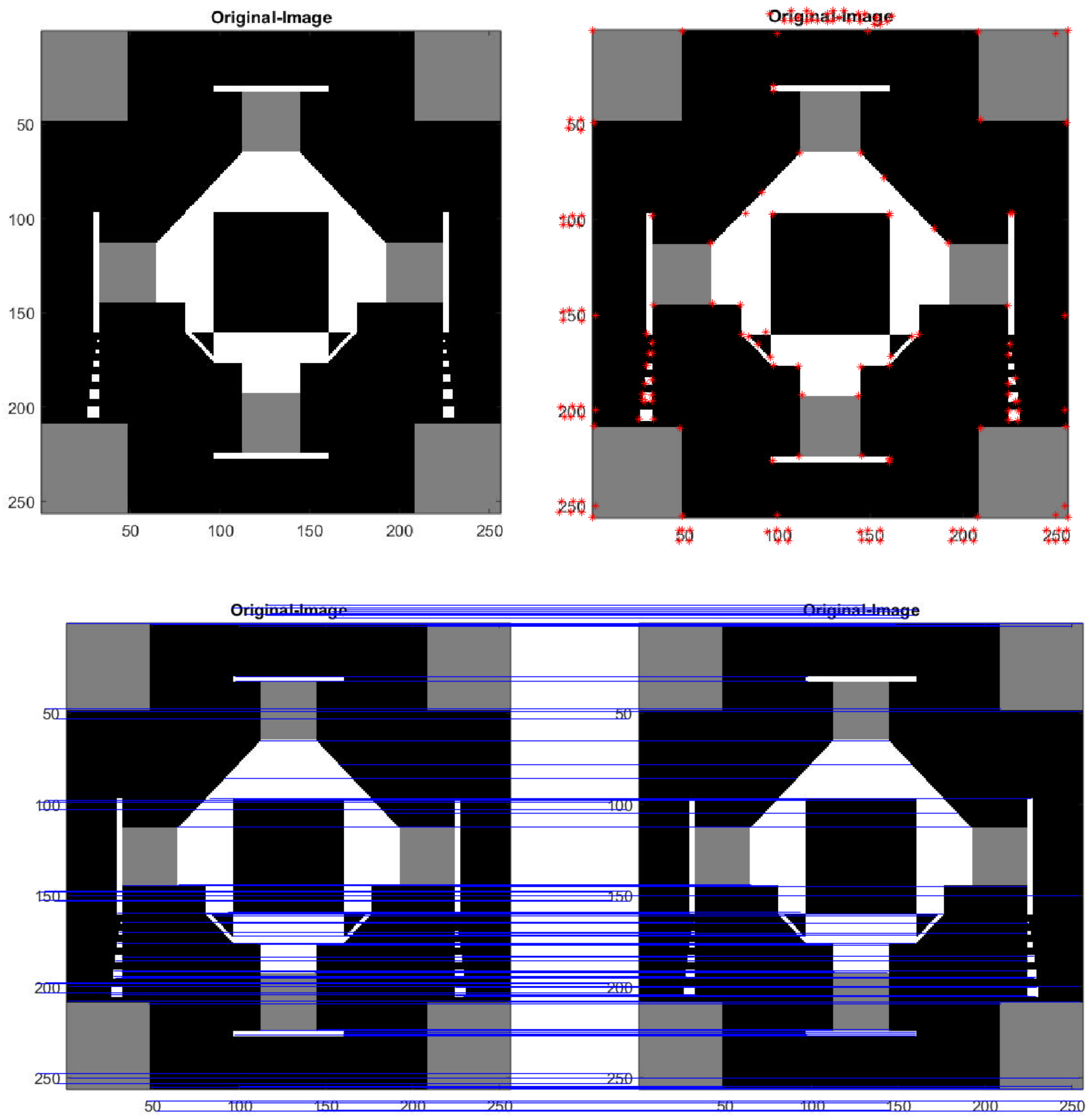


Figure 14: Testing the Matching Algorithm seeing if the same image matches.

Next was to see what happened when the feature detection was used between the original test image and an altered version of the same image. This altered test image (Figure 15), made is Microsoft Paint, was meant to test if the feature detection can still recognize an object in another image despite being shrunken down and not the only object in the image. The results of the altered image test can be seen below (Figures 16 and 17). Most of the blue matching lines that went toward the object go to the other object being detected in the image. This means that the ORB function could successfully locate objects in an image with a small degree of error.

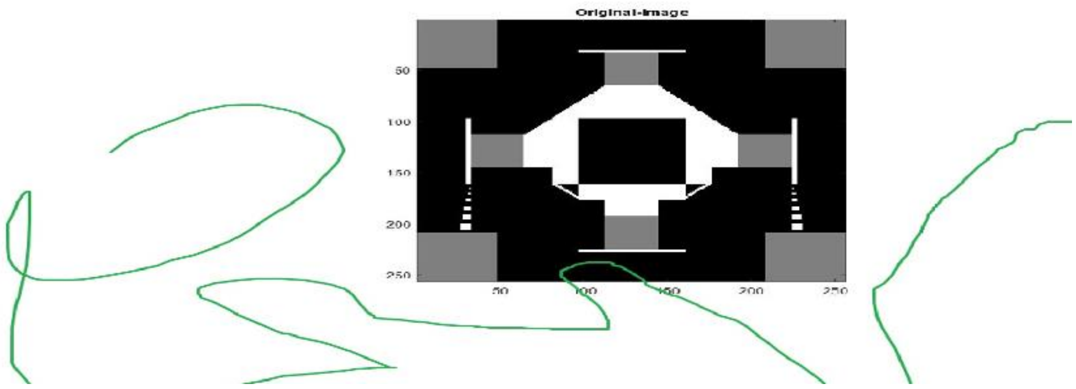


Figure 15: Altered Test Image

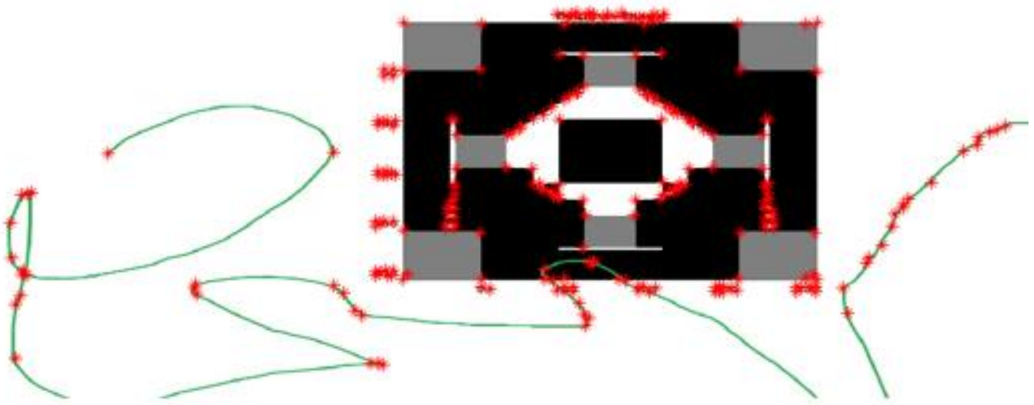


Figure 16: Altered Image with ORB Points.

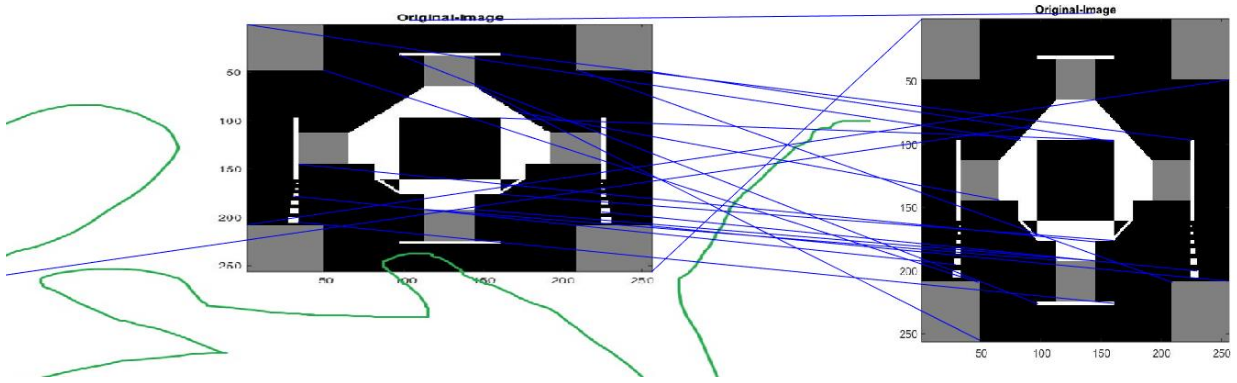


Figure 17: Matching the ORB points

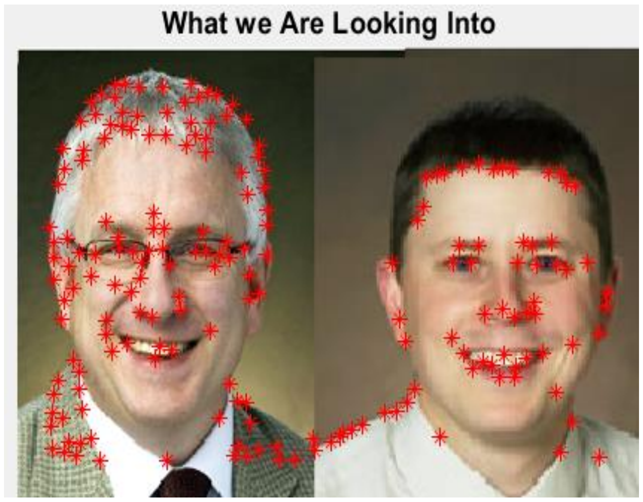
A few of the lines are incorrect because the test image had vertical symmetry and that can confuse the ORB matching function.

-Some Applications of ORB

A real-world example of how ORB can be used is in facial detection. For example, if a machine had video footage of Dr Roger Green (Figure 19) and wanted to be able to identify him. The machine would do by first taking the ORB points of both images (Figure 20). And then seeing which of the ORB points roughly match up (Figure 20).

Figure 18: Professor Index

Figure 19: Zoomed In Camera Footage



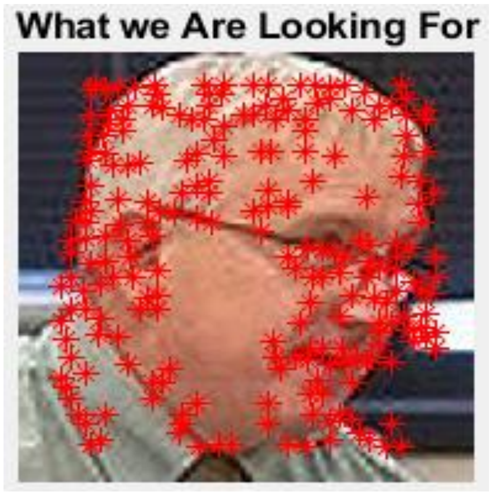


Figure 20: Finding ORB Features

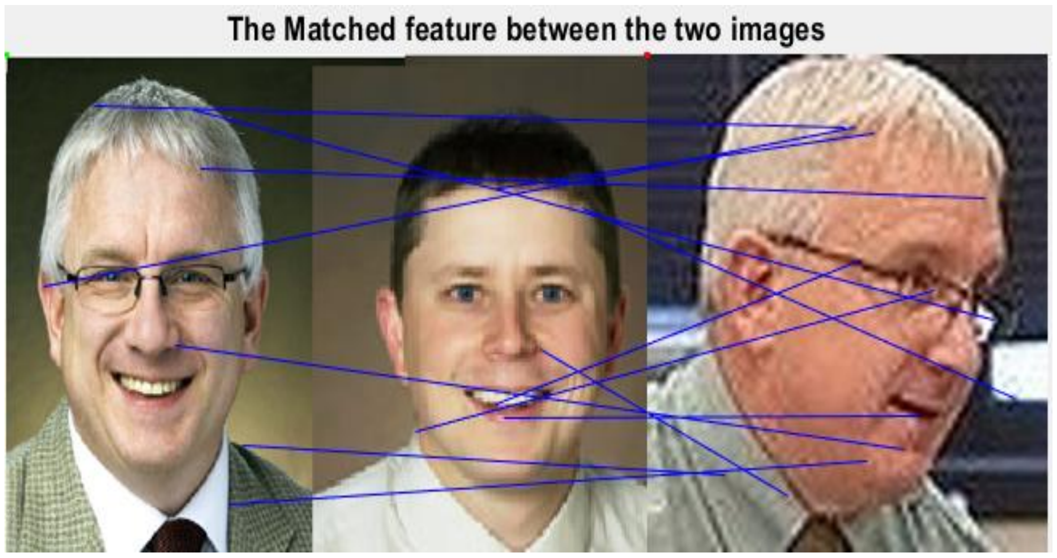


Figure 21: Matching ORB Features

Although a few of the lines did not match up, the machine would be able to identify Dr. Green with a high probability base on the number of features that matched up. Since the lines can sometimes be misleading, for visual purposes, below there was a stitched image made from combining the images together (Figure 22). On close inspection, almost all the distortion in this image was on the side of Dr Green's face. This showed that the machine would be able to identify that, with high probability, that it was most likely looking at one of the sides of Dr Green's face. In theory, if the machine had more pictures of Dr. Green from different angles and distances, it would be able to identify him more accurately and precisely.

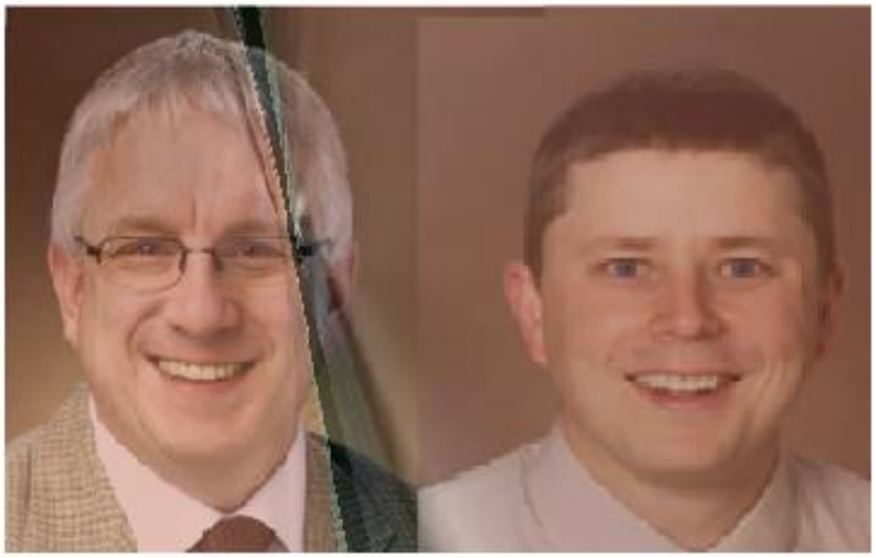


Figure 22: Image ORB Stitching based on ORB Features

Besides Feature detection, ORB can also be used for filtering. For example, the Cameraman image from earlier can be visually improved. On inspection, one of these images is dark and blurry (Figure 23) while the other one is an oversaturated (Figure 24). However, when matched and combined they became the image in Figure 25. This was the result of Image stitching. What happened was that the parts in the image that the ORB feature recognized in Figure 24 were combined with Figure 25 while the features in Figure 24 that were not recognized were ignored. This is an effective way of balancing the greyscale histogram of an image without losing any important details.

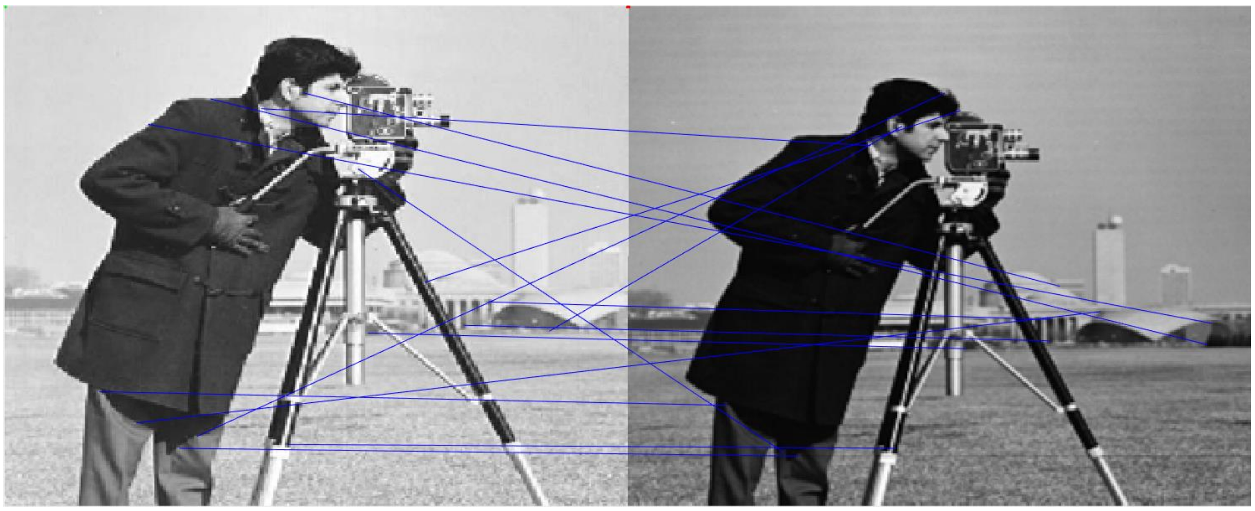


Figure 23: Oversaturated Cameraman   Figure 24: Dark and Blurry Cameraman

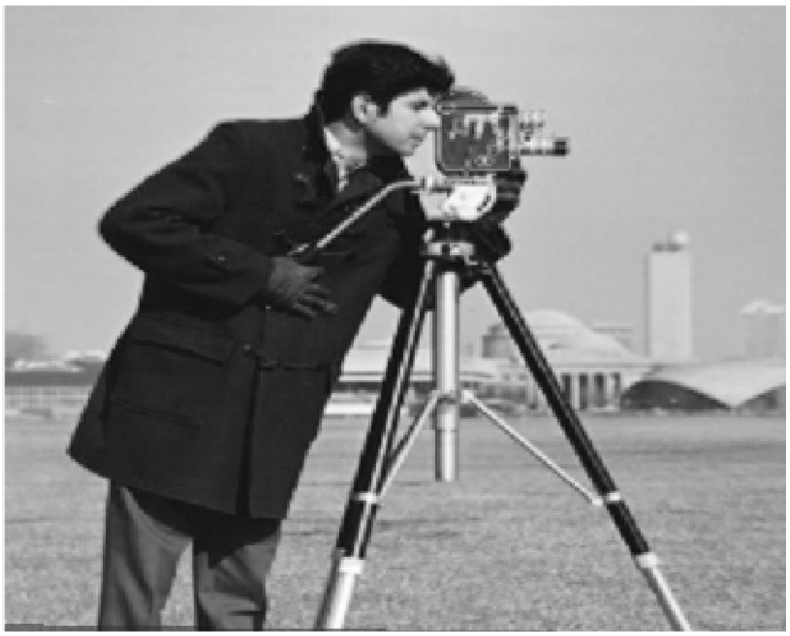


Figure 25: Image Stitching Result

-Conclusion

ORB feature detection was a useful tool that took advantage of computed FAST features and angles in order to efficiently orientate and BRIEF an image. This made it easier for processors to perform this method compared to most other advanced feature detection algorithms. Therefore, it takes less computation time. Once ORB points are found for two or more images, they can be compared in order to find matches. These matches allow for feature detection which can be used for identifying objects or people in an image. The facial detection from professor pictures showed that the process is an effective method for machine comprehension of images. For future tests, it would be interesting to see if the margin of error lowers if there are more images of the object being compared when going through the matching process. Regardless, ORB is an effective method for searching for image features.

-References

- Ethan, R., Rabaud, V., Konolige, K. and Bradski, G. (2017). *http://www.willowgarage.com/sites/default/files/orb_final.pdf*.

- Mordvintsev , Alexander. “ORB (Oriented FAST and Rotated BRIEF)¶.” *OpenCV*, 2013, https://opencv-python tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_orb/py_orb.html.

- Tyagi, Deepanshu. “Introduction to ORB (Oriented FAST and Rotated BRIEF).” *Medium*, Medium, 29 May 2019, medium.com/@deepanshut041/introduction-to-orb-oriented-fast-and-rotated-brief-4220e8ec40cf.