Project 2: Feature Detection and Matching

Sherwyn Braganza (NO)

October 29, 2022

Summary of Contents

- The Aim of the Project
- The Approach and Algorithm
- Extra Credit
- Appendices

1 The Project

The aim of this project revolved around learning about three major subsections of Computer Vision - Image Feature Detection, Feature Description and Feature Matching. To elaborate more on this, our goal was to identify interest points in an image (corners), learn to differentiate them from other features we have detected in the same image and try to match these features detected with those found in another similar image taken from a different perspective. The baseline requirements for this program were as follows:

1. Implement the Harris Stephens Feature Detector as mentioned in Combined Corner and Edge Detector - Harris, Stephens
2. Implement a method of describing a feature such that it is invariant to rotation and illumination
3. Match features that have a similar description
4. Implement the SSD distance and ratio test feature metrics

Additionally, the entire source code and results of this project can be found here.

2 The Algorithm and Approach

2.1 Feature Detection

The Harris Corner Detector is a more Robust version of the Hessian Corner detector, in the light that every response value less than 0 corresponds to an edge, values equal to 0 correspond to flat regions and values greater than 0 correspond to a corner. The strength of how good of a corner a pixel is, is given by the magnitude of the positive response value. Since the Harris Response Values are an outcome of the image gradients, it is both invariant to illumination and rotation. However, we should keep in mind that it is not invariant to changes in scale or contrast.

In my implementation, I created a class called FeatureDetection, who's whole purpose was to identify features in an image and record them. From a broad level perspective the main functions in this class were as follows:

1. **showFeature** - Overlays the image oriented squares at locations where features were found.
2. **plotFeature** - Helper Function to showFeature, plots an individual feature square at a given location.
3. **generatePlotPoints** - Helper function to plotFeature, generates points to along the edges of the buffer square.
4. **computeHarrisStephensResponse** - Implements the Harris-Stephens Corner Detector to identify interest points in the image.
5. **computeBrownSzeliskiWinder** - Implements the Brown-Szeliski-Winder version of Harris Corner Detector to identify interest points.
6. **suppressNonMaximals** - Implements non maximal suppression to sift through the features detected and remove features that are not maximas in a given window size.
7. **getFeatures** - Goes through the Harris Response Matrix and picks out values above a given threshold.

The functions **showFeature**, **plotFeature** and **generatePlotPoints** work together to create and plot a oriented buffer square around a feature. The main idea behind their working is that they are provided with a plotting parameters; the location of the feature (loc), the size of the square which is an outcome of scale (span) and the orientation of the square (orientation). A function called **grabBufferSquare** grabs the extremities or edge points of this generated hypothetical square and then passes them through a function called **rotateMatrix** that rotates it according to orientation. These new rotated points are then passed to **generatePlotPoints** that interpolates between them to get the points corresponding to the square that need to be plotted. The code for these functions can be found in Appendix I. Figure 1 shows the output of this function.

The main feature identification is handled by **getFeatures**. It calculates first order derivatives of the image by convolving it with a Sobel Kernel as shown below.

```
1         SOBEL_X = np.asarray([[-1,0,1], [-2,0,2], [-1,0,1]])
2         SOBEL_Y = SOBEL_X.T
3
4         # convolve the sobel with the image to get gradients in each dir
5         self.x_grad = g_x = signal.convolve2d(self.grayscale, SOBEL_X, mode='same')
6         self.y_grad = g_y = signal.convolve2d(self.grayscale, SOBEL_Y, mode='same')
```

Figure 1: **showFeature** and **getFeature** working together

It then calls one of the Harris Corner Detection Methods and corner strength response matrix. One our biggest problems at this point is that the edges of the image turn up as exceptionally high response values, so we overcome this hurdle cropping the image by 5% at each edge. The function then goes through the values of the Harris Response and cherry picks out values that surpass a minimum threshold. Each feature that manages to get picked, is stored in a 'features' list that is then pruned by the **suppressNonMaximals** function that applies non maximal suppression in a 5x5 window as show below. Each element in the features list is a 3-tuple that represents the (location, scale, orientation) of the feature. The orientation was calculated by computing the inverse tangent of the $\frac{Y-gradient}{X-gradient}$ at the location using the **np.atan2** function. The **getFeature** function along with the implementations of its subfunctions can be found in Appendix II. Figure 1 shows the feature detection mechanism working along with the feature plotting mechanism.

```
# Set exclusion bounds so we dont pick up corners as features
        # set to default 5% exclusion on each edge
        bounds = int(5 / 100 * self.image.shape[0]), int(5 / 100 * self.image.shape[1])

        # iterate through the response matrix and grab features
        scale = 1
        for row, x in enumerate(self.response):
            for col, y in enumerate(x):
                if row < bounds[0] or row > (self.image.shape[0] - bounds[0]) \
                        or col < bounds[1] or col > (self.image.shape[1] - bounds[1]):
                    continue
                if y > response_threshold:
                    self.features.append([(row, col), scale, getGradientOrientation(g_x, g_y, (row, col))])

        # suppress non maximums in a 5x5 window
        self.suppressNonMaximals((-2,2))
```

The Harris Corner Matrix was calculated according to the equation shown below.

$$H = w(x,y) * \begin{bmatrix} G_x^2 & G_xG_y \\ G_xG_y & G_y^2 \end{bmatrix} \tag{1}$$

where G_x and G_y are the gradients of the image in the x direction and y direction or the image convoluted with the X-Sobel and Y-Sobel.

The matrix was then convoluted with a Gaussian 'w' that was arbitrarily chosen. The corner strength was then calculated based on whether the user chose the **computeHarrisStephensResponse** or the **computeBrownSzeliskiWinder** functions. The mathematical representations of these functions are shown below

Harris Stephens Method

$$c(H) = det(H) - \alpha * trace(H)^2 \tag{2}$$

where H is the Harris Corner Matrix and $\alpha = [0.04, 0.08]$

Brown-Szeliski-Winder Adaptation

$$c(H) = \frac{det(H)}{trace(H)} \tag{3}$$

As mentioned above, these response values were then iterated through, and those that passed the minimum threshold were stored in the features list along with its scale and orientation. The code for these supporting functions can be found in Appendix II.

2.2 Feature Description

Feature Description and Feature Matching in my program are implemented by the same class - ***FeatureMatching***. The ***FeatureMatching*** class contains three main functions **getDescriptor**, **getMatches** and **plotMatches**. In this sub-section I will be talking about the **getDescriptor** function and how it describes a feature; and the in the next section, about the other two.

When selecting a descriptor, the requirements that needed to be met were that it was supposed to be illumination and rotation invariant.

My approach involved picking out chunks from X-gradient matrix and Y-gradient matrix, around the feature location in a $N * N$ window (I chose 5x5 in my implementation and testing). To make it rotation invariant, a chunk 2x the window size was picked out; the bigger chunk was then rotated by the orientation value of the feature, albeit in the opposite direction, to get it to a uniform orientation of 0 (line 9 and line 10). The chunk was then cropped to the correct window size from twice the amount(line 12 - line 14). This chunk was then normalized to make it illumination invariant. A visual representation of the gradient chunks can be found in Figure 2.

One thing that was noticed was that if the feature descriptor window size was increased, it would improve the matching accuracy as the feature descriptor better defines the point. This however increases the computation time.

```
1  # basic window size is 5x5
2  side_length = int(np.rint(scale * 5))
3          min_x, max_x = loc[0] - side_length, loc[0] + side_length
4          min_y, max_y = loc[1] - side_length, loc[1] + side_length
5          x_chunk = featureImage.x_grad[min_x: max_x + 1, min_y: max_y + 1]
6          y_chunk = featureImage.y_grad[min_x: max_x + 1, min_y: max_y + 1]
7
8          # rotate the chunk in the opposite direction of orientation
9          x_chunk = ndimage.rotate(x_chunk, angle=-orientation, reshape=False)
10         y_chunk = ndimage.rotate(y_chunk, angle=-orientation, reshape=False)
11
12         center = [x_chunk.shape[0]//2, x_chunk.shape[1]//2]
13         min_x, max_x = center[0] - side_length//2, center[0] + side_length//2
14         min_y, max_y = center[1] - side_length//2, center[1] + side_length//2
15
16         x_chunk = x_chunk[min_x:max_x + 1, min_y:max_y+1]
17         y_chunk = y_chunk[min_x:max_x + 1, min_y:max_y + 1]
18         return normalizeMatrix(x_chunk), normalizeMatrix(y_chunk)
```

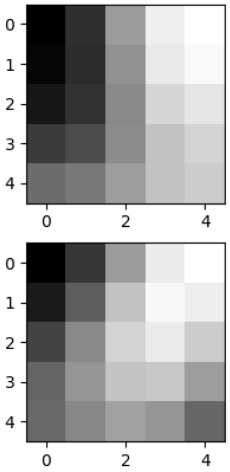

Figure 2: Feature Descriptors - Gx (top) and Gy (bottom)

2.3 Feature Matching

The process of image matching involves comparing descriptions of image points across two different images to see if any of them match. Interest points with similar feature descriptions would most likely be the same point across the images. In other words, if the average absolute difference or average sum of squared differences differs by a value δ, that is below a predetermined threshold, then they should be the same feature across images.

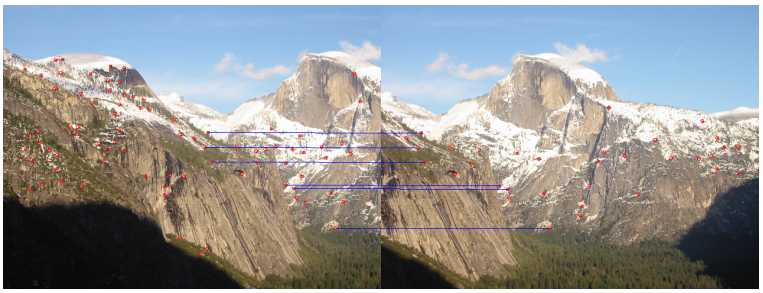

Figure 3: Matched Features

The function **match_features** implements the same approach mentioned above. It does an exhaustive compare of features from one image to the features of another. It computes the average sum of squared differences of the feature descriptors under consideration and stores it in the ssd_distance matrix. Then depending on what metric we choses, calls the respective function to find matches.

```
        for idx_x, x in enumerate(self.featureImage1.features):
            for idx_y, y in enumerate(self.featureImage2.features):
                image_a_grad = self.grabDescriptor(self.featureImage1, x[0], x[1], x[2])
                image_b_grad = self.grabDescriptor(self.featureImage2, y[0], y[1], y[2])
                grad_diff = np.sqrt((image_a_grad[0] - image_b_grad[0]) ** 2 +
                                    (image_a_grad[1] - image_b_grad[1]) ** 2)

                self.ssd_distance[idx_x, idx_y] = np.sum(grad_diff)

        self.ssdDistTest(5)
        # self.ratioTest(0.5)
```

The function **plotMatches** draws lines across matched pairs using linear interpolation, almost exactly how **plotFeatures** does. The code to this function as well as all other helper functions in the class ***FeatureMatching*** can be found in Appendix III

The strength of matches were measured by two metrics:

1. The SSD Distance
2. The 'Ratio Test'

Fortunately, based on our implementation, we already calculated the Sum of Squared Differences for each match. We just got to check for matches that pass a certain threshold and are the maximum for a given feature matching.

```
def ssdDistTest(threshold):
    for idx_x, x in enumerate(self.ssd_distances):
        y = max(x)
        y_idx = np.where(x == y, x)
        if y < threshold:
            self.match_pairs.append(
            self.features[idx_x], self.features[idx_y]
            )
```

The ratio test metric calculation was simple as you took the two smallest values from the ssd_distance list and computed their ratio. Every other match when divided against the min SSD distance needed to produce a fairly close to the original one to be considered as a valid match.

```
    def ratioTest(threshold):
        max_val = self.ssd_distances.max()

        for idx_x, x in enumerate(self.ssd_distances):
            for idx_y, y in enumerate(x):
                if max_val/y > threshold:
                    self.match_pairs.append(
                    self.features[idx_x], self.features[idx_y]
                    )
```

The SSD value for the best match in Figure 3 was **3.079** and the value of the base ratio test was **0.9364**. Thresholds of < 5 and > 0.5 were set for each of the tests.

From further testing, the SSD Distance test did better than the ratio test in terms of matching and was adopted as a norm for producing the results we got. The results for other images from the dataset and can found in Appendix IV. **This also includes matching that exhibits illumination and rotational invariance.**

3 Extra Credit Attempts

3.1 Adaptive Non Maximal Suppression

The main idea behind Adaptive Non Maximal Suppression is to suppress points in densely populated regions and permit points in sparse regions until a certain number of points are reached.

The requirements for implementing this method are:

1. A sorted Harris Corner Response Matrix, sorted by corner strength
2. A variable called radius that keeps track of an exclusion circle
3. A feature list that keeps track of features that are selected.

The radius for the exclusion circle is calculated according to the following equation.

$$r = min_j |\mathbf{x}_i - \mathbf{x}_j| \tag{4}$$

where x_j is chosen such that $f(x_i) < 0.9 * f(x_j)$. Each point in the feature list is mapped to an exclusion circle centered at itself with a radius r. New points are picked one at a time from the sorted list, as long as they don't belong in the exclusion circles of all the previous elements in the feature list. This results in a more distributed selection of features. The code snippet below shows how I calculate my suppression radius. The variable 'bounded_response' is the Harris Matrix Response excluding those corresponding to the image edges.

```
        bounds = int(5 / 100 * self.image.shape[0]), int(5 / 100 * self.image.shape[1])
        bounded_response = np.reshape(self.response[bounds[0]:-bounds[0], bounds[1]:-bounds[1]]), (1, -1)
        bounded_response = np.sort(bounded_response)
        suppression_radius = 0
        max_idx = np.where(self.response == bounded_response[0], self.response)

        for i in range(1, bounded_response):
            if bounded_response[0] < 0.9 * bounded_response[i]:
                temp_idx = np.where(self.response == bounded_response[i], self.response)
                suppression_radius = min(
                    ((max_idx[0] - temp_idx[0])**2 + (max_idx[1] - temp_idx[1])**2) ** 0.5
                )
```

Instead of a suppression circle, I use a suppresion box along with the Polygon Describing class - Shapely. I use my **grabBufferSquare** function to grab the extremities of the box and then create a Shapely Close Polygon Object. The main benefit of using Shapely is that I can just call shapely.object.contains(Point), everytime I need to check if a point is contained in the exclusion squares.

The following code snippet implements the above mentioned algorithm and stores the point and updates the shapely polygon list if its a valid new feature.

```
points = grabBufferSquare((max_idx[0], max_idx[1]), 2*suppression_radius)
        self.features.append(max_idx)
        polygon_list = [shapely.geometry.polygon(points+points[0])]

        for i in range(1, bounded_response):
            temp_idx = np.where(self.response == bounded_response[i], self.response)
            contains = False
            for x in polygon_list:
                if x.contains(shapely.geometry.point(temp_idx)):
                    contains = True
                    break
            if not contains:
                self.features.append(temp_idx)
                points = grabBufferSquare(temp_idx, 2*suppression_radius)
                polygon_list.append(shapely.geometry.polygon(points+points[0]))
```

Unfortunately there was a bug in this implementation that prevented me from being able to provide valid results of this working.

4 Appendix I

```
    def plotFeature(self, loc: tuple, span: int = 5, orientation: float = 0) -> None:
        """
            Shows (overlays) the features detected on the image. Plots them in the form of boxes with
            orientation.

            :param loc: the centroid of where the feature was detected
            :param span: The span of the feature
            :param orientation: The orientation of the feature.
            :return: None
        """
        buffer_size = span
        buffer = grabBufferSquare(loc, buffer_size)
        rotated_buffer = rotateMatrix(loc, buffer, orientation)
        plotpoints = self.generatePlotPoints(loc, rotated_buffer, buffer_size)

        for x in plotpoints:
            self.image[x[0], x[1], :] = np.asarray([255, 0, 0])

    def generatePlotPoints(self, loc: tuple, points: np.ndarray, size: int) -> np.ndarray:
        """
            Interpolate between extremities and get points along the edges of the buffer
            square for plotting

            Author: Sherwyn Braganza

            :param loc: The center of the feature
            :param points: The endpoints of edges of the feature
            :param size: The span of the feature
            :return: Points along the edges of the feature buffer
        """
        plotpoints = np.asarray([0, 0])

        density_factor = 3
        for i in range(len(points)):
            temp_x = np.linspace(points[i % 4, 0], points[(i+1)%4, 0], num=size*density_factor+1,
                                 endpoint=True, dtype=int)
            temp_y = np.linspace(points[i % 4, 1], points[(i+1)%4, 1], num=size*density_factor+1,
                                 endpoint=True, dtype=int)
            plotpoints = np.vstack((plotpoints, np.hstack((temp_x.reshape(-1, 1), temp_y.reshape(-1, 1)))))

        plotpoints = np.vstack((
            plotpoints,
            np.hstack((
                np.linspace(loc[0], points[-1, 0], num=size*density_factor+1, endpoint=True,
                            dtype=int).reshape(-1, 1),
                np.linspace(loc[1], points[-1, 1], num=size*density_factor+1, endpoint=True,
                            dtype=int).reshape(-1, 1),
            ))
        ))

        return np.delete(plotpoints, 0, axis=0)

    def showFeatures(self) -> None:
        """
            Goes through the each elemenent of the feature array and
            calls plotFeature to plot the feature on the image.

            Author: Sherwyn Braganza

            :return: None
        """
        for x in self.features:
            self.plotFeature(x[0], int(np.rint(5 * x[1])), x[2])

    def rotateMatrix(center: tuple[int, int], matrix: np.ndarray, orientation):
    """
        Rotates a matrix about a given center. From a broad level view,
        uses the Change of Basis Theorem to bring the matrix to (0, 0) and
        uses the standard rotation matrix about (0, 0) to rotate.

        :param center: The centre about which the rotation should be performed
        :param matrix: The matrix to be rotated
        :param orientation: The angle by which it should be rotated by
        :return: The rotated matrix.
    """
    loc = np.asarray(center)
    cob_points = matrix - center  # change of basis points
    theta = np.radians(orientation)
    c, s = np.cos(theta), np.sin(theta)

    # rotation Matrix about 0
    rotMatrix = np.array(((c, s),
                          (-s, c)))

    cob_points = np.matmul(cob_points, rotMatrix)

    def grabBufferSquare(loc: tuple, size: int) -> np.ndarray:
    """
```

```
        Grabs the end points of a hypothetical buffer square, centered
        at loc. The size of the square is determined by size

        The 4 end points corresponds to the edges of the buffer square.
        The points are listed in the bottomLeft, bottomRight,
        topRight, topLeft order.

        Additionally, it grabs a 5th point that corresponds to the projection
        of loc on the Right Edge of the buffer square.

        Author: Sherwyn Braganza

        :param loc: The centroid of the buffer square
        :param size: The edge size of the square
        :return: np.array corresponding to the end points.
    """
    sideLength = size
    bottomLeft = np.asarray([loc[0] - sideLength // 2, loc[1] - sideLength // 2])
    bottomRight = np.asarray([loc[0] + sideLength // 2, loc[1] - sideLength // 2])
    topRight = np.asarray([loc[0] + sideLength // 2, loc[1] + sideLength // 2])
    topLeft = np.asarray([(loc[0] - sideLength // 2, loc[1] + sideLength // 2)])
    bottomTopRightCenter = np.asarray([(loc[0] + sideLength // 2, loc[1])])

    return np.vstack((bottomLeft,bottomRight,topRight,topLeft, bottomTopRightCenter))

    return np.rint(cob_points + loc).astype(int, casting='unsafe')
```

5 Appendix II

```
def getFeatures(self, response_threshold: float = 0):
    """
        Implements a corner strength detection method to identify interest points.
        The performs non-Maximal Suppression on the identified points to remove
        duplicates of features really close to each other.

        Author Sherwyn Braganza

        :return: None
    """
    SOBEL_X = np.asarray([[-1,-1, 0, 1, 1],
                  [-2,-1, 0, 1, 2],
                  [-4, -2, 0, 2, 4],
                  [-2, -1, 0, 1, 2],
                  [-1,-1, 0, 1, 1]])
    SOBEL_Y = SOBEL_X.T

    # convolve the sobel with the image to get gradients in each dir
    self.x_grad = g_x = signal.convolve2d(self.grayscale, SOBEL_X, mode='same')
    self.y_grad = g_y = signal.convolve2d(self.grayscale, SOBEL_Y, mode='same')

    # Pick a feature Detection Method
    self.computeBrownSzeliskiWinder(g_x, g_y)
    self.response = normalizeMatrix(self.response)

    self.features = []

    # Set exclusion bounds so we dont pick up corners as features
    # set to default 5% exclusion on each edge
    bounds = int(5 / 100 * self.image.shape[0]), int(5 / 100 * self.image.shape[1])

    # iterate through the response matrix and grab features
    scale = 1
    for row, x in enumerate(self.response):
        for col, y in enumerate(x):
            if row < bounds[0] or row > (self.image.shape[0] - bounds[0]) \
                    or col < bounds[1] or col > (self.image.shape[1] - bounds[1]):
                continue
            if y > response_threshold:
                self.features.append([(row, col), scale, getGradientOrientation(g_x, g_y, (row, col))])

    # suppress non maximums in a 5x5 window
    self.suppressNonMaximals((-2,2))

def computeHarrisStephensResponse(self, g_x, g_y):
    """
        Computes the corner strength based on the Harris-Stephens
        Algorithm.

        Author: Sherwyn Braganza

        :param g_x: The derivative of the image in the X direction
        :param g_y: The derivative of the image in the Y direction
        :return: The corner strength response
    """
    w_1d = np.asarray([1, 4, 7, 4, 1]).reshape(-1, 1)
    w_2d = np.matmul(w_1d, w_1d.T)
    w = w_2d / np.sum(w_2d)
    g_xX = signal.convolve2d(g_x ** 2, w, mode='same')
    g_yY = signal.convolve2d(g_y ** 2, w, mode='same')
    g_xY = signal.convolve2d(g_x * g_y, w, mode='same')
    alpha = 0.04

    # determinant
    det = g_xX * g_yY - g_xY * g_xY
    # trace
    trace = g_xX + g_yY

    self.response = det - alpha * (trace ** 2)

def computeBrownSzeliskiWinder(self,  g_x, g_y):
    w_1d = np.asarray([1, 4, 7, 4, 1]).reshape(-1, 1)
    w_2d = np.matmul(w_1d, w_1d.T)
    w = w_2d / np.sum(w_2d)
    g_xX = signal.convolve2d(g_x ** 2, w, mode='same')
    g_yY = signal.convolve2d(g_y ** 2, w, mode='same')
    g_xY = signal.convolve2d(g_x * g_y, w, mode='same')
    # determinant
    det = g_xX * g_yY - g_xY * g_xY
    # trace
    trace = g_xX + g_yY

    self.response = det/trace

def suppressNonMaximals(self, suppression_window) -> None:
    """
        Implements non maximal suppression in a specified window size (suppression_window).
        Manipulates the feature array internally and deletes points that are supposed to
```

```
89              be suppressed.
90
91              Author: Sherwyn Braganza
92
93              :param suppression_window: The window size
94              :return: None
95          """
96          count = 0
97          idx = 0
98          while idx < len(self.features):
99              feature = self.features[idx]
100             if self.response[feature[0][0], feature[0][1]] < \
101                     np.max(
102                         self.response[feature[0][0] + suppression_window[0]: feature[0][0] + suppression_window[1]+1,
103                         feature[0][1] + suppression_window[0]: feature[0][1] + suppression_window[1]+1]
104                     ) and feature:
105                 del self.features[idx]
106                 count += 1
107             else:
108                 idx += 1
109
110         print('{} features suppressed'.format(count))
```

6 Appendix III

```
class FeatureMatching:

    def __init__(self, featureImage1: FeatureDetection, featureImage2: FeatureDetection):
        self.featureImage1 = featureImage1
        self.featureImage2 = featureImage2
        self.combinedImage = np.hstack((featureImage1.image, featureImage2.image))
        self.shift_amount = (0, featureImage1.image.shape[1])
        self.match_pairs = []
        self.ssd_distance = []
        self.ratio_test = -1

    def grabDescriptor(self, featureImage: FeatureDetection, loc, scale, orientation):
        side_length = int(np.rint(scale * 10))
        min_x, max_x = loc[0] - side_length, loc[0] + side_length
        min_y, max_y = loc[1] - side_length, loc[1] + side_length

        x_chunk = featureImage.x_grad[min_x: max_x + 1, min_y: max_y + 1]
        y_chunk = featureImage.y_grad[min_x: max_x + 1, min_y: max_y + 1]

        x_chunk = ndimage.rotate(x_chunk, angle=-orientation, reshape=False)
        y_chunk = ndimage.rotate(y_chunk, angle=-orientation, reshape=False)

        center = [x_chunk.shape[0]//2, x_chunk.shape[1]//2]
        min_x, max_x = center[0] - side_length//2, center[0] + side_length//2
        min_y, max_y = center[1] - side_length//2, center[1] + side_length//2

        x_chunk = x_chunk[min_x:max_x + 1, min_y:max_y+1]
        y_chunk = y_chunk[min_x:max_x + 1, min_y:max_y+1]

        # fig, axs = plt.subplots(2)
        # axs[0].imshow(x_chunk, cmap='gray')
        # axs[1].imshow(y_chunk, cmap='gray')
        plt.show()

        return normalizeMatrix(x_chunk), normalizeMatrix(y_chunk)

    def getMatches(self):
        gradient_tolerance = 5
        min_dist = 0
        for idx_x, x in enumerate(self.featureImage1.features):
            for idx_y, y in enumerate(self.featureImage2.features):
                image_a_grad = self.grabDescriptor(self.featureImage1, x[0], x[1], x[2])
                image_b_grad = self.grabDescriptor(self.featureImage2, y[0], y[1], y[2])
                grad_diff = np.sqrt((image_a_grad[0] - image_b_grad[0]) ** 2 +
                                    (image_a_grad[1] - image_b_grad[1]) ** 2)

                if np.sum(grad_diff)/(2*len(image_a_grad)) < gradient_tolerance:
                    self.match_pairs.append((x,y))
                    min_dist = min((grad_diff.min(), min_dist))
                    self.ssd_distance.append(grad_diff)
                    del self.featureImage1.features[idx_x]
                    del self.featureImage2.features[idx_y]
                    break

        print(len(self.match_pairs))

    def plotMatches(self):
        for match in self.match_pairs:
            point_density = int(np.rint(2 * np.sqrt(
                (match[0][0][0] - match[1][0][0] + self.shift_amount[0])**2 +
                (match[0][0][1] - match[1][0][1] + self.shift_amount[1])**2
            )))
            x_points = np.linspace(match[0][0][0], match[1][0][0] + self.shift_amount[0] + 1,
                                   num=point_density, dtype='int32')
            y_points = np.linspace(match[0][0][1], match[1][0][1] + self.shift_amount[1] + 1,
                                   num=point_density, dtype='int32')
            for i in range(point_density):
                self.combinedImage[x_points[i], y_points[i], :] = np.asarray([0,0,255])
```

7 Appendix IV

Figure 4 shows the resilience of the Feature Detection Program to Illumination. The min SSD value for this image was 1.13 and ratio test result was 0.881.

Figure 4: Illumination Invariant Feature Matching

Figure 5 shows the resilience of the Feature Detection Program to Rotation on the Graffiti dataset. The minimum SSD value was 13.23 and ratio was 0.8932.

Figure 5: Rotation and Some Level of Skew Invariant Feature Matching

Figure 6: Ill Posed Problem Feature Matching

Figure 7: Effect of Blur on Feature Detection