Filtering, Object Recognition and Features

Adam Jaamour, Andrea Lissak

April 1, 2019

1 Choice of Language: Python

For this project, the programming language of choice was Python. It was chosen over MATLAB due to the availability of functions similar to MATLAB's functions through libraries, and due to the syntax and flexibility of the language. With libraries such as OpenCV for image manipulations, Numpy and SciPy for more advanced mathematical functions including array manipulations, and MatPlotLib for plotting data, Python has all the tools for the task at hand.

2 Task 1: Image Convolution

For the image convolution task, a function to perform convolution on a grey image was written. This function takes an image to operate on and a kernel to apply on the image. In order to preserve image size, extra padding is added on the edges of the original image. The function can be found in Listing A.

Multiple filters are used to test the convolution function, including a Gaussian kernel, a sharpen kernel, a horizontal edge detector (Sobel filter) and an identity matrix. To confirm that the results are correct, the resulting filtered image compared to a library function. SciPy's "*convolve*" function was used to carry out this task as it is the equivalent of MATLAB's "*conv2*" function. The comparison is done by subtracting the function's result from the library's result and checking that all the values are equal to 0. See Listing B for the filters used and the test.

3 Task 2: Intensity-Based Template Matching

3.1 Pre-Processing

Multiple steps are followed to pre-process the training dataset. Firs, The background is set to 0 for each image in the training dataset. This is done by looking for white pixels (value 255) and replacing them with black pixels (value 0), as depicted in Listing C line 3. Next, each rotated and scaled template in the pyramid is then normalised using OpenCV's "*normalize*" function, which can be found in Listing C line 10. Finally, images are processed in RGB (3 channels) rather than being converting to grey scale (1 channel), therefore avoiding the loss of information and accuracy.

3.2 Gaussian Pyramid Generation

A Gaussian Pyramid is generated for each image in the training dataset. The scaling and rotation values for the pyramids were chosen by manually inspecting the training images to find out what angles the images were rotated to and by how much the images were scaled down. Therefore, scales of 50%, 25%, 12.5% and 6.25% were chosen, along with rotations ranging from 0°to 330°with steps of 30°for each scale. A brief overview of the training function used to generate the templates can be found in Listing D.

The rotation of the templates was achieved through SciPy's "*rotate*" function (Listing D, line 22), while the scaling down was done manually with a custom subsampling function that recursively scales the image down by half by subsampling one pixel every two pixels (see Listing E). The subsampling blurring uses a Gaussian Filter with a size of 5x5 and a standard deviation $\sigma = 15$.

Each template is saved in separate binary "*.dat*" files for quicker I/O operations using Python's "*pickle*" library for object serialisation conversion in byte streams (see Listing D, line 26).

3.3 Output

3.3.1 Testing

For the testing phase, each template previously stored in binary files is loaded in memory and is slid over one of the images from the testing dataset. All the templates for a single class are used to calculate the correlation score using Equation 1. The template with the highest score for a class is kept as the best match for that class, meaning there are 50 potential matching templates. Each template is then filtered based on its correlation score: it is kept if it is higher than a threshold, set at 0.5. This threshold is set empirically by testing different values until one that gave the most matches is found.

$$cor(x, y) = \sum_{i,j} T(i, j)I(i + x, j + y) \qquad (1)$$

3.3.2 Box Drawing

The final step in producing the output is to draw a box around the detected objects that passed the threshold. The scaling and the rotation of the selected templates are used to draw the rectangle at the correct scale and at the correct rotation (see

Listing G). The class name is drawn along with the box rather than class number to improve the output's clarity.

A rotated rectangle needs 4 pairs of coordinates to be drawn. Obtaining one pair will make it trivial to find the others. So, the real objective is to find the x and y shift from the one of the hypothetical straight square. Let us denote the x shift by Δx. Starting from a system of equation defining that the height/width of the original image is $h = \Delta x + \Delta y$, that $\sin(\alpha) * n = \Delta y$ and that $n^2 = \Delta x^2 + \Delta y^2$ (where *n* is the length of the newly rotated and scaled square). So, resulting from the above, we can find Δx (as shown in Equation 2), thus all the corners of the rotated square: $(\sin(\alpha) = k)$

$$\Delta x = \frac{\sqrt{h^2k^2 - h^2k^4} + hk^2 - h}{2k^2 - 1} \qquad (2)$$

3.4 Results Evaluation

To evaluate the results of our intensity-based template matching algorithm, the number of true and false positives are counted for each test along with the runtime measured in seconds. The results are reported in Figure 1. The average number of True Positives is 3.65, whereas the average number of False Positives is 31.2. This betrays the fact that this kind of matching algorithm is inefficient.

Attempts to maximise True Positives and minimise False Positives included increasing the threshold, which caused the number of False Positive matches to decrease but had a ripple effect on True Positive matches that were filtered out as well. A case that was often observed was that many small templates often matched with large images with white backgrounds such as "airport" or "bank", meaning they could either not be matched with the correct object or were not filtered out by the threshold due to their high correlation score with the white background. The correlation method does not provide a difference that important enough to efficiently filter out False Positives without affecting the True Positives.

Other more successful improvements consist in fine-tuning the Gaussian filter. Using a Gaussian filter of size $5x5$ with $\sigma = 15$ allows a minimal amount of artefacts to appear in the templates when scaling down images, thus increasing accuracy. Additionally, the templates are generated in RGB rather than grey scale. The improvement in accuracy was drastic as less templates matched with white backgrounds when using colours. With more time, an additional enhancement could have been implemented by drawing a box for only the best matching template around a single location with an object on the test image, which would limit the number of false positives around a single object.

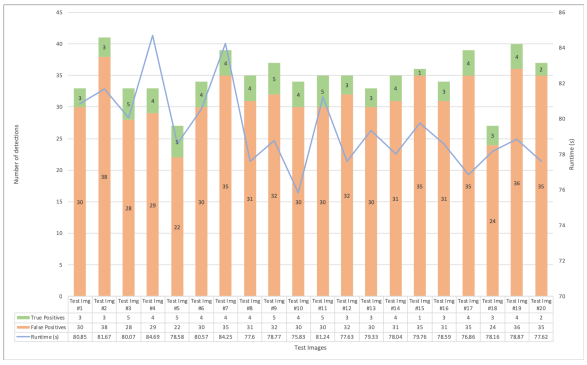

Figure 1: Results of Task 2 for each image in the testing dataset. In green, the number of true positive matches, in red the number of false positive matches, and in blue the runtime measured in seconds.

The complexity of the training function that generates the Gaussian Pyramid is $O(p * r * n^2)$ per image per colour channel, where *n* corresponds to the image size (e.g. *n* would be 262144 for a 512 by 512 image, and not 512), *p* to the number of scales and *r* to the number of rotations. Therefore the problem size increases linearly for each image added to the training dataset. However, the scalability is extremely inefficient when it comes to the testing. Indeed, with our current choice of scales and rotations, each new image in the training set would add 48 additional templates to slide across the test image, which currently takes on average 79.45 seconds with 2400 templates to slide across the test image.

4 Task 3: SIFT Features

Scale-invariant feature transform, as described by Lowe (2004), can be interpreted as a pipeline. So, the following paragraphs will go through the results of an attempt of implementation of SIFT. (The choice of constants for the current implementation consisted in following Lowe (2004)'s recommendations). As a high-level indication, all that is explained below was carried out through three different samplings originating from the original image (three octaves).

4.1 Key-point Filtering

Key-points are points of interest of an image. Therefore, when looking for them, a threshold for low contrast can be used for early filtering less relevant key-points. Now that only high-change areas have been spotted, extrema can be found, i.e. sharper changes which refer to edges and corners in the original image. Applying a Laplacian filter is an appropriate solution for such task. However, in discrete domains it has been found that the *Difference of Gaussians* (*DoG*) is a relatively accurate approximation to enable interest point localisation (Lowe et al., 1999). Peaks and dips are spotted by comparing multiple differences of Gaussians together. So, *DoG* (σ_2, σ_3) will be compared with the adjacent layers (σ_1, σ_2) and (σ_3, σ_4). More specifically, each analysed candidate will be compared with its 26 neighbouring candidates (8 within the same scale, 18 ranging between the above and below scales). Once all the minima and maxima are extracted, more keypoint can be filtered out by eliminating edges, which contain less information than corners. Because eigenvalues are discrete indicators, they can be used to discriminate these two cases. Consequently, a threshold to filter out edges can be applied. The code for this section can be found in Listing H.

4.2 Key-point Magnitude and Orientation

At this point, the coordinates of the key-points have been selected. In order to diversify them further, given their neighbourhood, both their magnitude and their orientation can be calculated. This is achieved by looking at pixel differences, meaning that pixel intensities arranged in certain ways can output how sharp a feature is and what its rotation is. This is applied across the entire image as it is a useful foundation for the following step, which corresponds to assigning a noise-free orientation and magnitude for each of the key-points. To enforce this, each key-point's neighbourhood needs to be taken into account. The code for this section can be found in Listing I.

A Gaussian distribution centred around the key-point is used to give more importance to pixels that are close to the key-point than to further ones. This means that each time one of these neighbours is visited, one of 36 rotation buckets corresponding to its orientation (divided by a factor of ten, to be more easily computed in discrete form) is increased by the calculated weight. In the end, once the whole neighbourhood is visited, the algorithm looks at the bucket that is the most full. In Lowe (2004)'s implementation, buckets which are 80% (or more) of the largest one are used to spawn new key-points with different rotations. However, in the current, this has not been implemented given the consequences being fairly negligible. The output is thus the key-point matrix, which includes (x, y), $m(x, y)$ and $\theta(x, y)$.

4.3 Box Representation

The key-point matrix can be used to visualise the location, magnitude and orientation of interest points on the original images. In our implementation, this has been done by placing markers with the radius directly proportional to the key-point's scale and with segments matching the orientation, as demonstrated in Figure 2 (see the function used to draw these in Listing K).

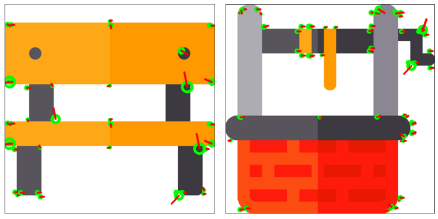

Figure 2: SIFT key points on 2 training images.

4.4 Descriptors

Descriptors are used to keep track of the bucket-style histogram representation of neighbouring areas. Lowe (2004)'s measures have been used for this, which means that this program generates 16 histograms, each made up of 16 pixels. The number of buckets used to maximise performance is 8 (Lowe, 2004). Therefore, a total of 16 histograms with 8 buckets each (128 values) are used to define the descriptor for each key-point. The code for this section can be found in Listing J.

References

Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110.

Lowe, D. G. et al. (1999). Object recognition from local scale-invariant features. In *iccv*, volume 99, pages 1150–1157.

Table 1: Table of contributions

Student Name	Adam Jaamour	Andrea Lissak
Student Username	*aj645*	*al746*
Student ID	*159327001*	*159044728*
Contributions	*- Convolution* *- Intensity-based template matching training & testing* *- Report formatting*	*- SIFT key points* *- Intensity-based template matching testing*
Contribution Percentage	*50%*	*50%*

Appendices

A Convolution Function

```
def convolution(image, kernel):
    # initialise output image
    output_img = np.zeros((img_width, img_height), dtype=np.uint8)

    # extend the image with 0s
    extended_img = np.pad(image, [r,c], mode="constant", constant_values=0)

    # flip the kernel horizontally and vertically
    kernel = np.flipud(np.fliplr(kernel))

    # perform convolution on image
    r = int((kernel_width - 1) / 2)
    c = int((kernel_height - 1) / 2)
    for i in range(r, extended_img_width-r+1):
        for j in range(c, extended_img_height-c+1):
            accumulator = 0
            for k in range(-r-1, r):
                for l in range(-c-1, c):
                    accumulator += kernel[r+k+1, c+l+1] * extended_img[i+k, j+l]
            output_img[i-r-1, j-c-1] = accumulator

    return output_img
```

B Convolution Filters Test

```
# filters to test convolution with
filters = {
    'gaussian_filter': gaussian_kernel(5, 5, 3),
    'sharpen': np.array([[0, -1, 0], [-1, 5, -1], [0, -1, 0]]),
    'horizontal_edge_detector': np.array([[1, 0, -1], [2, 0, -2], [1, 0, -1]]),
    'identity': np.array([[0, 0, 0], [0, 1, 0], [0, 0, 0]])
}

# testing example for Gaussian Filter (identical for other filters)
custom_conv = convolution(img, filters['gaussian_filter'])
library_conv = ndimage.convolve(img,filters['gaussian_filter'],mode='constant',cval=0)
difference = library_conv - custom_conv
is_different = np.all(difference == 0)  # boolean: True if images are identical

# ... code for plotting output seen in Figure 1 ...
```

C Intensity-Based Template Matching Pre-processing

```
def fill_black(img):
    """
    Replace white pixels (value 255) with black pixels (value 0) for an RGB image.
    """
    img[img >= [255, 255, 255]] = 0
    return img

def normalize(img):
    normalized_img = np.zeros((img.shape[0], img.shape[1]))
    normalized_img = cv2.normalize(img, normalized_img, 0, 255, cv2.NORM_MINMAX)
    return normalized_img
```

D Intensity-Based Template Matching Gaussian Pyramid

```
# Gaussian Pyramid parameters
scaling_pyramid_depths = [1, 2, 3, 4]  # 50% 25% 12.5% 6.25%
rotations = [0.0,30.0,60.0,90.0,120.0,150.0,180.0,210.0,240.0,270.0,300.0,330.0]
gaussian_filter = gaussian_kernel(5, 5, 15)

for image in get_image_filenames("dataset/Training/png/"):
    img = fill_black(img)  # replace white background pixels with black pixels
    img_b, img_g, img_r = cv2.split(img)  # split image into 3 images (R-G-B channels)

    for p in scaling_pyramid_depths:
        # scale the image by subsampling it p times
        # using our custom function "subsample_image" found in Listing B.5
        scaled_img_b = subsample_image(img_b, p, gaussian_filter)
        scaled_img_g = subsample_image(img_g, p, gaussian_filter)
        scaled_img_r = subsample_image(img_r, p, gaussian_filter)

        # merge back the 3 R-G-B channels into one image
        scaled_img = cv2.merge((scaled_img_b, scaled_img_g, scaled_img_r))

        for r in rotations:
            # rotate the scaled image by r degrees and normalize it
            rotated_scaled_img = ndimage.rotate(scaled_img, r)
            rotated_scaled_norm_img = normalize_image(rotated_scaled_img)

            # save the rotated & scaled template to a binary .dat
            pickle.dump(rotated_scaled_norm_img, "dataset/Training/templates/{classname
    }/rot{r}-sca{p}.dat".format(classname, int(r), p))
```

E Intensity-Based Template Matching Subsampling Function

```
def subsample_image(img, pyramid_depth, gaussian_filter):
    downsizing_factor = 2  # scale down by 50% for each depth

    # blur the image using the gaussian kernel
    blurred_img = signal.convolve2d(img,gaussian_filter,mode="full",boundary="fill",
    fillvalue=0)
    blurred_img_reduced = reduce_size(blurred_img, gaussian_filter)

    # recursively downsample image by half
    prev_scaled_img = blurred_img_reduced
    for _ in range(0, pyramid_depth):
        scaled_img = np.zeros((int(img_height/2), int(img_width/2)), dtype=np.uint8)
        i = 0
        for m in range(0, img_height, downsizing_factor):
            j = 0
            for n in range(0, img_width, downsizing_factor):
                scaled_img[i, j] = prev_scaled_img[m, n]
                j += 1
            i += 1
        prev_scaled_img = scaled_img
        prev_scaled_img = signal.convolve2d(prev_scaled_img,gaussian_filter,mode="full"
    ,boundary="fill",fillvalue=0)
        prev_scaled_img = reduce_size(prev_scaled_img, gaussian_filter)

    return scaled_img
```

F Intensity-Based Template Matching Testing

```
# used to calculate the number of false positives and false negatives
number_of_boxes_drawn = 50

# read and normalise test image
testing_img = cv2.imread("{}test_3.png".format(directory))
testing_img = normalize_image(testing_img)

# start measuring runtime
start_time = time.time()

for classname in os.listdir(templates_dir):
    template_tracker += 1

    cur_best_template = {'class_name':"", 'template_name':"", 'max_val':0.0, 'max_loc':0.0}

    for t in os.listdir(templates_dir + "/" + classname + "/"):
        template = np.load(templates_dir + "/" + classname + "/" + t)  # load template

        # use OpenCV's matchTemplate to calculate correlation score
        correlation_results = cv2.matchTemplate(testing_img,template,cv2.TM_CCORR_NORMED)

        # retrieve maxima value and location
        _, max_val, _, max_loc = cv2.minMaxLoc(correlation_results)

        # apply penalties for smaller images
        scale = get_scaling_from_template(t)
        if scale == '6':
            max_val = max_val * 0.6
        elif scale == '12':
            max_val = max_val * 0.7
        elif scale == '25':
            max_val = max_val * 0.7
        elif scale == '50':
            max_val = max_val * 0.7

        #  keep best template only and data associated to it
        if max_val > cur_best_template['max_val']:
            cur_best_template = {
                'class_name': classname,
                'template_name': t,
                'max_val': max_val,
                'max_loc': max_loc,
                'h': h,
                'rotation': get_rotation_from_template(t)
            }

    threshold = 0.5  # threshold to ignore low values
    if cur_best_template['max_val'] > threshold:
        top_left = cur_best_template['max_loc']
        rotation = cur_best_template['rotation']
        height = cur_best_template['h']
        box_corners = find_box_corners(rotation, height)

        # draw the rectangle by drawing 4 lines between the 4 corners
        cv2.line(
            img=testing_img,
            pt1=(box_corners["P1"][0]+top_left[0],box_corners["P1"][1]+top_left[1]),
            pt2=(box_corners["P2"][0]+top_left[0],box_corners["P2"][1]+top_left[1]),
            color=250,
            thickness=2
        )
```

```
        # ... repeat line drawing for the 3 other lines to draw the box...

        # print classname above the box
        cv2.putText(
            img=testing_img,
            text=classname,
            org=top_left
        )
    else:
        number_of_boxes_drawn -= 1

print("\n--- Runtime: {} seconds ---".format(time.time() - start_time))
print("Number of boxes drawn = {}".format(number_of_boxes_drawn))
```

G Intensity-Based Template Matching Box Scaling/Rotating

```
def find_box_corners(angle, img_height):
    angle = int(angle)
    if angle < 90:
        alpha = np.deg2rad(angle)
    elif 90 <= angle < 180:
        alpha = np.deg2rad(angle-90)
    elif 180 <= angle < 270:
        alpha = np.deg2rad(angle-180)
    elif 270 <= angle < 360:
        alpha = np.deg2rad(angle-270)
    else:
        alpha = np.deg2rad(0)

    k = (np.sin(alpha)) * (np.sin(alpha))
    h = img_height
    d = (np.sqrt((h**2 * k**2) - (h**2 * k**4)) + (h * k**2 - h)) / (2 * k**2 - 1)
    c = img_height - d

    p1x, p1y, p2x, p2y, p3x, p3y, p4x, p4y = (0,) * 8
    if angle < 90:
        p1x = c, p1y = h  # P1
        p2x = h, p2y = h - c  # P2
        p3x = h - c, p3y = 0  # P3
        p4x = 0, p4y = c  # P4
    elif 90 <= angle < 180:
        # ... repeat ...
    elif 180 <= angle < 270:
        # ... repeat ...
    elif 270 <= angle < 360:
        # ... repeat ...

    return {
        "P1": [int(p1x), int(p1y)],
        "P2": [int(p2x), int(p2y)],
        "P3": [int(p3x), int(p3y)],
        "P4": [int(p4x), int(p4y)]
    }
```

H SIFT Key Points Filtering

```
# SIFT Parameters
k = np.sqrt(2)  # base of the exponential which represents the scale coefficient
sig = 1.6       # sigma value suggested by Lowe (2004)
levels = 5      # number of different-size kernels
n_octaves = 3   # number of sampled scales

# assign different kernel sizes for different image scales and different levels. More specifically, start from sigma and k and increase k's exponent by 1 for progression within a level and increase k's exponent by 2 between scales
all_scales = np.zeros((levels + ((n_octaves - 1) * 2)))
octave_at_scale = np.zeros((n_octaves, levels))
for n in range(0, n_octaves):  # all octaves
    for m in range(0, levels):  # all levels
        octave_at_scale[n][m] = sig * (k ** (2 * n + m))
        all_scales[(n_octaves - 1) * n + m] = sig * (k ** (2 * n + m))

# create multiple image scales
img_at_scale = []
for n in range(0, n_octaves):
    if n == 0:
        # double image size as recommended by Lowe (2004)
        double = misc.imresize(start_img, 200, 'bilinear').astype(int)
        img_at_scale.append(double)
    else:
        img_sc = misc.imresize(img_at_scale[n - 1], 50, 'bilinear').astype(int)
        img_at_scale.append(img_sc)

# initialise Gaussian pyramids
gauss_at_scale = []
for n in range(0, n_octaves):
    gauss_at_scale.append(np.zeros((img_at_scale[n].shape[0],img_at_scale[n].shape[1], levels)))

# calculate Gaussian pyramids
for n in range(0, n_octaves):
    for i in range(0, levels):
        gauss_at_scale[n][:, :, i] = misc.imresize(
            ndimage.filters.gaussian_filter(img_at_scale[0],octave_at_scale[1][i]),1/(2**n),'bilinear')

# initialise DoG pyramids
diff_at_scale = []
for n in range(0, n_octaves):
    diff_at_scale.append(np.zeros((img_at_scale[n].shape[0],img_at_scale[n].shape[1], levels)))

# calculate DoG pyramids
for n in range(0, n_octaves):
    for i in range(0, levels-1):
        diff_at_scale[n][:,:,i] = gauss_at_scale[n][:,:,i+1]-gauss_at_scale[n][:,:,i]

# get min and max
max_min_at_scale = []
for n in range(0, n_octaves):
    max_min_at_scale.append(np.zeros((img_at_scale[n].shape[0], img_at_scale[n].shape[1], 3)))

# 3-taks loop:
# Task 1: ignore pixels where the difference of Gaussians is low
# Task 2: compute maxima and minima points in the DoG
# Task 3: reject edge cases and keep corner cases
for n in range(0, 3):  # all octaves
    # skip 0 and 4 because you don't want top and bottom layer because they only have
```

```
    17 neighbours, not 26
    for i in range(1, 4):
        for j in range(1, img_at_scale[n].shape[0] - 1):
            for k in range(1, img_at_scale[n].shape[1] - 1):
                # first filtering strategy explained in section 4 of Lowe's paper
                if np.absolute(diff_at_scale[n][j, k, i]) < thr:
                    continue

                # elements of each pyramids that are larger or smaller than its
                # 26 immediate neighbours in space and scale are labelled as extrema
                max = (diff_at_scale[n][j, k, i] > 0)
                min = (diff_at_scale[n][j, k, i] < 0)
                # one among min and max will be true and one will be false
                for di in range(-1, 2):
                    for dj in range(-1, 2):
                        for dk in range(-1, 2):
                            if di == 0 and dj == 0 and dk == 0:  # skip itself
                                continue
                            max = max and (diff_at_scale[n][j, k, i] > diff_at_scale[n
][j + dj, k + dk, i + di])
                            min = min and (diff_at_scale[n][j, k, i] < diff_at_scale[n
][j + dj, k + dk, i + di])
                            if not max and not min:
                                break
                        if not max and not min:
                            break
                    if not max and not min:
                        break

                # at this point we have all minima and maxima,
                # the below code will filter them further by using a threshold based on
 the eigenvalues of the Hessian matrix
                if max or min:

                    # direct application of second derivatives with help from the 2-tap
 filter [-1, 1] technique
                    dxx = (diff_at_scale[n][j, k, i] + diff_at_scale[n][j - 2, k, i] -
2 * diff_at_scale[n][
                        j - 1, k, i])
                    dyy = (diff_at_scale[n][j, k, i] + diff_at_scale[n][j, k - 2, i] -
2 * diff_at_scale[n][
                        j, k - 1, i])
                    dxy = (diff_at_scale[n][j, k, i] - diff_at_scale[n][j, k - 1, i] -
diff_at_scale[n][
                        j - 1, k, i] + diff_at_scale[n][j - 1, k - 1, i])

                    # computation of the trace following relative formulas
                    tr_d = dxx + dyy

                    # computation of the determinant following relative formulas
                    det_d = dxx * dyy - (dxy ** 2)

                    r = 10.0 # value for r suggested by Lowe (2004)

                    # inequality used for separation of interest points referring to
                    # edges in contrast to the ones referring to corners
                    thr_ratio = (tr_d ** 2) / det_d
                    r_ratio = ((r + 1) ** 2) / r

                    if thr_ratio > r_ratio:
                        # mark the matrix with a value to know which are the
                        # coordinates of the interest points
                        max_min_at_scale[n][j, k, i - 1] = 1
```

I SIFT Magnitude and Orientation

```
# initialisation of gradient magnitude and orientation matrices
mag_at_scale = []
for n in range(0, n_octaves):
    mag_at_scale.append(np.zeros((img_at_scale[n].shape[0],img_at_scale[n].shape[1],3)))
ori_at_scale = []
for n in range(0, n_octaves):
    ori_at_scale.append(np.zeros((img_at_scale[n].shape[0],img_at_scale[n].shape[1],3)))

# calculate all magnitudes and orientations using simple trigonometry
# (i.e. Pythagoras' theorem and inverse tangent)
for n in range(0, 3):
    for i in range(0, 3):
        for j in range(1, img_at_scale[n].shape[0] - 1):
            for k in range(1, img_at_scale[n].shape[1] - 1):
                mag_at_scale[n][j, k, i] = (((img_at_scale[n][j + 1, k] - img_at_scale[n][j - 1, k]) ** 2) + ((img_at_scale[n][j, k + 1] - img_at_scale[n][j, k - 1]) ** 2)) ** 0.5
                # converted from radians to degrees,
                # but divided by 10 (will be useful later)
                ori_at_scale[n][j, k, i] = (36 / (2 * np.pi)) * (np.pi + np.arctan2((img_at_scale[n][j, k + 1] - img_at_scale[n][j, k - 1]), (img_at_scale[n][j + 1, k] - img_at_scale[n][j - 1, k])))

# getting the length of the keypoint matrix
key_sum = int(np.sum(max_min_at_scale[0]) + np.sum(max_min_at_scale[1]) + np.sum(max_min_at_scale[2]))

# initialising keypoint matrix
keypoints = np.zeros((key_sum, 4))

# the below loop assigns orientations and magnitudes to the selected keypoints
key_count = 0
for n in range(0, 3):  # all octaves
    for i in range(0, 3):  # all levels

        # loop through every pixel of the image
        for j in range(1, img_at_scale[n].shape[0] - 1):
            for k in range(1, img_at_scale[n].shape[1] - 1):

                # pick only pre-selected and filtered maxima and minima
                if max_min_at_scale[n][j, k, i] == 1:

                    # 1.5 times all_scales[i+n*2] as suggested by Lowe (2004)
                    window=multivariate_normal(mean=[j,k],cov=(1.5*all_scales[i+n*2]))
                    win_radius = np.floor(1.5 * all_scales[i + n * 2])

                    # this is the reason earlier we converted radians to this
                    # different type of degree
                    buckets = np.zeros([36, 1])

                    # for each bin, compute weight inside it,
                    # keeping in mind that the magnitudes close to the keypoint of
                    # interest are more important
                    for x in range(int(-1 * win_radius * 2), int(win_radius * 2) + 1):
                        # circle equation
                        square_to_circle_shape = int((((win_radius * 2) ** 2) - (np.absolute(x) ** 2)) ** 0.5)

                        # from 0 when x=-18, r
                        for y in range(-1 * square_to_circle_shape, square_to_circle_shape + 1):
```

```
                            # if window out of image
                            if j + x < 0 or \
                                    j + x > img_at_scale[n].shape[0] - 1 or \
                                    k + y < 0 or \
                                    k + y > img_at_scale[n].shape[1] - 1:
                                continue

                            # if center of circle, i.e. j and k are reached, i.e.
                            # when x and y are 0, then maximum weight is reached,
                            # for borders, weight is smaller
                            weight = mag_at_scale[n][j + x, k + y, i] * window.pdf([j + x, k + y])  # PDF

                            # refactor float to an int within bounded range
                            bucket_to_choose = int(np.clip(np.floor(ori_at_scale[n][j + x, k + y, i]) - 1, 0, 35))

                            # add the weight to the selected bin
                            buckets[bucket_to_choose] += weight

                    loc_of_max = np.argmax(buckets)
                    # matches scale of image (will be normalised for calculation
                    # of descriptors)
                    scale_factor = (2 ** (n - 1))
                    # add orientation and scale
                    keypoints[key_count, :] = np.array(
                        [int(j*scale_factor),int(k*scale_factor),all_scales[i+n*2], loc_of_max])
                    key_count += 1
```

J SIFT Descriptors

```
descriptors = np.zeros([keypoints.shape[0], 128])

# bin conversion follows proportion -> 8:36=?:ori
for i in range(0, keypoints.shape[0]):
    window = multivariate_normal(mean=[keypoints[i, 0], keypoints[i, 1]], cov=8)

    # blocks hosting the 16 histograms
    for j in range(-2, 2):
        for k in range(-2, 2):
            # if window out of image
            if keypoints[i, 0] + j * 4 < 0 or \
                    keypoints[i, 0] + (j + 1) * 4 > img_at_scale[1].shape[0] - 1 or \
                    keypoints[i, 1] + k * 4 < 0 or \
                    keypoints[i, 1] + (k + 1) * 4 > img_at_scale[1].shape[1] - 1:
                continue

            # 8 buckets for the 8 directions
            # (8 was also a value suggested by Lowe (2004))
            buckets = np.zeros([8, 1])

            # go through the 16 cells which make up each block
            # and fit its direction in the the relative bin of the histogram
            for m in range(0, 4):
                for n in range(0, 4):
                    x = j * 4 + m
                    y = k * 4 + n

                    # weight depends on the magnitude of each cell
                    # and on its distance from the centre of the Gaussian
                    weight = mag_at_scale[1][int(keypoints[i, 0] + x), int(keypoints[i, 1] + y), 0] * window.pdf([keypoints[i, 0] + x, keypoints[i, 1] + y])
                    # pdf = probability density function

```

```
33
34                      # if center of circle, i.e. j and k are reached
35                      # i.e. when x and y are 0, then maximum weight
36                      # (the same way as before)
37                      bucket_to_choose = int(np.clip(  # fit into buckets
38                          a=np.floor(
39                              ori_at_scale[1][int(keypoints[i, 0] + x), int(keypoints[i,
    1] + y), 0] * 8 / 36
40                          ) - 1,
41                          a_min=0,
42                          a_max=7
43                      ))
44                      buckets[bucket_to_choose] += weight
45
46                      # setting up relative coordinate environment to simplify indices
47                      j_helper = j + 2
48                      k_helper = k + 2
49
50                      # histogram number
51                      current_4by4_chunck = j_helper * 4 + k_helper
52
53                      # range for 1D interpretation of histograms
54                      current_location_in_megaarray = current_4by4_chunck * 8
55
56                      # location of exact bin (among the 128 possible bins)
57                      # in the 128-dimensional descriptor array
58                      descriptors[i,current_location_in_megaarray+bucket_to_choose]+=
    weight
```

K SIFT Box Drawing with Direction

```
1  def draw_sift_boxes_and_directions(img, keypoints):
2
3      for i in range(0, keypoints.shape[0]):
4          x = int(keypoints[i][1])
5          y = int(keypoints[i][0])
6
7          scale = int(keypoints[i][2])
8          rotation = keypoints[i][3] * 10
9          alpha = np.deg2rad(rotation)
10
11         # draw circle
12         cv2.circle(img, (x, y), scale, (0, 255, 0), thickness=6, lineType=8, shift=0)
13
14         # draw line for direction (extend beyond circle perimeter)
15         cv2.line(
16             img=img,
17             pt1=(x, y),
18             pt2=(int(x +(np.cos(alpha)*scale*3)), int(y+(np.sin(alpha)*scale*3))),
19             color=(0,0,255),
20             thickness=4
21         )
22
23     return img
```