Assignment 3

Computer Vision

DR. Ahmed M. Badawi
T.A. Laila Abbas
T.A. Peter Salah

STUDENT NAME	SEC	BN.
GHOFRAN MOHAMMED	2	8
KAREMAN YASER	2	9
MAYAR FAYEZ	2	42
NADA AHMED	2	46
NAIRA YOUSSEF	2	48

Description:

A small web application based app developed with python and streamlit, to apply different image processing techniques.

Requirements:

- Python 3.
- Streamlit 1.13.0
- Numpy 1.23.4
- Matplotlib 3.6.2

Running command:

Streamlit run server.py

- The UI contains two main tabs Features Generation, Matching

Tab1:

- Harris
- SIFT

❖ Harris:

Harris Corner Detector is a corner detection operator that is commonly used in computer vision algorithms to extract corners and infer features of an image. Corners are the important features in the image, and they are generally termed as interest points which are invariant to translation, rotation, and illumination. Harris, and Stephens developed the Harris Corner Detector [1], a mathematical approach to detect corners and edges in images. They picked the statements of Moravec and gave it a mathematical signification, Equation 1.

$$E(u,v) = \sum_{x,y} \underbrace{w(x,y)}_{\text{window function}} \, [\underbrace{I(x+u,y+v)}_{\text{shifted intensity}} - \underbrace{I(x,y)}_{\text{intensity}}]^2$$

Equation 1 – Matematical formulation to find the difference in intensity for a sift of (u,v) in a image.

After apply Taylor expansion it is possible to obtain the following approximation of Equation2.

$$E(u,v) \approx \begin{bmatrix} u & v \end{bmatrix} M \begin{bmatrix} u \\ v \end{bmatrix}$$

where:

$$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x I_x & I_x I_y \\ I_x I_y & I_y I_y \end{bmatrix}$$

Equation 2 – Taylor expansion.

The Ix and Iy present in Equation 2 are the x and y image derivatives. To conclude, the response of the corner detector is obtained with Equation 3. Depending on the value obtained by this response, is possible to determine if a region contains a flat region, an edge, or a corner.

$$R = \det(M) - k(\operatorname{trace}(M))^2$$

Equation 3 – Response function of the corner detector.

Algorithm:

```
def harris(im, sigma=1.0, relTh=0.0001, k=0.04):

    im = im.astype(np.float) # Make sure im is float

    # Get smoothing and derivative filters
    g, _, _, _, _, _ = gaussian2(sigma)
    _, gx, gy, _, _, _ = gaussian2(np.sqrt(0.5))

    # Partial derivatives
    Ix = conv2(im, -gx, mode='constant')
    Iy = conv2(im, -gy, mode='constant')

    # Components of the second moment matrix
    Ix2Sm = conv2(Ix**2, g, mode='constant')
    Iy2Sm = conv2(Iy**2, g, mode='constant')
    IxIySm = conv2(Ix*Iy, g, mode='constant')

    # Determinant and trace for calculating the corner response
    detC = (Ix2Sm*IxIySm)-(Iy2Sm**2)
    traceC = Ix2Sm+IxIySm

    # Corner response function R
    # "Corner": R > 0
    # "Edge": R < 0
    # "Flat": |R| = small
    R = detC-k*traceC**2
    maxCornerValue = np.amax(R)

    # Take only the local maxima of the corner response function
    fp = np.ones((3,3))
    fp[1,1] = 0
    maxImg = maximum_filter(R, footprint=fp, mode='constant')

    # Test if cornerness is larger than neighborhood
    cornerImg = R>maxImg

    # Threshold for low value maxima
    y, x = np.nonzero((R>relTh*maxCornerValue)*cornerImg)

    # Convert to float
    x = x.astype(np.float)
    y = y.astype(np.float)

    # Remove responses from image borders to reduce false corner detections
    r, c = R.shape
    idx = np.nonzero((x<2)+(x>c-3)+(y<2)+(y>r-3))[0]
    x = np.delete(x,idx)
```

```python
    y = np.delete(y,idx)

    # Parabolic interpolation
    for i in range(len(x)):
        _,dx=maxinterp((R[int(y[i]), int(x[i])-1], R[int(y[i]), int(x[i])],
R[int(y[i]), int(x[i])+1]))
        _,dy=maxinterp((R[int(y[i])-1, int(x[i])], R[int(y[i]), int(x[i])],
R[int(y[i])+1, int(x[i])]))
        x[i]=x[i]+dx
        y[i]=y[i]+dy

    return x, y, cornerImg
```

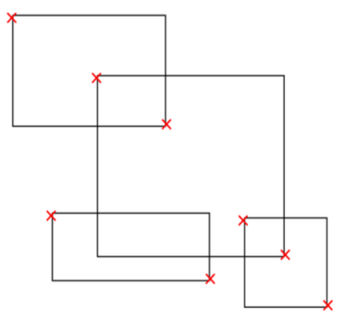
in this algorithm we change the image type to float then we do the next steps

1. Calculate image x and y derivatives.
2. Derivate again the previous values to obtain the second derivative;
3. For each pixel, sum the last step obtained derivatives. Here we are making a 1 pixel sift of the windows over the image;
4. For each pixel and using the sums of the previous step, define H matrix;
5. Calculate the response of the detector;
6. Use a threshold value in order to exclude some of the detections.

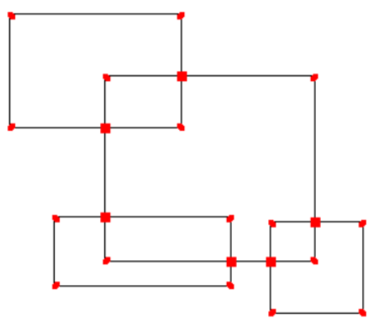
Results:

Computation time equals 0.11477828025817871 seconds on average

Our results



CV2 results



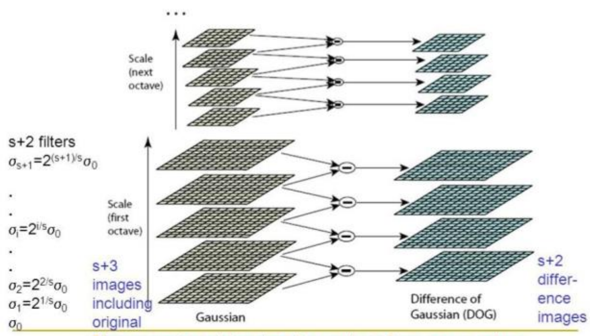
- # SIFT:

  Scale Invariant Feature Transform, used to extract features from images using 4 steps:

  1. Scale space construction
  2. Scale space extrema detection
  3. Orientation Assignment
  4. Key point descriptor

1. Scale Space Construction

   Search over multiple scales and image locations.



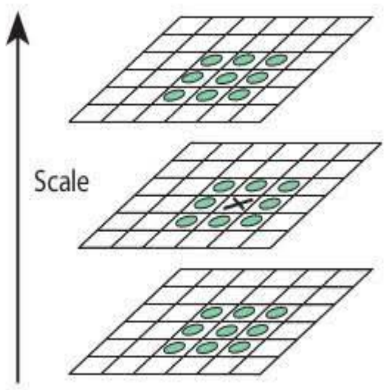
The parameter **s** determines the number of images per octave.

Algorithm

```
def generat_Scales(sigma,S):
    #Generate list of gaussian Scales at which to blur the input image.
    # S :the parameter determines the number of scales per octave
    scales = S+3                        #the number of images per octave
    k = 2 ** (0.5)
    gaussian_Scales = zeros(scales)  # scale of gaussian blur necessary to go from one blur scale to the next within an octave
    gaussian_Scales[0] = sigma

    for image_index in range(1, scales):
        sigma_previous = (k ** (image_index - 1)) * sigma
        sigma_total = k * sigma_previous
        gaussian_Scales[image_index] = sqrt(sigma_total ** 2 - sigma_previous ** 2)
    return gaussian_Scales

def generate_Octaves(image,sigma, num_octaves, gaussian_Scales):
    image = resize(image, (0, 0), fx=2, fy=2, interpolation=INTER_LINEAR)
    sigma_diff = sqrt(max((sigma ** 2) - ((2 *0.5) ** 2), 0.01))
    image=GaussianBlur(image, (0, 0), sigmaX=sigma_diff, sigmaY=sigma_diff)
    gaussian_images = []

    for octave_index in range(num_octaves):
        gaussian_images_in_octave = []
        gaussian_images_in_octave.append(image)  # first image in octave already has the correct blur
        for gaussian_kernel in gaussian_Scales[1:]:
            image = GaussianBlur(image, (0, 0), sigmaX=gaussian_kernel, sigmaY=gaussian_kernel)
            gaussian_images_in_octave.append(image)
        gaussian_images.append(gaussian_images_in_octave)
```

```
        octave_base = gaussian_images_in_octave[-3]
        image = resize(octave_base, (int(octave_base.shape[1] / 2),
int(octave_base.shape[0] / 2)), interpolation=INTER_NEAREST)
    return array(gaussian_images, dtype=object)

def generateDoGImages(gaussian_images):
    #Generate Difference-of-Gaussians image pyramid
    dog_images = []

    for gaussian_images_in_octave in gaussian_images:
        dog_images_in_octave = []
        for first_image, second_image in zip(gaussian_images_in_octave,
gaussian_images_in_octave[1:]):
            dog_images_in_octave.append(subtract(second_image, first_image))
        dog_images.append(dog_images_in_octave)
    return array(dog_images, dtype=object)
```

2. Scale Space Extrema Detection



●Detect maxima and minima of differences of Gaussian in scale space.

●Each point is compared to its 8 neighbours in the current image and 9 neighbours each in the scales above and below.

●Reject flats

●The Hessian matrix was used to eliminate edge responses.

Algorithm

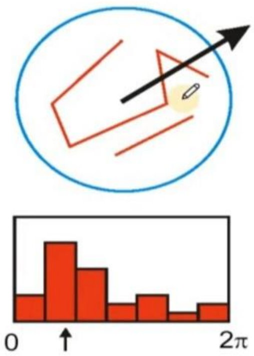
```
def findScaleSpaceExtrema(gaussian_images, dog_images, S, sigma, image_border_width,
contrast_threshold=0.04):
    #Find pixel positions of all scale-space extrema in the octave
    threshold = floor(0.5 * contrast_threshold / S * 255)
    keypoints = []

    for octave_index, dog_images_in_octave in enumerate(dog_images):
        for image_index, (first_image, second_image, third_image) in
enumerate(zip(dog_images_in_octave, dog_images_in_octave[1:],
dog_images_in_octave[2:])):
            # (i, j) is the center of the 3x3 array
            for i in range(image_border_width, first_image.shape[0] -
image_border_width):
                for j in range(image_border_width, first_image.shape[1] -
image_border_width):
```

```
                    if key_points(first_image[i-1:i+2, j-1:j+2], second_image[i-1:i+2,
j-1:j+2], third_image[i-1:i+2, j-1:j+2], threshold):
                        localization_result = localizeExtremumViaQuadraticFit(i, j,
image_index + 1, octave_index, S, dog_images_in_octave, sigma, contrast_threshold,
image_border_width)
                        if localization_result is not None:
                            keypoint, localized_image_index = localization_result
                            keypoints_with_orientations =
computeKeypointsWithOrientations(keypoint, octave_index,
gaussian_images[octave_index][localized_image_index])
                            for keypoint_with_orientation in
keypoints_with_orientations:
                                keypoints.append(keypoint_with_orientation)
    return keypoints

def key_points(first_subimage, second_subimage, third_subimage, threshold):
    #Return True if the center element of the 3x3x3 input array is strictly greater
than or less than all its neighbors, False otherwise

    center_pixel_value = second_subimage[1, 1]
    if abs(center_pixel_value) > threshold:
        if center_pixel_value > 0:
            return all(center_pixel_value >= first_subimage) and \
                   all(center_pixel_value >= third_subimage) and \
                   all(center_pixel_value >= second_subimage[0, :]) and \
                   all(center_pixel_value >= second_subimage[2, :]) and \
                   center_pixel_value >= second_subimage[1, 0] and \
                   center_pixel_value >= second_subimage[1, 2]
        elif center_pixel_value < 0:
            return all(center_pixel_value <= first_subimage) and \
                   all(center_pixel_value <= third_subimage) and \
                   all(center_pixel_value <= second_subimage[0, :]) and \
                   all(center_pixel_value <= second_subimage[2, :]) and \
                   center_pixel_value <= second_subimage[1, 0] and \
                   center_pixel_value <= second_subimage[1, 2]
    return False

def localizeExtremumViaQuadraticFit(i, j, image_index, octave_index, S,
dog_images_in_octave, sigma, contrast_threshold, image_border_width,
eigenvalue_ratio=10, num_attempts_until_convergence=5):
    #Iteratively refine pixel positions of scale-space extrema via quadratic fit around
each extremum's neighbors
    extremum_is_outside_image = False
    image_shape = dog_images_in_octave[0].shape
    for attempt_index in range(num_attempts_until_convergence):
        # need to convert from uint8 to float32 to compute derivatives and need to
rescale pixel values to [0, 1] to apply Lowe's thresholds
```

```python
        first_image, second_image, third_image = dog_images_in_octave[image_index-1:image_index+2]
        pixel_cube = stack([first_image[i-1:i+2, j-1:j+2],
                            second_image[i-1:i+2, j-1:j+2],
                            third_image[i-1:i+2, j-1:j+2]]).astype('float32') / 255.
        gradient = computeGradientAtCenterPixel(pixel_cube)
        hessian = computeHessianAtCenterPixel(pixel_cube)
        extremum_update = -lstsq(hessian, gradient, rcond=None)[0]
        if abs(extremum_update[0]) < 0.5 and abs(extremum_update[1]) < 0.5 and abs(extremum_update[2]) < 0.5:
            break
        j += int(round(extremum_update[0]))
        i += int(round(extremum_update[1]))
        image_index += int(round(extremum_update[2]))
        # make sure the new pixel_cube will lie entirely within the image
        if i < image_border_width or i >= image_shape[0] - image_border_width or j < image_border_width or j >= image_shape[1] - image_border_width or image_index < 1 or image_index > S:
            extremum_is_outside_image = True
            break
    if extremum_is_outside_image:
        # Updated extremum moved outside of image before reaching convergence. Skipping...
        return None
    if attempt_index >= num_attempts_until_convergence - 1:
        # Exceeded maximum number of attempts without reaching convergence for this extremum. Skipping...
        return None
    functionValueAtUpdatedExtremum = pixel_cube[1, 1, 1] + 0.5 * dot(gradient, extremum_update)
    if abs(functionValueAtUpdatedExtremum) * S >= contrast_threshold:
        xy_hessian = hessian[:2, :2]
        xy_hessian_trace = trace(xy_hessian)
        xy_hessian_det = det(xy_hessian)
        if xy_hessian_det > 0 and eigenvalue_ratio * (xy_hessian_trace ** 2) < ((eigenvalue_ratio + 1) ** 2) * xy_hessian_det:
            keypoint = KeyPoint()
            keypoint.pt = ((j + extremum_update[0]) * (2 ** octave_index), (i + extremum_update[1]) * (2 ** octave_index))
            keypoint.octave = octave_index + image_index * (2 ** 8) + int(round((extremum_update[2] + 0.5) * 255)) * (2 ** 16)
            keypoint.size = sigma * (2 ** ((image_index + extremum_update[2]) / float32(S))) * (2 ** (octave_index + 1))  # octave_index + 1 because the input image was doubled
            keypoint.response = abs(functionValueAtUpdatedExtremum)
            return keypoint, image_index
    return None
```

```
def computeGradientAtCenterPixel(pixel_array):
    # Approximate gradient at center pixel [1, 1, 1] of 3x3x3 array using central
difference formula of order O(h^2), where h is the step size
    # With step size h, the central difference formula of order O(h^2) for f'(x) is
(f(x + h) - f(x - h)) / (2 * h)
    # Here h = 1, so the formula simplifies to f'(x) = (f(x + 1) - f(x - 1)) / 2
    # NOTE: x corresponds to second array axis, y corresponds to first array axis, and
s (scale) corresponds to third array axis
    dx = 0.5 * (pixel_array[1, 1, 2] - pixel_array[1, 1, 0])
    dy = 0.5 * (pixel_array[1, 2, 1] - pixel_array[1, 0, 1])
    ds = 0.5 * (pixel_array[2, 1, 1] - pixel_array[0, 1, 1])
    return array([dx, dy, ds])

def computeHessianAtCenterPixel(pixel_array):
    # Approximate Hessian at center pixel [1, 1, 1] of 3x3x3 array using central
difference formula of order O(h^2), where h is the step size

    # With step size h, the central difference formula of order O(h^2) for f''(x) is
(f(x + h) - 2 * f(x) + f(x - h)) / (h ^ 2)
    # Here h = 1, so the formula simplifies to f''(x) = f(x + 1) - 2 * f(x) + f(x - 1)
    # With step size h, the central difference formula of order O(h^2) for (d^2) f(x,
y) / (dx dy) = (f(x + h, y + h) - f(x + h, y - h) - f(x - h, y + h) + f(x - h, y - h))
/ (4 * h ^ 2)
    # Here h = 1, so the formula simplifies to (d^2) f(x, y) / (dx dy) = (f(x + 1, y +
1) - f(x + 1, y - 1) - f(x - 1, y + 1) + f(x - 1, y - 1)) / 4
    # NOTE: x corresponds to second array axis, y corresponds to first array axis, and
s (scale) corresponds to third array axis
    center_pixel_value = pixel_array[1, 1, 1]
    dxx = pixel_array[1, 1, 2] - 2 * center_pixel_value + pixel_array[1, 1, 0]
    dyy = pixel_array[1, 2, 1] - 2 * center_pixel_value + pixel_array[1, 0, 1]
    dss = pixel_array[2, 1, 1] - 2 * center_pixel_value + pixel_array[0, 1, 1]
    dxy = 0.25 * (pixel_array[1, 2, 2] - pixel_array[1, 2, 0] - pixel_array[1, 0, 2] +
pixel_array[1, 0, 0])
    dxs = 0.25 * (pixel_array[2, 1, 2] - pixel_array[2, 1, 0] - pixel_array[0, 1, 2] +
pixel_array[0, 1, 0])
    dys = 0.25 * (pixel_array[2, 2, 1] - pixel_array[2, 0, 1] - pixel_array[0, 2, 1] +
pixel_array[0, 0, 1])
    return array([[dxx, dxy, dxs],
                  [dxy, dyy, dys],
                  [dxs, dys, dss]])
```

3. Orientation Assignment



●Create histogram of local gradient directions at selected scale.

●Assign canonical orientation at peak of smoothed histogram

●Each key specifies stable 2D coordinates

●Histogram of gradient orientation bin-counts are weighted by gradient magnitudes and a Gaussian Weighting function. Usually,36 bins are chosen for the orientation.

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2}$$

$$\theta(x,y) = \tan^{-1}((L(x,y+1) - L(x,y-1))/(L(x+1,y) - L(x-1,y)))$$

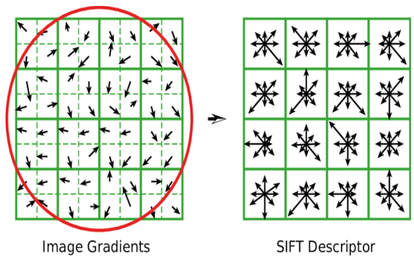
<u>**Algorithm**</u>

```
def computeKeypointsWithOrientations(keypoint, octave_index, gaussian_image,
radius_factor=3, num_bins=36, peak_ratio=0.8, scale_factor=1.5):
    #Compute orientations for each keypoint
    keypoints_with_orientations = []
    image_shape = gaussian_image.shape

    scale = scale_factor * keypoint.size / float32(2 ** (octave_index + 1))  # compare
with keypoint.size computation in localizeExtremumViaQuadraticFit()
    radius = int(round(radius_factor * scale))
    weight_factor = -0.5 / (scale ** 2)
    raw_histogram = zeros(num_bins)
    smooth_histogram = zeros(num_bins)

    for i in range(-radius, radius + 1):
        region_y = int(round(keypoint.pt[1] / float32(2 ** octave_index))) + i
        if region_y > 0 and region_y < image_shape[0] - 1:
            for j in range(-radius, radius + 1):
                region_x = int(round(keypoint.pt[0] / float32(2 ** octave_index))) + j
                if region_x > 0 and region_x < image_shape[1] - 1:
                    dx = gaussian_image[region_y, region_x + 1] -
gaussian_image[region_y, region_x - 1]
                    dy = gaussian_image[region_y - 1, region_x] -
gaussian_image[region_y + 1, region_x]
                    gradient_magnitude = sqrt(dx * dx + dy * dy)
                    gradient_orientation = rad2deg(arctan2(dy, dx))
                    weight = exp(weight_factor * (i ** 2 + j ** 2))  # constant in
front of exponential can be dropped because we will find peaks later
                    histogram_index = int(round(gradient_orientation * num_bins /
360.))
```

```
                    raw_histogram[histogram_index % num_bins] += weight * 
gradient_magnitude

    for n in range(num_bins):
        smooth_histogram[n] = (6 * raw_histogram[n] + 4 * (raw_histogram[n - 1] + 
raw_histogram[(n + 1) % num_bins]) + raw_histogram[n - 2] + raw_histogram[(n + 2) % 
num_bins]) / 16.
    orientation_max = max(smooth_histogram)
    orientation_peaks = where(logical_and(smooth_histogram > roll(smooth_histogram, 1), 
smooth_histogram > roll(smooth_histogram, -1)))[0]
    for peak_index in orientation_peaks:
        peak_value = smooth_histogram[peak_index]
        if peak_value >= peak_ratio * orientation_max:
            left_value = smooth_histogram[(peak_index - 1) % num_bins]
            right_value = smooth_histogram[(peak_index + 1) % num_bins]
            interpolated_peak_index = (peak_index + 0.5 * (left_value - right_value) / 
(left_value - 2 * peak_value + right_value)) % num_bins
            orientation = 360. - interpolated_peak_index * 360. / num_bins
            if abs(orientation - 360.) < float_tolerance:
                orientation = 0
            new_keypoint = KeyPoint(*keypoint.pt, keypoint.size, orientation, 
keypoint.response, keypoint.octave)
            keypoints_with_orientations.append(new_keypoint)
    return keypoints_with_orientations
```

4. Key point descriptor

Use local image gradients at selected scale and rotation to describe each key point region.



Algorithm

```
def unpackOctave(keypoint):
    #Compute octave, layer, and scale from a keypoint
    octave = keypoint.octave & 255
    layer = (keypoint.octave >> 8) & 255
    if octave >= 128:
        octave = octave | -128
    scale = 1 / float32(1 << octave) if octave >= 0 else float32(1 << -octave)
    return octave, layer, scale

def generateDescriptors(keypoints, gaussian_images, window_width=4, num_bins=8, 
scale_multiplier=3, descriptor_max_value=0.2):
    #Generate descriptors for each keypoint
    descriptors = []

    for keypoint in keypoints:
        octave, layer, scale = unpackOctave(keypoint)
```

```
        gaussian_image = gaussian_images[octave + 1, layer]
        num_rows, num_cols = gaussian_image.shape
        point = round(scale * array(keypoint.pt)).astype('int')
        bins_per_degree = num_bins / 360.
        angle = 360. - keypoint.angle
        cos_angle = cos(deg2rad(angle))
        sin_angle = sin(deg2rad(angle))
        weight_multiplier = -0.5 / ((0.5 * window_width) ** 2)
        row_bin_list = []
        col_bin_list = []
        magnitude_list = []
        orientation_bin_list = []
        histogram_tensor = zeros((window_width + 2, window_width + 2, num_bins))   #
first two dimensions are increased by 2 to account for border effects

        # Descriptor window size (described by half_width)
        hist_width = scale_multiplier * 0.5 * scale * keypoint.size
        half_width = int(round(hist_width * sqrt(2) * (window_width + 1) * 0.5))   #
sqrt(2) corresponds to diagonal length of a pixel
        half_width = int(min(half_width, sqrt(num_rows ** 2 + num_cols ** 2)))     #
ensure half_width lies within image

        for row in range(-half_width, half_width + 1):
            for col in range(-half_width, half_width + 1):
                row_rot = col * sin_angle + row * cos_angle
                col_rot = col * cos_angle - row * sin_angle
                row_bin = (row_rot / hist_width) + 0.5 * window_width - 0.5
                col_bin = (col_rot / hist_width) + 0.5 * window_width - 0.5
                if row_bin > -1 and row_bin < window_width and col_bin > -1 and col_bin
< window_width:
                    window_row = int(round(point[1] + row))
                    window_col = int(round(point[0] + col))
                    if window_row > 0 and window_row < num_rows - 1 and window_col > 0
and window_col < num_cols - 1:
                        dx = gaussian_image[window_row, window_col + 1] -
gaussian_image[window_row, window_col - 1]
                        dy = gaussian_image[window_row - 1, window_col] -
gaussian_image[window_row + 1, window_col]
                        gradient_magnitude = sqrt(dx * dx + dy * dy)
                        gradient_orientation = rad2deg(arctan2(dy, dx)) % 360
                        weight = exp(weight_multiplier * ((row_rot / hist_width) ** 2 +
(col_rot / hist_width) ** 2))
                        row_bin_list.append(row_bin)
                        col_bin_list.append(col_bin)
                        magnitude_list.append(weight * gradient_magnitude)
                        orientation_bin_list.append((gradient_orientation - angle) *
bins_per_degree)
```

```
        for row_bin, col_bin, magnitude, orientation_bin in zip(row_bin_list, col_bin_list, magnitude_list, orientation_bin_list):
            # Smoothing via trilinear interpolation
            # Note that we are really doing the inverse of trilinear interpolation here (we take the center value of the cube and distribute it among its eight neighbors)
            row_bin_floor, col_bin_floor, orientation_bin_floor = floor([row_bin, col_bin, orientation_bin]).astype(int)
            row_fraction, col_fraction, orientation_fraction = row_bin - row_bin_floor, col_bin - col_bin_floor, orientation_bin - orientation_bin_floor
            if orientation_bin_floor < 0:
                orientation_bin_floor += num_bins
            if orientation_bin_floor >= num_bins:
                orientation_bin_floor -= num_bins

            c1 = magnitude * row_fraction
            c0 = magnitude * (1 - row_fraction)
            c11 = c1 * col_fraction
            c10 = c1 * (1 - col_fraction)
            c01 = c0 * col_fraction
            c00 = c0 * (1 - col_fraction)
            c111 = c11 * orientation_fraction
            c110 = c11 * (1 - orientation_fraction)
            c101 = c10 * orientation_fraction
            c100 = c10 * (1 - orientation_fraction)
            c011 = c01 * orientation_fraction
            c010 = c01 * (1 - orientation_fraction)
            c001 = c00 * orientation_fraction
            c000 = c00 * (1 - orientation_fraction)

            histogram_tensor[row_bin_floor + 1, col_bin_floor + 1, orientation_bin_floor] += c000
            histogram_tensor[row_bin_floor + 1, col_bin_floor + 1, (orientation_bin_floor + 1) % num_bins] += c001
            histogram_tensor[row_bin_floor + 1, col_bin_floor + 2, orientation_bin_floor] += c010
            histogram_tensor[row_bin_floor + 1, col_bin_floor + 2, (orientation_bin_floor + 1) % num_bins] += c011
            histogram_tensor[row_bin_floor + 2, col_bin_floor + 1, orientation_bin_floor] += c100
            histogram_tensor[row_bin_floor + 2, col_bin_floor + 1, (orientation_bin_floor + 1) % num_bins] += c101
            histogram_tensor[row_bin_floor + 2, col_bin_floor + 2, orientation_bin_floor] += c110
            histogram_tensor[row_bin_floor + 2, col_bin_floor + 2, (orientation_bin_floor + 1) % num_bins] += c111

        descriptor_vector = histogram_tensor[1:-1, 1:-1, :].flatten()  # Remove histogram borders
```

```
        # Threshold and normalize descriptor_vector
        threshold = norm(descriptor_vector) * descriptor_max_value
        descriptor_vector[descriptor_vector > threshold] = threshold
        descriptor_vector /= max(norm(descriptor_vector), float_tolerance)
        # Multiply by 512, round, and saturate between 0 and 255 to convert from float32 to unsigned char (OpenCV convention)
        descriptor_vector = round(512 * descriptor_vector)
        descriptor_vector[descriptor_vector < 0] = 0
        descriptor_vector[descriptor_vector > 255] = 255
        descriptors.append(descriptor_vector)
```

Results:

Computation time equals `66.9212273999999` in seconds.

Our results

CV2 results

Tab2:

- Using SSD
- Using NCC

Matching Harris corner points

❖ Using SSD:

The aim is to first detect Harris corners from two images of the same scene. Then, image patches of size 15x15 pixels around each detected corner point is extracted following a matching step where mutually nearest neighbors are found using the sum of squared differences (SSD) similarity measure.

The SSD measure for two image patches, f and g, is defined as follows

$SSD(f,g) = \sum_{k,l}(g(k,l) - f(k,l))^2$ so that the larger the SSD value the more dissimilar the patches are.

Steps:

- Harris corner extraction
- We pre-allocate the memory for the 15*15 image patches extracted around each corner point from both images, then extracts the patches using bilinear interpolation
- We compute the sum of squared differences (SSD) of pixels' intensities for all pairs of patches extracted from the two images
- Next we compute pairs of patches that are mutually nearest neighbors according to the SSD measure
- We sort the mutually nearest neighbors based on the SSD
- Estimate the geometric transformation between images
- Next we visualize the 40 best matches which are mutual nearest neighbors and have the smallest SSD values
- Finally, since we have estimated the planar projective transformation, we can check that how many of the nearest neighbor matches actually are correct correspondences

Algorithm :

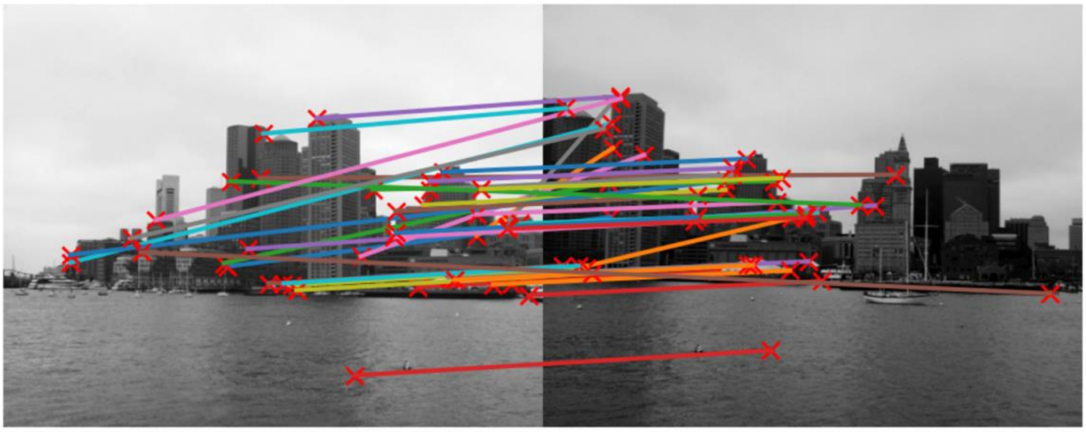
```
def match_SSD (I1,I2):
  start = time.time()
  # Harris corner extraction
  x1, y1, image1 = harris(I1)
  x2, y2, image2 = harris(I2)
  # We pre-allocate the memory for the 15*15 image patches extracted
  # around each corner point from both images
  patch_size=15
  npts1=x1.shape[0]
  npts2=x2.shape[0]
```

```python
patches1=np.zeros((patch_size, patch_size, npts1))
patches2=np.zeros((patch_size, patch_size, npts2))
# The following part extracts the patches using bilinear interpolation
k=(patch_size-1)/2.
xv,yv=np.meshgrid(np.arange(-k,k+1),np.arange(-k, k+1))
for i in range(npts1):
  patch = map_coordinates(I1, (yv + y1[i], xv + x1[i]))
  patches1[:,:,i] = patch
for i in range(npts2):
  patch = map_coordinates(I2, (yv + y2[i], xv + x2[i]))
  patches2[:,:,i] = patch
# We compute the sum of squared differences (SSD) of pixels' intensities
# for all pairs of patches extracted from the two images
distmat = np.zeros((npts1, npts2))
for i1 in range(npts1):
    for i2 in range(npts2):
        distmat[i1,i2]=np.sum((patches1[:,:,i1]-patches2[:,:,i2])**2)
# Next we compute pairs of patches that are mutually nearest neighbors
# according to the SSD measure
ss1 = np.amin(distmat, axis=1)
ids1 = np.argmin(distmat, axis=1)
ss2 = np.amin(distmat, axis=0)
ids2 = np.argmin(distmat, axis=0)
pairs = []
for k in range(npts1):
    if k == ids2[ids1[k]]:
        pairs.append(np.array([k, ids1[k], ss1[k]]))
pairs = np.array(pairs)

# We sort the mutually nearest neighbors based on the SSD
sorted_ssd = np.sort(pairs[:,2], axis=0)
id_ssd = np.argsort(pairs[:,2], axis=0)

# Estimate the geometric transformation between images
src=[]
dst=[]
for k in range(len(id_ssd)):
    l = id_ssd[k]
    src.append([x1[int(pairs[l, 0])], y1[int(pairs[l, 0])]])
    dst.append([x2[int(pairs[l, 1])], y2[int(pairs[l, 1])]])
src=np.array(src)
dst=np.array(dst)
rthrs=2
tform,_ = ransac((src, dst), ProjectiveTransform, min_samples=4,
              residual_threshold=rthrs, max_trials=1000)
H1to2p = tform.params

# Next we visualize the 40 best matches which are mutual nearest neighbors
```

```
# and have the smallest SSD values
Nvis = 40

montage = np.concatenate((I1, I2), axis=1)

fig = plt.figure()
plt.imshow(montage, cmap='gray')
plt.axis('off')

for k in range(np.minimum(len(id_ssd), Nvis)):
    l = id_ssd[k]
    plt.plot(x1[int(pairs[l, 0])], y1[int(pairs[l, 0])], 'rx')
    plt.plot(x2[int(pairs[l, 1])] + I1.shape[1], y2[int(pairs[l, 1])], 'rx')
    plt.plot([x1[int(pairs[l, 0])], x2[int(pairs[l, 1])]+I1.shape[1]],
       [y1[int(pairs[l, 0])], y2[int(pairs[l, 1])]])

p1to2=np.dot(H1to2p, np.hstack((src, np.ones((src.shape[0],1)))).T)
p1to2 = p1to2[:2,:] / p1to2[2,:]
p1to2 = p1to2.T
pdiff=np.sqrt(np.sum((dst-p1to2)**2, axis=1))

n_correct = len(pdiff[pdiff<rthrs])

# Ouput the execution time
exe_time = str(time.time() - start)

return fig, n_correct, exe_time
```

Results :

```
24 Correct Matches.
```



```
Total time elapsed (s): 2.823293924331665
```

Computation time equals 2.8232933924331665 in seconds.

❖ Using NCC

from SSD to NCC:

- You need to determine the mutually nearest neighbors by finding pairs for which NCC is maximized (i.e. not minimized like SSD).
- Also, you need to sort the matches in descending order in terms of NCC
- in order to find the best matches (i.e. not ascending order as with SSD).

$$NCC(f,g) = \frac{\sum_{k,l}(g(k,l)-\bar{g})(f(k,l)-\bar{f})}{\sum_{k,l}(g(k,l)-\bar{g})^2\sum_{k,l}(f(k,l)-\bar{f})^2}$$

where $\bar{g}$ and $\bar{f}$ are the mean intensity values of patches g and f. The values of NCC are always between -1 and 1, and the larger the value the more similar the patches are.

Steps:

- Harris corner extraction
- We pre-allocate the memory for the 15*15 image patches extracted around each corner point from both images, then extracts the patches using bilinear interpolation
- We compute the sum of squared differences (SSD) of pixels' intensities for all pairs of patches extracted from the two images
- Compute Normalized cross correlation for each windows
- Next we compute pairs of patches that are mutually nearest neighbors according to the ncc measure
- We sort the mutually nearest neighbors based on the ncc
- Estimate the geometric transformation between images
- Next we visualize the 40 best matches which are mutual nearest neighbors and have the smallest ncc values
- Finally, since we have estimated the planar projective transformation, we can check that how many of the nearest neighbor matches actually are correct correspondences

Algorithm:

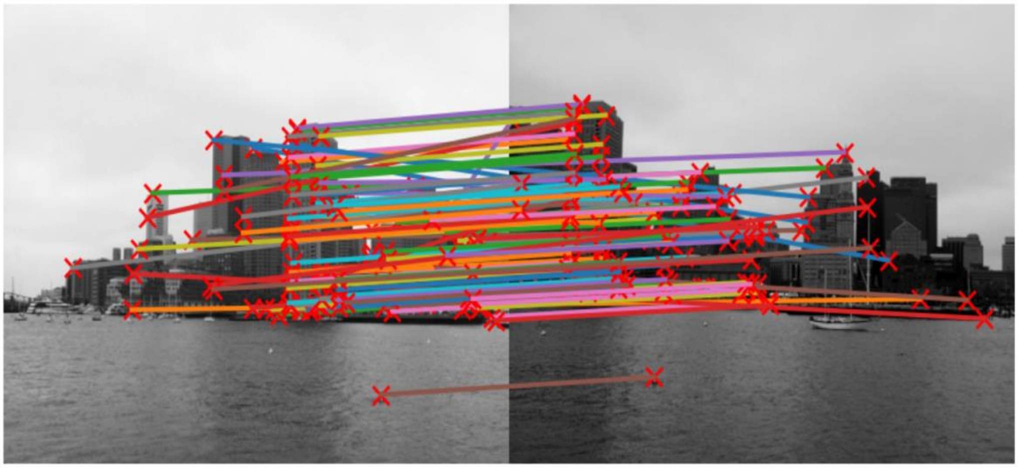
```
def match_NCC (I1,I2):

  start = time.time()
  # Harris corner extraction
  x1, y1, cimg1 = harris(I1)
  x2, y2, cimg2 = harris(I2)
  # According to SSD Function
  patch_size=15
  npts1=x1.shape[0]
  npts2=x2.shape[0]
  patches1=np.zeros((patch_size, patch_size, npts1))
  patches2=np.zeros((patch_size, patch_size, npts2))
```

```
distmat = np.zeros((npts1, npts2))
# The following part extracts the patches using bilinear interpolation
k=(patch_size-1)/2.
xv,yv=np.meshgrid(np.arange(-k,k+1),np.arange(-k, k+1))

for i in range(npts1):
  patch = map_coordinates(I1, (yv + y1[i], xv + x1[i]))
  patches1[:,:,i] = patch

for i in range(npts2):
  patch = map_coordinates(I2, (yv + y2[i], xv + x2[i]))
  patches2[:,:,i] = patch

for i1 in range(npts1):
  for i2 in range(npts2):
    distmat[i1,i2]=np.sum((patches1[:,:,i1]-patches2[:,:,i2])**2)

ss1 = np.amin(distmat, axis=1)
# Compute Normalized cross correlation for each windows
ncc = np.zeros((npts1, npts2))
for i1 in range(npts1):
  for i2 in range(npts2):
    n1 = patches1[:,:,i1] - np.mean(patches1[:,:,i1])
    n2 = patches2[:,:,i2] - np.mean(patches2[:,:,i2])
    ncc[i1,i2] = np.sum(n1*n2)/np.sqrt(np.sum(n1**2)*np.sum(n2**2))
# Next we compute pairs of patches that are mutually nearest neighbors
# according to the ncc measure
ncc1 = np.amax(ncc, axis=1)
ids1 = np.argmax(ncc, axis=1)
ncc2 = np.amax(ncc, axis=0)
ids2 = np.argmax(ncc, axis=0)

pairs = []
for k in range(npts1):
  if k == ids2[ids1[k]]:
    pairs.append(np.array([k, ids1[k], ss1[k]]))
pairs = np.array(pairs)

# We sort the mutually nearest neighbors based on the ncc
sorted_ncc = np.sort(pairs[:,2], axis=0)[::-1]
id_ncc = np.argsort(pairs[:,2], axis=0)[::-1]
# Estimate the geometric transformation between images
src=[]
dst=[]
for k in range(len(id_ncc)):
  l = id_ncc[k]
  src.append([x1[int(pairs[l, 0])], y1[int(pairs[l, 0])]])
  dst.append([x2[int(pairs[l, 1])], y2[int(pairs[l, 1])]])
```

```
src=np.array(src)
dst=np.array(dst)
rthrs=2
tform,_ = ransac((src, dst), ProjectiveTransform, min_samples=4,
      residual_threshold=rthrs, max_trials=1000)
H1to2p = tform.params
# Next we visualize the 40 best matches which are mutual nearest neighbors
# and have the smallest ncc values
Nvis = 40

montage = np.concatenate((I1, I2), axis=1)

fig = plt.figure()
plt.imshow(montage, cmap='gray')
plt.axis('off')

for k in range(np.maximum(len(id_ncc), Nvis)):
    l = id_ncc[k]
    plt.plot(x1[int(pairs[l, 0])], y1[int(pairs[l, 0])], 'rx')
    plt.plot(x2[int(pairs[l, 1])] + I1.shape[1], y2[int(pairs[l, 1])], 'rx')
    plt.plot([x1[int(pairs[l, 0])], x2[int(pairs[l, 1])]+I1.shape[1]],
         [y1[int(pairs[l, 0])], y2[int(pairs[l, 1])]])
# Finally, since we have estimated the planar projective transformation
# we can check that how many of the nearest neighbor matches actually
# are correct correspondences
p1to2=np.dot(H1to2p, np.hstack((src, np.ones((src.shape[0],1)))).T)
p1to2 = p1to2[:2,:] / p1to2[2,:]
p1to2 = p1to2.T
pdiff=np.sqrt(np.sum((dst-p1to2)**2, axis=1))
# The criterion for the match being a correct is that its correspondence in
# the second image should be at most rthrs=2 pixels away from the transformed
n_correct = len(pdiff[pdiff<rthrs])

# Ouput the execution time
exe_time = str(time.time() - start)

return fig, n_correct, exe_time
```

Results:

```
81 Correct Matches.
```



```
Total time elapsed (s): 5.329760551452637
```

Computation time equals 5.329760551452637 in seconds.

Comments:

1) Using NCC, 81 correct correspondences were found compared to the 24 found with SSD.
2) SDD is very sensitive to pixel intensity differences in images, so in this case, NCC works better because it normalizes the pixel intensities to account for the intensity diffence between images. Also these two images only differs in translation and brightness, there is no significant rotation, scale or non-linear photometric differences, and NCC still works well under such circumstances.

Matching Points Using SIFT

Steps:

- Find the keypoints and descriptors with SIFT detector
- Initiate BruteForce matcher with default params
- Perform matching and save k=2 nearest neighbors for each descriptor
- Apply Lowe's ratio test
- Sort matches
- Collect feature points and scales from the match objects
- Estimate the geometric transformation between images

Algorithm:

```
def match_sift(img1,img2):

  start = time.time()
  #Get keypoints, descriptors
  kp1, des1= computeKeypointsAndDescriptors(img1)
  kp2, des2= computeKeypointsAndDescriptors(img2)
  # Initiate BruteForce matcher with default params
  bf = cv2.BFMatcher()
  # Perform matching and save k=2 nearest neighbors for each descriptor
  matches = bf.knnMatch(des1, des2, k=2)
  # Apply Lowe's ratio test
  good_matches = []
  for m,n in matches:
    if m.distance < 0.75*n.distance:
      good_matches.append(m)
  # Sort matches
  good_matches = sorted(good_matches, key = lambda x:x.distance)
  # Collect feature points and scales from the match objects
  source_pts = []
  target_pts = []
  source_radii = []
  target_radii = []

  for match in good_matches:
    # Collect feature point coords and scale query (img1)
    x, y = kp1[match.queryIdx].pt
    pt = np.array([np.round(x), np.round(y)]).astype(np.int)
    source_pts.append(pt)
    radius = kp1[match.queryIdx].size / 2.
    source_radii.append(radius)
```

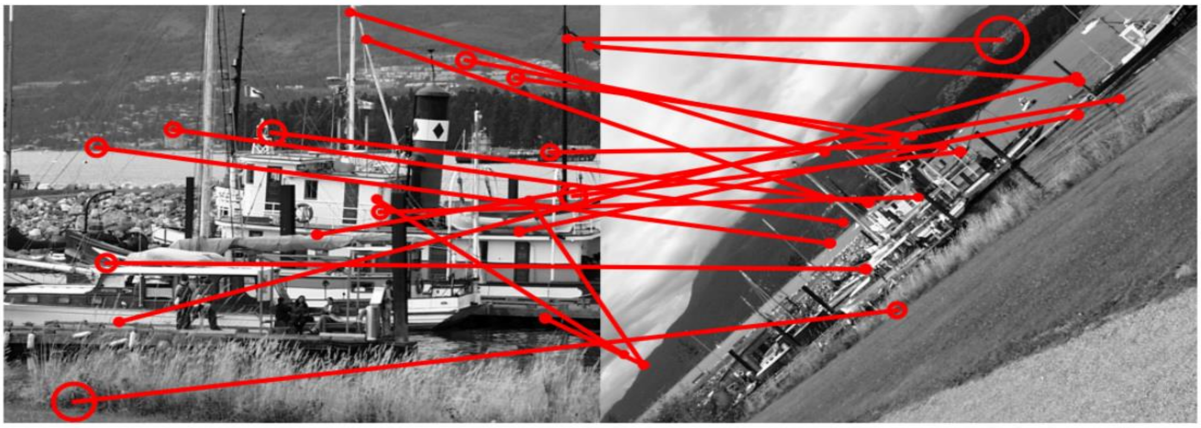
```
  # Collect feature point coords and scale query (img2)
  x, y = kp2[match.trainIdx].pt
  pt = np.array([np.round(x), np.round(y)]).astype(np.int)
  target_pts.append(pt)
  radius = kp2[match.trainIdx].size / 2.
  target_radii.append(radius)

source_pts = np.array(source_pts)
source_radii = np.array(source_radii)
target_pts = np.array(target_pts)
target_radii = np.array(target_radii)

## Estimate the geometric transformation between images
rthrs=10
tform,_ = ransac((source_pts, target_pts), SimilarityTransform, min_samples=2,
                              residual_threshold=rthrs, max_trials=1000)
H1to2p = tform.params
s = np.sqrt(np.linalg.det(H1to2p[0:2,0:2]))
R = H1to2p[0:2,0:2] / s
t = H1to2p[0:2,2]

montage = np.concatenate((img1, img2), axis=1)
Nvis = 20
fig=plt.figure()
plt.title("Matching points using SIFT")
plt.imshow(montage, cmap='gray')
plt.axis("off")


for k in range(0, Nvis):
    plt.plot([source_pts[k,0], target_pts[k,0]+img1.shape[1]],\
             [source_pts[k,1], target_pts[k,1]], 'r-')

    x,y=circle_points(source_pts[k,0], source_pts[k,1],\
                      3*np.sqrt(2)*source_radii[k])
    plt.plot(x, y, 'r', linewidth=1.5)

    x,y=circle_points(target_pts[k,0]+img1.shape[1], target_pts[k,1],\
                      3*np.sqrt(2)*target_radii[k])
    plt.plot(x, y, 'r', linewidth=1.5)


# Ouput the execution time
exe_time = str(time.time() - start)

return fig, exe_time
```

Results:

Select

SIFT

Matching points using SIFT



```
Total time elapsed (s): 911.9581384658813
```

Computation time equals 911.9581384658813 in seconds (15 min 11 sec) .