Shiraz University Of Technology

Javid Norouzi

SIFT (Scale Invariant Feature Transform)

Based on procedure in

Anatomy Of the SIFT Method

Impoting Required Libraries

In [1]:

```
import matplotlib.pyplot as plt
import cv2
import numpy as np
from math import *
import copy
```

Class to hold information about each interest point

x,y : Refined Location
m,n : Discrete Location
octave : Octave in which key point is detected in
scale : Scale in which key point is detected in
w : DoG responce in refined location
orientation : Key Point Orientation
descriptor : Key Point Descriptor
pt() : Returns refined location as a Tuple

In [2]:

```
class keyPoint:
    def __init__(self, octave=None, scale=None, m=None, n=None, x=None, y=None, sigma=None, w=None, ori=None, desc=None):
        self.x = x
        self.y = y
        self.n = n
        self.m = m
        self.octave = octave
        self.scale = scale
        self.sigma = sigma
        self.w = w
        self.orientation = ori
        self.descriptor = desc

    def pt(self):
        return int(self.x), int(self.y)
```

Some Utility Functions

2D Hessian in surface

In [3]:

```
def hessian2d(surface):
    h11 = surface[2, 1] + surface[0, 1] - 2 * surface[1, 1]
    h22 = surface[1, 2] + surface[1, 0] - 2 * surface[1, 1]
    h12 = (surface[2, 2] - surface[2, 0] - surface[0, 2] + surface[0, 0]) / 4
    return np.array([[h11, h12],
                     [h12, h22]])
```

3D Gradient

In [4]:

```
def gradientAtCubeCenter(cube):
    ds = 0.5 * (cube[2, 1, 1] - cube[0, 1, 1])
    dm = 0.5 * (cube[1, 2, 1] - cube[1, 0, 1])
    dn = 0.5 * (cube[1, 1, 2] - cube[1, 1, 0])
    return np.array([ds, dm, dn])
```

3D Hessian

In [5]:

```
def hessianAtCubeCenter(cube):
    center_point = cube[1, 1, 1]
    h11 = cube[2, 1, 1] - 2 * center_point + cube[0, 1, 1]
    h22 = cube[1, 2, 1] - 2 * center_point + cube[1, 0, 1]
    h33 = cube[1, 1, 2] - 2 * center_point + cube[1, 1, 0]
    h12 = 0.25 * (cube[2, 2, 1] - cube[2, 0, 1] - cube[0, 2, 1] + cube[0, 0, 1])
    h13 = 0.25 * (cube[2, 1, 2] - cube[2, 1, 0] - cube[0, 1, 2] + cube[0, 1, 0])
    h23 = 0.25 * (cube[1, 2, 2] - cube[1, 2, 0] - cube[1, 0, 2] + cube[1, 0, 0])
    return np.array([[h11, h12, h13],
                     [h12, h22, h23],
                     [h13, h23, h33]])
```

Displays a 4D Space

In [6]:

```
def showSpace(space, color_map='gray'):
    n_octave, n_image_per_octave = space.shape
    figure, ax = plt.subplots(n_octave, n_image_per_octave, figsize=(24, 20))

    for octave in range(n_octave):
        for image_index in range(n_image_per_octave):
            pos = ax[octave, image_index].imshow(space[octave, image_index], cmap=color_map)
            figure.colorbar(pos, ax=ax[octave, image_index])

    figure.canvas.draw()
#     plt.show()
    return  np.array(figure.canvas.buffer_rgba())
```

Smoothes a 1D vector

In [7]:

```
def smoothHist(histogram):
    h = np.zeros_like(histogram)
    h_size = len(histogram)
    for iteration in range(6):
        for i in range(len(histogram)):
            h_p = histogram[(i - 1) % h_size]
            h_c = histogram[i]
            h_n = histogram[(i + 1) % h_size]
            h[i] = (h_p + h_n + h_c) / 3
    return h
```

Draw Key Points on given image based on key points list

Each Circle's radius aquals to the keypoint's sigma

In [8]:

```
def drawKeyPoints(img, key_points, title=""):
    img_cp = img.copy() * 255
    for point in key_points:
        x, y = point.pt()
        pt1 = (y, x)
        r = int(point.sigma)
        ori = point.orientation
        cv2.circle(img_cp, pt1, r, (255, 255, 255), 1)
        if r > 2:
            a = cos(ori) * r
            b = sin(ori) * r
            pt2 = (pt1[0] + round(b),pt1[1] + round(a))
            cv2.line(img_cp, pt1, pt2, (255, 255, 255))

#     plt.figure(figsize=(24, 20)).add_subplot(title=title).imshow(img_cp)
#     plt.show()
    return img_cp
```

Exactly like the previous function but within a Rotated Rectangles

In [9]:

```
def drawRotRect(img, key_points, title=""):
    img_cp = img.copy() * 255
    img_height, img_width = img.shape
    for point in key_points:
        cy, cx = point.pt()
        ori = point.orientation
        r_rect = int(point.sigma * sqrt(2))

        pt1 = (cx + int(cos(ori + pi / 4) * r_rect), cy + int(sin(ori + pi / 4) * r_rect))
        pt2 = (cx + int(cos(ori + 3 * pi / 4) * r_rect), cy + int(sin(ori + 3 * pi / 4) * r_rect))
        pt3 = (cx + int(cos(ori + 5 * pi / 4) * r_rect), cy + int(sin(ori + 5 * pi / 4) * r_rect))
        pt4 = (cx + int(cos(ori + 7 * pi / 4) * r_rect), cy + int(sin(ori + 7 * pi / 4) * r_rect))
        cv2.line(img_cp, pt1, pt2, (255, 255, 255))
        cv2.line(img_cp, pt2, pt3, (255, 255, 255))
        cv2.line(img_cp, pt3, pt4, (255, 255, 255))
        cv2.line(img_cp, pt1, pt4, (255, 255, 255))
        cv2.line(img_cp, (cx, cy), (cx + int(cos(ori) * r_rect), cy + int(sin(ori) * r_rect)), (255, 255, 255))

    return img_cp
```

Constructs a 2D numpy array consisting total bluring for each image in gaussian Space

InterPixel Distance is assumed distance between image pixels in the input image unlike what is assumed in the original algorithm

In [10]:

```
def generateBlurringMatrix2(n_octave=5, n_scale_per_octave=3, sigma_min=.8, inter_pixel_distance=1):
    k = 2 ** (1 / n_scale_per_octave)
    iteration_per_octave = 3 + n_scale_per_octave
    blurring_matrix = np.zeros((n_octave, iteration_per_octave))
    for octave in range(n_octave):
        iteration_sigma_min = 2 ** octave * sigma_min / inter_pixel_distance
        blurring_matrix[octave, 0] = iteration_sigma_min
        for iteration in range(1, iteration_per_octave):
            total_blur = k ** iteration * iteration_sigma_min
            blurring_matrix[octave, iteration] = total_blur

    return blurring_matrix
```

In [11]:

```
# As an example we have the following
generateBlurringMatrix2(n_scale_per_octave=2)
```

Out[11]:

```
array([[ 0.8       ,  1.13137085,  1.6       ,  2.2627417 ,  3.2       ],
       [ 1.6       ,  2.2627417 ,  3.2       ,  4.5254834 ,  6.4       ],
       [ 3.2       ,  4.5254834 ,  6.4       ,  9.0509668 , 12.8       ],
       [ 6.4       ,  9.0509668 , 12.8       , 18.1019336 , 25.6       ],
       [12.8       , 18.1019336 , 25.6       , 36.2038672 , 51.2       ]])
```

SIFT Implementations

Generate Digital Guassian Space

In [12]:

```
def generateDigitalGaussianScaleSpace(input_image, n_octave, n_sample_per_octave, sigma_min, inter_pixel_distance, sigma_in):
    delta_min = inter_pixel_distance / 2
    scale_factor = 1 / delta_min
    interpolated_img = cv2.resize(input_image, (0, 0), fx=scale_factor, fy=scale_factor, interpolation=cv2.INTER_LINEAR)
    blur_to_apply =  sqrt(sigma_min ** 2 - sigma_in ** 2) / delta_min
    img_per_octave = n_sample_per_octave + 3
    gaussian_scale_space = np.zeros(shape=(n_octave, img_per_octave), dtype=np.ndarray)
    gaussian_scale_space[0, 0] = cv2.GaussianBlur(interpolated_img, (0, 0), blur_to_apply)

    for s in range(1, img_per_octave):
        blur_to_apply = sigma_min / delta_min * sqrt(2 ** (2 * s / n_sample_per_octave) - 2 ** (2 * (s - 1) / n_sample_per_octave))
        gaussian_scale_space[0, s] = cv2.GaussianBlur(gaussian_scale_space[0, s - 1], (0, 0), sigmaX=blur_to_apply)
    for o in range(1, n_octave):
        gaussian_scale_space[o, 0] = cv2.resize(gaussian_scale_space[o - 1, n_sample_per_octave], (0, 0), fx=.5, fy=.5, interpolation=cv2.INTER_LINEAR)
        for s in range(1, img_per_octave):
            blur_to_apply = sigma_min / delta_min * sqrt(2 ** (2 * s / n_sample_per_octave) - 2 ** (2 * (s - 1) / n_sample_per_octave))
            gaussian_scale_space[o, s] = cv2.GaussianBlur(gaussian_scale_space[o, s - 1], (0, 0), blur_to_apply)

    return gaussian_scale_space
```

Generates the DoG Space

In [13]:

```
def generateDoG(scale_space):
    octave_num, iteration_num = scale_space.shape
    DoG = np.zeros_like(scale_space)
    for i in range(1, iteration_num):
        DoG[:, i - 1] = scale_space[:, i] - scale_space[:, i - 1]
    return DoG[:, :-1]
```

Detects the extremums in the DoG Space

In [14]:

```
def detectExtremums(difference_of_gaussian, contrast_dog, sigma_min, inter_pixel_distance):
    positions = []
    candidate_key_points = []
    n_octave, n_img_per_octave = difference_of_gaussian.shape
    n_sample_per_octave = n_img_per_octave - 3
    total_blurring_map = generateBlurringMatrix2(n_octave=n_octave, n_scale_per_octave=n_sample_per_octave, sigma_min=sigma_min, inter_pixel_distance=inter_pixel_distance)
    total_blurring_map = np.array(total_blurring_map)
    for octave in range(n_octave):
        img_height, img_width = difference_of_gaussian[octave, 0].shape
        for img_index in range(1, n_img_per_octave - 1):
            # fix
            three_successive_img = np.stack(difference_of_gaussian[octave, img_index - 1:img_index + 2])
            for m in range(1, img_height - 1):
                for n in range(1, img_width - 1):
                    cube = three_successive_img[:, m - 1:m + 2, n - 1:n + 2]
                    # pt = cube[1, 1, 1]
                    # cube = np.delete(cube, 13)
                    # if pt >= cube.max() or pt <= cube.min():
                    # if np.argmax(np.abs(cube)) == 13:
                    if (np.argmax(cube) == 13 or np.argmin(cube) == 13) and abs(cube[1, 1, 1]) > .8 * contrast_dog:
                        x = inter_pixel_distance * 2 ** (octave - 1) * m
                        y = inter_pixel_distance * 2 ** (octave - 1) * n
                        candidate_key_points.append(keyPoint(m=m, n=n, x=x, y=y, octave=octave, scale=img_index, ori=0, sigma=total_blurring_map[octave, img_index]))
                        positions.append((m + 1, n + 1))
    return candidate_key_points
```

Calculates Gradient of Gaussian Scale Space

In [15]:

```
def space_gradient(space):
    n_octave, n_image_per_octave = space.shape
    gradient_m = np.zeros_like(space, dtype=np.ndarray)
    gradient_n = np.zeros_like(space, dtype=np.ndarray)

    for octave in range(n_octave):
        for img_index in range(n_image_per_octave):
            gradient_m[octave, img_index] = cv2.Sobel(space[octave, img_index], cv2.CV_64F, 0, 1, 3)
            gradient_n[octave, img_index] = cv2.Sobel(space[octave, img_index], cv2.CV_64F, 1, 0, 3)
    return gradient_m, gradient_n
```

Discarding low contrast points in DoG space (A conservative test to prevent unneccacery calculations)

In [16]:

```
def discardLowContrastPointsConservative(key_points, difference_of_gaussian, contrast_dog):
    candidate_key_points = []
    for point in key_points:
        o = point.octave
        s = point.scale
        m = point.m
        n = point.n
        if abs(difference_of_gaussian[o, s][m, n]) > .8 * contrast_dog:
            candidate_key_points.append(point)

    return candidate_key_points
```

Discarding low contrast points after refining position and response in DoG

In [17]:

```
def discardLowContrastPoints(keyPoints, contrast):
    final_list = []
    for point in keyPoints:
        if abs(point.w) > contrast:
            final_list.append(point)
    return final_list
```

Refining KeyPoints positions by quadratic interpolation

In [18]:

```
def refinePositions(candidate_key_points, difference_of_gaussian_space, n_attempts, inter_pixel_distance, sigma_min):
    n_octave, n_img_per_octave = difference_of_gaussian_space.shape
    n_sample_per_octave = n_img_per_octave - 2
    final_key_point_list = []
    for point in candidate_key_points:
        o = point.octave
        s = point.scale
        m = point.m
        n = point.n
        outside_bound = False
        img_height, img_width = difference_of_gaussian_space[o, 0].shape
        for attempt in range(n_attempts):
            if not 0 < m < img_height - 1 or not 0 < n < img_width - 1 or not 0 < s < n_img_per_octave - 1:
                outside_bound = True
                break
            offset, w = quadraticInterPolate(difference_of_gaussian_space=difference_of_gaussian_space, point=(o, s, m, n))
            sigma = 2 ** o * sigma_min * 2 ** ((offset[0] + s) / n_sample_per_octave)
            x = inter_pixel_distance * 2 ** (o - 1) * (offset[1] + m)
            y = inter_pixel_distance * 2 ** (o - 1) * (offset[2] + n)
            s += np.round(offset).astype(np.int)[0]
            m += np.round(offset).astype(np.int)[1]
            n += np.round(offset).astype(np.int)[2]
            if np.all(offset < .5):
                break

        if outside_bound:
            continue

        if np.any(np.abs(offset) > 1):
            continue

        point.x = x
        point.y = y
        point.w = w
        point.sigma = sigma

        final_key_point_list.append(point)

    return final_key_point_list
```

Quadratic InterPolation at a single point in DoG

In [19]:

```
def quadraticInterPolate(difference_of_gaussian_space, point):
    o, s, m, n = point
    three_successive_img = np.stack(difference_of_gaussian_space[o, s - 1:s + 2])
    cube = three_successive_img[:, m - 1:m + 2, n - 1:n + 2]
    hessian = hessianAtCubeCenter(cube)
    gradient = gradientAtCubeCenter(cube)
    offset = -np.linalg.lstsq(hessian, gradient, rcond=None)[0]
    # offset = -np.linalg.inv(hessian) @ gradient
    w = difference_of_gaussian_space[o, s][m, n] + .5 * gradient.T @ offset
    return offset, w
```

Discarding edge response

In [20]:

```
def discardEdgeResponse(key_point_list, difference_of_gaussian_space, r_edge, n_octave, n_sample_per_octave, sigma_min, inter_pixel_distance):
    total_blurring_map = generateBlurringMatrix2(n_octave=n_octave, n_scale_per_octave=n_sample_per_octave, sigma_min=sigma_min, inter_pixel_distance=inter_pixel_distance)
    total_blurring_map = np.array(total_blurring_map)
    final_list = []
    for point in key_point_list:
        o = point.octave
        m = point.m
        n = point.n
        sigma_dist = np.abs(total_blurring_map[point.octave] - point.sigma)
        closest_scale = np.argmin(sigma_dist)
        current_plane = np.stack(difference_of_gaussian_space[o, closest_scale])
        surface = current_plane[m - 1:m + 2, n - 1:n + 2]
        hessian = hessian2d(surface)
        hessian_trace = np.trace(hessian)
        hessian_det = np.linalg.det(hessian)
        edgeness = hessian_trace ** 2 / hessian_det
        if edgeness < (r_edge + 1) ** 2 / r_edge:
            final_list.append(point)
    return final_list
```

Detecting each keypoint's orientations

In [21]:

```
def computeKeyPointOrientation(key_point_list, gradient_of_scale_space, l_ori, n_bins, t_secondary, sigma_min, n_octave, n_sample_per_octave, inter_pixel_distance):
    total_blurring_map = generateBlurringMatrix2(n_octave=n_octave, n_scale_per_octave=n_sample_per_octave, sigma_min=sigma_min, inter_pixel_distance=inter_pixel_distance)
    total_blurring_map = total_blurring_map[:, 1:-2]
    gradient_m = gradient_of_scale_space[0]
    gradient_n = gradient_of_scale_space[1]
    gradient_m = gradient_m[:, 1:-2]
    gradient_n = gradient_n[:, 1:-2]
    plane_height, plane_width = gradient_m[1, 0].shape
    result_key_points_list = []
    for point in key_point_list:
        histogram = np.zeros(shape=n_bins)
        sigma_dist = np.abs(total_blurring_map - point.sigma)
        closest_scale = np.argmin(sigma_dist) % (n_sample_per_octave)
        current_octave = int(np.argmin(sigma_dist) // n_sample_per_octave)

        safety_dist = 3 * l_ori * point.sigma
        inter_pixel_distance_current_octave = inter_pixel_distance * 2 ** (current_octave - 1)
        if not safety_dist < point.x < (plane_height - safety_dist) or not safety_dist < point.y < (plane_width - safety_dist):
            continue
        patch_corner_m_0 = int((point.x - safety_dist) / inter_pixel_distance_current_octave)
        patch_corner_m_1 = int((point.x + safety_dist) / inter_pixel_distance_current_octave)
        patch_corner_n_0 = int((point.y - safety_dist) / inter_pixel_distance_current_octave)
        patch_corner_n_1 = int((point.y + safety_dist) / inter_pixel_distance_current_octave)

        for m in range(patch_corner_m_0, patch_corner_m_1):
            for n in range(patch_corner_n_0, patch_corner_n_1):
                grad_m = gradient_m[current_octave, closest_scale][m, n]
                grad_n = gradient_n[current_octave, closest_scale][m, n]

                gradient_magnitude = sqrt(grad_m ** 2 + grad_n ** 2)
                relative_m = m * inter_pixel_distance_current_octave - point.x
                relative_n = n * inter_pixel_distance_current_octave - point.y
                pt_orientation_share = exp(-(relative_m ** 2 + relative_n ** 2) / (2 * (l_ori * point.sigma) ** 2)) * gradient_magnitude
                ori = atan2(grad_n, grad_m) % (2 * pi)
                pt_orientation_bin = floor(n_bins / (2 * pi) * ori) % (n_bins)
                histogram[pt_orientation_bin] += pt_orientation_share

        smoothed_histogram = smoothHist(histogram)
        histogram_max_value = smoothed_histogram.max()

        refrence_angles = []
        for k in range(n_bins):
            h_p = smoothed_histogram[k - 1]
            h_n = smoothed_histogram[(k + 1) % n_bins]
            h_c = smoothed_histogram[k]
            if h_p < h_c and h_c > h_n and h_c > t_secondary * histogram_max_value:
                # fix
                angle_at_bin_k = 2 * pi * (k + 1) / n_bins
                key_ori = angle_at_bin_k + pi / n_bins * (h_p - h_n) / (h_p - 2 * h_c + h_n)
                refrence_angles.append(key_ori)

        for angle in refrence_angles:
            finalized_key_point = copy.deepcopy(point)
            finalized_key_point.orientation = angle
            result_key_points_list.append(finalized_key_point)

    return result_key_points_list
```

Computing descriptor for each key point

In [22]:

```
def computeDescriptor(key_point_list, gradient_of_scale_space, inter_pixel_distance, l_desc, n_ori, n_hist, n_octave, n_sample_per_octave, sigma_min):
    total_blurring_map = generateBlurringMatrix2(n_octave=n_octave, n_scale_per_octave=n_sample_per_octave, sigma_min=sigma_min, inter_pixel_distance=inter_pixel_distance)
    total_blurring_map = total_blurring_map[:, 1:-2]
    gradient_m = gradient_of_scale_space[0]
    gradient_n = gradient_of_scale_space[1]
    gradient_m = gradient_m[:, 1:-2]
    gradient_n = gradient_n[:, 1:-2]
#     counter = 0
    # img_to_draw = gradient_of_scale_space[0][1, 0]
    result_key_points_list = []
    h, w = gradient_m[1, 0].shape
    for point in key_point_list:
        sigma_dist = np.abs(total_blurring_map - point.sigma)
        closest_scale = np.argmin(sigma_dist) % n_sample_per_octave
        current_octave = int(np.argmin(sigma_dist) // n_sample_per_octave)
        inter_pixel_distance_in_current_octave = inter_pixel_distance * 2 ** (current_octave - 1)

        safety_dis = ceil(sqrt(2) * l_desc * point.sigma * (n_hist + 1) / n_hist)
        if not safety_dis < point.x < h - safety_dis or not safety_dis < point.y < w - safety_dis:
            continue

        feature_desc = np.zeros(shape=n_hist ** 2 * n_ori, dtype=np.float32)
        patch_hist = np.zeros(shape=(n_hist, n_hist, n_ori))
        r_outer_patch = sqrt(2) * l_desc * point.sigma * (n_hist + 1) / n_hist

        outer_patch_corner_m0 = int((point.x - r_outer_patch) / inter_pixel_distance_in_current_octave)
        outer_patch_corner_m1 = int((point.x + r_outer_patch) / inter_pixel_distance_in_current_octave)
        outer_patch_corner_n0 = int((point.y - r_outer_patch) / inter_pixel_distance_in_current_octave)
        outer_patch_corner_n1 = int((point.y + r_outer_patch) / inter_pixel_distance_in_current_octave)

        inner_patch_width = 2 * l_desc * point.sigma * (n_hist + 1) / n_hist
        for m in range(outer_patch_corner_m0, outer_patch_corner_m1):
            for n in range(outer_patch_corner_n0, outer_patch_corner_n1):
                x_dist = m * inter_pixel_distance_in_current_octave - point.x
                y_dist = n * inter_pixel_distance_in_current_octave - point.y
                x_patch = (x_dist * cos(point.orientation) + y_dist * sin(point.orientation)) / point.sigma
                y_patch = (-x_dist * sin(point.orientation) + y_dist * cos(point.orientation)) / point.sigma
                if max(abs(x_patch), abs(y_patch)) < l_desc * (n_hist + 1) / n_hist:

                    grad_m = gradient_m[current_octave, closest_scale][m, n]
                    grad_n = gradient_n[current_octave, closest_scale][m, n]
                    gradient_magnitude = sqrt(grad_m ** 2 + grad_n ** 2)
                    ori = (atan2(grad_n, grad_m) % (2 * pi) - point.orientation) % (2 * pi)
                    # ori_contrib = exp(-(x_patch ** 2 + y_patch ** 2) / (2 * (l_desc * point.sigma) ** 2)) * gradient_magnitude
                    ori_contrib = exp(-(x_patch ** 2 + y_patch ** 2) / (2 * l_desc ** 2)) * gradient_magnitude

                    failed = True

                    for i in range(n_hist):
                        for j in range(n_hist):
                            x_i = (i + 1 - (1 + n_hist) / 2) * 2 * l_desc / n_hist
                            y_j = (j + 1 - (1 + n_hist) / 2) * 2 * l_desc / n_hist
                            if max(abs(x_i - x_patch), abs(y_j - y_patch)) <= 2 * l_desc / n_hist:
                                for k in range(n_ori):
                                    ori_k = 2 * pi / n_ori * (k + 1)
                                    angle_dis = abs(ori_k - ori)
                                    if abs(ori_k - ori) < 2 * pi / n_ori:
                                        patch_hist[i, j, k] += (1 - n_hist / (2 * l_desc) * abs(x_patch - x_i)) * (1 - n_hist / (2 * l_desc) * abs(y_patch - y_j)) * (1 - n_ori / (2 * pi) * abs(ori - ori_k)) * ori_contrib
#                                         failed = False
#                     if failed:
#                         counter += 1

        feature_desc = patch_hist.flatten()
        l2_feature_desc = np.linalg.norm(feature_desc)

        feature_desc = np.minimum(feature_desc, .2 * l2_feature_desc)
        feature_desc = np.minimum(feature_desc * 512 / l2_feature_desc, 255)

        point.descriptor = feature_desc
        result_key_points_list.append(point)
#         print(len(result_key_points_list))
#     print("failed:", counter)

    return result_key_points_list
```

Detecting Key point default configurations is applied as default arguments

In [23]:

```
def sift(img_in, n_octave=5, n_sample_per_octave=3, sigma_min=.8, sigma_in=.5, c_dog=.015, r_edge=10, l_ori=1.5, l_desc=6, inter_pixel_dist=1):
    figures = []
    print("input image size:")
    print(img_in.shape)

    print("Generating Scale Space ... ")
    s_space = generateDigitalGaussianScaleSpace(img_in, n_octave=n_octave, n_sample_per_octave=n_sample_per_octave, sigma_min=sigma_min, inter_pixel_distance=inter_pixel_dist, sigma_in=sigma_in)
    figures.append(showSpace(s_space))

    print("Calculating DoG space ...")
    dog = generateDoG(s_space)
    figures.append(showSpace(dog, color_map='gist_heat'))

    print("Detecting Extremums ...")
    candidate_points = detectExtremums(dog, c_dog, sigma_min=sigma_min, inter_pixel_distance=inter_pixel_dist)
    print(len(candidate_points))
    figures.append(drawKeyPoints(img_in, candidate_points, "Extremums"))

    print("Discarding low contrast DOG points(Conservative Test) .... ")
    candidate_points = discardLowContrastPointsConservative(candidate_points, dog, c_dog)
    drawKeyPoints(img_in, candidate_points, "low contrase extremums discarded (conservative)")
    print(len(candidate_points))
    figures.append(drawKeyPoints(img_in, candidate_points, "low contrase extremums discarded (conservative)"))

    print("Refining key points positions ... ")
    candidate_points = refinePositions(candidate_points, dog, n_attempts=5, inter_pixel_distance=inter_pixel_dist, sigma_min=sigma_min)
    figures.append(drawKeyPoints(img_in, candidate_points, title="refined key points"))
    print(len(candidate_points))

    print("Discarding low contrast points after position refinement")
    candidate_points = discardLowContrastPoints(candidate_points, contrast=c_dog)
    figures.append(drawKeyPoints(img_in, candidate_points, title="final thresholding dog discarding low contrast points"))
    print(len(candidate_points))

    print("Discarding edge points ...")
    candidate_points = discardEdgeResponse(candidate_points, dog, r_edge, n_octave, n_sample_per_octave, sigma_min, inter_pixel_dist)
    figures.append(drawKeyPoints(img_in, candidate_points, title="edge response discarded"))
    print(len(candidate_points))

    print("Computing Scale space gradient ... ")
    gradient_of_scale_space = space_gradient(s_space)

    print("Detecting Keypoints Orientation ... ")
    candidate_points = computeKeyPointOrientation(candidate_points, gradient_of_scale_space, l_ori, n_bins=36, t_secondary=.8, sigma_min=sigma_min, n_octave=n_octave, n_sample_per_octave=n_sample_per_octave, inter_pixel_distance=inter_pixel_dist)
    figures.append(drawKeyPoints(img_in, candidate_points, title="orientation computed"))
    print(len(candidate_points))

    print("Computing keypoints descriptor ...")
    candidate_points = computeDescriptor(candidate_points, gradient_of_scale_space, inter_pixel_distance=inter_pixel_dist, l_desc=l_desc, n_ori=8, n_hist=4, n_octave=n_octave, n_sample_per_octave=n_sample_per_octave, sigma_min=sigma_min)
    figures.append(drawKeyPoints(img_in, candidate_points, title="descriptors computed"))
    print(len(candidate_points))

    print("The End ...")
    return candidate_points, figures
```

In [24]:

```
img1=cv2.imread("reeses_puffs.png",0).astype(np.float64) / 255
img2=cv2.imread("many_cereals.jpg",0).astype(np.float64) / 255
fig,axis=plt.subplots(1,2,figsize=(24,20))
axis[0].imshow(img1)
axis[1].imshow(img2)
```

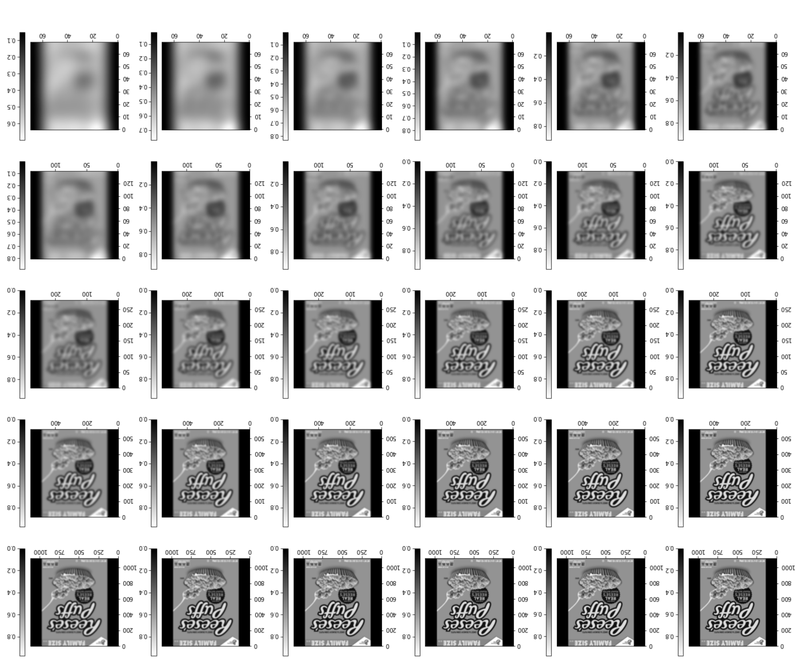
Out[24]:

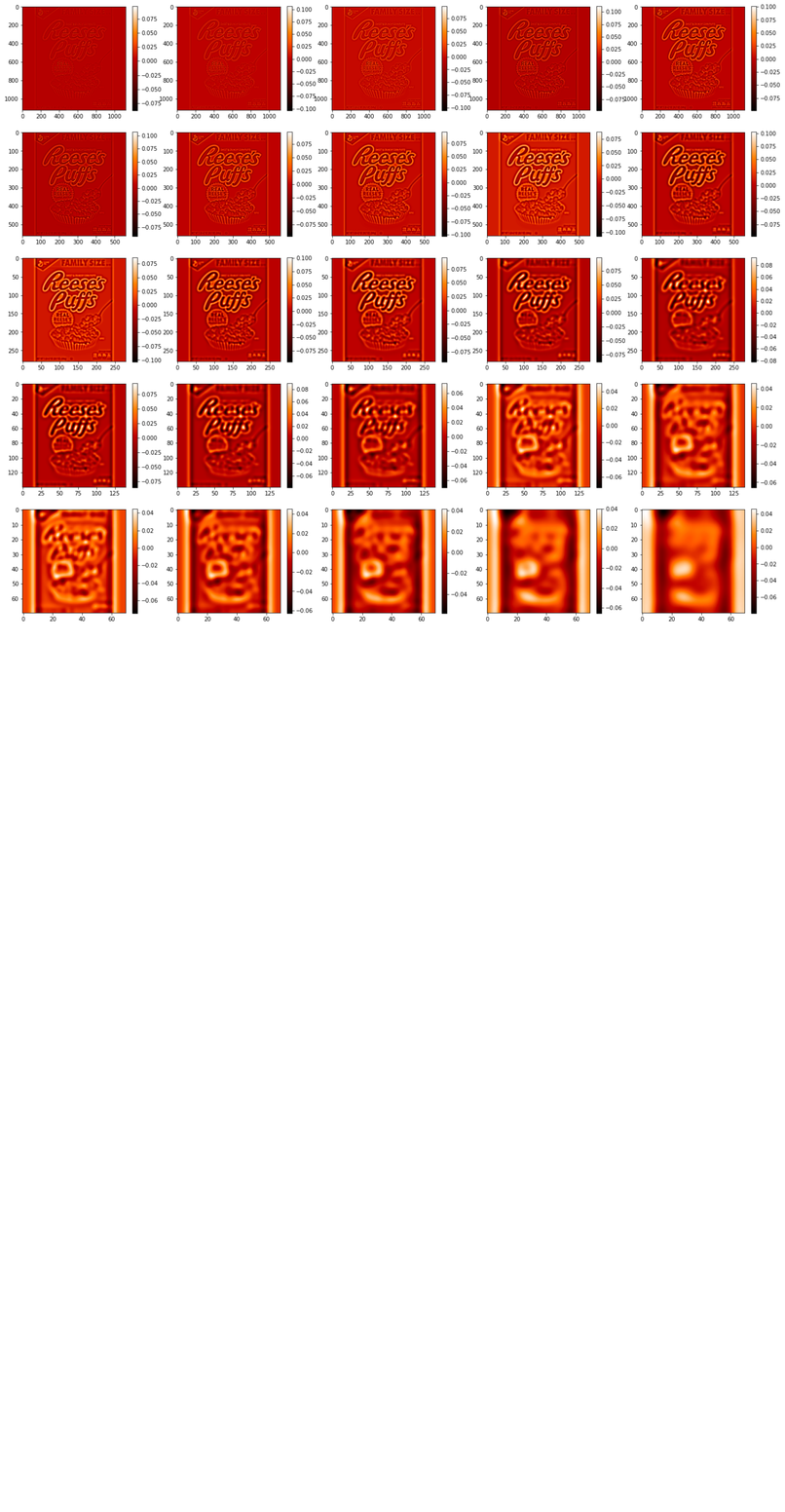
```
<matplotlib.image.AxesImage at 0x7f331bcab210>
```

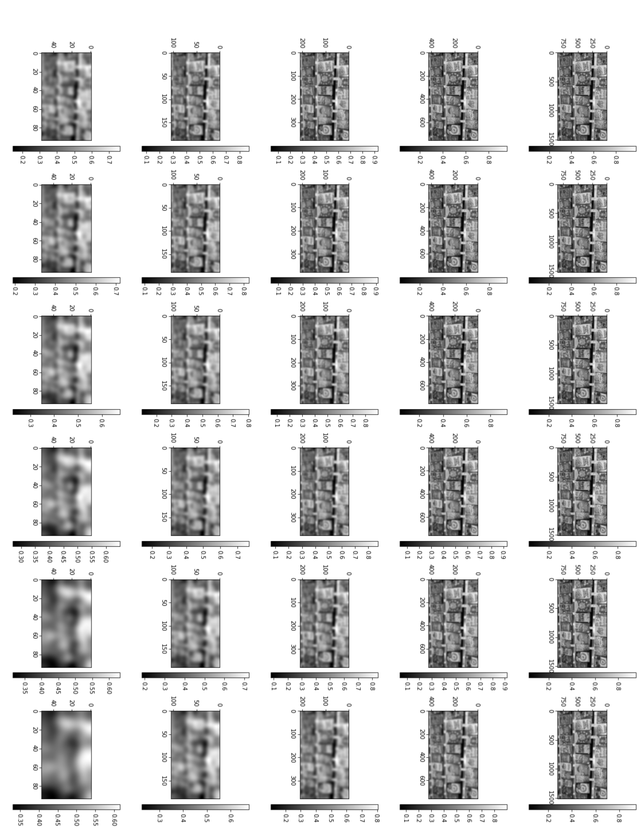
In [25]:

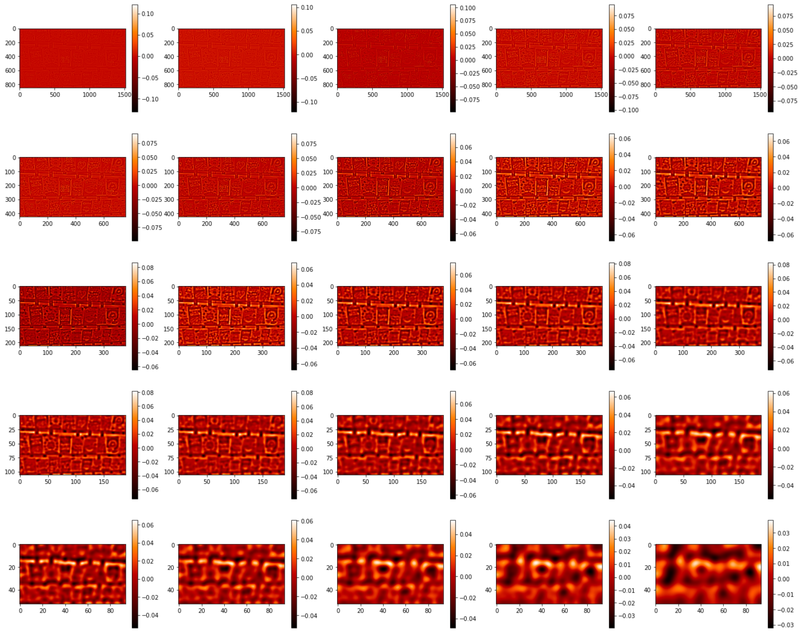
```
kp1,fig1=sift(img1)
kp2,fig2=sift(img2)
```

```
Input image size:
(560, 560)
Generating Scale Space ...
Calculating DoG space ...
3488
Detecting Extremums ...
3488
Discarding low contrast DoG points(Conservative Test) ....
3486
Refining key points positions ...
2615
Discarding low contrast points after position refinement
2556
Discarding edge points ...
1287
Computing Scale space gradient ...
1748
Detecting Keypoints Orientation ...
1656
Computing keypoints descriptor ...
The End ...
Input image size:
(425, 757)
Generating Scale Space ...
Calculating DoG space ...
Detecting Extremums ...
5252
Discarding low contrast DoG points(Conservative Test) ....
5252
Refining key points positions ...
4679
Discarding low contrast points after position refinement
4526
Discarding edge points ...
3393
Computing Scale space gradient ...
Detecting Keypoints Orientation ...
4116
Computing keypoints descriptor ...
3711
The End ...
```




Reese's Puffs
FAMILY SIZE



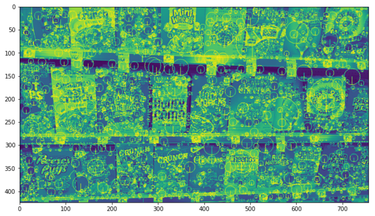


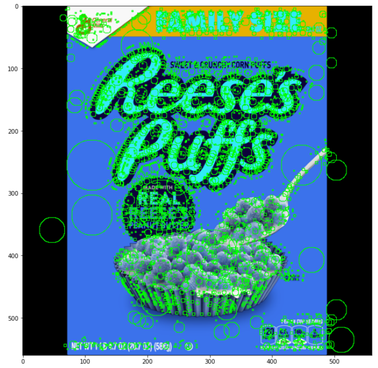
In [26]:

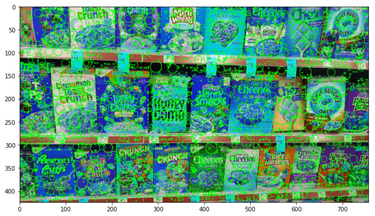
```
fig,ax=plt.subplots(1,2,figsize=(24,20))
ax[0].imshow(fig1[2])
ax[1].imshow(fig2[2])
ax[0].set(title="Detecting Extremums")
fig,ax=plt.subplots(1,2,figsize=(24,20))
ax[0].imshow(cv2.imread("extra_NES_im0.png"))
ax[1].imshow(cv2.imread("extra_NES_im1.png"))


fig,ax=plt.subplots(1,2,figsize=(24,20))
ax[0].imshow(fig1[3])
ax[1].imshow(fig2[3])
ax[0].set(title="Conservative Thresholding")
fig,ax=plt.subplots(1,2,figsize=(24,20))
ax[0].imshow(cv2.imread("extra_DoGSoftThresh_im0.png"))
ax[1].imshow(cv2.imread("extra_DoGSoftThresh_im1.png"))


fig,ax=plt.subplots(1,2,figsize=(24,20))
ax[0].imshow(fig1[4])
ax[1].imshow(fig2[4])
ax[0].set(title="Refining Positions")
fig,ax=plt.subplots(1,2,figsize=(24,20))
ax[0].imshow(cv2.imread("extra_ExtrInterp_im0.png"))
ax[1].imshow(cv2.imread("extra_ExtrInterp_im1.png"))



fig,ax=plt.subplots(1,2,figsize=(24,20))
ax[0].imshow(fig1[5])
ax[1].imshow(fig2[5])
ax[0].set(title="Final Thresholding DoG")
fig,ax=plt.subplots(1,2,figsize=(24,20))
ax[0].imshow(cv2.imread("extra_DoGThresh_im0.png"))
ax[1].imshow(cv2.imread("extra_DoGThresh_im1.png"))

fig,ax=plt.subplots(1,2,figsize=(24,20))
ax[0].imshow(fig1[6])
ax[1].imshow(fig2[6])
ax[0].set(title="Discarding Edge Responses")
fig,ax=plt.subplots(1,2,figsize=(24,20))
ax[0].imshow(cv2.imread("extra_OnEdgeResp_im0.png"))
ax[1].imshow(cv2.imread("extra_OnEdgeResp_im1.png"))


fig,ax=plt.subplots(1,2,figsize=(24,20))
ax[0].imshow(fig1[7])
ax[1].imshow(fig2[7])
ax[0].set(title="Detecting Orientation")
fig,ax=plt.subplots(1,2,figsize=(24,20))
ax[0].imshow(cv2.imread("keys_im0.png"))
ax[1].imshow(cv2.imread("keys_im1.png"))

fig,ax=plt.subplots(1,2,figsize=(24,20))
ax[0].imshow(fig1[8])
ax[1].imshow(fig2[8])
ax[0].set(title="Computing Descriptors")
```

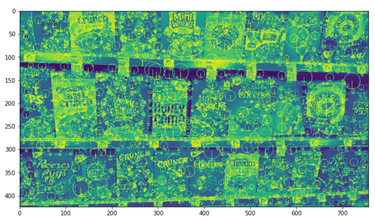
Out[26]:

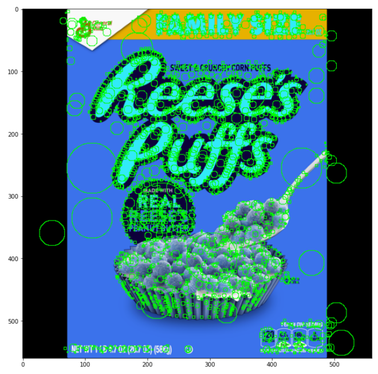
[Text(0.5, 1.0, 'Computing Descriptors')]

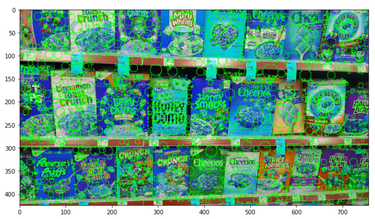




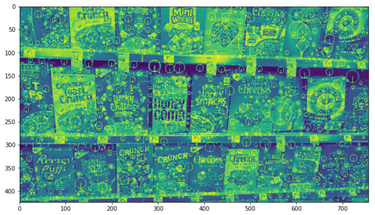


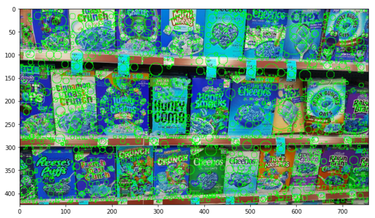




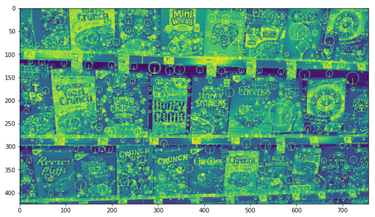


Refining Positions

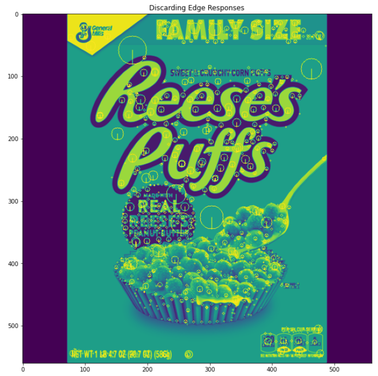


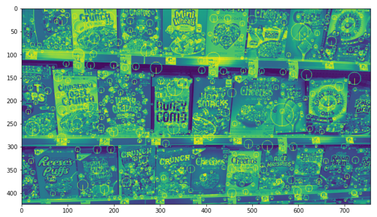


Final Thresholding DoG

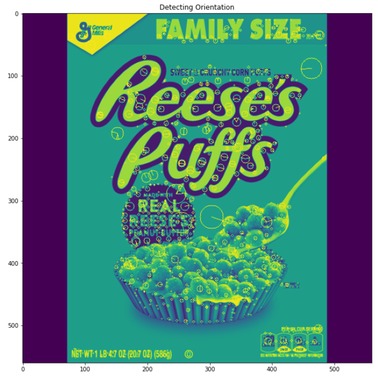


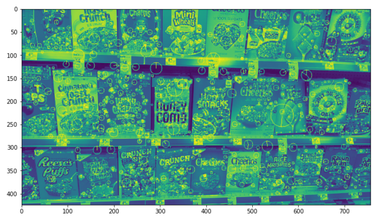
Discarding Edge Responses

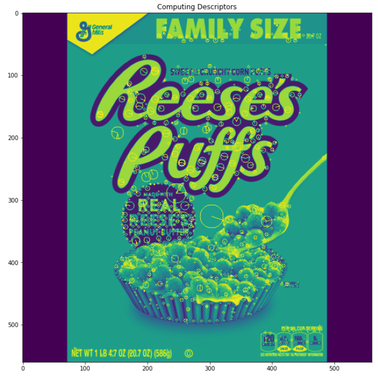


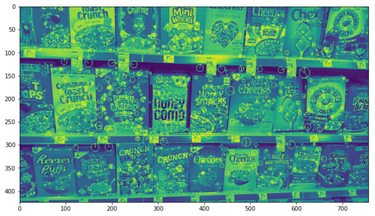


Detecting Orientation









In [27]:

```
des1 = np.array([point.descriptor for point in kp1]).astype(np.float32)
des2 = np.array([point.descriptor for point in kp2]).astype(np.float32)
```

In [28]:

```
MIN_MATCH_COUNT = 10
FLANN_INDEX_KDTREE = 0
index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
search_params = dict(checks=50)
flann = cv2.FlannBasedMatcher(index_params, search_params)
matches = flann.knnMatch(des1, des2, k=2)

# Lowe's ratio test
good = []
for m, n in matches:
    if m.distance < .7 * n.distance:
        good.append(m)


if len(good) > 0:
    # Estimate homography between template and scene
    src_pts = [kp1[m.queryIdx] for m in good]
    dst_pts = [kp2[m.trainIdx] for m in good]
    img1_height, img1_width = img1.shape
    img2_height, img2_width = img2.shape

    newimg = np.zeros(shape=(max(img1_height, img2_height), img1_width + img2_width))
    newimg[:img1_height, :img1_width] = drawKeyPoints(img1,src_pts)* 255
    newimg[:img2_height, img1_width:] = drawKeyPoints(img2,dst_pts) * 255
    matches=newimg.copy()

    for i in range(len(good)):
        pt1 = (int(src_pts[i].pt()[1]), int(src_pts[i].pt()[0]))
        pt2 = (int(dst_pts[i].pt()[1] + img1_width), int(dst_pts[i].pt()[0]))
        print(pt1, pt2)
        newimg = cv2.line(newimg, pt1, pt2, (0, 0, 0), 1)

else:
    print("Not enough matches are found - %d/%d" % (len(good), MIN_MATCH_COUNT))


print ("good Matches found :")
print (len(good))
```

```
(239, 39) (1147, 387)
(341, 14) (745, 273)
(486, 51) (723, 183)
(193, 304) (1110, 324)
(178, 328) (1110, 324)
(223, 136) (614, 352)
(372, 136) (614, 352)
(231, 131) (621, 333)
(231, 131) (621, 333)
(231, 131) (621, 333)
(231, 131) (621, 333)
(287, 131) (621, 333)
(380, 131) (621, 333)
(244, 202) (626, 347)
(249, 214) (627, 350)
(194, 223) (614, 352)
(421, 333) (669, 372)
(165, 343) (611, 376)
(194, 94) (612, 325)
(194, 94) (612, 325)
(172, 126) (607, 332)
(312, 126) (640, 331)
(312, 126) (640, 331)
(257, 130) (627, 332)
(299, 136) (659, 333)
(299, 136) (659, 333)
(393, 136) (659, 333)
(393, 136) (637, 334)
(393, 136) (659, 333)
(159, 138) (605, 334)
(159, 138) (605, 334)
(268, 137) (630, 334)
(173, 140) (608, 335)
(222, 153) (619, 337)
(222, 153) (619, 337)
(277, 153) (632, 337)
(277, 153) (632, 337)
(370, 153) (654, 337)
(425, 155) (658, 355)
(425, 155) (658, 355)
(331, 159) (644, 338)
(230, 176) (622, 342)
(318, 178) (642, 342)
(318, 178) (642, 342)
(378, 199) (656, 346)
(198, 219) (608, 328)
(244, 226) (626, 352)
(255, 240) (629, 355)
(255, 240) (629, 355)
(365, 244) (654, 355)
(262, 266) (631, 360)
(308, 264) (642, 359)
(185, 290) (614, 365)
(190, 306) (616, 368)
(199, 364) (619, 380)
(220, 434) (625, 393)
(284, 441) (640, 394)
(284, 441) (640, 394)
(299, 445) (643, 394)
(341, 458) (653, 397)
(194, 94) (612, 325)
(194, 94) (612, 325)
(343, 136) (647, 333)
(343, 136) (647, 333)
(361, 138) (651, 333)
(212, 138) (617, 334)
(205, 157) (616, 338)
(170, 161) (608, 339)
(258, 189) (628, 344)
(323, 230) (644, 353)
(323, 230) (644, 353)
(288, 237) (636, 354)
(288, 237) (636, 354)
(255, 240) (629, 355)
(255, 240) (629, 355)
(286, 240) (636, 354)
(269, 251) (632, 357)
(309, 263) (642, 359)
(309, 263) (642, 359)
(416, 308) (667, 367)
(416, 308) (667, 367)
(331, 331) (648, 372)
(182, 366) (615, 380)
(335, 380) (650, 382)
(244, 394) (630, 385)
(189, 137) (612, 334)
(343, 174) (648, 341)
(176, 231) (611, 353)
(357, 233) (652, 353)
(357, 233) (652, 353)
(198, 263) (615, 361)
(172, 309) (612, 369)
(181, 364) (615, 380)
(391, 334) (662, 372)
(391, 334) (662, 372)
(218, 380) (624, 383)
(174, 400) (614, 387)
(246, 289) (629, 365)
good Matches found :
98
```

In [29]:

```
plt.figure(figsize=(24,20)).add_subplot().imshow(matches)
fig,ax=plt.subplots(1,2,figsize=(24,20))
ax[0].imshow(cv2.imread("matching_keys_im0.png"))
ax[1].imshow(cv2.imread("matching_keys_im1.png"))

plt.figure(figsize=(24, 20)).add_subplot().imshow(newimg)
plt.show()

plt.figure(figsize=(24, 20)).add_subplot().imshow(cv2.imread("OUTmatches.png"))
plt.show()
```


FAMILY SIZE
SWEET & CRUNCHY CORN PUFFS
Reese's
Puffs
NET WT 1 LB 4.7 OZ (20.7 OZ) (586g)


FAMILY SIZE
SWEET & CRUNCHY CORN PUFFS
Reese's
Puffs
MADE WITH
REAL
REESE'S
PEANUT BUTTER
NET WT 1 LB 4.7 OZ (20.7 OZ) (586g)


FAMILY SIZE
Reese's
Puffs


FAMILY SIZE
Reese's
Puffs