Contents

[Usage 1](#_Toc67005605)

[General Principles 1](#_Toc67005606)

[Task 1 : Scale Space 1](#_Toc67005607)

[Loading the image 1](#_Toc67005608)

[Creating 12 Gaussian kernels 2](#_Toc67005609)

[Task 2: Feature Point Locations 4](#_Toc67005610)

[Difference of Gaussians 4](#_Toc67005611)

[Finding Key Points 5](#_Toc67005612)

[Task 3: Feature Point Orientation 8](#_Toc67005613)

[Calculating the Derivatives of Gaussians in scale space 8](#_Toc67005614)

[Task 4: Feature Descriptors 14](#_Toc67005615)

# Usage

The usage of this python file is as follows

./keypoints.py -f <filename>

Or

./keypoints.py –file <filename>

# General Principles

1. To keep the code clean, instead of a lot of bounds checks, exception handling has been used. Instead of checking for going out of the bounds of the image, we rely on out-of-bounds exception.
2. Wherever possible, results once calculated have been cached. This has been done in two ways:
   1. Decorating some functions with @lru\_cache
   2. Creating a cross reference from sigma to different values that we’ll need to use repeatedly

# Task 1 : Scale Space

## Loading the image

"""

This function just loads teh image

"""

def load\_original\_image()->np.ndarray:

    global g\_filename

    return cv2.imread(g\_filename)

"""

This function loads the image and converts it to grayscale

"""

def load\_image\_and\_convert\_to\_grayscale()->np.ndarray:

    """

    Open the image and convert to grayscale

    """

    image = load\_original\_image()

    image\_gray = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)

    image\_gray = np.float32(image\_gray)

    return image\_gra

The above lines load the image and convert it to grayscale. The corresponding lines in the code are from 23 to 40.

## Creating 12 Gaussian kernels

The following code creates 12 Gaussian kernels:

"""This controls the size of the Gaussian kernel

"""

GAUSSIAN\_KERNEL\_SIZE\_MULTIPLIER = 3

"""

Given a sigma, return a gaussian smoothing kernel of the appropriate size

"""

def get\_gaussian\_smoothing\_kernel(sigma:float)->np.ndarray:

    p\_x, p\_y = np.meshgrid(\

           np.arange(-GAUSSIAN\_KERNEL\_SIZE\_MULTIPLIER \* sigma, \

                GAUSSIAN\_KERNEL\_SIZE\_MULTIPLIER \* sigma),\

           np.arange(-GAUSSIAN\_KERNEL\_SIZE\_MULTIPLIER \* sigma,\

                GAUSSIAN\_KERNEL\_SIZE\_MULTIPLIER \* sigma))

    return np.exp(-(p\_x\*\*2 + p\_y\*\*2) / (2 \* sigma\*\*2)) / (2 \* np.pi \* sigma\*\*2)

"""

Get 12 gaussian smoothing kernels with different sigmas

where sigma = (2 \*\* (k/2)), k = 0,1,...,11

"""

def get\_twelve\_smoothed\_images(image:np.ndarray)->tuple:

    sigma\_arr = []

    images = []

    for k in range(12):

        sigma = (2 \*\* (k/2))

        sigma\_arr.append(sigma)

        g = get\_gaussian\_smoothing\_kernel(sigma)

        im = cv2.filter2D(image.copy(), -1, g)

        images.append(im)

    return images, sigma\_arr

The corresponding line numbers are 94 – 122. This is called from the following location:

"""

Gets twelve gaussian kernels in the sigma range

For each gaussian kernel, it applies it to the image

It then displays two things:

    1. The Gaussian kernel

    2. The image after application of the gaussian kernel (scale space)

"""

def task\_1b():

    images, sigma\_arr = get\_twelve\_smoothed\_images(\

            load\_image\_and\_convert\_to\_grayscale())

    title\_arr = []

    for s in sigma\_arr:

        title\_arr.append(f"sigma = {s:2.2f}")

    display\_images\_meshgrid(images, \

            gray=True, title\_arr=title\_arr, suptitle="Gaussian Smoothed Image")

    gaussian\_kernels = []

    for s in sigma\_arr:

        gaussian\_kernels.append(get\_gaussian\_smoothing\_kernel(s))

    display\_images\_meshgrid(gaussian\_kernels, \

            gray=False, title\_arr=title\_arr, suptitle="gaussian kernels")

    plt.show()

The above function displays the Gaussian kernels and the images after the Gaussian kernels are applied to it. The corresponding line numbers are 231-251.

The result of application of this follows.

The Guassian kernels are visualized below.

A picture containing text, sky, light

Description automatically generated

When these kernels are applied to the image, the scale space is created and is visualized below.

Diagram, engineering drawing

Description automatically generated

# Task 2: Feature Point Locations

## Difference of Gaussians

The difference of Gaussians is created by the following function in lines 124-135

"""

Get difference of gaussian images, with different sigma values

"""

def get\_difference\_of\_gaussian\_images(images:np.ndarray,\

        sigma\_arr:list)->list:

    dog\_arr = []

    for i in range(len(images)-1):

        i1, i2 = images[i], images[i+1]

        s1, s2 = sigma\_arr[i], sigma\_arr[i+1]

        dog = i2 - i1 + GAUSSIAN\_ADD

        dog\_arr.append({"dog": dog, "sigma1": s1, "sigma2": s2, })

    return dog\_arr

In the above, images is an array of images to which a Gaussian kernel has already been applied, and the sigma\_arr is an array containing the sigma values which have been applied to the corresponding image in the images array.

This function is called from task\_2() in lines 279-280.

*# Get the difference of Gaussian by calling this function*

    dog\_arr = get\_difference\_of\_gaussian\_images(new\_images, new\_sigma\_arr)

Finally all the difference of Gaussian images are displayed from lines 282-292:

*# Now display all the difference of Gaussian images. Most of the code here*

*# Is to get the right title appended to each dog*

    display = []

    display\_text = []

    for x in dog\_arr:

        dog, s1, s2 = x["dog"], x["sigma1"], x["sigma2"]

        display\_text.append(f"{s2} - {s1}")

        display.append(dog)

    display\_images\_meshgrid(display,\

            True, display\_text, "Difference of Gaussian")

    plt.show(

The resultant images are represented below:

A picture containing diagram

Description automatically generated

## Finding Key Points

Non Maxima suppression with a threshold of 10 is applied to get the scale space. For every layer the layer above and below it is considered. The first function is is\_maximal\_pixel() in lines 138-179. This function is called thrice for each layer – for the layer itself, for the layer above it, and for the layer below it, for every pixel in the layer. Calling this function thrice makes the condition for comparison somewhat more readable.

"""

This function is called multiple times. It checks if a pixel is maximal

or not.

There is an additional parameter check\_same\_level

If this parameter is True, then it will check [x,y], if not, it will not

check [x,y]

This is because this will be called thrice for each layer,

1. once for sigma = 2\*\*(k/2)

2. Once for sigma = 2\*\*((k-1)/2)

3. Once for sigma = 2\*\*((k+1)/2)

For 1, check\_same\_level=False and True otherwise

"""

def is\_maximal\_pixel(\

        metric:np.ndarray,\

        x:int,\

        y:int,\

        pixel\_value:float,\

        T:int,\

        check\_same\_level:bool=True)->bool:

    try:

        if ((pixel\_value > T) and

            (pixel\_value > metric[x-1,y-1]) and

            (pixel\_value > metric[x-1,y])   and

            (pixel\_value > metric[x-1,y+1]) and

            (pixel\_value > metric[x,y-1])   and

            (pixel\_value > metric[x,y+1])   and

            (pixel\_value > metric[x+1,y-1]) and

            (pixel\_value > metric[x+1,y])   and

            (pixel\_value > metric[x+1,y+1])):

                if check\_same\_level and pixel\_value > metric[x, y]:

                    return True

                elif not check\_same\_level:

                    return True

                else:

                    return False

        else:

            return False

    except:

        return False

    return False

The above function is called by get\_non\_maxima\_suppression\_pixels(). For every pixel, it calls the above function for the pixel in the current layer, the corresponding pixel in the layer above, and the corresponding pixel in the layer below it. If all of these indicate that the pixel is the maximal, then it is considered a key point. This is then added to a list of all key points. (lines 180-206)

"""

For a particular level in scale-space, get all non-maxima suppressed pixels

at that level.

This is done by calling is\_maximal\_pixel for every pixel

"""

def get\_non\_maxima\_suppression\_pixels(\

        image:np.ndarray,\

        lower:np.ndarray,\

        higher:np.ndarray,\

        T:int,\

        sigma:list)->list:

    points = []

    for x in range(1,len(image)-1):

        for y in range(1,len(image[0])-1):

            m1, m2, m3 = True, True, True

            if lower is not None:

                m1 = is\_maximal\_pixel(lower, x, y, image[x, y], T, True)

            if not m1:

                continue

            if higher is not None:

                m3 = is\_maximal\_pixel(higher, x, y, image[x, y], T, True)

            if not m3:

                continue

            m2 = is\_maximal\_pixel(image, x, y, image[x, y], T, False)

            if m1 and m2 and m3:

                points.append((x, y, sigma, ))

    return point

Now that all these supporting functions have been described, these are used by get\_all\_non\_maxima\_suppression\_pixels() to get the complete list of all key points. This function takes as argument an array of all difference of Gaussian images, and a threshold T, and for each layer it calls get\_non\_maxima\_suppression\_pixels, and collects the results. (lines 208-229).

"""

Get all keypoints for all levels in scale space

"""

def get\_all\_non\_maxima\_suppression\_pixels(dog\_arr:list, T:int)->list:

    points = []

    for i in range(len(dog\_arr)):

        lower = higher = image = None

        x = dog\_arr[i]

        image = x["dog"]

        sigma = x["sigma1"]

        if i - 1 >= 0:

            lower = dog\_arr[i - 1]["dog"]

        else:

            continue

        if i + 1 < len(dog\_arr):

            higher = dog\_arr[i + 1]["dog"]

        else:

            continue

        p = get\_non\_maxima\_suppression\_pixels(image, lower, higher, T, sigma)

        [points.append(x) for x in p]

        print(f"Finding keypoints: sigma {sigma:2.2f} - {len(points)} points")

    return points

The resulting pixels are displayed by drawing circles around the points, with the radius sigma. (lines 294-311).

*# Get all key points by non-maxima suppression*

    points = get\_all\_non\_maxima\_suppression\_pixels(dog\_arr,\

                GAUSSIAN\_ADD+THRESHOLD)

    print(f"Number of points found = {len(points)}")

*# For each of the key points, draw a circle*

    for point in points:

        x, y, sigma = point

        x = int(x)

        y = int(y)

        radius = math.floor(3 \* sigma)

        cv2.circle(save\_image, (y, x,), radius, (0,255,0), 1)

*# Display the image, and also save it to a file*

    cv2.imwrite("circles\_only.jpg", save\_image)

    cv2.imshow("result", save\_image)

    cv2.waitKey(0)

    cv2.destroyAllWindows()

The resultant image is as follows.

A picture containing grass, outdoor, athletic game, green

Description automatically generated

For a better visualization, a different color and radius was also used:

A picture containing text

Description automatically generated

# Task 3: Feature Point Orientation

## Calculating the Derivatives of Gaussians in scale space

The following function get\_derivative\_of\_gaussian() calculates the derivate of Gaussians in scale space. The scale space is passed to it, along with an array containing the sigma values. (lines 322-331)

"""

Get gaussian derivatives of a list of images in different sigma scales

The output is a list, each element of the list is a tuple.

The first item in the tuple is gx, and the second item in the tuple

is gy

"""

def get\_derivative\_of\_gaussian(gaussian\_images:list, sigma\_list)->list:

    retval = []

    dx = np.array([[1, 0, -1]])

    dy = dx.T

    for image in gaussian\_images:

        gx = cv2.filter2D(image.copy(), -1, dx)

        gy = cv2.filter2D(image.copy(), -1, dy)

        retval.append((gx, gy, ))

    return retva

This function is called from task\_3(). Additionally, once the derivative of Gaussians have been created, a cross reference is created so that any value that will be used for a particular sigma later in the function can be cached so that they don’t have to be calculated again and again. There is a lot of repetition in the pixels as we iterate through the points, and that repetition may be cached for speed. This includes the gradient direction (theta), magnitude, square of derivatives (lines 407-446)

def task\_3()->tuple:

*# Get the grayscale image*

    image = load\_image\_and\_convert\_to\_grayscale()

*# Also load the original color image, which we will use*

*# to draw everything at the end*

    save\_image = load\_original\_image()

*# Get the twelve smoothed images*

    images, sigma\_arr = get\_twelve\_smoothed\_images(image)

*# Get the key points from task 2, this will be augmented by the direction*

    key\_points = task\_2()

*# Get the derivative of gaussian images in x and y direction*

    dog\_images = get\_derivative\_of\_gaussian(images, sigma\_arr)

    magnitudes = []

    theta\_array = []

    sigma\_cross\_ref = {}

*# Calculate the magnitude and angle for all points*

*# We will select the points that matter to us later*

*# This cross reference allows us to have a quick way to access*

*# stuff that we've already calculated*

*# task 4 will also use this cross ref, so we're calculating*

*# some stuff here that is not strictly required in this task*

*# as they will be required in task 4*

    for im, sigma, (gx, gy) in zip(images, sigma\_arr, dog\_images):

        magnitude = np.sqrt(np.square(gx) + np.square(gy))

        magnitudes.append(magnitude)

        theta = np.arctan2(gy, gx)

        theta\_array.append(theta)

        sigma\_cross\_ref[sigma] = {

                "gx": gx,

                "gy": gy,

                "gx2": np.square(gx.copy()),

                "gy2": np.square(gy.copy()),

                "image": im,

                "theta": theta,

                "magnitude": magnitude

                }

Finally the derivative of Gaussian images are plotted by the following code (lines 449-451, 392-404)

*# Now plot the derivative of gaussian in x and y directions*

    plot\_gx\_gy(sigma\_cross\_ref)

    plt.show()

"""

Display derivative of gaussian kernels

The cross-ref we create in task\_3() comes in very handy as we can just

lookup anything that we have already computed earlier in O(1) time.

"""

def plot\_gx\_gy(sigma\_cross\_ref:dict)->None:

    title\_arr = []

    gx\_arr = []

    gy\_arr = []

    for i in sigma\_cross\_ref.keys():

        title\_arr.append(f"sigma = {i:2.2f}")

        gx\_arr.append(sigma\_cross\_ref[i]["gx"])

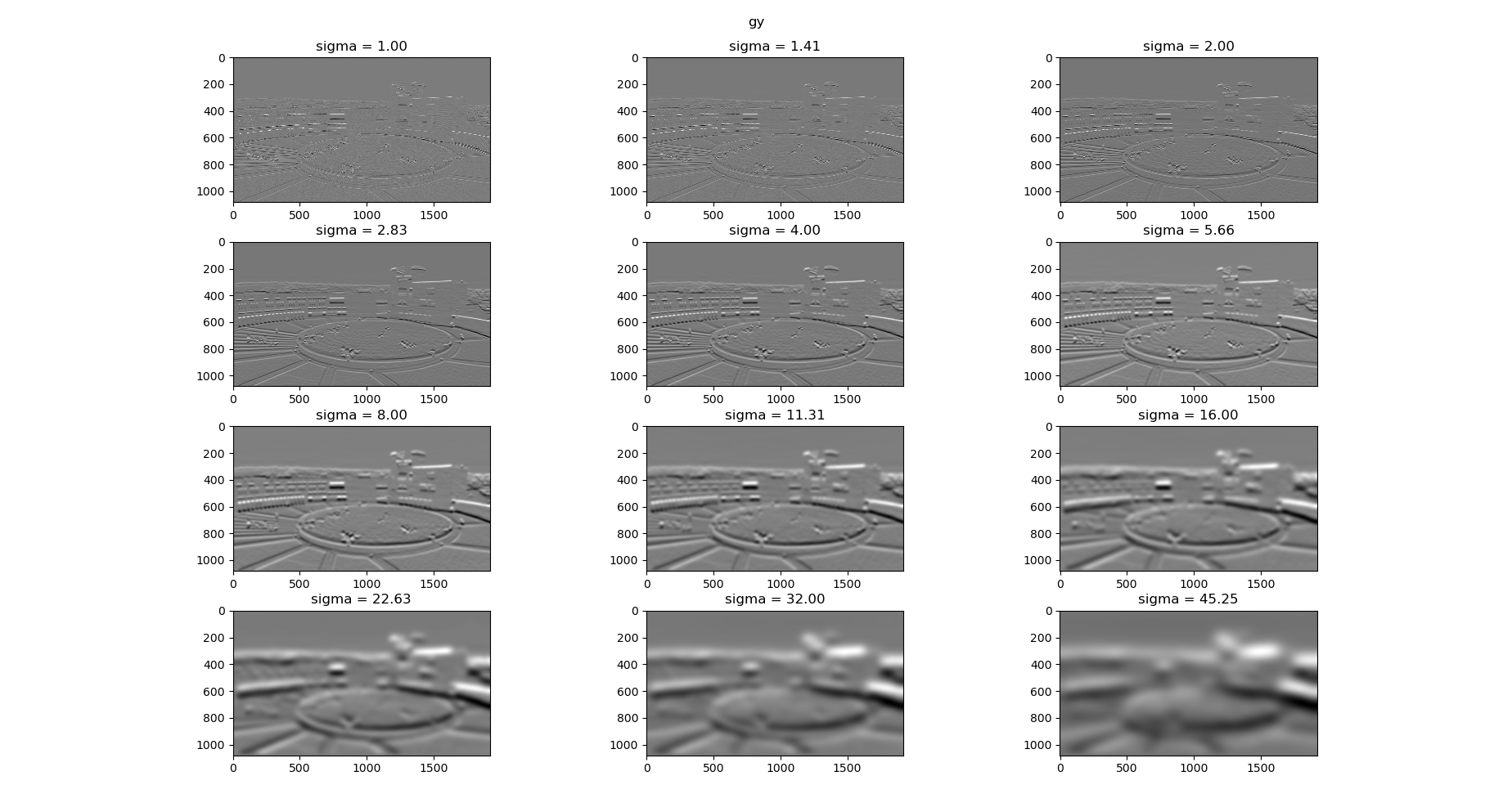
        gy\_arr.append(sigma\_cross\_ref[i]["gy"])

    display\_images\_meshgrid(gx\_arr, True, title\_arr, "gx")

    display\_images\_meshgrid(gy\_arr, True, title\_arr, "gy")

The resulting derivatives of Gaussians in x and y direction are represented below.

Diagram, calendar

Description automatically generated

The Gaussian weighting function for this step is implemented as follows. (lines 332-341).

"""

In task 3C, we need the weights for each pixel to multiply with the magnitude

of each point. This function gives us the weight of each point.

LRU cache is used to memoize the result once it is calculated

"""

@lru\_cache

def get\_weight(q:int, r:int, sigma:float)->float:

    wqr = np.exp( -(q\*\*2 + r\*\*2) / (9 \* sigma\*\*2 / 2 ))

    wqr = wqr / (9 \* np.pi \* sigma\*\*2 / 2)

    return wqr

The above function is decorated with @lru\_cache so that the values don’t need to be calculated again and again.

This calculated weight is used in get\_augmented\_point\_with\_direction() to get the final direction of each key point. This is done by considering each point on a 7x7 grid around the key point, and for each point, bin its weight into 36 bins indexed by the angle. Finally the bin with the highest weight is taken and converted back into the angle and that is assumed to be angle for the key point.

"""

For task 3C, this function multiplies the magnitude with the weight

for each point, and then uses this product to bin the angles

and then appends the angle to each key point

The magnitude and angle have already been calculated for all the points

and saved so that they needn't be calculated again and again

"""

def get\_augmented\_point\_with\_direction(\

        im:np.ndarray,\

        x:int,\

        y:int,\

        sigma:np.ndarray,\

        m\_gx:np.ndarray,\

        m\_gy:np.ndarray,\

        m\_theta:np.ndarray,\

        m\_magnitude:np.ndarray)->tuple:

    hist = [0.0] \* 36

    for k in range(-3, 4):

        for j in range(-3, 4):

            q = round(3/2 \* k \* sigma)

            r = round(3/2 \* j \* sigma)

            xq = x + q

            yr = y + r

            mqr = m\_magnitude[xq, yr]

            tqr = m\_theta[xq, yr]

            wqr = get\_weight(q, r, sigma)

            i = int(math.floor((tqr + np.pi) / (2 \* np.pi / 36)))

            hist[i] += (wqr \* mqr)

    thetahat = (2 \* np.pi / 36) \* (0.5 + np.argmax(np.array(hist))) - np.pi

    return (x, y, sigma, thetahat, )

Lastly, the key points are visualized by drawing a circle of 3\*sigma around the key point, along with the radius in the direction of the point calculated above. (lines 453-487).

*# For each key point, find the direction and augment that point*

    augmented\_points = []

    exception\_count = 0

    for point in key\_points:

        try:

            x, y, sigma = point

            augmented\_points.append(get\_augmented\_point\_with\_direction(\

                    sigma\_cross\_ref[sigma]["image"],\

                    x,\

                    y,\

                    sigma,\

                    sigma\_cross\_ref[sigma]["gx"],\

                    sigma\_cross\_ref[sigma]["gy"],\

                    sigma\_cross\_ref[sigma]["theta"],\

                    sigma\_cross\_ref[sigma]["magnitude"]))

        except:

            exception\_count += 1

*# Now that we have found the direction theta for each point,*

*# Draw a circle and a radius in the direction that we just found out*

*# Also save the window*

    save\_image = load\_original\_image()

    for point in augmented\_points:

        x, y, sigma, angle = point

        x = int(x)

        y = int(y)

        radius = math.floor(3 \* sigma)

        cv2.circle(save\_image, (y, x,), radius, (0,255,0), 1)

        x1 = int(round(x + radius \* math.cos(angle)))

        y1 = int(round(y + radius \* math.sin(angle)))

        cv2.line(save\_image, (y, x, ), (y1, x1, ), (0,0,255,), 2)

    cv2.imwrite("circles\_with\_radius.jpg", save\_image)

    cv2.imshow("result", save\_image)

    cv2.waitKey(0)

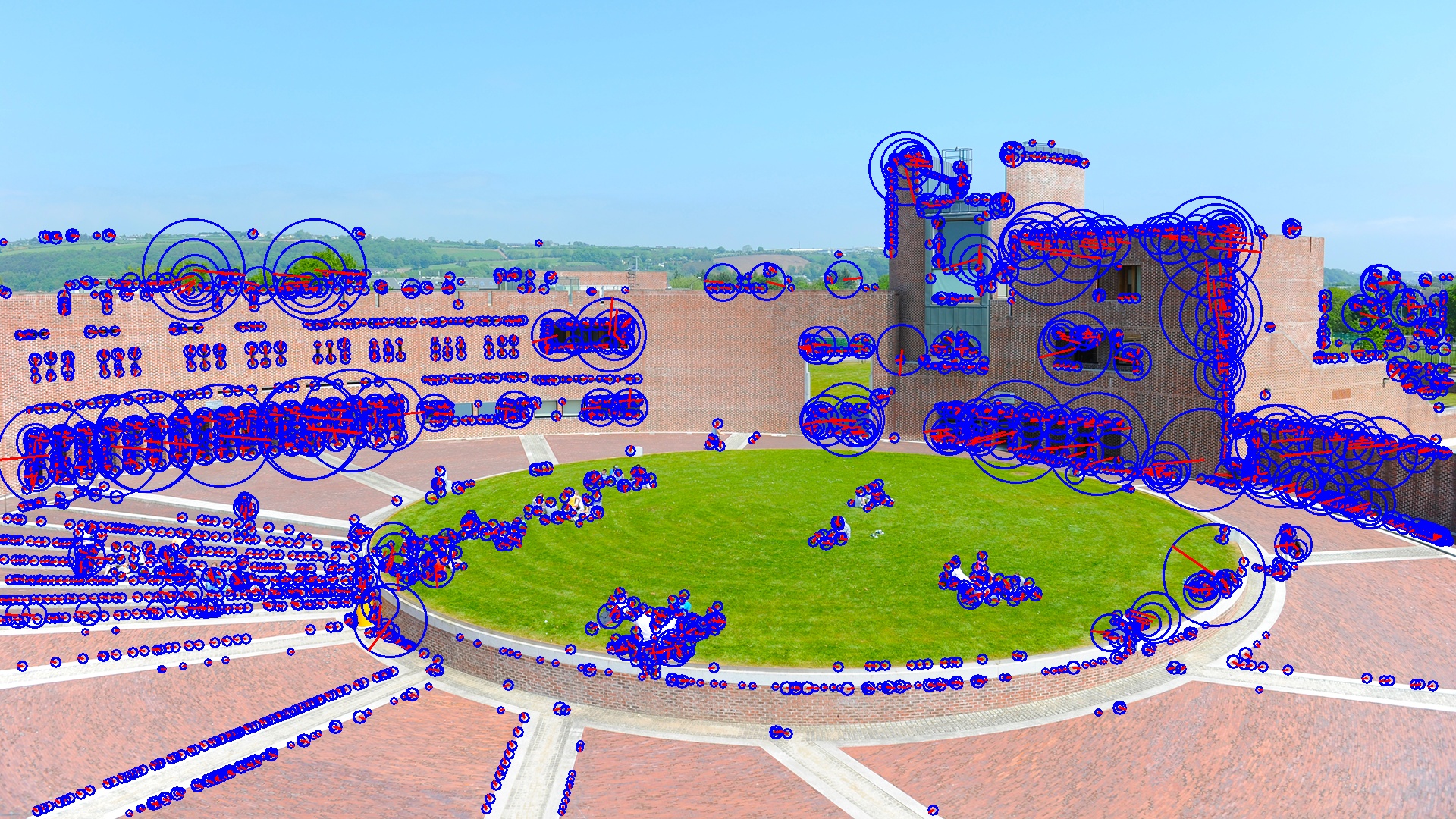
    cv2.destroyAllWindows()

The resultant image with the represented circles and directions is pasted below.

A picture containing grass, outdoor, building, stadium

Description automatically generated

For a better visualization a different color and thickness was also used.



# Task 4: Feature Descriptors

The area around each key point is 16 grids, each containing 4x4 pixels. For each grid, a 8-bin histogram is created, and the product of magnitude and the weight is binned into them. This time the weighting function used is slightly different and is implemented as follows in lines 491-497:

"""The magnitude at each point will be multiplied by this weight in

task 4

"""

@lru\_cache

def get\_weight2(s:int, t:int, sigma:float)->float:

    return np.exp(-(s\*\*2 + t\*\*2) / (81 \* sigma\*\*2 /2)) / \

            (81 \* np.pi \* sigma\*\*2 / 2)

These 8 bins are calculated for each of the 16 grids and appended to form the final key point descriptor. The implementation is from lines 499-570

def task\_4()->None:

*# Get the key points augmented with the direction of each point,*

*# as well as the sigma cross reference of all the stuff we've already*

*# calculated at that sigma in task 2*

    augmented\_points, sigma\_crf = task\_3()

*# This array will hold all the SIFT descriptors for all the points*

    all\_descriptors = []

*# Now iterate all the points and find out the SIFT descriptor for each*

*# point*

    for point in augmented\_points:

        try:

            x, y, sigma, theta\_hat = point

            theta\_arr = sigma\_crf[sigma]["theta"]

            magnitude\_arr = sigma\_crf[sigma]["magnitude"]

            all\_hist = []

*# Break the pixels around the key point into 16 4x4 grids*

*# two nested loops to form 16 4x4 grids*

            for i in range(-2, 2):

                for j in range(-2, 2):

*# These will be used to calculate the coordinates inside*

*# each 4x4 grid*

                    kk1 = [4 \* i + x for x in range(4)]

                    kk2 = [4 \* j + x for x in range(4)]

*# This is the orientation histogram for each 4x4 subgrid*

                    hist = [0.0] \* 8

*# For each of the pixels in the 4x4 grid*

                    for k1 in kk1:

                        for k2 in kk2:

                            s = int(round(9/16 \* (k1 + 0.5) \* sigma))

                            t = int(round(9/16 \* (k2 + 0.5) \* sigma))

                            wst = get\_weight2(s, t, sigma)

                            xs = x + s

                            yt = y + t

*# Find the magnitude*

                            mst = magnitude\_arr[xs, yt]

*# Find the angle and subtract theta\_hat from it*

                            theta\_st = theta\_arr[xs, yt] - theta\_hat

*# Find the index in the histogram for this*

*# 4x4 subgrid from the angle*

                            ind = int(math.floor(theta\_st / (2 \* np.pi / 8)))

*# Update the weight in that index of the histogram*

                            hist[ind] += (wst \* mst)

*# Now that we're done calculating the histogram for this*

*# 4x4 sub-grid, append it to the list of histograms for*

*# this point to form the descriptor*

*# At the end of all loops for this point, this will have*

*# 128 floats*

                    all\_hist.extend(hist)

*# append the descriptors for this point into the global list*

*# of descriptors.*

*# Each descriptor for each point has 128 floats*

            all\_hist = np.array(all\_hist)

            all\_hist = all\_hist / np.sqrt(np.sum(np.square(all\_hist)))

            all\_hist[all\_hist > 0.2] = 0.2

            all\_descriptors.append(all\_hist)

        except:

            pass

    for descriptor in all\_descriptors[0:5]:

        print(len(descriptor), descriptor)

Finally all the descriptor is normalized, and then all values are capped at 0.2.