Contents

[Usage 1](#_Toc67003376)

[Task 1 : Scale Space 1](#_Toc67003377)

[Loading the image 1](#_Toc67003378)

[Creating 12 Gaussian kernels 2](#_Toc67003379)

[Task 2: Feature Point Locations 4](#_Toc67003380)

[Difference of Gaussians 4](#_Toc67003381)

[Finding Key Points 5](#_Toc67003382)

[Task 3: Feature Point Orientation 9](#_Toc67003383)

# Usage

The usage of this python file is as follows

./keypoints.py -f <filename>

Or

./keypoints.py –file <filename>

# 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

# Task 3: Feature Point Orientation