Computer Vision

Phase 2

Team Member Names	Section	Bench Number
Osama Faisal AbdulLatef	1	11
Beshara Safwat Fahim	1	22
Shuaib AbdulSalam Ahmed	1	48
AbdElRahman Sameh Mohamed	1	53
Mariam Mounier AbdElRehim	2	35
Sara Tarek		

Canny Edge Detection for Circled Shapes

Overview:

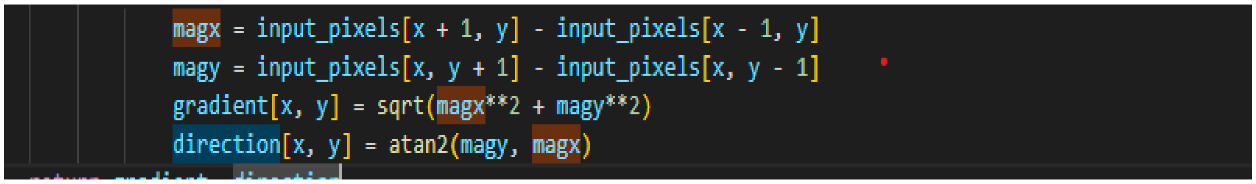
The source codes are written in Python with the libraries PIL and Numpy. However, no advanced features are used and it should be trivial to reimplement the algorithm in any other language.

In order to detect the circles, or any other geometric shape, we first need to detect the edges of the objects present in the image.

The edges in an image are the points for which there is a sharp change of color. For instance, the edge of a red ball on a white background is a circle. In order to identify the edges of an image, a common approach is to compute the image gradient.

Since an image is composed of a set of discrete values, the derivative functions must be approximated. The most common way to approximate the image gradient is to convolve an image with a kernel, such as the Sobel operator

The simplest way to approximate the gradient image is to compute, for each point:



```
            magx = input_pixels[x + 1, y] - input_pixels[x - 1, y]
            magy = input_pixels[x, y + 1] - input_pixels[x, y - 1]
            gradient[x, y] = sqrt(magx**2 + magy**2)
            direction[x, y] = atan2(magy, magx)
```

Where intensity[x, y] is the luminosity of the pixel situated at (x, y).

The Canny edge detector is a multi-stage algorithm that will clean the image and only keep the strongest edges.

The Canny edge detector successively apply the following operations:

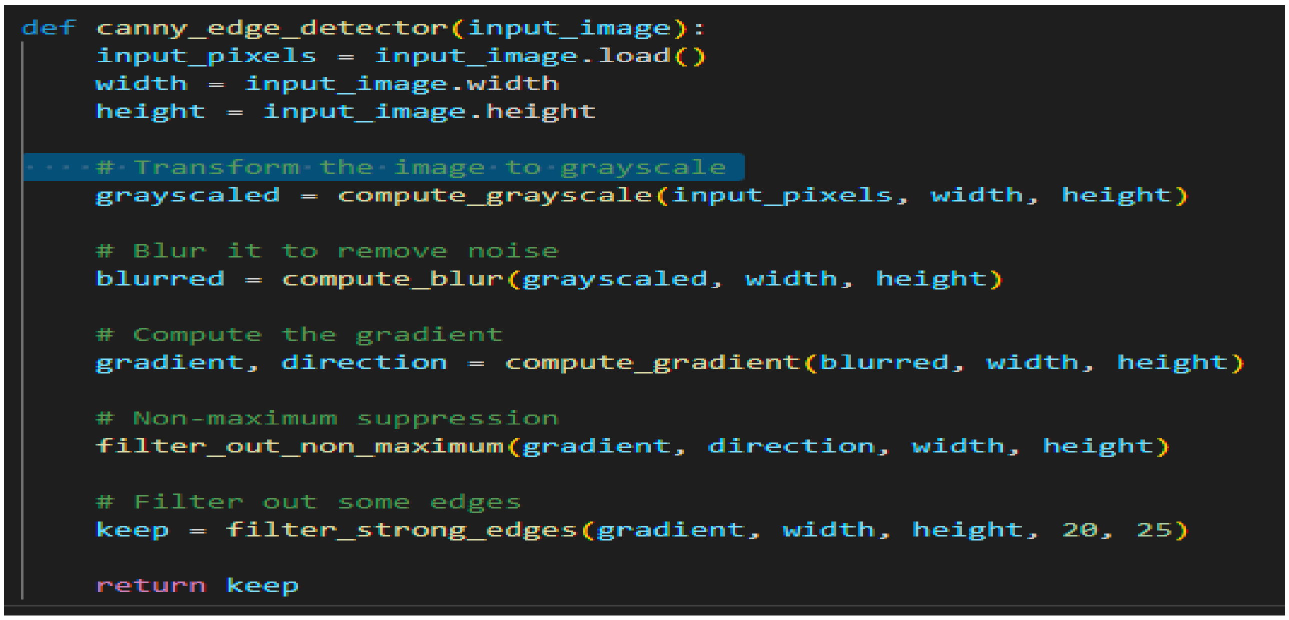
- Gaussian filter
- Compute image gradient
- Non-maximum suppression
- Edge tracking

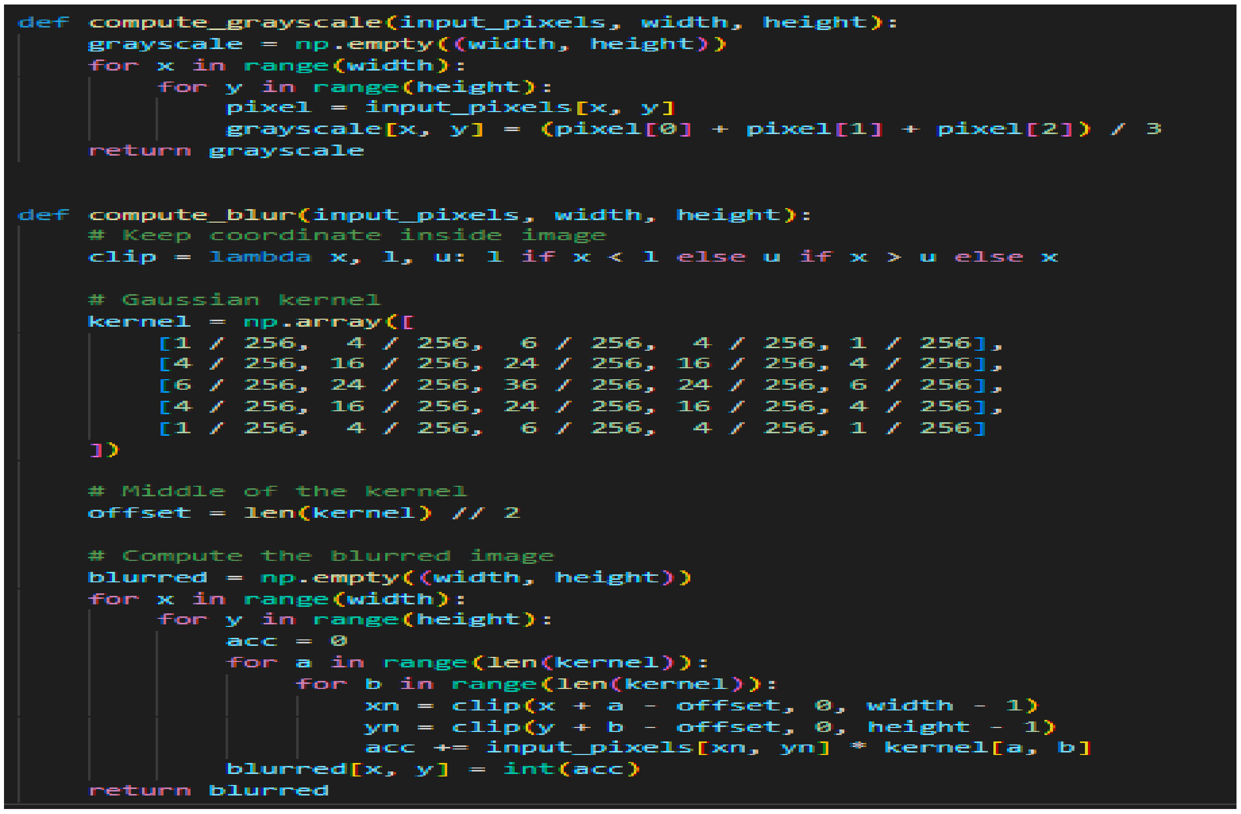
The gaussian filter aims at smoothing the image to remove some noise. And, as just shown, the image gradient will identify the edges. The objective of the next two steps is to remove some edges to only keep those which are the most relevant.

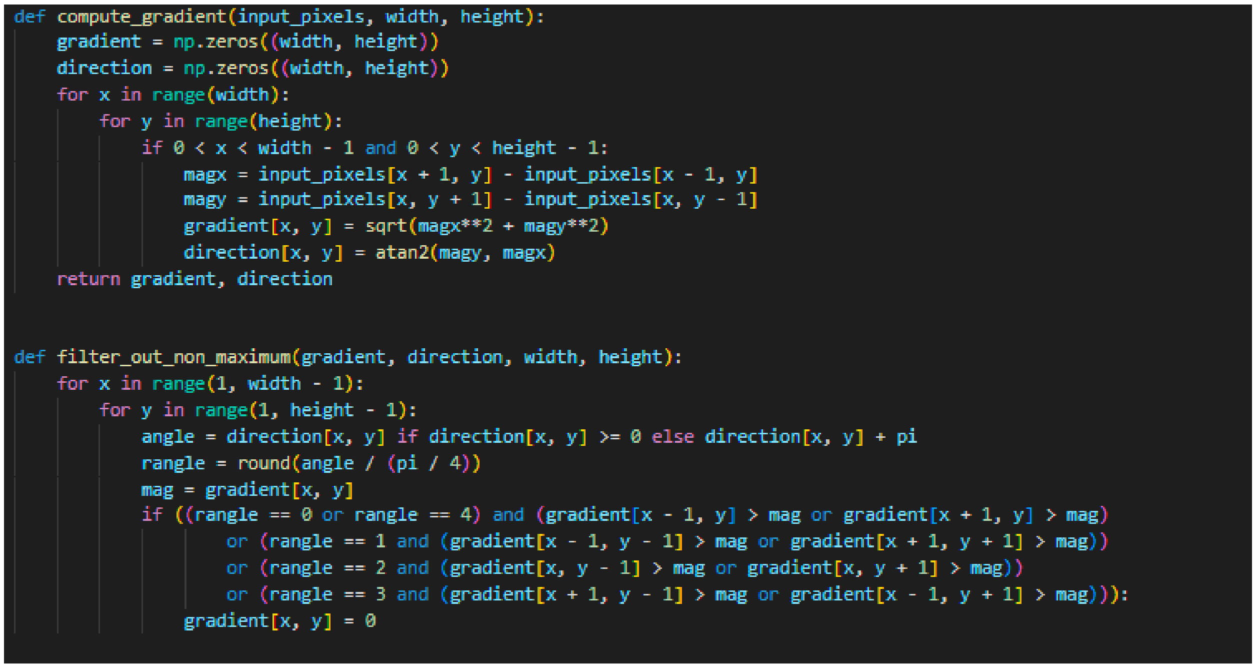
When computing the gradient image, we also compute the direction of the gradient `atan2(magy, magx)`.
Using this, we only keep the pixels that are the maximum among their neighbors in the direction of the gradient. This will thin the edges.

The next step consist of applying two threshold: the pixels with a gradient magnitude lower that `low` are removed, those greater than `high` are marked as strong edges, and those between are marked weak edges. Then, we iterate over the weak edges and mark as strong those which are next to a strong edge.

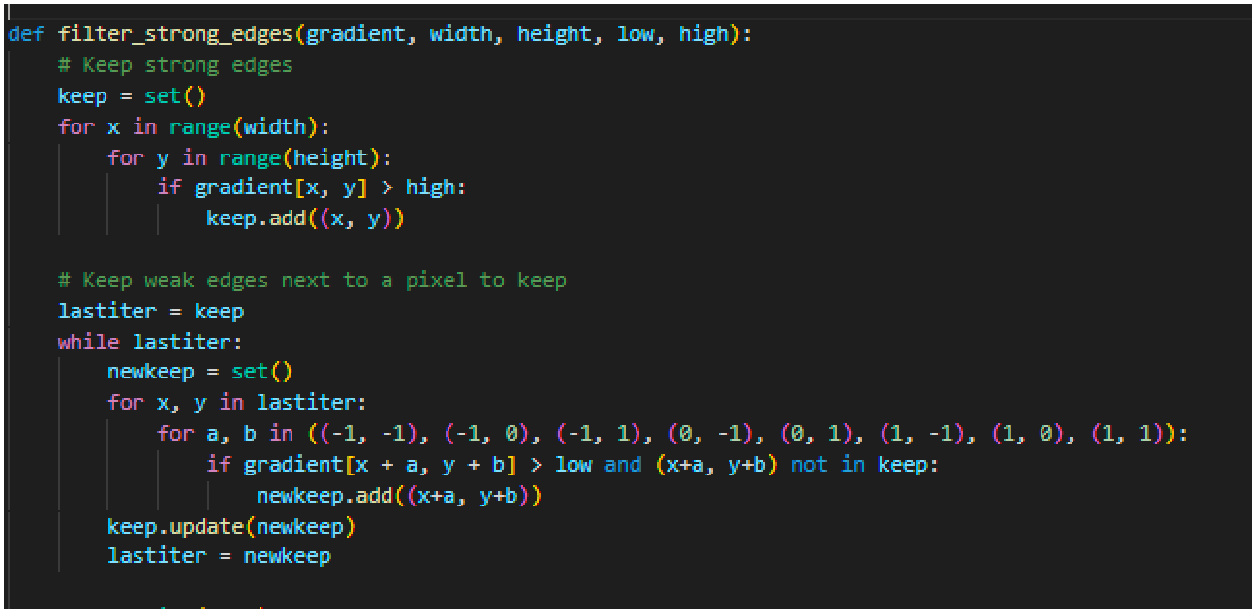
This will allow us to improve the edge continuity.


```
def canny_edge_detector(input_image):
    input_pixels = input_image.load()
    width = input_image.width
    height = input_image.height

    # Transform the image to grayscale
    grayscaled = compute_grayscale(input_pixels, width, height)

    # Blur it to remove noise
    blurred = compute_blur(grayscaled, width, height)

    # Compute the gradient
    gradient, direction = compute_gradient(blurred, width, height)

    # Non-maximum suppression
    filter_out_non_maximum(gradient, direction, width, height)

    # Filter out some edges
    keep = filter_strong_edges(gradient, width, height, 20, 25)

    return keep
```


```
def compute_grayscale(input_pixels, width, height):
    grayscale = np.empty((width, height))
    for x in range(width):
        for y in range(height):
            pixel = input_pixels[x, y]
            grayscale[x, y] = (pixel[0] + pixel[1] + pixel[2]) / 3
    return grayscale


def compute_blur(input_pixels, width, height):
    # Keep coordinate inside image
    clip = lambda x, l, u: l if x < l else u if x > u else x

    # Gaussian kernel
    kernel = np.array([
        [1 / 256,  4 / 256,  6 / 256,  4 / 256, 1 / 256],
        [4 / 256, 16 / 256, 24 / 256, 16 / 256, 4 / 256],
        [6 / 256, 24 / 256, 36 / 256, 24 / 256, 6 / 256],
        [4 / 256, 16 / 256, 24 / 256, 16 / 256, 4 / 256],
        [1 / 256,  4 / 256,  6 / 256,  4 / 256, 1 / 256]
    ])

    # Middle of the kernel
    offset = len(kernel) // 2

    # Compute the blurred image
    blurred = np.empty((width, height))
    for x in range(width):
        for y in range(height):
            acc = 0
            for a in range(len(kernel)):
                for b in range(len(kernel)):
                    xn = clip(x + a - offset, 0, width - 1)
                    yn = clip(y + b - offset, 0, height - 1)
                    acc += input_pixels[xn, yn] * kernel[a, b]
            blurred[x, y] = int(acc)
    return blurred
```



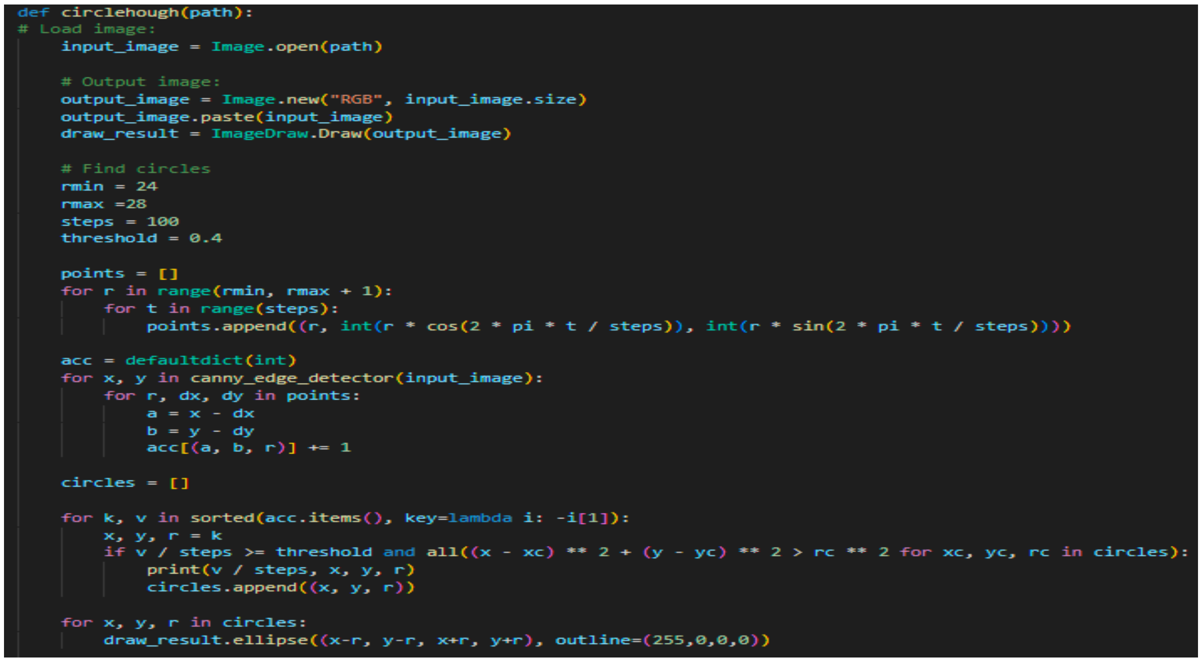
```python
def compute_gradient(input_pixels, width, height):
    gradient = np.zeros((width, height))
    direction = np.zeros((width, height))
    for x in range(width):
        for y in range(height):
            if 0 < x < width - 1 and 0 < y < height - 1:
                magx = input_pixels[x + 1, y] - input_pixels[x - 1, y]
                magy = input_pixels[x, y + 1] - input_pixels[x, y - 1]
                gradient[x, y] = sqrt(magx**2 + magy**2)
                direction[x, y] = atan2(magy, magx)
    return gradient, direction


def filter_out_non_maximum(gradient, direction, width, height):
    for x in range(1, width - 1):
        for y in range(1, height - 1):
            angle = direction[x, y] if direction[x, y] >= 0 else direction[x, y] + pi
            rangle = round(angle / (pi / 4))
            mag = gradient[x, y]
            if ((rangle == 0 or rangle == 4) and (gradient[x - 1, y] > mag or gradient[x + 1, y] > mag)
                    or (rangle == 1 and (gradient[x - 1, y - 1] > mag or gradient[x + 1, y + 1] > mag))
                    or (rangle == 2 and (gradient[x, y - 1] > mag or gradient[x, y + 1] > mag))
                    or (rangle == 3 and (gradient[x + 1, y - 1] > mag or gradient[x - 1, y + 1] > mag))):
                gradient[x, y] = 0
```



```python
def filter_strong_edges(gradient, width, height, low, high):
    # Keep strong edges
    keep = set()
    for x in range(width):
        for y in range(height):
            if gradient[x, y] > high:
                keep.add((x, y))

    # Keep weak edges next to a pixel to keep
    lastiter = keep
    while lastiter:
        newkeep = set()
        for x, y in lastiter:
            for a, b in ((-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)):
                if gradient[x + a, y + b] > low and (x+a, y+b) not in keep:
                    newkeep.add((x+a, y+b))
        keep.update(newkeep)
        lastiter = newkeep
```

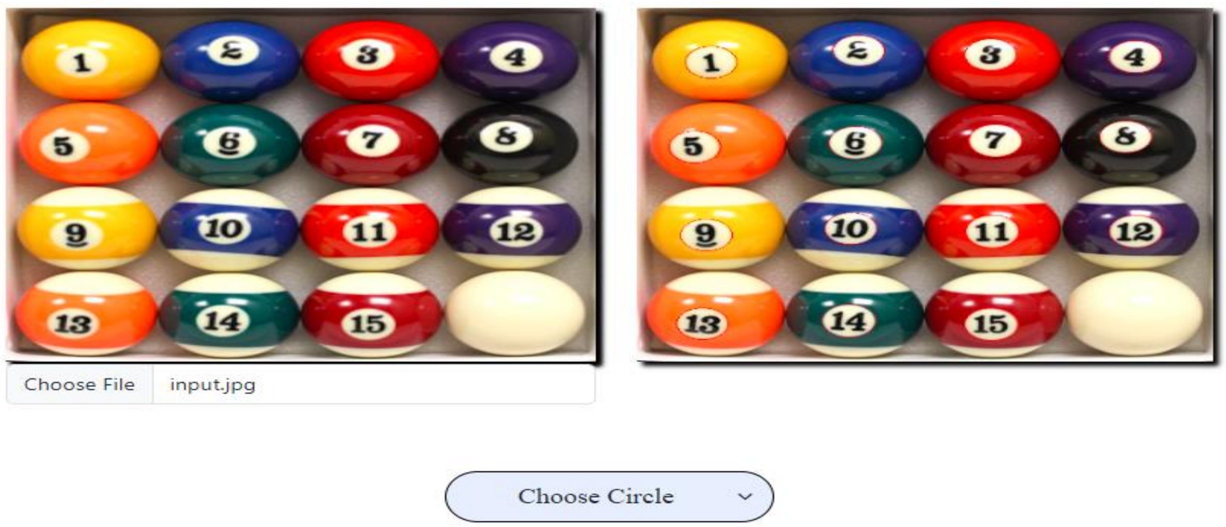
Using the edges given by the Canny edge detector and for each possible circle, we count the number of edges that are part of each circle. For each of these circles, we increment a counter.

In order to select which circles are good enough, we use two criteria: a threshold (here, at least 40% of the pixels of a circle must be detected) and we exclude circles that are too close of each other (here, once a circle has been selected, we reject all the circles whose center is inside that circle).



```
def circlehough(path):
# Load image:
    input_image = Image.open(path)

    # Output image:
    output_image = Image.new("RGB", input_image.size)
    output_image.paste(input_image)
    draw_result = ImageDraw.Draw(output_image)

    # Find circles
    rmin = 24
    rmax =28
    steps = 100
    threshold = 0.4

    points = []
    for r in range(rmin, rmax + 1):
        for t in range(steps):
            points.append((r, int(r * cos(2 * pi * t / steps)), int(r * sin(2 * pi * t / steps))))

    acc = defaultdict(int)
    for x, y in canny_edge_detector(input_image):
        for r, dx, dy in points:
            a = x - dx
            b = y - dy
            acc[(a, b, r)] += 1

    circles = []

    for k, v in sorted(acc.items(), key=lambda i: -i[1]):
        x, y, r = k
        if v / steps >= threshold and all((x - xc) ** 2 + (y - yc) ** 2 > rc ** 2 for xc, yc, rc in circles):
            print(v / steps, x, y, r)
            circles.append((x, y, r))

    for x, y, r in circles:
        draw_result.ellipse((x-r, y-r, x+r, y+r), outline=(255,0,0,0))
```

OUTPUT



Hough Transform for Line Detection

Overview

The general expression of a line is $y=mx+c$. It means that you can map a line into a coordinate pair, m, c. However, vertical lines have a problem, since the slope value is unbounded. In this case, the hough transform uses a different parametric representation. The parameters are θ and ρ. Here, θ is the angle between the axis and the origin line connecting that closest point. P corresponds to the distance from the origin to the nearest point on the straight line.

Conceptually, Hough transform is the mapping of image coordinate in the x and y into the parametric coordinate θ,ρ

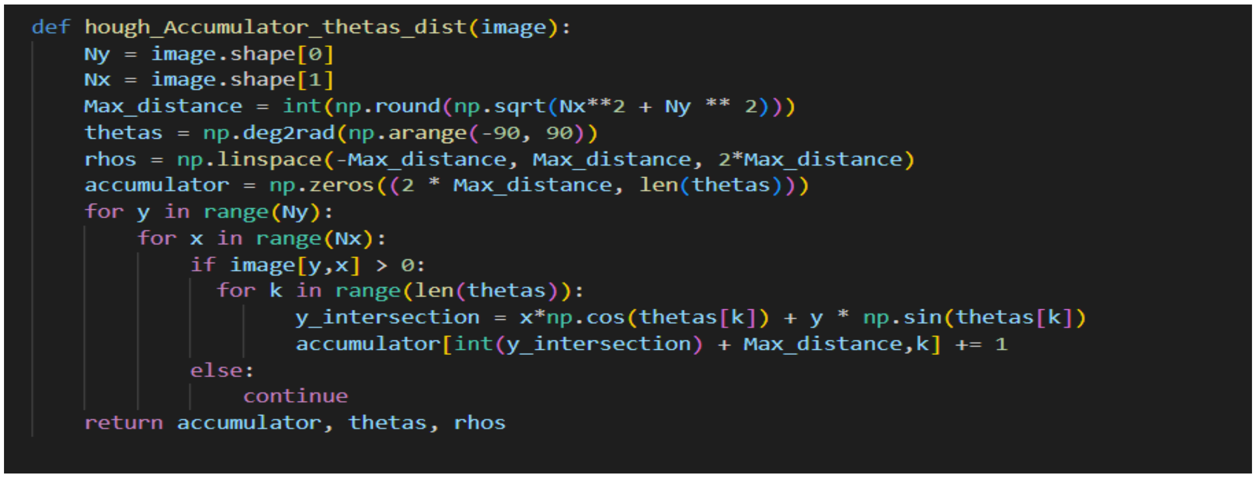
A line is made up of multiple points. Similarly, multiple lines can pass through the same point. We visualize multiple lines in a hough transform plane where the parameters θ are on the x-axis and ρ on the y-axis. Every point on the image is transformed into a sinusoid. A sinusoid is a signal in the shape of a sine wave.

It makes sense because many lines may pass through a given point in an image plane. It translates into a sinusoid in the Hough plane. If two or more of these lines coincide to form a line in the image plane, they will intersect in the Hough transform plane. This intersection corresponding to the θ,ρ pair corresponds to the detected line.

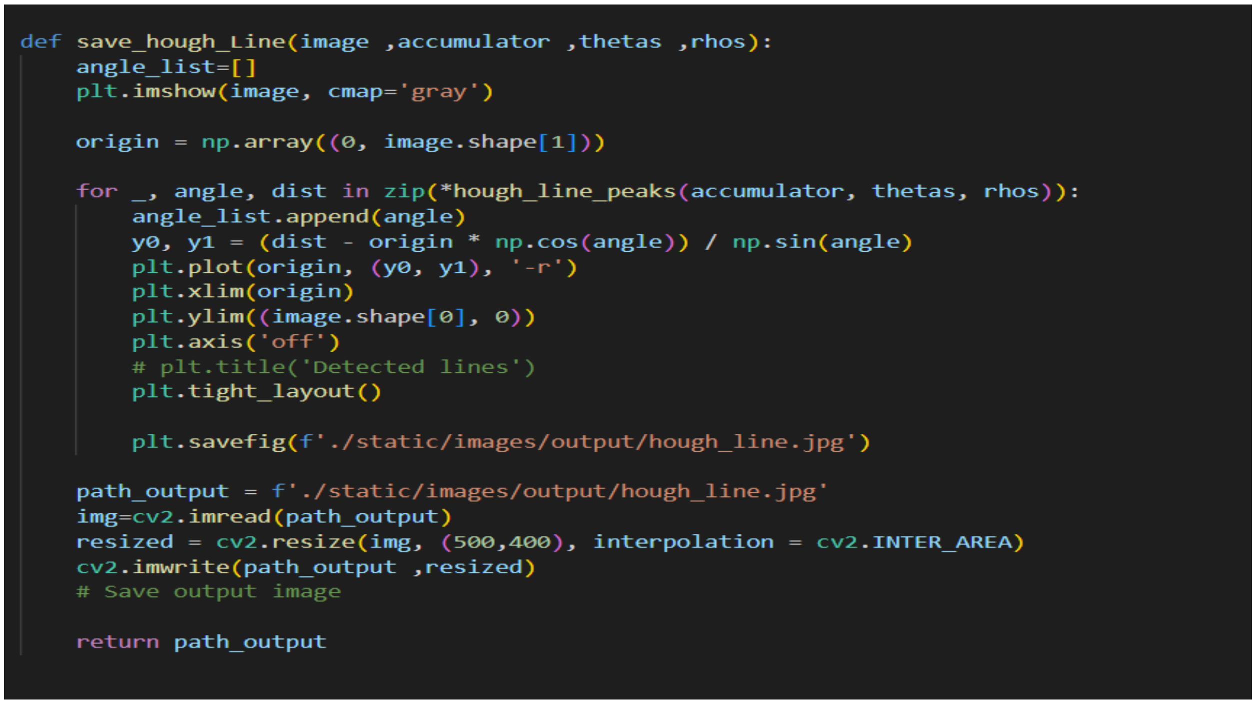
Suppose you have an image with a line and you want to extract the line segments; the algorithm is:

1. Create a hough transform matrix using the hough function.
2. Locate the peaks in the Hough transform matrix function. The peaks correspond to the intersection point in the parameter plane. Each of these peaks is a detected line.

3. Use a hough transform function to map the peak back into the image plane. You then extract the line segments.

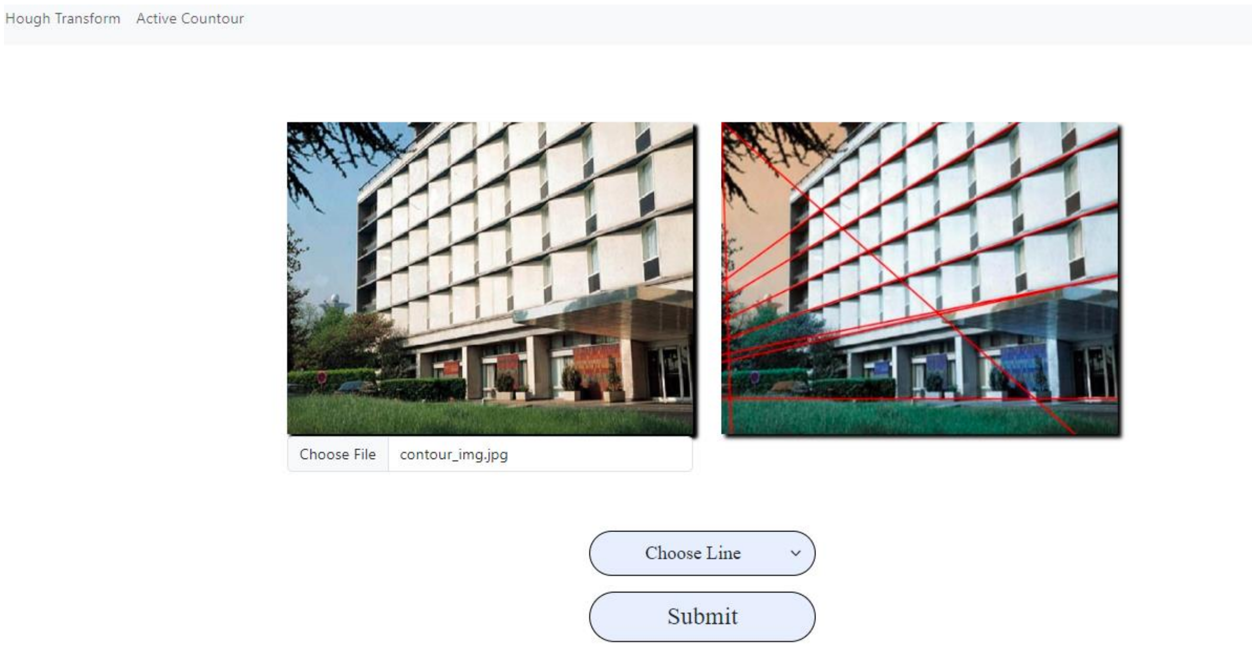


```python
def hough_Accumulator_thetas_dist(image):
    Ny = image.shape[0]
    Nx = image.shape[1]
    Max_distance = int(np.round(np.sqrt(Nx**2 + Ny ** 2)))
    thetas = np.deg2rad(np.arange(-90, 90))
    rhos = np.linspace(-Max_distance, Max_distance, 2*Max_distance)
    accumulator = np.zeros((2 * Max_distance, len(thetas)))
    for y in range(Ny):
        for x in range(Nx):
            if image[y,x] > 0:
              for k in range(len(thetas)):
                    y_intersection = x*np.cos(thetas[k]) + y * np.sin(thetas[k])
                    accumulator[int(y_intersection) + Max_distance,k] += 1
            else:
                continue
    return accumulator, thetas, rhos
```



```python
def save_hough_Line(image ,accumulator ,thetas ,rhos):
    angle_list=[]
    plt.imshow(image, cmap='gray')

    origin = np.array((0, image.shape[1]))

    for _, angle, dist in zip(*hough_line_peaks(accumulator, thetas, rhos)):
        angle_list.append(angle)
        y0, y1 = (dist - origin * np.cos(angle)) / np.sin(angle)
        plt.plot(origin, (y0, y1), '-r')
        plt.xlim(origin)
        plt.ylim((image.shape[0], 0))
        plt.axis('off')
        # plt.title('Detected lines')
        plt.tight_layout()

        plt.savefig(f'./static/images/output/hough_line.jpg')

    path_output = f'./static/images/output/hough_line.jpg'
    img=cv2.imread(path_output)
    resized = cv2.resize(img, (500,400), interpolation = cv2.INTER_AREA)
    cv2.imwrite(path_output ,resized)
    # Save output image

    return path_output
```

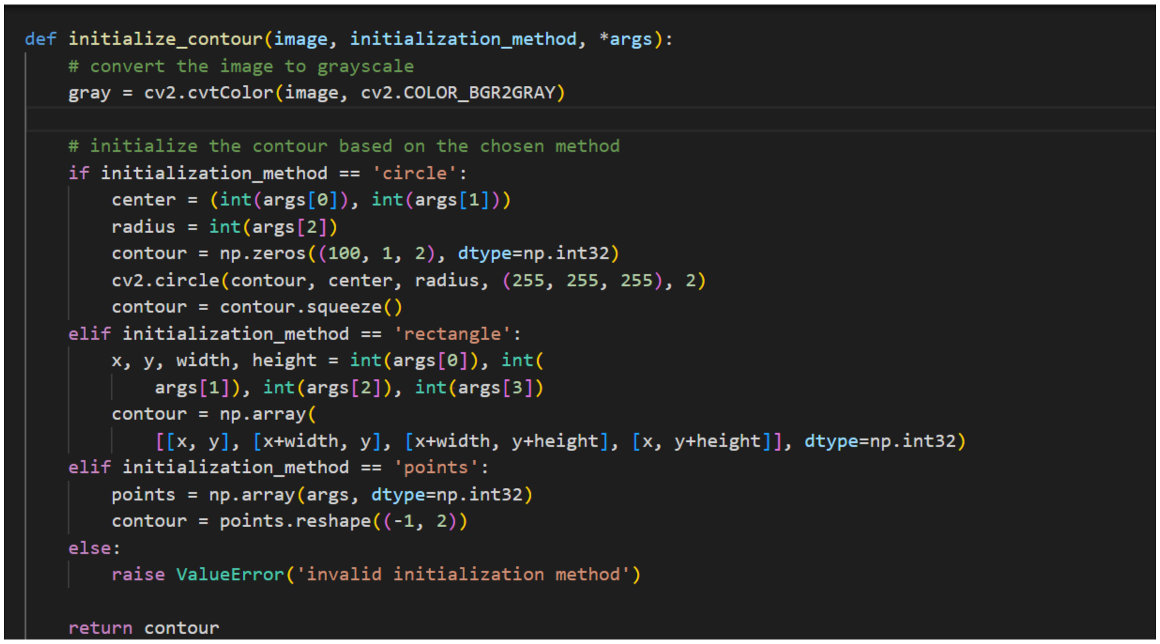
Output



Active contour using greedy algorithm

The initialize_contour function initializes the contour for segmentation using one of three choices: circle, rectangle, or points.

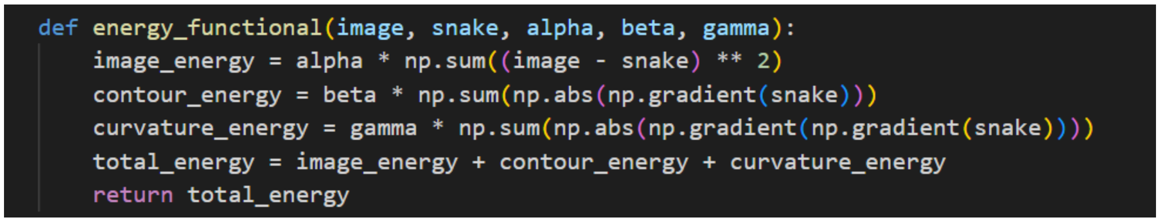
The input is image and the initialization method of choice, along with any additional arguments required by the method (like: center and radius for the circle method). And returns the initialized contour.



```python
def initialize_contour(image, initialization_method, *args):
    # convert the image to grayscale
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # initialize the contour based on the chosen method
    if initialization_method == 'circle':
        center = (int(args[0]), int(args[1]))
        radius = int(args[2])
        contour = np.zeros((100, 1, 2), dtype=np.int32)
        cv2.circle(contour, center, radius, (255, 255, 255), 2)
        contour = contour.squeeze()
    elif initialization_method == 'rectangle':
        x, y, width, height = int(args[0]), int(
            args[1]), int(args[2]), int(args[3])
        contour = np.array(
            [[x, y], [x+width, y], [x+width, y+height], [x, y+height]], dtype=np.int32)
    elif initialization_method == 'points':
        points = np.array(args, dtype=np.int32)
        contour = points.reshape((-1, 2))
    else:
        raise ValueError('invalid initialization method')

    return contour
```

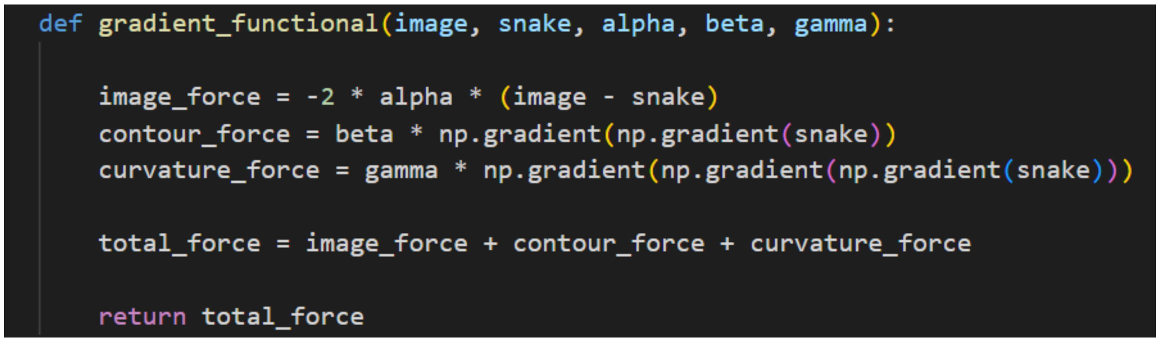
The energy_functional function calculates the energy for the given image and contour. The energy functional is composed of three terms: an image term that measures how well the contour fits the image, a contour term that penalizes deviations from a desired contour shape, and a curvature term that smooths the contour.

This function takes the image, contour, and weights for each term (alpha, beta, gamma), and returns the total energy.



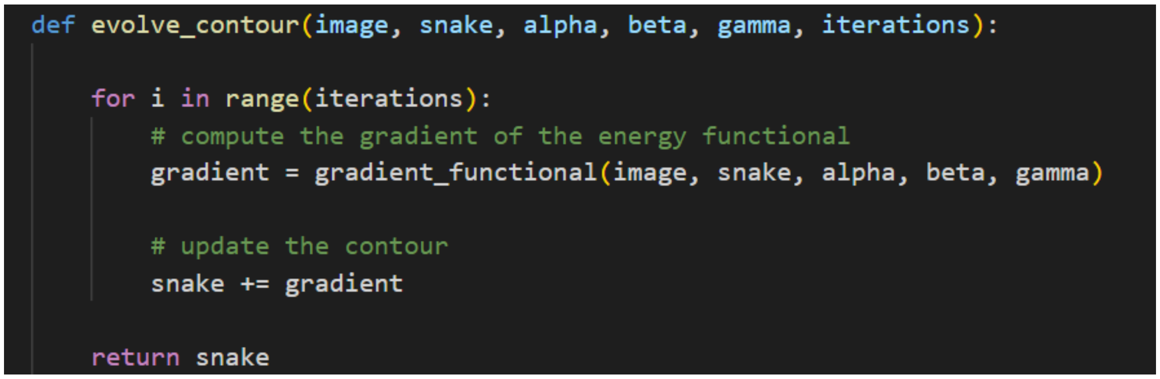
```
def energy_functional(image, snake, alpha, beta, gamma):
    image_energy = alpha * np.sum((image - snake) ** 2)
    contour_energy = beta * np.sum(np.abs(np.gradient(snake)))
    curvature_energy = gamma * np.sum(np.abs(np.gradient(np.gradient(snake))))
    total_energy = image_energy + contour_energy + curvature_energy
    return total_energy
```

The gradient_functional function calculates the gradient of the energy with respect to the contour. This gradient represents the force acting on each point on the contour, and is used to update the contour in the evolve_contour function. This function takes the image, contour, and weights for each term as arguments, and returns the gradient.



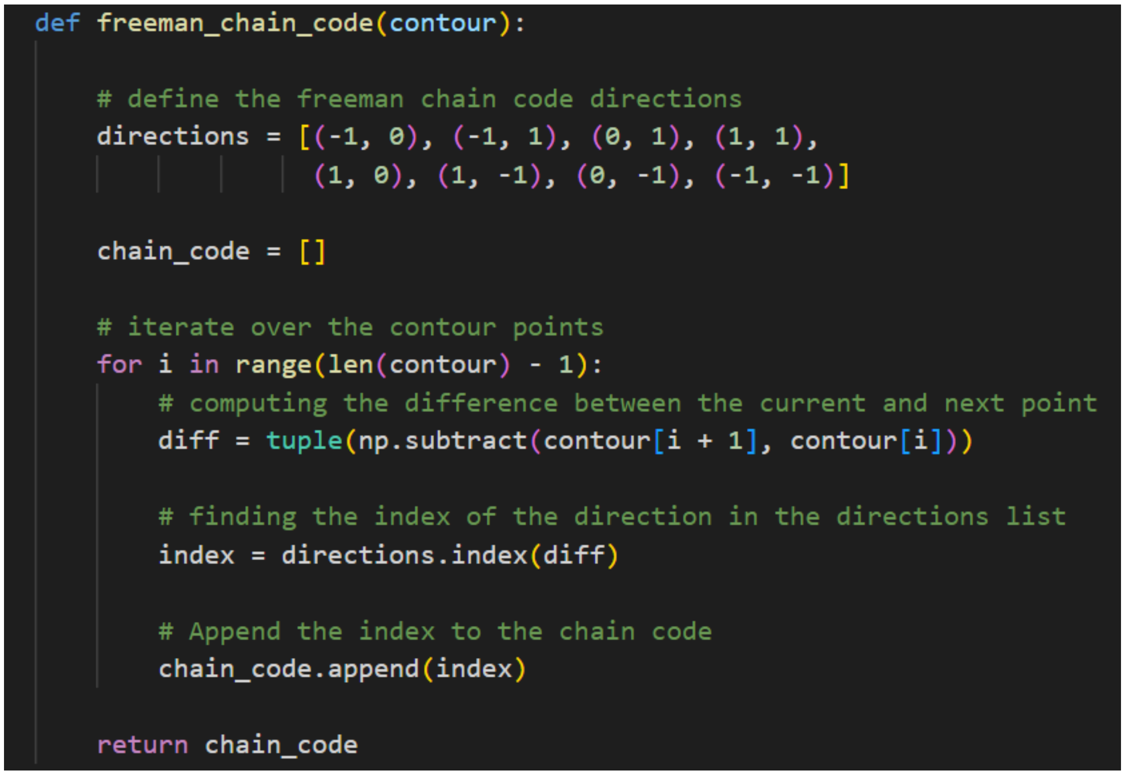
```
def gradient_functional(image, snake, alpha, beta, gamma):

    image_force = -2 * alpha * (image - snake)
    contour_force = beta * np.gradient(np.gradient(snake))
    curvature_force = gamma * np.gradient(np.gradient(np.gradient(snake)))

    total_force = image_force + contour_force + curvature_force

    return total_force
```

The evolve_contour function updates the contour using a greedy algorithm that minimizes the energy functional. This function takes the image, contour, and weights as arguments, along with the number of iterations to perform. It iteratively computes the gradient of the energy functional, and updates the contour by adding this gradient to each point on the contour.



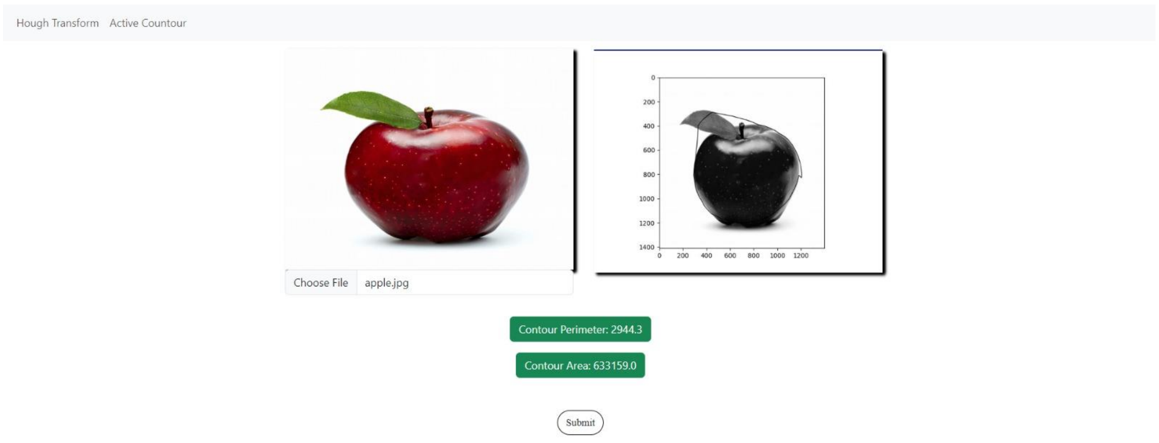
```
def evolve_contour(image, snake, alpha, beta, gamma, iterations):

    for i in range(iterations):
        # compute the gradient of the energy functional
        gradient = gradient_functional(image, snake, alpha, beta, gamma)

        # update the contour
        snake += gradient

    return snake
```

The freeman_chain_code function converts the contour points into chain codes using the Freeman chain code algorithm. Chain codes are a compact representation of contours that can be used for further analysis or storage. This function takes the contour as input and returns the chain code representation.



```python
def freeman_chain_code(contour):

    # define the freeman chain code directions
    directions = [(-1, 0), (-1, 1), (0, 1), (1, 1),
                  (1, 0), (1, -1), (0, -1), (-1, -1)]

    chain_code = []

    # iterate over the contour points
    for i in range(len(contour) - 1):
        # computing the difference between the current and next point
        diff = tuple(np.subtract(contour[i + 1], contour[i]))

        # finding the index of the direction in the directions list
        index = directions.index(diff)

        # Append the index to the chain code
        chain_code.append(index)

    return chain_code
```

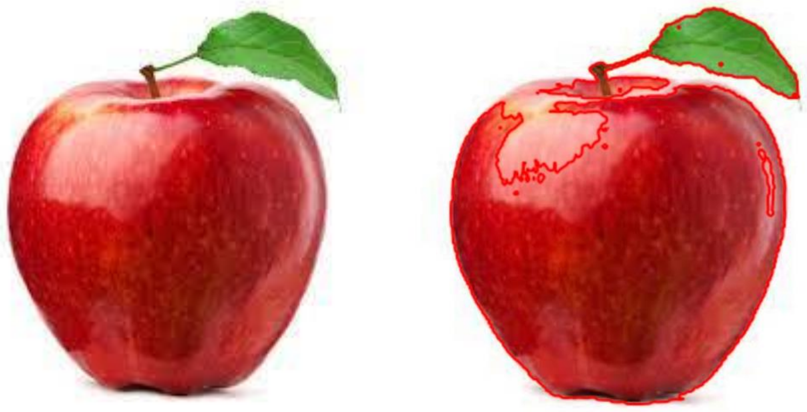
Output



Contour Perimeter: 2944.3

Contour Area: 633159.0

Another Active contour algorithm which made the output look like this:



Chan_Vese function calculates some parameters:

LSF: the level set function, which is used to represent the evolving contour of the segmented object

Img: the input grayscale image to be segmented

Mu: a regularization parameter that controls the smoothness of the contour

Nu: a weighting factor for the length term, which encourages the contour to conform to the edges of the object

Epsilon: a small constant added to avoid division by zero in some calculations

Step: the step size for the iterative update of the level set function

Drc: the Dirac delta function of the level set function at each point in the image, as explained in the previous answer.

Hea: a smoothed version of the Heaviside step function of the level set function, which is used to partition the image into two regions (inside and outside the contour).

Cur: the curvature of the evolving contour, which is used in the length and penalty terms.

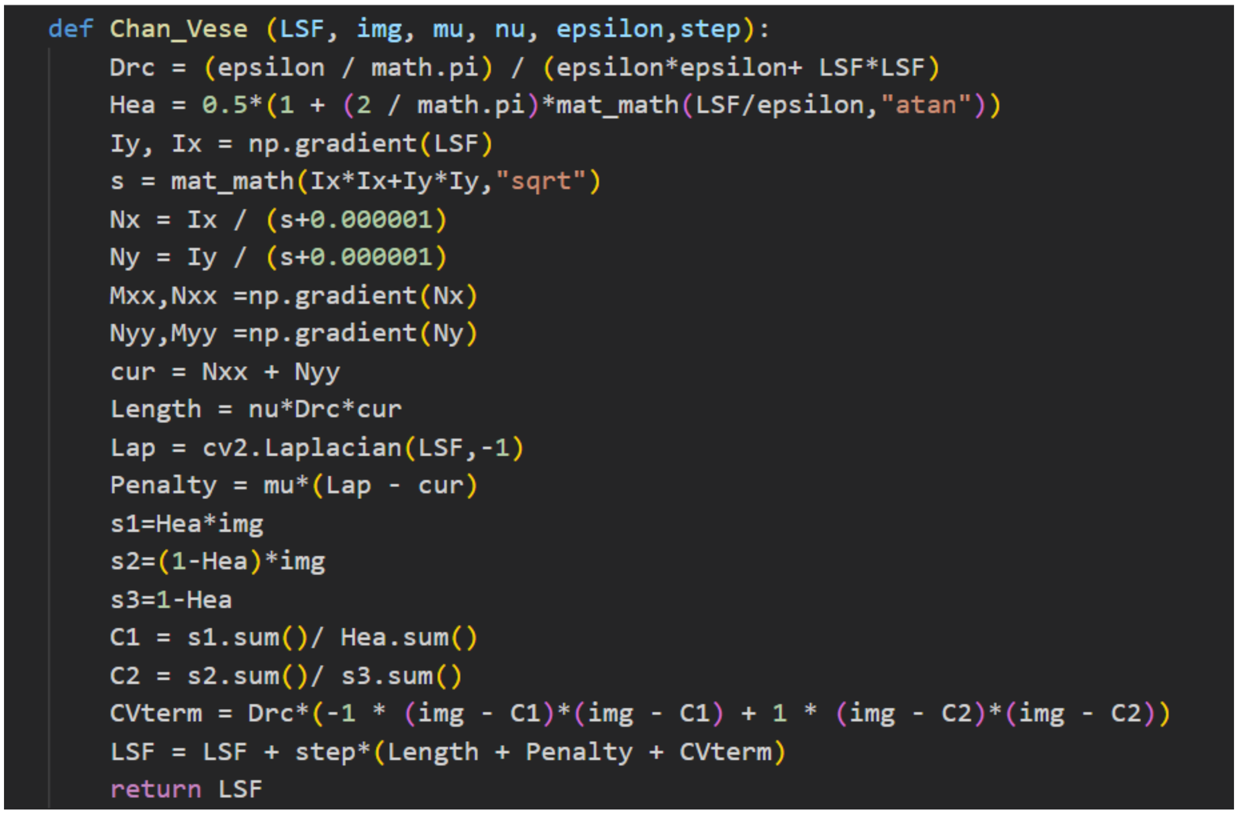
The algorithm works by repeating to update the level set function based on three terms: length, penalty, and data fidelity.

The length term: encourages the contour to follow the edges of the object

The penalty term: penalizes deviations from a smooth contour.

The data fidelity term: encourages the mean pixel values inside and outside the contour to be as different as possible.

Then the level set function is updated using a weighted combination of these terms, as well as a step size that controls the speed of evolution. The updated level set function is returned as the output of the function.



```
def Chan_Vese (LSF, img, mu, nu, epsilon,step):
    Drc = (epsilon / math.pi) / (epsilon*epsilon+ LSF*LSF)
    Hea = 0.5*(1 + (2 / math.pi)*mat_math(LSF/epsilon,"atan"))
    Iy, Ix = np.gradient(LSF)
    s = mat_math(Ix*Ix+Iy*Iy,"sqrt")
    Nx = Ix / (s+0.000001)
    Ny = Iy / (s+0.000001)
    Mxx,Nxx =np.gradient(Nx)
    Nyy,Myy =np.gradient(Ny)
    cur = Nxx + Nyy
    Length = nu*Drc*cur
    Lap = cv2.Laplacian(LSF,-1)
    Penalty = mu*(Lap - cur)
    s1=Hea*img
    s2=(1-Hea)*img
    s3=1-Hea
    C1 = s1.sum()/ Hea.sum()
    C2 = s2.sum()/ s3.sum()
    CVterm = Drc*(-1 * (img - C1)*(img - C1) + 1 * (img - C2)*(img - C2))
    LSF = LSF + step*(Length + Penalty + CVterm)
    return LSF
```