Task 2 Report

Computer Vision

Hough transform, Active Contour

Team Members

Ahmed Abd Elraouf
Zeyad Amr
Mo’men Mohamed
Mazen Tarek
Michael Hany

Table of content

Introduction

It’s a web project that aims to enhance images using various image processing techniques. As mentioned in the previous task report, in addition to hough transform for detecting lines, circles and ellipses in image and applying active contour to image.

Overall, the web application is user-friendly and offers a range of image processing options, from basic filtering to advanced techniques such as hough and active contour.

Technologies

- Frontend: React ts
- Backend: Django
- Processing: Python, opencv, matplotlib, numpy

Features

1. Filters:

The given class Filters contains various image processing algorithms for adding noise, applying smoothing filters and detecting edges to an image.

2. Histogram:

The given class Histogram contains methods for performing image processing tasks such as histogram equalization, normalization, global and local thresholding, and color channel splitting.

3. Frequency:

The given class Frequency contains methods for applying frequency-based image processing techniques, specifically Fourier transforms, high-pass/low-pass filtering and hybrid images.

4. Hough:

The given class Hough is a class that implements the Hough transform algorithm for detecting lines, circles, and ellipses in an image.

The Hough Transform is a technique that allows detecting shapes (such as lines, circles, or ellipses) in an image by looking for patterns in a transformed version of the image. The basic idea is to represent the image in a parameter space that describes the geometric properties of the shapes to be detected, and then look for high-density regions in that space, which correspond to the presence of shapes in the original image.

a. getImg():

- Get the image

b. __get_edges(img, min_edge_threshold, max_edge_threshold):

- The function uses the OpenCV library to perform edge detection on the input image. Specifically, it first converts the input image from BGR color space to grayscale using the cv2.cvtColor() function.
- Then, it applies the Canny edge detection algorithm using the cv2.Canny() function, with the minimum and maximum edge thresholds as inputs. The Canny algorithm is a popular edge detection technique that uses a multi-stage process to detect a wide range of edges in an image.
- Finally, the function returns the resulting edge image as a NumPy array.

c. __superimpose(lines, color):

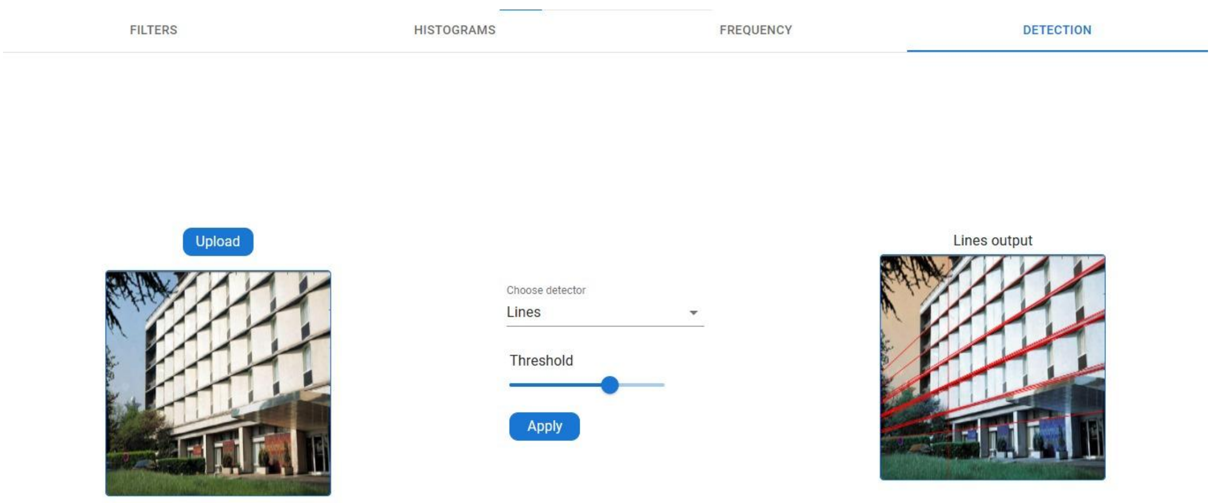
- This function takes in two arguments - lines and color.
- The lines parameter is a list of lines represented in the polar coordinate system, where each line is a tuple containing two values - the distance r and the angle theta.
- The color parameter is a tuple representing the color in RGB format.
- The function first retrieves the image using the getImg() method of the class, and then creates a copy of the image using np.copy().
- It then iterates over each line in the lines list and calculates the two endpoints of the line segment in the Cartesian coordinate system.
- The Cartesian coordinates of the endpoints are calculated using the formula: pt = (x0 + 1000*(-b), y0 + 1000*(a)) and pt2 = (x0 - 1000*(-b), y0 - 1000*(a)).
- Here, x0 and y0 are the coordinates of the closest point on the line to the origin (0,0) in the polar coordinate system. a and b are the sine and cosine values of theta respectively, which are calculated using math.sin() and math.cos().
- Finally, it draws a line segment between the two endpoints using the cv2.line() function of the OpenCV library. The src image is modified in-place by this operation, and the modified image is returned at the end of the function.

d. __hough_lines(src, threshold):

- This function performs the Hough Transform algorithm to detect lines in an input image. The Hough Transform is a technique used to detect straight lines in an image by converting the image space into a parameter space. In this parameter space, each point represents a possible line in the original image.
- The function takes two inputs: src is the binary edge image in which lines are to be detected, and threshold is a value that represents the minimum number of points that must be associated with a line in the parameter space in order for that line to be considered a valid line.
- The first step of the function is to compute the diagonal of the input image using the Pythagorean theorem. This is used to determine the dimensions of the accumulator matrix, which is a two-dimensional array used to store the votes for each possible line in the parameter space. The size of the accumulator matrix is based on the maximum possible distance between a pixel in the image and the origin, which is given by the diagonal.
- Next, the function creates an empty accumulator matrix of size (2 * diagonal, 180), where the first dimension represents the distance of the line from the origin and the second dimension represents the angle of the line with respect to the x-axis.
- The function then iterates over each edge pixel in the input image and calculates the Hough Transform for each edge. This involves looping over all possible angles and distances and incrementing the corresponding cell in the accumulator matrix for each point that lies on a line with that angle and distance. This process generates a matrix with peaks at the positions of the lines in the input image.
- Finally, the function loops over all the cells in the accumulator matrix and checks if the number of votes for a particular line exceeds the threshold value. If so, the function adds the corresponding line to a list of detected lines, where each line is represented as a tuple of (distance, angle).
- The function returns a list of detected lines.

d. detect_lines(threshold, color):

- This function is used to detect lines in an image using the Hough Transform algorithm. It takes two arguments - threshold and color.
- The threshold argument sets the minimum number of pixels required to detect a line. A higher threshold will result in fewer lines being detected, while a lower threshold will result in more lines being detected.
- The color argument sets the color of the lines that are drawn on the image. By default, the color is set to (255, 0, 0), which is red.
- First, the function calls the getImg method of the Image class to retrieve the image. Then it calls the private method __get_edges to detect the edges in the image using the Canny edge detection algorithm with a minimum threshold of 50.
- Next, it calls the private method __hough_lines to detect the lines in the image using the Hough Transform algorithm. This method creates an accumulator matrix with dimensions that cover the range of possible values for r and theta in the Hough space. It then finds the location of the edges in the edge image and calculates the Hough transform for each edge, incrementing the corresponding cells in the accumulator matrix. Finally, it extracts the lines with a number of votes above the specified threshold value.
- Finally, the function calls the private method __superimpose to draw the detected lines on the original image. This method creates a copy of the original image and iterates over the detected lines, drawing each one in the specified color. The resulting image is returned as the output of the detect_lines function.



Choose detector
Lines

Threshold

e. __hough_circles(image, edge_image, r_min, r_max, delta_r, num_thetas, bin_threshold, post_process=True):

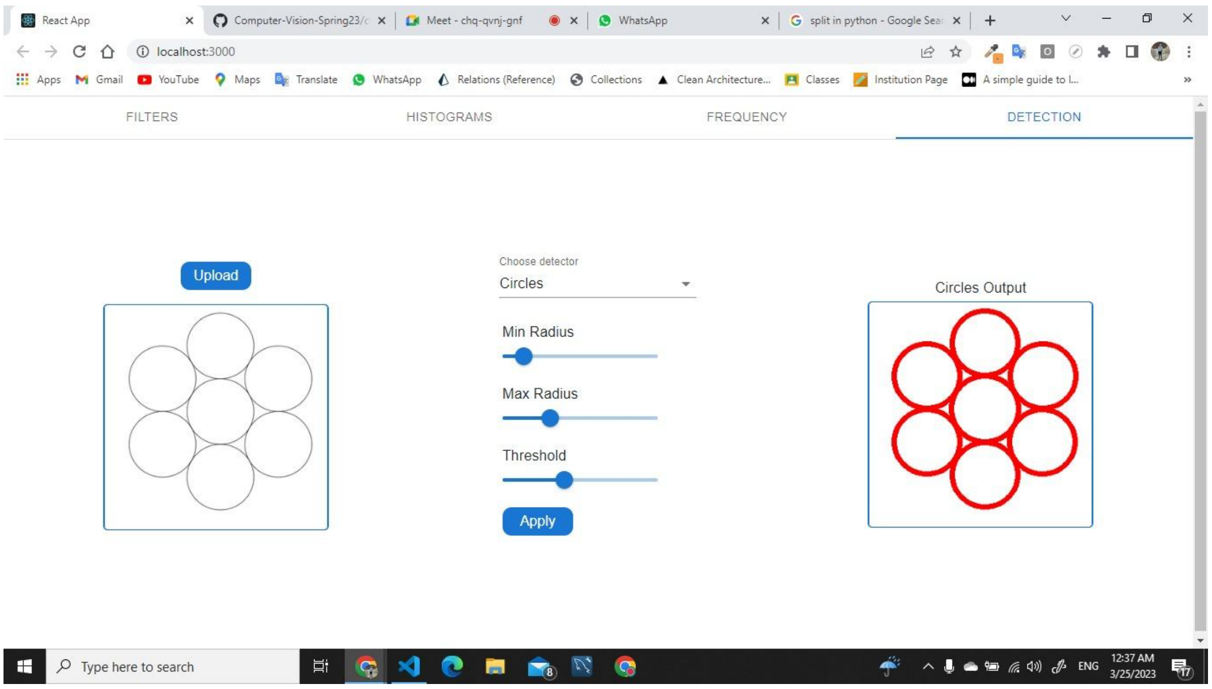
$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1.$$

- This is an implementation of the Hough transform algorithm for detecting circles in an image.
- The function takes as input the original image, an edge-detected image, and various parameters such as the minimum and maximum radius of the circles to be detected, the step size for the radius, the number of angles to use in the Hough transform, and a threshold for how many votes a circle must receive to be considered a valid candidate.
- The function first generates a list of all possible circles in the image based on the given radius range and number of angles.
- It then loops through each edge pixel in the edge image and for each candidate circle, calculates the center of the circle and votes for it in the accumulator.
- The accumulator is a dictionary that keeps track of the number of votes each candidate circle has received.
- Once all edge pixels have been processed, the circles are sorted by their vote count and those with a percentage of votes above the given threshold are shortlisted.
- A post-processing step can be optionally applied to remove duplicate circles that are too close to each other.
- Finally, the shortlisted circles are drawn on the output image and returned.

f. detect_circles(self, min_radius, max_radius, threshold, color):

- This Function detects circles in an input image using the Hough Circle Transform. The method takes four arguments: min_radius, max_radius, threshold, and color.

- min_radius and max_radius set the minimum and maximum radii of circles to be detected, respectively. threshold is a value between 0 and 1 that determines the minimum percentage of votes required for a circle to be considered valid. color is a tuple that specifies the color of the detected circles.
- The method first gets the input image using the getImg() method of the class. It then applies the __get_edges() method to the image to extract the edges. The __get_edges() method uses the Canny Edge Detection algorithm to extract the edges.
- If the edge image is not None, the method proceeds to call the __hough_circles() method with the input image, edge image, minimum radius, maximum radius, delta radius, number of thetas, and the bin threshold as arguments.
- The __hough_circles() method applies the Hough Circle Transform to the edge image to detect circles.
- Finally, it returns the output image which is the input image with the detected circles superimposed on it.



Choose detector
Circles

Min Radius

Max Radius

Threshold

g. __hough_ellipses(self, image, edge_image, a_min, a_max, delta_a, b_min, b_max, delta_b, num_thetas, bin_threshold, post_process=True):

$$\frac{(x\cos\alpha + y\sin\alpha)^2}{a^2} + \frac{(x\sin\alpha - y\cos\alpha)^2}{b^2} = 1$$

- This is a private method called __hough_ellipses which takes an image, an edge image (binary image with detected edges), the minimum and maximum values for the semi-major and semi-minor axes (a and b), their respective increments (delta_a and delta_b), the number of angles to consider (num_thetas), a bin threshold value (bin_threshold) and a boolean flag to perform post-processing (post_process).
- The method uses the Hough transform to detect ellipses in the edge image.
- The accumulator is a dictionary where the key is a tuple of ellipse parameters (x_center, y_center, a, b, t) and the value is the number of votes for that ellipse.
- The loop over edge pixels is used to vote for all possible ellipses that pass through that point, using the ranges of a, b, and t specified as arguments.
- The parameter t is the angle at which the major axis is oriented.
- After all edge pixels have voted, the method sorts the accumulator by the number of votes in descending order and loops through each candidate ellipse.
- The current_vote_percentage is the percentage of angles that voted for that ellipse.
- If it is greater than or equal to the bin threshold, the method checks if there are any previously found ellipses that overlap with this one. If there is no overlap, the ellipse is added to the list of ellipses found.
- If post-processing is enabled, the method applies a pixel threshold value to filter out very similar ellipses and draws the shortlisted ellipses on the output image in red.
- Finally, the method returns the output image with the detected ellipses drawn on it.

h. def detect_ellipses(self, min_radius1, max_radius1, min_radius2, max_radius2, threshold, color):

- This is a method that uses the Hough transform to detect ellipses in an input image.
- The method takes several arguments:
    - min_radius1 and max_radius1: the minimum and maximum values for the first radius of the ellipse.
    - min_radius2 and max_radius2: the minimum and maximum values for the second radius of the ellipse.
    - threshold: the minimum percentage of votes required for an ellipse to be considered a valid detection.
    - color: the color of the ellipses to be drawn on the output image.
- The method first calls a private method __get_edges() to extract edges from the input image using the Canny edge detection algorithm.
- Then, it calls another private method __hough_ellipses() to perform the Hough transform on the edge image to detect ellipses.
- The __hough_ellipses() method takes the edge image, the ranges of the two radii, the step sizes for the two radii, the number of angles to consider, and the bin threshold as arguments.
- It initializes an accumulator dictionary and loops over all edge pixels, voting for all possible ellipses that pass through the current pixel.
- It then finds ellipses with enough votes to be considered a valid detection, and post-processes the results to remove similar detections.
- Finally, it draws the shortlisted ellipses on the output image and returns it.

5. Active Contour:

The given class Active Contour is a class that provides code that implements an Active Contour model, also known as Snake model, which is a framework for performing image segmentation tasks. The model consists of a set of contour points that are iteratively adjusted to align with the boundaries of the object to be segmented.

a. points_distance(x1, y1, x2, y2):

- This function is used for calculating the distance between any two points in a two-dimensional space.

b. circle_contour(center, radius, numberOfPoints, x_coordinates, y_coordinates):

- This is a function that generates the contour points of a circle given its center, radius, and the desired number of points on the contour. The function takes in five parameters:
  - center: a tuple of the (x,y) coordinates of the center of the circle
  - radius: the radius of the circle
  - numberOfPoints: the number of points on the contour
  - x_coordinates: an array to store the x-coordinates of the contour points
  - y_coordinates: an array to store the y-coordinates of the contour points
- The function first computes the angular resolution of the contour using the formula resolution = 2 * pi / numberOfPoints. Then, it loops over numberOfPoints to compute the x and y coordinates of each point on the contour using the formulae x = center[0] + radius * cos(angle) and y = center[1] - radius * sin(angle), where angle is the angle of the current point, computed as i * resolution.
- The x and y coordinates of each contour point are rounded to the nearest integer using the round() function and stored in the output arrays x_coordinates and y_coordinates.
- Finally, the function returns the two arrays containing the x and y coordinates of the contour points.

c. draw_contour(Image, numberOfPoints, x_coordinates, y_coordinates):

- This function draw_contour() takes four arguments: Image, numberOfPoints, x_coordinates, and y_coordinates. The Image argument is the image on which the contour will be drawn.
- numberOfPoints, x_coordinates, and y_coordinates are the output of the circle_contour() function, which returns the x and y coordinates of the contour points.
- The function then creates a variable img and sets its initial value to 0. It then loops over each point in the contour using a for loop, and for each point, it calculates the coordinates of the next point using modular arithmetic.
- It then uses the OpenCV line() function to draw a line between the current point and the next point, with a color of (0, 255, 0) and a line thickness of 2.
- Finally, the function converts the image from BGR format to RGB format using the cvtColor() function from OpenCV, and returns the resulting image.
- The resulting image should show the contour drawn on top of the original image.

d. contour_area(numberOfPoints, x_coordinates, y_coordinates):

- This function takes in the number of points, x and y coordinates of a contour and calculates its area using the shoelace formula.
- The shoelace formula is a mathematical algorithm that calculates the area of a polygon given the coordinates of its vertices.
- The formula gets its name from the way it works, which involves multiplying pairs of coordinates and then adding them up in a pattern that resembles lacing up a shoe.
- The function first initializes the area variable to zero.
- It then uses a for loop to iterate over each vertex of the contour.

- The formula requires the use of two vertices at a time, so the function initializes a variable j to the index of the last vertex in the array.
- The formula multiplies the x coordinate of the current vertex and the y coordinate of the previous vertex and subtracts the x coordinate of the previous vertex and the y coordinate of the current vertex. The result is added to the area variable.
- After iterating over all vertices, the function returns the absolute value of area divided by 2, which gives the area of the polygon.

e. contour_perimeter(x_points, y_points, points_n):

- The function contour_perimeter takes in the x and y coordinates of a set of points that define a contour, as well as the number of points. It then iterates over each point, calculates the distance to the next point in the sequence, and adds it to a running sum of distances.
- Finally, the function returns the sum of distances, which corresponds to the perimeter of the contour.
- Here's a step-by-step breakdown of what the function does:
    - Initialize a variable distance_sum to zero.
    - For each point i in the range 0 to points_n - 1, do the following:
        - a. Compute the index of the next point in the sequence using next_point = i + 1, unless i is the last point in the sequence, in which case next_point is set to zero.
        - b. Compute the distance between the current point (x_points[i], y_points[i]) and the next point (x_points[next_point], y_points[next_point]) using the points_distance function.
        - c. Add the computed distance to distance_sum.
    - Return distance_sum.
- In summary, the contour_perimeter function calculates the length of a path formed by connecting a sequence of points.

f. window_neighbours(size):

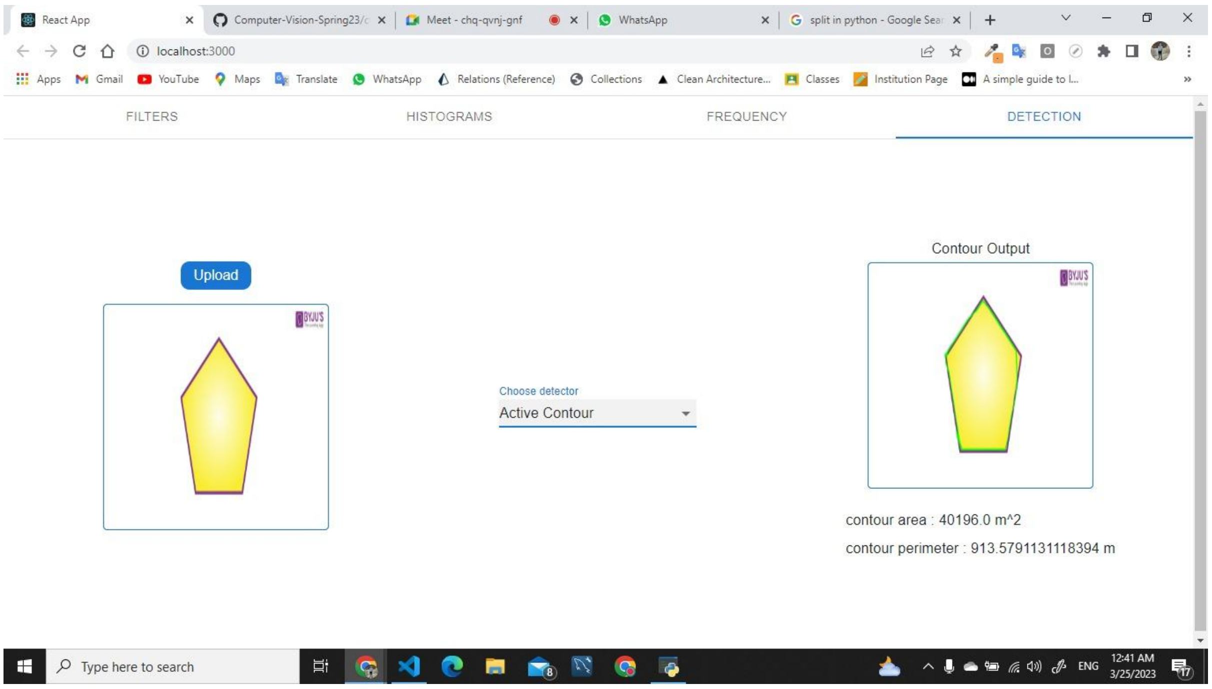
- The function window_neighbours(size) takes an input parameter size, which is the size of the square window.
- It returns a list of coordinates representing all the neighboring points in a square window around a central point.
- The function initializes two empty lists, window and point.
- It then loops through each pixel in the square window by iterating over a range of (-size // 2, size // 2 + 1) for both x and y coordinates. It creates a new list point to store the current (x, y) coordinate of the current pixel.
- This point is then appended to the window list, which contains all neighboring points in the window.
- For example, if the input size is 3, the window list will contain the following points:
    - [[-1, -1], [-1, 0], [-1, 1], [0, -1], [0, 0], [0, 1], [1, -1], [1, 0], [1, 1]]
- This can be used, for example, to implement a convolution operation, where the window is centered around each pixel in an image and the neighboring pixels are multiplied with a filter and summed to compute the output value for the central pixel.
- Finally, the function returns the list of points that represents the window.

g. convert(list):

- The convert() function takes a list of integers as an input argument, and returns an integer formed by concatenating all the integers in the list.
- Here's how the function works:
    - First, it converts each integer element in the list to a string using a list comprehension: s = [str(i) for i in list]
    - Then it concatenates all the string elements in s using the join() method and converts it to an integer: res = int("".join(s))
    - Finally, it returns the integer res as the output.
- For example, if we call convert([1, 2, 3, 4]), the function will return the integer 1234.

h. greedy_contour(source, iterations, alpha, beta, gamma, x_points, y_points, points_n, window_size, plot):

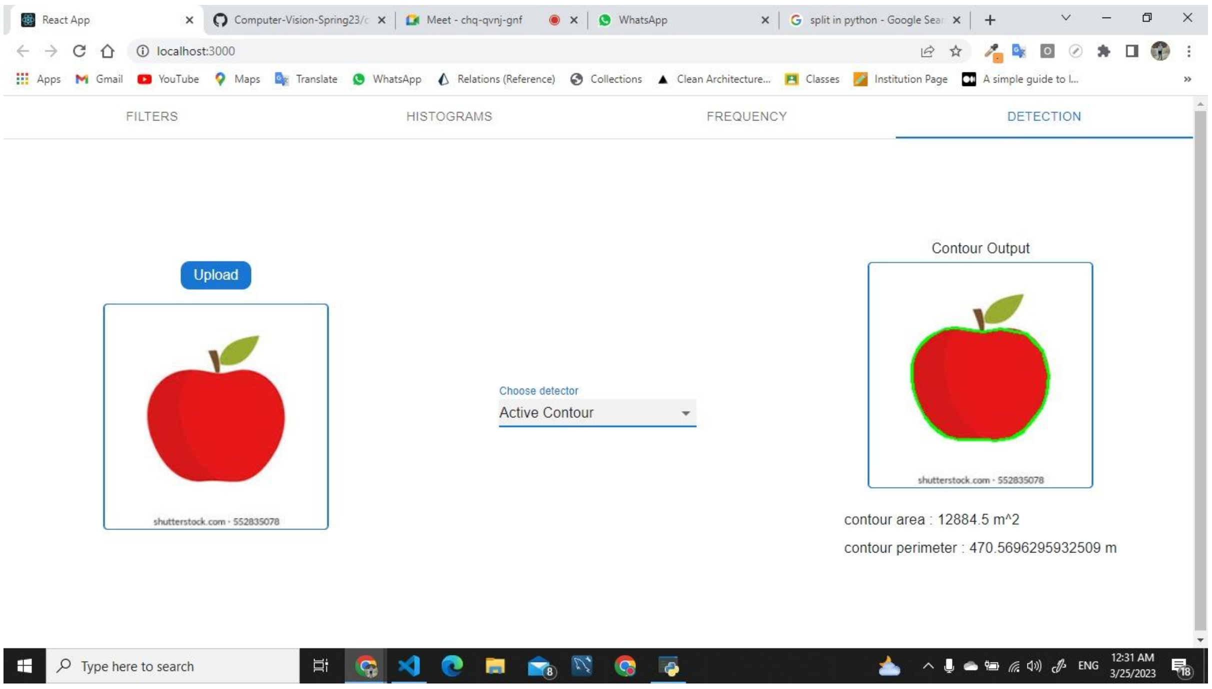
- This function implements the greedy algorithm for active contours, which is a method for image segmentation.
- It takes as input an image (source), the number of iterations (iterations), and various parameters (alpha, beta, gamma) that are used to compute the internal energy of the contour.
- It also takes as input an initial set of points (x_points, y_points) that define the initial contour, the number of points in the contour (points_n), the size of the neighborhood window (window_size), and a boolean flag (plot) that controls whether or not to display the contour during the algorithm.
- The function first computes the external energy of the image using the Sobel operator, which is used to attract the contour to edges in the image.
- It then initializes some variables and enters a loop that iterates until either the number of iterations is exceeded or the number of movements of the contour is less than a threshold value (threshold).
- In each iteration of the loop, the function considers each point in the contour and evaluates the energy of the contour if the point were to be moved to each of the neighboring points in the neighborhood window.
- It then selects the neighboring point that results in the minimum energy, and moves the point to that position.
- If the point is moved, the function increments a counter (movements).
- The function also computes the internal energy of the contour, which is a measure of how smooth the contour is.
- This is done using the internal_energy function, which takes as input the current set of points (current_x, current_y) and the various parameters (alpha, beta, gamma).
- The internal energy is used to prevent the contour from becoming too jagged.
- The function returns the final set of points that define the contour after the algorithm has converged.



Choose detector
Active Contour

contour area : 40196.0 m^2

contour perimeter : 913.5791131118394 m



Choose detector
Active Contour

contour area : 12884.5 m^2

contour perimeter : 470.5696295932509 m

i. internal_energy(x_points, y_points, points_n, alpha, beta):

- This code defines a function internal_energy which takes as input x_points and y_points which represent the x and y coordinates of the points of a contour, points_n which is the number of points in the contour, alpha and beta which are weighting factors for the two types of internal energy computed, respectively.
- The function first computes the average distance between adjacent points in the contour, which is used to compute the deviation of each distance from the average, which is squared and summed over all points to compute the "contour smoothness" energy.
- Next, the function computes the average angle between adjacent line segments in the contour, which is used to compute the deviation of each angle from the average, which is squared and summed over all points to compute the "curvature" energy.
- Finally, the function returns the sum of these two energy terms, weighted by alpha and beta, respectively.
- This represents the total internal energy of the contour.

j. external_energy(source):

- The external_energy function takes an input image source and applies Gaussian filtering, edge detection using Canny algorithm, and then returns the resulting edges as a binary image.
- The function first applies Gaussian filtering on the input image using the cv2.GaussianBlur function with a kernel size of (3, 3) and standard deviation of 0.
- This helps to reduce noise in the image.
- Then, it converts the filtered image to grayscale using the cv2.cvtColor function with the COLOR_BGR2GRAY flag.
- Next, it applies edge detection on the grayscale image using the Canny algorithm with low threshold of 0 and high threshold of 255, and stores the resulting binary image in edges.
- Finally, the function writes the resulting edges to a file named 'edges.jpg' and returns the binary image edges.

k. normaliseToRotation(chain_code):

- The function normaliseToRotation takes a list of chain codes as input and returns a modified version of the list where the chain codes have been normalized to rotation.
- The function first creates a copy of the input chain code list using the copy() method.
- It then iterates through the list using a for loop. Inside the loop, it calculates the difference between the current chain code symbol and the next symbol by subtracting the former from the latter.
- The result is then modulo 8 to normalize it.
- The normalized symbol is then assigned to the next position in the copied chain code list using the index operator.
- Finally, the function returns the modified chain code list.

l. normaliseToStartingPoint(chaincode):

- This function takes a chain code as input and returns a normalized version of it, where the starting point of the contour is always the first element in the chain code.
- To do this, the function first finds the smallest element in the chain code by sorting it, and then finds all the occurrences of that smallest element using the getAllOccurences function (which is not shown here).
- Next, it rotates the chain code so that each occurrence of the smallest element becomes the first element in the list.
- It then converts each rotated chain code into an integer using the convert function (which is not shown here).
- This integer is calculated by concatenating the elements of the chain code and treating them as a base-10 number.
- The rotated chain code that yields the smallest integer value is chosen as the normalized chain code, and returned by the function.

m. parametersToAppend(mulByMn, mulByDx, mulByDy, mn, dx, dy):

- This function takes in six parameters: mulByMn, mulByDx, mulByDy, mn, dx, and dy.
- mulByMn, mulByDx, and mulByDy are values that will be appended to a list multiple times, while mn, dx, and dy are integer values used to determine how many times to append each value.
- The function starts by creating an empty list codeList.
- It then appends the value of mulByMn to the codeList mn number of times, effectively appending mulByMn to the list mn times.
- Next, it subtracts mn from the absolute value of dx and dy. If either value is negative, it takes the absolute value first.
- Finally, it appends mulByDx to the codeList dx number of times, and appends mulByDy to the codeList dy number of times.
- The function then returns the codeList which contains the appended values according to the given parameters.

n. getCodeBetweenTwoPoints(x1, y1, x2, y2):

- This function generates a chain code between two points (x1, y1) and (x2, y2) using a specific set of rules for encoding the directions between the points.
- The function first calculates the differences between the x and y coordinates of the two points, which are used to determine the direction between them.
- It then calculates the minimum absolute difference between dx and dy, which is used to determine the number of steps required to move in each direction to reach the endpoint.
- Based on the direction between the two points, the function selects a set of parameters to append to the chain code list using the parametersToAppend() function.
- The parameter values are determined by a set of rules, which are based on the direction between the two points. The rules assign specific values for the multipliers of mn, dx, and dy, which are used to generate the chain code.
- The resulting chain code is returned as a list.

o. getChainCode(contourX, contourY):

- This function takes two lists of x and y coordinates that represent the contour of an object in an image.
- It then iterates over the points in the contour, and for each point, it calls the getCodeBetweenTwoPoints function to calculate the chain code for the line segment between that point and the next point in the contour.
- The chain codes are appended to a list chaincode.
- After calculating the chain codes for all the line segments in the contour, the function normalizes the chain code twice: first to rotation, using the normaliseToRotation function, and then to starting point, using the normaliseToStartingPoint function.
- The three chain codes (original, normalized to rotation, and normalized to starting point) are then returned as a tuple.