Assignment 2

Computer Vision

DR. Ahmed M. Badawi
T.A. Laila Abbas
T.A. Peter Salah

STUDENT NAME	SEC	BN.
GHOFRAN MOHAMMED	2	8
KAREMAN YASER	2	9
MAYAR FAYEZ	2	42
NADA AHMED	2	46
NAIRA YOUSSEF	2	48

Description:

A small web application based app developed with python and streamlit, to apply different image processing techniques.

Requirements:

- Python 3.
- Streamlit 1.13.0
- Numpy 1.23.4
- Matplotlib 3.6.2

Running command:

Streamlit run server.py

- The UI contains two main tabs Hough Transformations, Active Contour

Tab1:

- Line Detection
- Circle Detection
- Ellipse Detection

❖ Line Detection:

Algorithm

First: We read image using cv2 library then, apply canny edge detection also using cv2 library.

Second: Call line detection function with parameters uploaded image, edged image and threshold from user, we want to determine the height & width of edged image to determine the diagonal then, determine "theta" resolution and "r" resolution.

Then, find all non-zeros edges pixels and cycle through them and cycle through theta to calculate "r" resolution to determine hough accumulator.

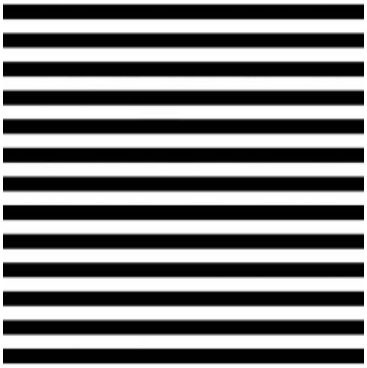
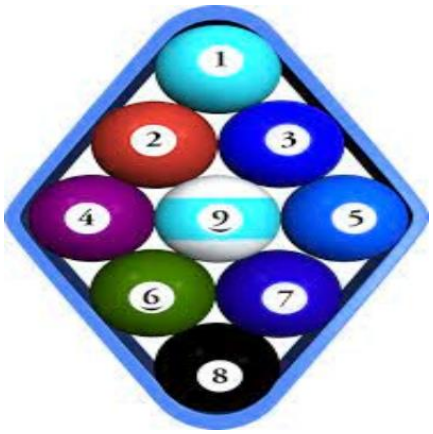
Last step: Cycle through accumulator to determine line points then, using cv2.line library to superimpose the detected line on the input image.

```
def line_detection(image, edge_image, threshold):
    edge_height, edge_width = edge_image.shape[:2]
    d = np.sqrt(np.square(edge_height) + np.square(edge_width)) #Diagonal
    thetas = np.arange(0, 180, 1) #theta:The resolution of the parameter θ in radians.
    rhos = np.arange(-d, d, 1)    #rho: The resolution of the parameter r in pixels.
    cos_thetas = np.cos(np.deg2rad(thetas))
    sin_thetas = np.sin(np.deg2rad(thetas))
    accumulator = np.zeros((len(rhos), len(thetas)))    #Hough Accumulator
    y_idxs, x_idxs = np.nonzero(edge_image) # find all edge (nonzero) pixel indexes
    for i in range(len(x_idxs)):            # cycle through edge points
        x = x_idxs[i]
        y = y_idxs[i]
        for theta_idx in range(len(thetas)):     # cycle through thetas and calc rho
            rho = int((x * cos_thetas[theta_idx] + y * sin_thetas[theta_idx]) + d )
            accumulator[rho, theta_idx] += 1

    for y in range(accumulator.shape[0]):
        for x in range(accumulator.shape[1]):
            if accumulator[y][x] > threshold:
                rho = rhos[y]
                theta = thetas[x]
                a = np.cos(np.deg2rad(theta))
                b = np.sin(np.deg2rad(theta))
```

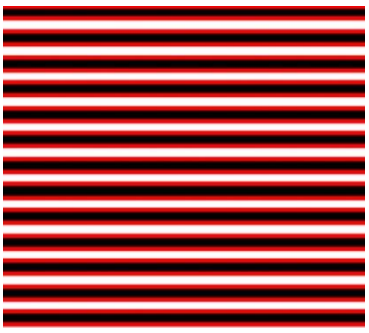
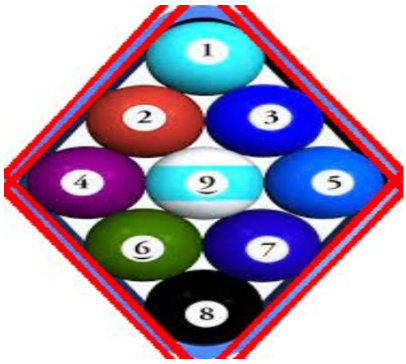
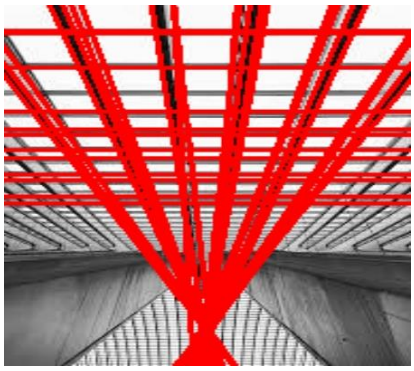
```
                x0 = (a * rho)   # x0 stores the value rcos(theta)
                y0 = (b * rho)   # y0 stores the value rsin(theta)
                x1 = int(x0 + 1000 * (-b))
                y1 = int(y0 + 1000 * (a))
                x2 = int(x0 - 1000 * (-b))
                y2 = int(y0 - 1000 * (a))
                out_img = cv2.line(image, (x1,y1), (x2,y2), (255, 0, 0), 2)
    return out_img
```

Parameters that user can enter them:

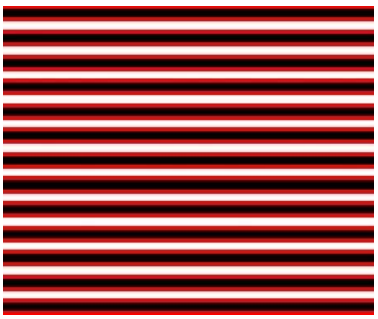
- **Line Detection Threshold**
- **Canny Edge Lower Threshold**
- **Canny Edge Higher Threshold**


Algorithm Output:

CV2 Output:



❖ Circle Detection:

Algorithm

First: We read image using cv2 library then, apply canny edge detection also using cv2 library.

Second: Call circle detection function with parameters uploaded image, edged image, min radius and max radius of detected circle and delta r change from min radius and max radius. Based on defined number of angles we calculate angles from 0 to 360 to build parametric equation of circle (x = x_center + r * cos(t) and y = y_center + r * sin(t)), then append it in array of candidate circles. loop on image height and width of an image to find an edge pixel which pass through any of candidate circle array then build accumulator from defined current circles in image.

Last step: Sort accumulator in addition to some post process to exclude too close circles and duplicated ones. Finally using cv2.cirlce library to superimpose the detected circles on the input image.

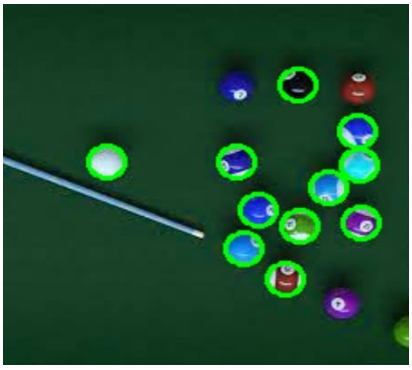
```
def circle_detection(input_img,edge_image,r_min,r_max,delta_r):
    num_thetas = 100
    bin_threshold = 0.4
    post_process = True
    img_height, img_width = edge_image.shape[:2]
    dtheta = int(360 / num_thetas)
    thetas = np.arange(0, 360, step=dtheta)
    rs = np.arange(r_min, r_max, step=delta_r)
    cos_thetas = np.cos(np.deg2rad(thetas))
    sin_thetas = np.sin(np.deg2rad(thetas))
    circle_candidates = []

    for r in rs:
        for t in range(num_thetas):
          circle_candidates.append((r, int(r * cos_thetas[t]), int(r * sin_thetas[t])))
    accumulator = defaultdict(int)

    for y in range(img_height):
        for x in range(img_width):
            if edge_image[y][x] != 0:
                for r, rcos_t, rsin_t in circle_candidates:
                    x_center = x - rcos_t
                    y_center = y - rsin_t
                    accumulator[(x_center, y_center, r)] += 1 #vote for current
candidate
    output_img = input_img.copy()
    out_circles = []
```

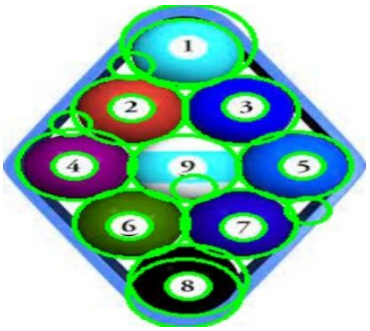
```
    for candidate_circle, votes in sorted(accumulator.items(), key=lambda i: -i[1]):
     x, y, r = candidate_circle
     current_vote_percentage = votes / num_thetas
     if current_vote_percentage >= bin_threshold:
          out_circles.append((x, y, r, current_vote_percentage))
    if post_process :
        pixel_threshold = 5
        postprocess_circles = []
        for x, y, r, v in out_circles:
            if all(abs(x - xc) > pixel_threshold or abs(y - yc) > pixel_threshold or
abs(r - rc) > pixel_threshold for xc, yc, rc, v in postprocess_circles):
                postprocess_circles.append((x, y, r, v))
        out_circles = postprocess_circles

    for x, y, r, v in out_circles:
        output_img = cv2.circle(output_img, (x,y), r, (0,255,0), 2)
    return output_img
```

Parameters that user can choose them:

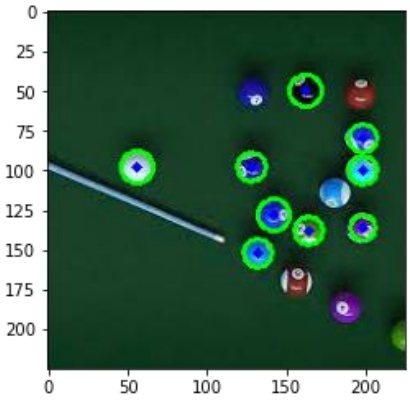
- **Canny Edge Lower Threshold**
- **Canny Edge Higher Threshold**
- **Minimum Circle Radius**
- **Maximum Circle Radius**
- **Change between min radius and max radius.**

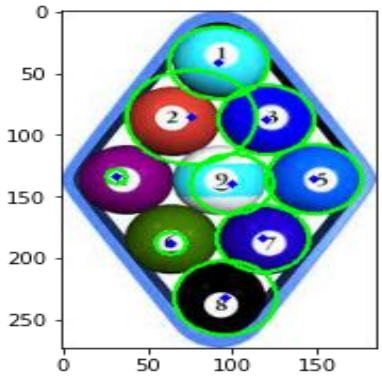
Algorithm output:





CV2 output:





❖ Ellipse Detection: (Just a Try)

Algorithm

```
def Ellipse_detection(image_rgb,edge_image, min2a = 10, min_votes = 10):

 w, h = edge_image.shape
 EdgePixel = []
 output_img = image_rgb.copy()
    # Step 1 - Save all Edge-Pixels in an array
 for x in range(w):
        for y in range(h):
            if edge_image[x, y] != 0:
             EdgePixel.append([x,y])
 acc = np.zeros(int(max(w, h)/2))
 for i in range(len(EdgePixel)-1):
     for j in range(i+2,len(EdgePixel)):
         x1= EdgePixel[i][0]
         y1 = EdgePixel[i][1]
         x2=EdgePixel[j][0]
         y2=EdgePixel[j][1]
         d12=np.linalg.norm(np.array([x1,y1])-np.array([x2,y2]))
         acc = acc * 0
         if  x1 - x2 > min2a and d12 > min2a :
            #  center
            x0 = (x1 + x2)/2
            y0 = (y1 + y2)/2
            # Half-length of the major axis.
            a = d12/2
            # Orientation.
            if x2!=x1:
             alpha = math.atan((y2 - y1)/(x2 - x1))
```

```
                else:
                    alpha=0
                # Distances between the two points and the center.
                d01 = np.linalg.norm(np.array([x1, y1]) - np.array([x0, y0]))
                d02 = np.linalg.norm(np.array([x2, y2]) - np.array([x0, y0]))
                for k in range(len(EdgePixel)):
                    if k == i and k == j:
                        continue

                    x3= EdgePixel[k][0]
                    y3 = EdgePixel[k][1]
                    d03 = np.linalg.norm(np.array([x3, y3]) - np.array([x0, y0]))
                    if  d03 >= a:
                        continue
                    # Distance f
                    f= np.linalg.norm(np.array([x3, y3]) - np.array([x2, y2]))
                    # Estimating the half-length of the minor axis.
                    cos2_tau = pow(((pow(a, 2)+ pow(d03, 2) - pow(f, 2)) /
                                    (2 * a * d03)), 2)
                    sin2_tau = 1 - cos2_tau
                    b = math.sqrt(abs((pow(a, 2) * pow(d03, 2) * sin2_tau) /
                                  (pow(a, 2) - pow(d03, 2) * cos2_tau)))
                    # Changing the score of the accumulator, if b is a valid value.
                    #  NOTE: the accumulator's length gives us the biggest expected
                    #  value for b, which means, in this current implementation,
                    #  we wouldn't detect ellipses whose half of minor axis is
                    #  greater than the image's size (look at the acc's declaration).
                    if b > 0 and b <= len(acc):
                        acc[int(b)]= acc[int(b)]+1

                sv=np.argmax(acc)
                si=np.max(acc)
                if sv > min_votes:
                    #  Ellipse detected!
                    #  The index si gives us the best b value.
                     parameters = [x0, y0, a, si, alpha]
                    #  return parameters
                     center_coordinates=(int(x0),int(y0))
                     axesLength=(int(2*a),int(2*si))
                     output_img=cv2.ellipse(output_img, center_coordinates,
axesLength,alpha,(0,255,0), 2)
                     return output_img
                else:
                    return output_img
```

Edge_image is the inputs image. Images composed of only edges
MIN2A is the minimum length of the major axis (default 10).

MIN_VOTES is the minimum number of votes on a "b" value (half-length of the minor axis) to validate the existence of an ellipse.

First: We find all nonzero pixels of the image, possible ellipse's pixels, then we initialize an accumulator for the minor axis' half-length. The indexes correspond to the possible b values.

Second: We get two pixels p1(x1,y1) & p2(x2,y2) and get the distance between them and check that x1 - x2 && d12 > min2a if true we calculate the center, half-length of the major axis and orientation of ellipse

Third: We calculate the Distances between the two points and the center and we get a third point, it must be a different point we get the distance between the third point and the center

Last step: Estimating the half-length of the minor axis, Changing the score of the accumulator, if b is a valid value, the accumulator's length gives us the biggest expected value for b which means we wouldn't detect ellipses whose half of minor axis is greater than the image's size. Lastly we take the highest score and use the parameters to draw the ellipse over the input RGB image.

Tab2:

- ❖ Active Contour (Snake)

Algorithm

Snake theory:

A snake (a.k.a. active contour) is a flexible 2D line, which is moved around the image to minimize an energy functional. The line is parametrized by through a variable p that goes from 0 to 1. Two functions of this variable, $x(p)$ and $y(p)$, define the coordinates of the points along the line. For simplicity, we use the vector $\vec{s}(p) = (x(p), y(p))^T$. More often than not, the snake is implemented as a closed contour, because it is most often used to segment an object. This is enforced setting $\vec{s}(0) = \vec{s}(1)$.

The energy functional that we want to minimize is defined as:

$$E = \int E_{\text{int}}(\vec{s}(p)) + E_{\text{ext}}(\vec{s}(p))\,\mathrm{d}p \ .$$

E_{int} is an internal energy, meant to force the snake to be small and smooth. It avoids obviously wrong solutions. E_{ext} is an external energy, the one responsible for the snake finding the edges of an object in the image. E_{ext} is therefore defined on the image. A common external energy is the inverse of the gradient magnitude: low energies at the location of the edges, higher energies everywhere else. This energy definition comes from the original paper by Kass, Witkin and Terzopoulos (1988).

Let's discuss the internal energy first. It is defined as

$$E_{\text{int}}(\vec{s}(p)) = \frac{1}{2}\left\{\alpha(p)|\vec{s}'(p)|^2 + \beta(p)|\vec{s}''(p)|^2\right\} \ .$$

This looks rather complicated, but it need not be. For starters, α and β are usually constant (the method has enough parameters as it is, there is no point in making two of the parameters different for every point along the snake!). So what we have left is the magnitude of the first derivative, which is larger for longer snakes, and the magnitude of the second derivative, which is larger for sharper bends. The first part keeps the snake short, the second part keeps it straight. The two parameters, α and β, define the relative importance of these two terms.

To minimize this energy function we use the Euler-Lagrange equation. After quite some work we end up with the following equation:

$$\alpha\vec{s}''(p) - \beta\vec{s}''''(p) - \nabla E_{\text{ext}}(\vec{s}(p)) = 0 \ .$$

(By the way, when you derive this is when you understand why the original definition of the internal energy has that ½ in it!) ∇E_{ext}, the gradient of the external energy, is a force. We call this the external force, $\vec{F}_{\text{ext}}$, and this is where most of the improvements to snakes were made. I won't discuss them here.

Solving the equation above is accomplished with the gradient descent method: you convert the snake $\vec{s}$ into a function of time, and replace the 0 with the partial derivative of $\vec{s}$ to time. You can see the reason for this if you assume that, after a long time when the snake has converged to a minimum, it's derivative to time will be zero, and so the equation above is satisfied.

$$\frac{\partial\vec{s}(p)}{\partial t} = \alpha\vec{s}''(p,t) - \beta\vec{s}''''(p,t) + \vec{F}_{\text{ext}}(\vec{s}(p,t)) \ .$$

Next we need to discretize the parameter p so we can actually implement this numerically. So we set $p = i/N$, with i going from 0 to $N-1$. Because the snake is periodic, $\vec{s}[N] = \vec{s}[0]$, $\vec{s}[N+1] = \vec{s}[1]$, etc. (Note I'm using the square brackets to denote the discrete indexing, as opposed to the round brackets for continuous-domain functions.) This is also the time when we split the vector $\vec{s}$ into its two components x and y. The two components of the external force $\vec{F}_{\text{ext}}$ are f_x and f_y. We use the finite difference approximation to the spatial derivatives:

$$x''[i] = x[i-1] - 2x[i] + x[i+1]$$

and

$$x''''[i] = x[i-2] - 4x[i-1] + 6x[i] - 4x[i+1] + x[i+2] \ .$$

This results in the following set of $2N$ equations:

$$\begin{aligned}
\frac{\partial x_t[i]}{\partial t} &= \alpha(x_t[i-1] - 2x_t[i] + x_t[i+1]) \\
&\quad + \beta(-x_t[i-2] + 4x_t[i-1] - 6x_t[i] + 4x_t[i+1] - x_t[i+2]) \\
&\quad + f_x(x_t[i], y_t[i]) \\
\frac{\partial y_t[i]}{\partial t} &= \alpha(y_t[i-1] - 2y_t[i] + y_t[i+1]) \\
&\quad + \beta(-y_t[i-2] + 4y_t[i-1] - 6y_t[i] + 4y_t[i+1] - y_t[i+2]) \\
&\quad + f_y(x_t[i], y_t[i]) \ .
\end{aligned}$$

We can write the whole thing, except the external force, as a N-by-N matrix A:

$$\begin{bmatrix} -2\alpha-6\beta & \alpha+4\beta & -\beta & 0 & 0 & 0 & \dots & -\beta & \alpha+4\beta \\ \alpha+4\beta & -2\alpha-6\beta & \alpha+4\beta & -\beta & 0 & 0 & \dots & 0 & -\beta \\ -\beta & \alpha+4\beta & -2\alpha-6\beta & \alpha+4\beta & -\beta & 0 & \dots & 0 & 0 \\ 0 & -\beta & \alpha+4\beta & -2\alpha-6\beta & \alpha+4\beta & -\beta & \dots & 0 & 0 \\ 0 & 0 & -\beta & \alpha+4\beta & -2\alpha-6\beta & \alpha+4\beta & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ -\beta & 0 & 0 & 0 & 0 & 0 & \dots & -2\alpha-6\beta & \alpha+4\beta \\ \alpha+4\beta & -\beta & 0 & 0 & 0 & 0 & \dots & \alpha+4\beta & -2\alpha-6\beta \end{bmatrix}$$

such that:

$$\frac{\partial X_t}{\partial t} = AX_t + f_x(X_t, Y_t)$$
$$\frac{\partial Y_t}{\partial t} = AY_t + f_y(X_t, Y_t) \ .$$

Note the simple structure of this matrix. Only 5 diagonals have values in them, the rest of the matrix is 0, no matter how large N is. This is called a cyclic penta-diagonal banded matrix, and can easily be inverted using Cholesky decomposition. If the snake were not circular (the two ends not connected) this matrix would be even simpler!

The last step is to discretize the time parameter, adding a step size γ:

$$\frac{\partial X_t}{\partial t} = \frac{X_t - X_{t-1}}{\gamma} \ .$$

To make this easy to solve, we need to do a little trick: we will assume that the movement of the snake is small enough that the external force doesn't change much from one time step to the next. We can then substitute in the set of equations above the external force at time $t-1$, yielding a simple to solve set of equations:

$$X_t = (I - \gamma A)^{-1} \{X_{t-1} + \gamma f_x(X_{t-1}, Y_{t-1})\}$$
$$Y_t = (I - \gamma A)^{-1} \{Y_{t-1} + \gamma f_y(X_{t-1}, Y_{t-1})\} \ .$$

```
def create_A(a, b, N):
    """
    a:alpha parameter
    b:beta parameter
    N:is the number of points sampled on the snake curve: (x(p_i), y(p_i)), i=0,...,N-1
    """
    row = np.r_[
        -2*a - 6*b,
        a + 4*b,
        -b,
        np.zeros(N-5),
        -b,
        a + 4*b
    ]
    A = np.zeros((N,N))
    for i in range(N):
        A[i] = np.roll(row, i)
    return A


def create_external_edge_force_gradients_from_img( img, sigma=30. ):
    """
    Given an image, returns 2 functions, fx & fy, that compute
    the gradient of the external edge force in the x and y directions.
    img: ndarray
        The image.
    """
    img=(img-img.min()) / (img.max()-img.min())
    # Gaussian smoothing.
    smoothed = cv2.GaussianBlur(img, (7,7), 30)
    # Gradient of the image in x and y directions.
    gix=cv2.Sobel(smoothed ,cv2.CV_64F, 1, 0, ksize=3)
    giy =cv2.Sobel(smoothed, cv2.CV_64F, 0, 1, ksize=3)
    # Gradient magnitude of the image.
    gmi = (gix**2 + giy**2)**(0.5)
    # Normalize. This is crucial (empirical observation).
    gmi = (gmi - gmi.min()) / (gmi.max() - gmi.min())
     # Gradient of gradient magnitude of the image in x and y directions.
    ggmiy, ggmix = np.gradient( gmi )
    def fx(x, y):
        """
        Return external edge force in the x direction.
        """
        # Check bounds.
        x[ x < 0 ] = 0.
        y[ y < 0 ] = 0.

        x[ x > img.shape[1]-1 ] = img.shape[1]-1
        y[ y > img.shape[0]-1 ] = img.shape[0]-1
```

```
        return ggmix[ (y.round().astype(int), x.round().astype(int)) ]
    def fy(x, y):
        """
        Return external edge force in the y direction.
        """
        # Check bounds.
        x[ x < 0 ] = 0.
        y[ y < 0 ] = 0.

        x[ x > img.shape[1]-1 ] = img.shape[1]-1
        y[ y > img.shape[0]-1 ] = img.shape[0]-1

        return ggmiy[ (y.round().astype(int), x.round().astype(int)) ]

    return fx, fy


def iterate_snake(img,x, y, a, b, gamma=0.1, n_iters=10, return_all=True):
    """
    x:intial x coordinates of the snake
    y:initial y coordinates of the snake
    a:alpha parameter
    b:beta parameter
    fx: callable
        partial derivative of first coordinate of external energy function. This is the
first element of the gradient of the external energy.
    fy: callable
        see fx.
    gamma:step size of the iteration

    n_iters:number of times to iterate the snake
    return_all: if True, a list of (x,y) coords are returned corresponding to each
iteration.
        if False, the (x,y) coords of the last iteration are returned.
    """
    # fx and fy are callable functions
    fx, fy = create_external_edge_force_gradients_from_img( img, sigma=10 )

    A = create_A(a,b,x.shape[0])
    B = np.linalg.inv(np.eye(x.shape[0]) - gamma*A)
    if return_all:
        snakes = []

    for i in range(n_iters):
        x_ = np.dot(B, x + gamma*fx(x,y))
        y_ = np.dot(B, y + gamma*fy(x,y))
        x, y = x_.copy(), y_.copy()
        if return_all:
```

```
            snakes.append( (x_.copy(),y_.copy()) )

    if return_all:
        return snakes
    else:
        return (x,y)
```

Using the next function to control the initial contour by entering Radius .

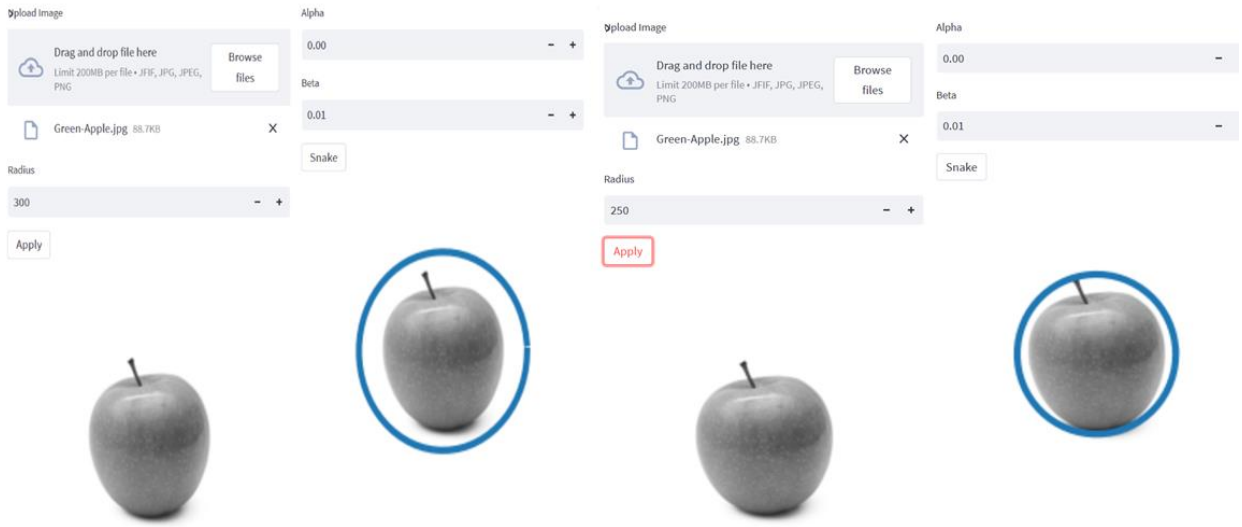
```
def initialcontour(img,R):        #R:radius
    cx=img.shape[0]/2            #cx,cy:center
    cy=img.shape[1]/2
    t = np.arange(0, 2*np.pi, 0.1)
    x = cx+R*np.cos(t)
    y = cy+R*np.sin(t)
    return x ,y
```

Parameters that the user can enter them:

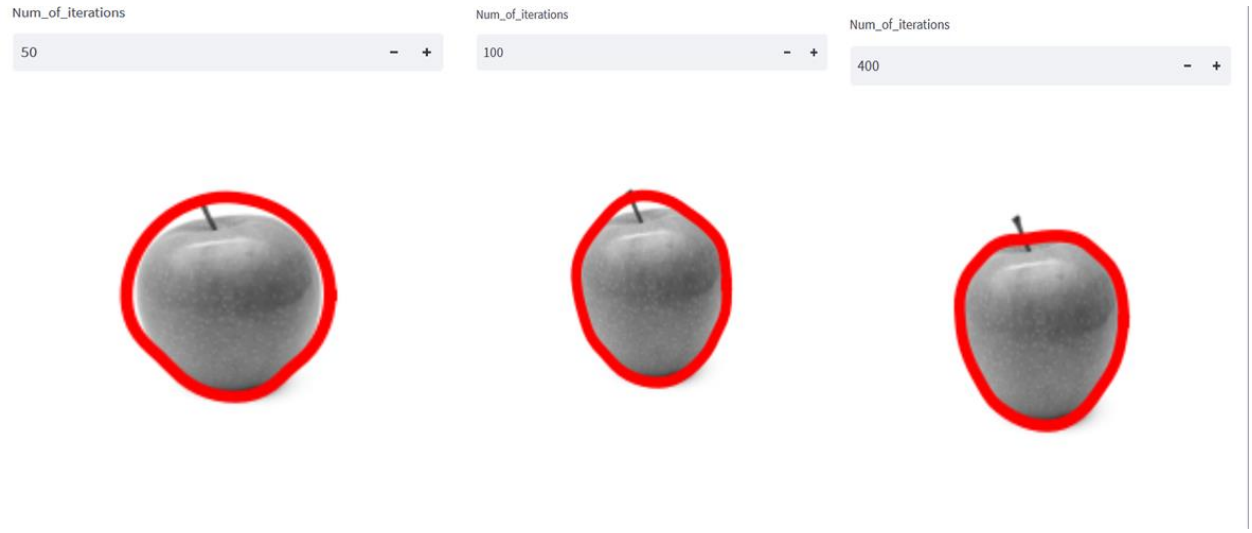
- Radius
- Alpha
- Beta
- Gamma
- Num_of_iterations

Result:

Initial contour



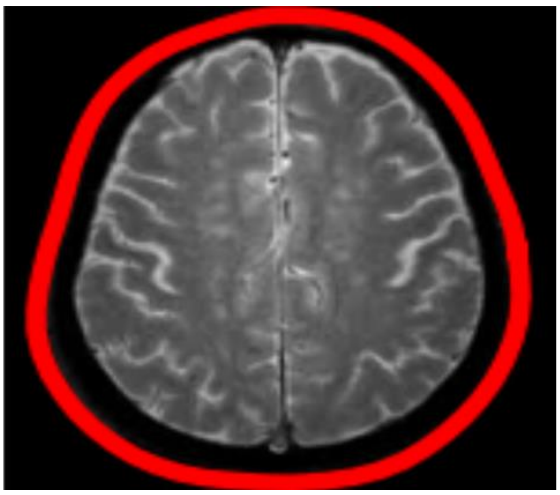
Snake



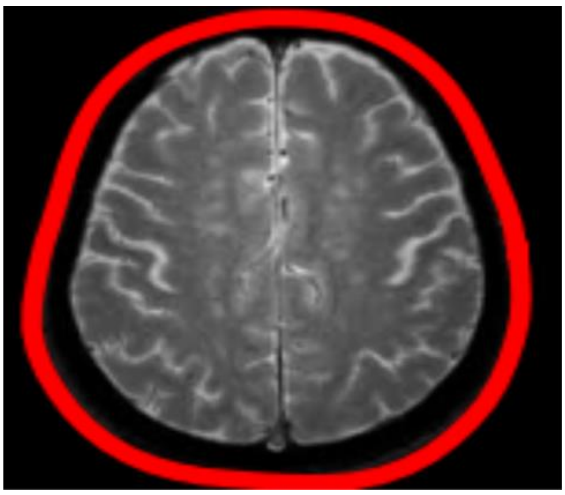
Note:

Num_of_iterations ↑ --- improve snake.

OpenCV



Implemented



❖ chain code

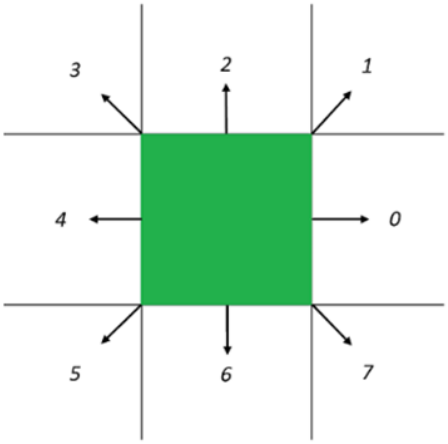
The chain codes could be generated by using conditional statements for each direction but it becomes very tedious to describe for systems having large number of directions (3-D grids can have up to 26 directions). Instead, we use a hash function. The difference in X (dx)and Y(dy) co-ordinates of two successive points are calculated and hashed to generate the key for the chain code between the two points.

Chain code list: [5,6,7,4, -1,0,3,2,1]

Hash function: C(dx,dy)=3dy+dx+4

Hash table: -

dx	dy	$C(dx, dy)$	$chainCode[C]$
1	0	5	0
1	1	8	1
0	1	7	2
-1	1	6	3
-1	0	3	4
-1	-1	0	5
0	-1	1	6
1	-1	2	7



Algorithm :

```
#-------------------------------------------------Chain Code-------------------------
--------------------------------
'''using them to call  passes_through_all_points function
x_axis=np.r_[snakes[-1][0], snakes[-1][0][0]]
y_axis=(np.r_[snakes[-1][1], snakes[-1][1][0]])
arr = np.stack((x_axis, y_axis), axis=1)'''

# Python3 code for generating 8-neighbourhood chain

codeList = [5, 6, 7, 4, -1, 0, 3, 2, 1]
# This function generates the chaincode
# for transition between two neighbour points
def getChainCode(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    hashKey = 3 * dy + dx + 4
    return codeList[hashKey]

'''This function generates the list ofchaincodes for given list of points'''
def generateChainCode(ListOfPoints):
    chainCode = []
    for i in range(len(ListOfPoints) - 1):
        a = ListOfPoints[i]
        b = ListOfPoints[i + 1]
```

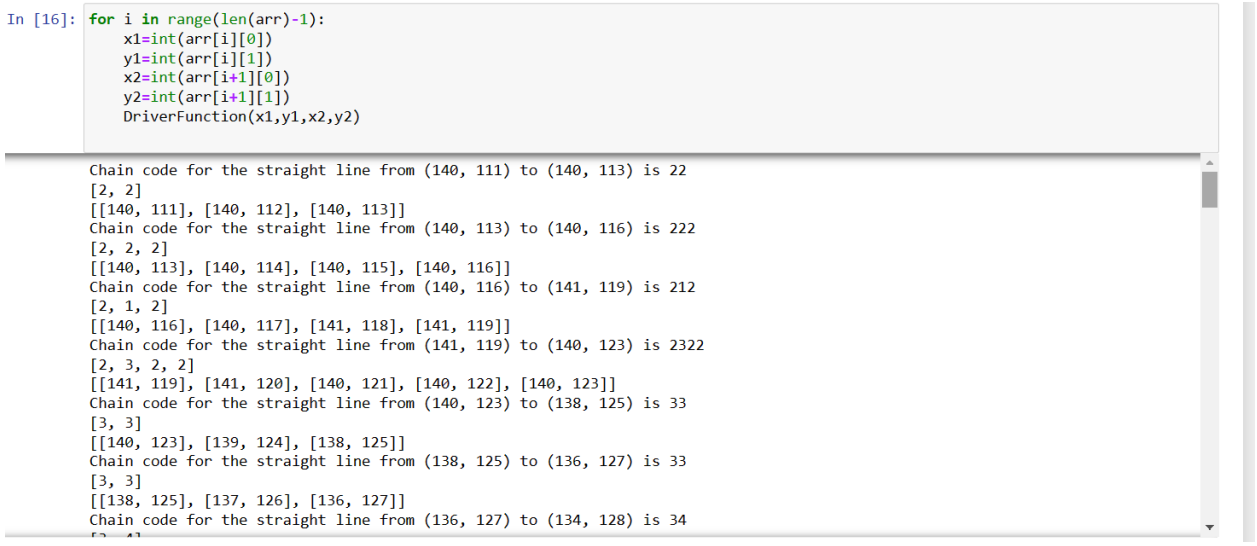
```
        chainCode.append(getChainCode(a[0], a[1], b[0], b[1]))
    return chainCode

'''This function generates the list of points for
a straight line using Bresenham's Algorithm'''
def Bresenham2D(x1, y1, x2, y2):
    ListOfPoints = []
    ListOfPoints.append([x1, y1])
    xdif = x2 - x1
    ydif = y2 - y1
    dx = abs(xdif)
    dy = abs(ydif)
    if(xdif > 0):
        xs = 1
    else:
        xs = -1
    if (ydif > 0):
        ys = 1
    else:
        ys = -1
    if (dx > dy):

        # Driving axis is the X-axis
        p = 2 * dy - dx
        while (x1 != x2):
            x1 += xs
            if (p >= 0):
                y1 += ys
                p -= 2 * dx
            p += 2 * dy
            ListOfPoints.append([x1, y1])
    else:

        # Driving axis is the Y-axis
        p = 2 * dx-dy
        while(y1 != y2):
            y1 += ys
            if (p >= 0):
                x1 += xs
                p -= 2 * dy
            p += 2 * dx
            ListOfPoints.append([x1, y1])
    return ListOfPoints

def DriverFunction(x1,y1,x2,y2):
    (x1, y1) = (x1,y1 )
    (x2, y2) = (x2, y2)
    ListOfPoints = Bresenham2D(x1, y1, x2, y2)
```

```
    chainCode = generateChainCode(ListOfPoints)
    chainCodeString = "".join(str(e) for e in chainCode)
    print ('Chain code for the straight line from', (x1, y1),'to', (x2, y2), 'is',
chainCodeString)
    print (chainCode)
    print (ListOfPoints)
```

Using the next function to calculate the chain code between each 2points in snake contours

```
def passes_through_all_points (arr):
    for i in range(len(arr)-1):
        x1=int(arr[i][0])
        y1=int(arr[i][1])
        x2=int(arr[i+1][0])
        y2=int(arr[i+1][1])
        DriverFunction(x1,y1,x2,y2)
```

The output:



```
In [16]: for i in range(len(arr)-1):
             x1=int(arr[i][0])
             y1=int(arr[i][1])
             x2=int(arr[i+1][0])
             y2=int(arr[i+1][1])
             DriverFunction(x1,y1,x2,y2)

Chain code for the straight line from (140, 111) to (140, 113) is 22
[2, 2]
[[140, 111], [140, 112], [140, 113]]
Chain code for the straight line from (140, 113) to (140, 116) is 222
[2, 2, 2]
[[140, 113], [140, 114], [140, 115], [140, 116]]
Chain code for the straight line from (140, 116) to (141, 119) is 212
[2, 1, 2]
[[140, 116], [140, 117], [141, 118], [141, 119]]
Chain code for the straight line from (141, 119) to (140, 123) is 2322
[2, 3, 2, 2]
[[141, 119], [141, 120], [140, 121], [140, 122], [140, 123]]
Chain code for the straight line from (140, 123) to (138, 125) is 33
[3, 3]
[[140, 123], [139, 124], [138, 125]]
Chain code for the straight line from (138, 125) to (136, 127) is 33
[3, 3]
[[138, 125], [137, 126], [136, 127]]
Chain code for the straight line from (136, 127) to (134, 128) is 34
```

Function to compate perimeter and area

```
def compute_perimeter(contour):
    """
    This function takes a contour as input and computes its perimeter
    """

    # Start with the first point of the contour
    current_point = contour[0]
```

```
    perimeter = 0

    # Iterate over all points of the contour
    for i in range(1, len(contour)):
        # Find the next point on the contour
        next_point = contour[i]
        # Compute the euclidean distance between the points
        distance = math.sqrt((next_point[0] - current_point[0])**2 + (next_point[1] -
current_point[1])**2)
        # Add the distance to the perimeter
        perimeter += distance
        # Move to the next point
        current_point = next_point

    return perimeter
```

```
def compute_area(contour):
    """
    This function takes a contour as input and computes the area inside it
    """

    # Convert the contour to a list of x-coordinates and y-coordinates
    x_values = [point[0] for point in contour]
    y_values = [point[1] for point in contour]

    # Apply the shoelace formula to compute the area
    area = 0.5 * abs(sum(x_values[i] * y_values[(i + 1) % len(contour)] - x_values[(i +
1) % len(contour)] * y_values[i]
                         for i in range(len(contour))))
    return area
```

the output:

```
In [32]: # Compute the perimeter of the contour
         perimeter = compute_perimeter(arr)
         print("Perimeter: ", perimeter)

         # Compute the area inside the contour
         area = compute_area(arr)
         print("Area: ", area)

         Perimeter:  207.46715481862017
         Area:  2688.2280718460133
```