Table of Contents

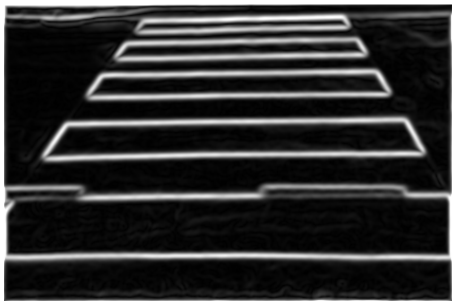
Whit Boland - jwb10028
Professor Gerig
CS6643 Computer Vision
20 March, 2024

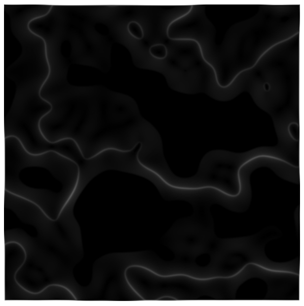
Lab 3 Report

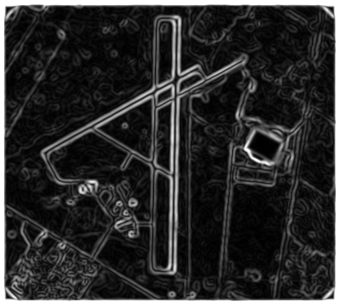
INTRODUCTION/ABSTRACT:

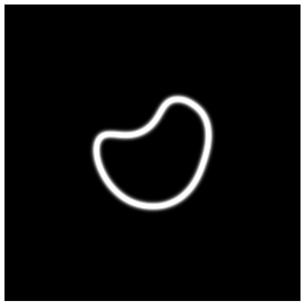
This report presents the implementation and evaluation of the snake, or active contour algorithm, a method for image analysis that dynamically adjusts to delineate object boundaries within an image. This study replicates the snake implementation as detailed in lecture by Trucco and Verri, with a particular focus on the interaction of internal and external forces that guide the contour's progression. A method for computing the gradient magnitude image has been developed in project two, forming the basis for the external force field that influences the snake. The iterative process is visualized through a series of images, showcasing the algorithm's responsiveness to the dynamic environment and its ability to converge to the minimum energy configuration. Special consideration is given to parameter values, including the sigma of the Gaussian filter and the intensity inversion of the gradient image, to optimize the snake's sensitivity and accuracy. The algorithm's performance is quantified through a step by step procedure, capturing the propagation of the snake from an initial polygon through successive iterations until the resulting snake is fully stabilized. This report not only illustrates the technical implementation but also provides insights into the decision-making process for adapting the snake to varying conditions, while offering a comprehensive understanding of its practical applications.

CONVOLUTION/GRADIENT MAGNITUDE RESULTS:


crosswalk.png


noise.png

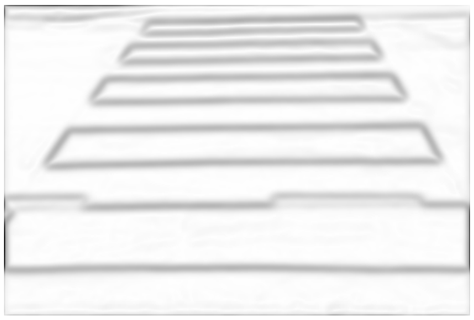

Ohio.jpg


shape.png

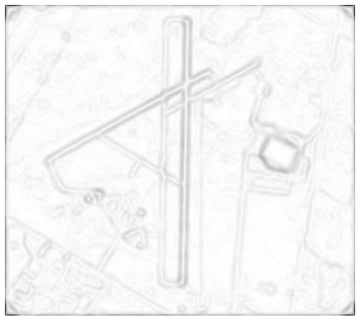
CONVOLUTION/GRADIENT MAGNITUDE DISCUSSION:

In order to provide data for the core snake algorithm it is necessary to first revisit the gradient magnitude implementation from project two. To reiterate, the above results represent assets from the project file after several convolution passes. The first pass through is used to convolve the source image with a gaussian blur filter. The reason the source image is first convolved with a gaussian filter is primarily to define the balance between edge smoothing and localization prior to the gradient magnitude calculation. To better explain, larger sigma values in the gaussian filter kernel will produce results with the broad edges of the image, as the sigma value is lowered the gradient magnitude calculation begins to define edges for the finer features in the source image. After the source image has been appropriately blurred using a gaussian filter the second two passes are needed to compute both the x, and y gradient images. After both the x, and y gradient images are calculated the gradient magnitude is calculated using the equation from project two $Gmag = \sqrt{(xderiv)^2 + (yderiv)^2}$. The only major difference in the process here is that instead of thresholding the source image to result in a binary edge image, the gradient magnitude is inverted to ensure that dark minimum intensities are found at the edges while the larger intensities are found outside of these detected edges. The primary reason for this step is to prepare the gradient magnitude image for input into the snake algorithm. To explain this preprocessing step further, active contours or”snakes” are designed to propagate towards the edges of objects/features within a source image. The effective use of inverting intensities at edges is to ensure that there is an energy minimization as the snake propagates towards lower intensity values of edges detected during the gradient magnitude process.With that being said, the primary reason for this inversion is to ensure that local minima within the image are avoided, which could cause the snake to get stuck inside non-edge structures found in the source image. By inverting the gradient magnitude image the snake algorithm is able to more effectively “lock onto” the edges of the image providing greater accuracy in the end result.

INVERTED GRADIENT RESULTS:


crosswalk.png

noise.png


Ohio.jpg

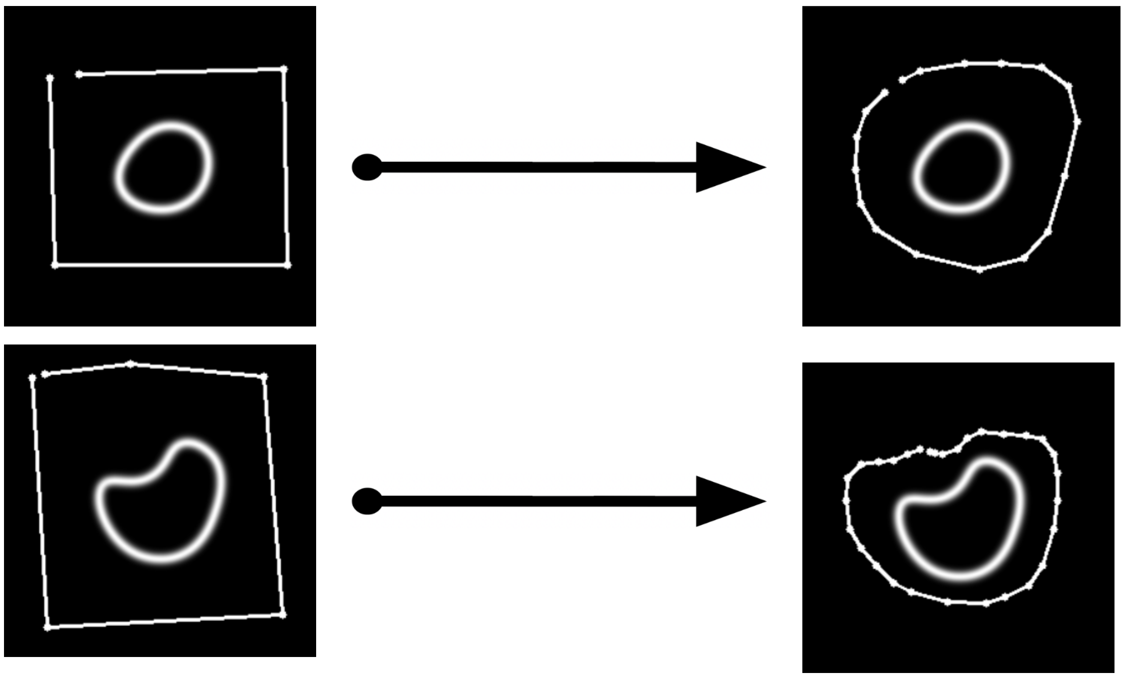
shape.png

PICKING POINTS/POLYGON GENERATION METHODOLOGY:

Throughout testing, a python-based method is employed to facilitate the selection of points and the initial drawing of a polygon on an image using openCV. The technique makes use of an external window to capture user interactions with the image. A mouse callback function, ‘click_event’, is registered to the openCV window to detect mouse clicks. As the user clicks on the image, the x and y coordinates of the mouse position are recorded and stored in a list named points. The program also provides immediate visual cues to the user by drawing a white circle of 3-pixel radius at each clicked point on the image, which is in this case the input gradient magnitude image used for edge detection in this context. When at least two points are selected, a white polygonal line is drawn connecting the points in the sequence they were clicked. This is written into the image using the cv2.polylines() function, which takes the accumulated points and plots the lines with a specified thickness, without closing the polygon. The primary reason that the polygon is not closed is to allow for the dynamic visual construction of the desired polygonal shape as more points are added. To better explain the point selection process, this method serves as an interactive tool to manually select regions of interest within the image, which are crucial for the later implementation of the core snake algorithm. These selected points make up the initial polygon snake shape that is used as a base for propagation throughout the active contour/snake procedure. From testing it can be noted that when dealing with complex objects/features within an image the more points the initial polygon has, it seems the more accurate the end result contour will be. It can be noted that in the below source code implementation the contour results are written to a result image which is overlaid with the input gradient magnitude (before inversion) image.

PICKING POINTS VISUALIZATION:

Needs more points in selection optimal points selection



CORE SNAKE/ACTIVE CONTOUR ALGORITHM:

The presented active contour model, commonly known as a "snake algorithm," is a method used for image segmentation and contour detection. It is an iterative process that relies on the concept of energy minimization to fit a contour to the perceived edges within an image. The algorithm is initialized with a contour defined by the user during the point selection process defined above, and through successive iterations, it moves towards the objects/features within the image. The behavior of the snake is guided by internal and external forces. According to the article on snakes written by Trucco & Verri, "The key idea of deformable contours is to associate an energy function to each possible contour shape, in such a way that the image contour to be detected corresponds to a minimum of the functional. Each energy term serves a different purpose…" (Trucco & Verri, 109). The internal forces are derived from the geometry of the snake itself and are controlled by two parameters that are defined as alpha (α) and beta (β). The alpha parameter influences the snake's tension, which is a measure of continuity and helps to prevent the points of the snake from spreading too far apart, higher alpha values will make the snake more "rubber-band" like which has a penalizing effect on the contours ability to stretch between points. The beta parameter affects the rigidity of the snake, controlling its curvature, and hence, resisting bending, analogous to a thin plate. To further define how the beta value acts as an internal force, a higher value will have a smoothing effect on the contour, but have a negative effect on the flexibility of the snake as it propagates. These parameters, therefore, dictate the smoothness and flexibility of the snake contour as it evolves over time. External forces are derived from the image data, designed to pull the contour towards objects/features, such as edges. In this implementation, the image gradient is used as a potential force field, attracting the snake towards the edges where the gradient is at a minimum. The gamma (γ) parameter serves as a damping coefficient, regulating the influence of both internal and external forces at each iteration. It essentially acts as a time step that maintains the displacement of points on the snake, which has a stabilizing effect on the evolution process. In this particular implementation, each point of the snake polygon is moved individually in each iteration cycle. The algorithm determines the net force on each point, combining the internal forces (controlled by α and β) with the external forces derived from the image data. With that being said, rather than circling around points until minimum energy is found this implementation recalculates and applies forces to all points for a specified number of iterations or until the energy of the system falls below a certain threshold. This energy comprises both the internal energy (from the snake's configuration) and the external energy (from the image features). To conclude the discussion on the snake algorithm, it is crucial to note that as mentioned above tuning these alpha, beta, and gamma values (in company with the sigma of the initial gaussian filtering) can have a large impact on the snake's ability to propagate to areas of minimum intensity within an image. By adjusting these values the programmer has the ability to influence the overall behavior of the snake.

CORE SNAKE/ACTIVE CONTOUR ALGORITHM RESULTS:

IMAGE: shape.png
STEPS: 1 - 6 PARAMETERS: alpha(.01), beta(.1), gamma(.04)

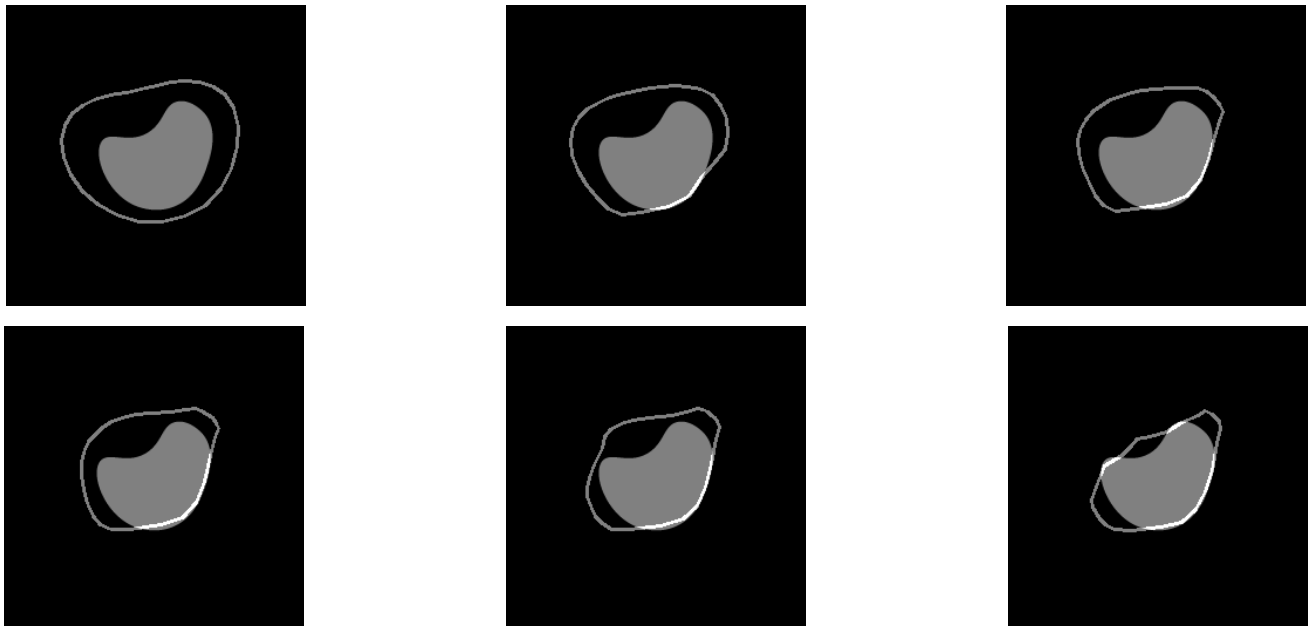
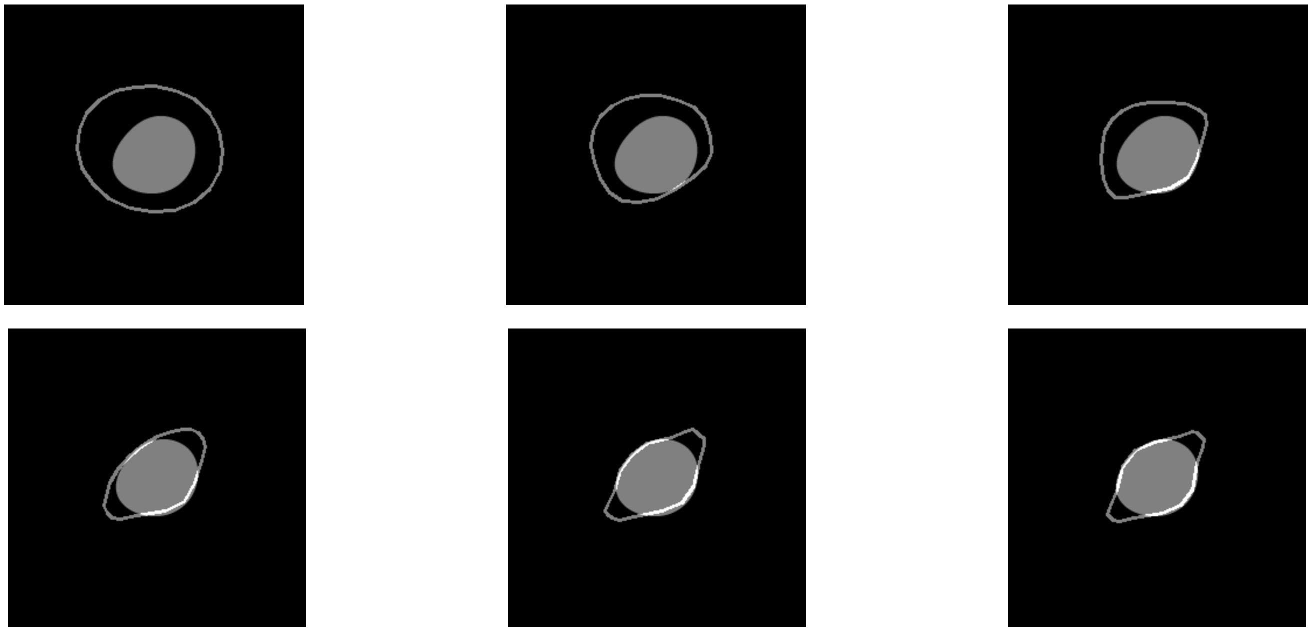


IMAGE: shape2.png
STEPS: 1 - 6 FORCE PARAMETERS: alpha(.01), beta(.1), gamma(.04)



MORE RESULTS:

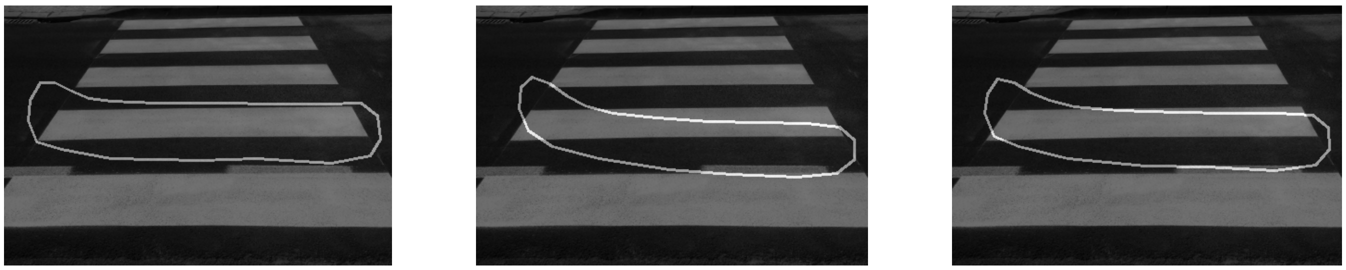


IMAGE: crosswalk.png
STEPS: 1-3
FORCE PARAMETERS: alpha(.01), beta(.01), gamma(.1)

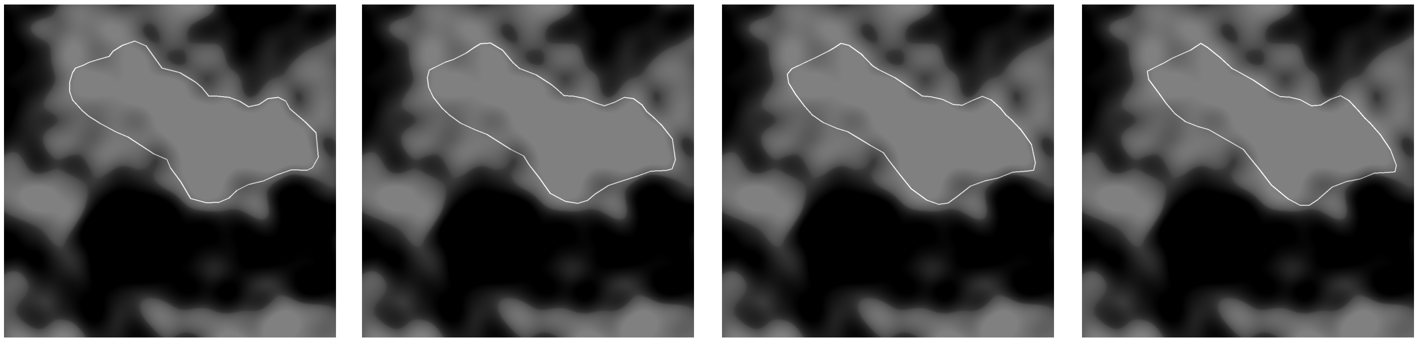


IMAGE: noise.png
STEPS: 1-4
FORCE PARAMETERS: alpha(.01), beta(.01), gamma(.1)

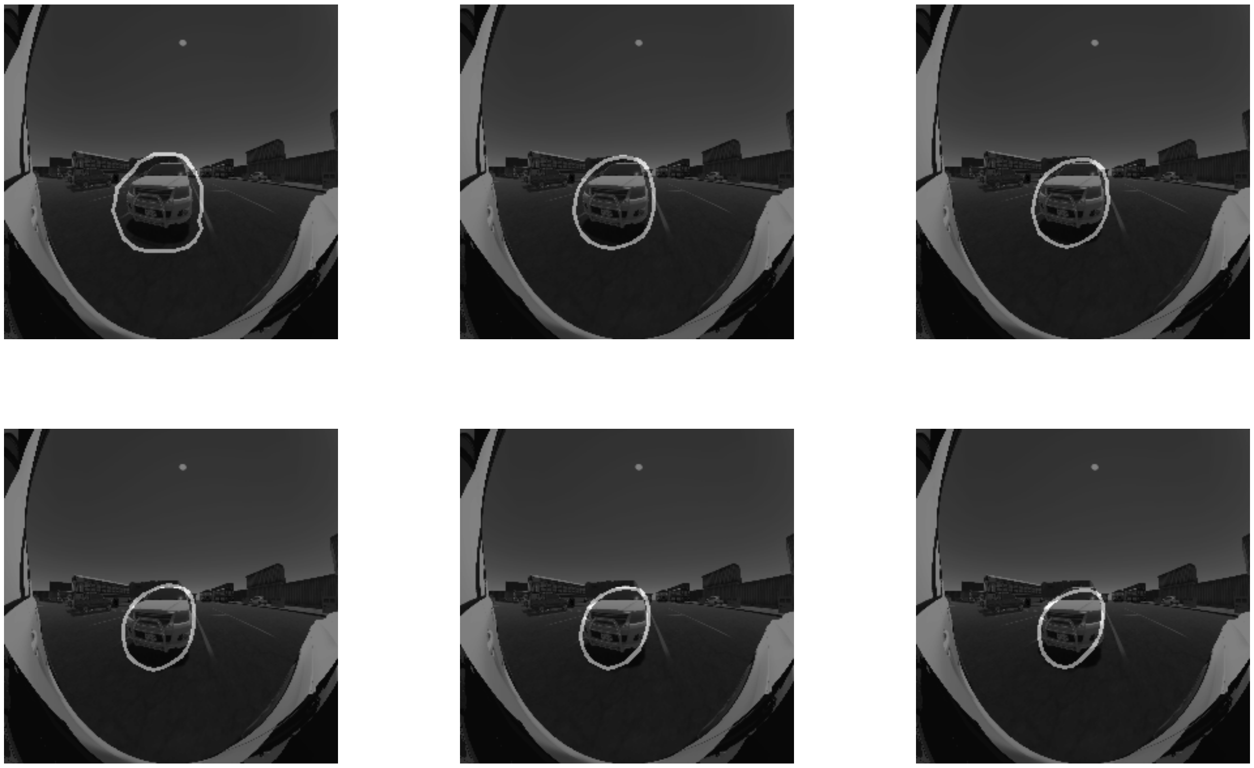
RESULTS DISCUSSION:

Looking at the above results from two different source images passed through the snake algorithm, it is fairly easy to see that as the complexity of objects/features within the source image increase, the harder it is for the snake algorithms to effectively lock onto desired features within the source image (in this case edges). These results suggest that while the snake algorithm is effective in identifying and latching onto clear edges, the complexity of object features in the source image significantly influence the efficiency and accuracy of the contouring process. For example, as seen above with the results on the 'crosswalk.png' source image it can be noted that using a snake to propagate towards a crosswalk block will be fairly difficult as the snake pushes/pulls towards the minimum edge at each point which in this case varies widely. To further define, the algorithm at this stage lacks the ability to differentiate between individual objects, but rather locks to the nearest edge. The algorithm's ability to hone in on the edges may require fine-tuning of its parameters and potentially more iterations, highlighting the importance of considering the source image's complexity when using active contour models for edge detection tasks.

APPLICATIONS/USES:

In terms of possible uses for the active contour/snake algorithm the possibilities are somewhat boundless, and use cases can vary widely. For example, as discussed in the above report a primary use of the snake algorithm is to perform object tracking. The snake algorithm works great as an object tracking algorithm especially for video analysis due to the ability to update the snake's contour position dynamically. Due to this trait, active contours can move with objects that are visibly in motion in the source video feed. The ability to move with objects detected in the feed insinuates practical applications for the algorithm like security/surveillance, human computer interaction (i.e. gesture control), vehicle navigation, and many others. Below are a few examples of a possible use case for the snake algorithm as a vehicle backup camera hazard detection system. The snake algorithm takes an input gradient magnitude image taken from the source feed on a vehicle bumper backup camera. Looking from the start of propagation to the end it is clear to see that the algorithm does a great job at detecting vehicles close enough to the lens to create distinct edges after gradient magnitude computation. The gaussian filter could be tweaked to establish a sense of proximity for object detection, that is with a higher sigma value the gradient magnitude computation would start to focus in on the broad edges of the source image.

IMAGE: fe.png
STEPS: 1 - 6
FORCE PARAMETERS: alpha(.1), beta(.61), gamma(.2)



References

Trucco, Emanuele, and Alessandro Verri. Introductory Techniques for 3-D Computer Vision. Prentice Hall, 2006.

SOURCE CODE:

```
# CODE HERE:
import matplotlib.pyplot as plt
import matplotlib.image as img
import numpy as np
import cv2
# reading in source image
# READING IMAGES:
in_img = cv2.imread('./ASSETS/fe.png', cv2.IMREAD_GRAYSCALE)
r_img = cv2.resize(in_img, (256, 256))
a_img = np.array(r_img)
n_img = ((a_img - np.min(a_img)) * (1/(np.max(a_img) - np.min(a_img)) *
255)).astype('uint8')
convolution_input = n_img.astype(float)
# FUNCTIONS 01:
def convolution(image, filter):
    # Convert image to float array if it's not already
    image = [[float(val) for val in row] for row in image]
    # Convert filter to float array if it's not already
    filter = [[float(val) for val in row] for row in filter]

    # Determine the dimensions of the image and filter
    image_height = len(image)
    image_width = len(image[0])
    filter_height = len(filter)
    filter_width = len(filter[0])

    # Calculate padding
    pad_height = filter_height // 2
    pad_width = filter_width // 2

    # Pad the image with zeros on all sides
    padded_image = [[0 for _ in range(image_width + 2 * pad_width)] for _
in range(image_height + 2 * pad_height)]
    for i in range(image_height):
        for j in range(image_width):
            padded_image[i + pad_height][j + pad_width] = image[i][j]

    # Prepare the img_conv array
    img_conv = [[0 for _ in range(image_width)] for _ in
range(image_height)]

    # Apply the filter
    for y in range(image_height):
        for x in range(image_width):
            # Extract the current region of interest
            region = [[padded_image[i][j] for j in range(x, x +
filter_width)] for i in range(y, y + filter_height)]
            # Perform element-wise multiplication and sum the result
            img_conv[y][x] = sum(sum(region[i][j] * filter[i][j] for j in
range(filter_width)) for i in range(filter_height))

    return img_conv

def manual_threshold(img_in, threshold):

    manual_thresh_img = []

    for row in img_in:
        thresholded_row = []
        for pixel in row:
            thresholded_row.append(255 if pixel > threshold else 0)
        manual_thresh_img.append(thresholded_row)

    threshold_img = manual_thresh_img

    return threshold_img

def gaussian_kernel_x(l, sig):

    ax = np.linspace(-(l - 1) / 2., (l - 1) / 2., l)
    gauss = np.exp(-0.5 * ax**2 / sig**2)
    kernel = np.outer(gauss, 1)

    return kernel / np.sum(kernel)

def gaussian_kernel_y(l, sig):

    ax = np.linspace(-(l - 1) / 2., (l - 1) / 2., l)
    gauss = np.exp(-0.5 * ax**2 / sig**2)
    kernel = np.outer(1, gauss)

    return kernel / np.sum(kernel)
# A1:
# Derivative Kernels:
sigma = 1.0
filter_x = [[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]]
filter_y = [[1, 2, 1], [0, 0, 0], [-1, -2, -1]]
kernel_1dx = gaussian_kernel_x(6*int(sigma)+1, sigma)
kernel_1dy = gaussian_kernel_y(6*int(sigma)+1, sigma)
gaussian_image_x = convolution(convolution_input, kernel_1dx)
gaussian_image = convolution(gaussian_image_x, kernel_1dy)
gradient_x = convolution(gaussian_image, filter_x)
gradient_y = convolution(gaussian_image, filter_y)
grad_mag = np.sqrt(np.array(gradient_x)**2 + np.array(gradient_y)**2)
cv2.imwrite('gradient_magnitude.png', grad_mag)
def invert_gradient_intensities(grad_magnitude):

    cv2.normalize(grad_magnitude, grad_magnitude, 0, 1, cv2.NORM_MINMAX)

    inverted_grad_magnitude = 1 - grad_magnitude

    inverted_grad_magnitude = (inverted_grad_magnitude *
255).astype(np.uint8)

    return inverted_grad_magnitude
inv_mag = invert_gradient_intensities(grad_mag)

cv2.imwrite('inv_gradient_magnitude.png', inv_mag)
# FUNCTIONS 02:
def draw_contour(image, contour):
    result_img = np.zeros_like(image)
    res_img = cv2.polylines(result_img, [contour], isClosed=True,
color=(255, 255, 255), thickness=2)
    return res_img

def overlay_images(image1, image2, alpha=.50):
    if not (0 <= alpha <= 1):
        raise ValueError("Alpha should be a float between 0 and 1.")
    # Calculate beta (the weight of the second image)
    beta = 1.0 - alpha

    blended_image = cv2.addWeighted(image1, alpha, image2, beta, 0)
    return blended_image
points = []

def click_event(event, x, y, flags, param):
    if event == cv2.EVENT_LBUTTONDOWN:
        # Append the new point (x, y)
        points.append((x, y))

        # Draw a small circle at the click point
        cv2.circle(img, (x, y), 3, (255, 255, 255), -1)

        # Draw the polygon if there are at least two points
        if len(points) >= 2:
            cv2.polylines(img, [np.array(points)], isClosed=False,
color=(255, 255, 255), thickness=2)

        cv2.imshow('image', img)

# Load an image
img = grad_mag
if img is None:
    print("Error: Image not found.")
else:
    # Create a window
    cv2.namedWindow('image')
    cv2.setMouseCallback('image', click_event)

    # Display the image
    cv2.imshow('image', img)

    # Wait until any key is pressed
    cv2.waitKey(0)
    cv2.destroyAllWindows()
contour = np.array(points ,dtype=np.int32)
```

```
contour_img = overlay_images(draw_contour(grad_mag, contour),
convolution_input)


cv2.imwrite('init_contour.png', contour_img)
contour_image = contour_img
init_contour = contour
def snake_algorithm(gradient_image, initial_contour, iterations,
alpha=0.1, beta=0.61, gamma=.20, energy_threshold=4.0):
    inverted_gradient = 255 - gradient_image

    snake = initial_contour.astype(np.float32)

    gray_image = cv2.cvtColor(inverted_gradient, cv2.COLOR_BGR2GRAY) if
len(inverted_gradient.shape) == 3 else inverted_gradient

    gray_image_8bit = cv2.convertScaleAbs(gray_image)

    external_force = cv2.distanceTransform(gray_image_8bit, cv2.DIST_L2,
5)
    internal_energy = 0
    external_energy = 0

    # Iteratively update the snake
    for i in range(iterations):


        # Internal force (continuity and curvature)
        for i in range(snake.shape[0]):
            prev_point = snake[i - 1 if i - 1 >= 0 else -1]
            next_point = snake[(i + 1) % snake.shape[0]]

            continuity_force = alpha * (prev_point - snake[i])
            curvature_force = beta * (prev_point - 2 * snake[i] +
next_point)

            snake[i] += gamma * (continuity_force + curvature_force)
        # External force (image gradient)
        for i in range(snake.shape[0]):
            point = np.round(snake[i]).astype(int)

            point[0] = np.clip(point[0], 0, external_force.shape[1] - 1)
            point[1] = np.clip(point[1], 0, external_force.shape[0] - 1)

            # Apply the external force
            gradient = np.array(np.gradient(external_force))
            external_force_at_point = gradient[:, point[1], point[0]]
            snake[i] += gamma * external_force_at_point

        internal_energy += continuity_force**2 + curvature_force**2
        external_energy += np.linalg.norm(external_force_at_point)**2


    snake = snake.astype(np.int32)



    return snake
snake_contour_01 = snake_algorithm(inv_mag, init_contour, 5)
snake_contour_02 = snake_algorithm(inv_mag, init_contour, 10)
snake_contour_03 = snake_algorithm(inv_mag, init_contour, 15)
snake_contour_04 = snake_algorithm(inv_mag, init_contour, 20)
snake_contour_05 = snake_algorithm(inv_mag, init_contour, 25)
snake_contour_06 = snake_algorithm(inv_mag, init_contour, 30)
snake_contour_img1 = overlay_images(draw_contour(grad_mag,
snake_contour_01), convolution_input)
snake_contour_img2 = overlay_images(draw_contour(grad_mag,
snake_contour_02), convolution_input)
snake_contour_img3 = overlay_images(draw_contour(grad_mag,
snake_contour_03), convolution_input)
snake_contour_img4 = overlay_images(draw_contour(grad_mag,
snake_contour_04), convolution_input)
snake_contour_img5 = overlay_images(draw_contour(grad_mag,
snake_contour_05), convolution_input)
snake_contour_img6 = overlay_images(draw_contour(grad_mag,
snake_contour_06), convolution_input)

cv2.imwrite('step1.png', snake_contour_img1)
```

```
cv2.imwrite('step2.png', snake_contour_img2)
cv2.imwrite('step3.png', snake_contour_img3)
cv2.imwrite('step4.png', snake_contour_img4)
cv2.imwrite('step5.png', snake_contour_img5)
cv2.imwrite('step6.png', snake_contour_img6)
```