Computer Vision Assignment 2

Submission Requirements

Format

1. Project materials including template: `proj2.zip`.
2. You should provide a `README.md` file to describe how to run the code.
3. Pack your `proj2_code` and `README.md` into a zip file, named with your student ID, like [student ID].zip.
4. **Note that, additional Python packages should not be introduced. Do not use absolute paths in your code. Use relative paths like the starter code already does.**

Submission Way

1. Please submit your results to https://docs.qq.com/form/page/DS3FQd0ZvUWNITmJ1
2. The deadline is 23:59 on May 22, 2025. No submission after this deadline is acceptable.

About Plagiarize

DO NOT PLAGIARIZE! We have no tolerance for plagiarizing and will penalize it with giving zero score. You may refer to some others' materials, please make citations such that one can tell which part is actually yours.

Evaluation Criterion

We mainly evaluate your submission according to your code and report. Efficient implementation, elegant code style, concise and logical report are all important factors towards a high score.

Setup

1. Install Anaconda.
2. Download and extract the project starter code.

3. Create a conda environment using following command.

```
conda create cv_proj2 python=3.8 -y
conda activate cv_proj2
pip install numpy opencv-python # no more packages should be installed
```

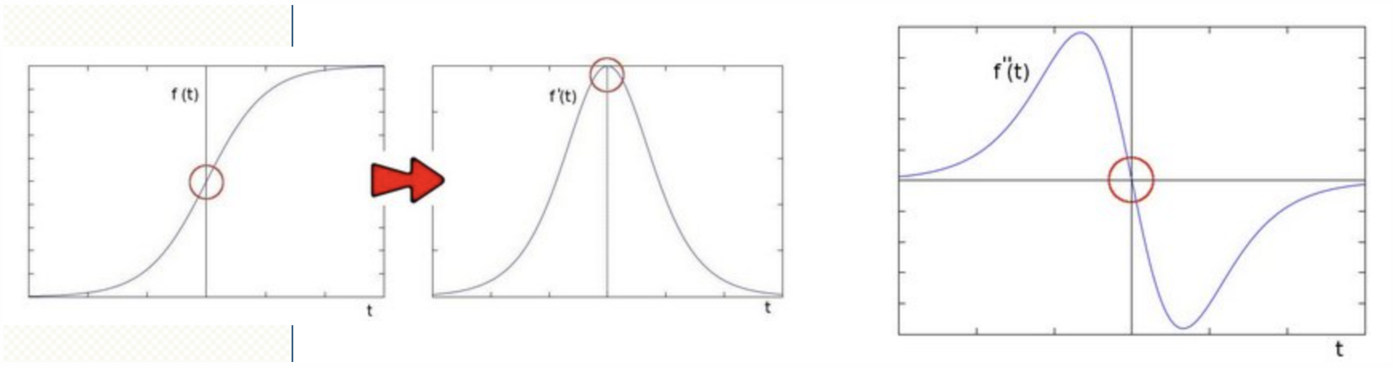
Part 1: Edge Detection (50%)

Overview: The main steps of edge detection are: (1) assign a score to each pixel; (2) find local maxima along the direction perpendicular to the edge. Sometimes a third step is performed where local evidence is propagated so that long contours are more confident or strong edges boost the confidence of nearby weak edges. Optionally, a thresholding step can then convert from soft boundaries to hard binary boundaries.

We have provided 50 test images BSDS, along with some code for main framework. Your job is to build gradient-based edge detectors, including Laplace Operator, Sobel Operator and Canny Operator (based on Sobel Operator), within the code interface we provide. The code will compare the results of your operations with the results of standard opencv operator in the form of a spliced image.

1.1 Laplace Operator (15\%)

The Laplacian operator detects edges by computing the second derivative of the image: in edge regions, pixel intensity exhibits a "jump" or a high variation in intensity. If we take the first derivative of the pixel intensity, we observe that the edge corresponds to a maximum value; if we take the second derivative of the pixel intensity, the value at the edge is zero.



Mathematically, the Laplacian operator is defined as follows:

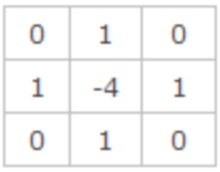
$$\nabla^2 f\$(x,y)\$ = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \tag{1}$$

where ∇^2 denotes the Laplacian operator, $f(x,y)$ is the grayscale value function of the image, $\frac{\partial^2 f}{\partial x^2}$ and $\frac{\partial^2 f}{\partial y^2}$ represent represent the second-order partial derivatives of the image in the x and y directions, respectively. By calculating the sum of these two partial derivatives, edges and texture features in the image can be detected.

The discrete form of the Laplacian operator is

$$\nabla^2 f\$(x,y)\$ = f(x+1,y) + f(x-1,y) + f(x,y+1) + f(x,y-1) - 4f(x,y) \tag{2}$$

. Therefore, a 3×3 Laplacian operator can be represented by the following convolution kernel (mask):



0	1	0
1	-4	1
0	1	0

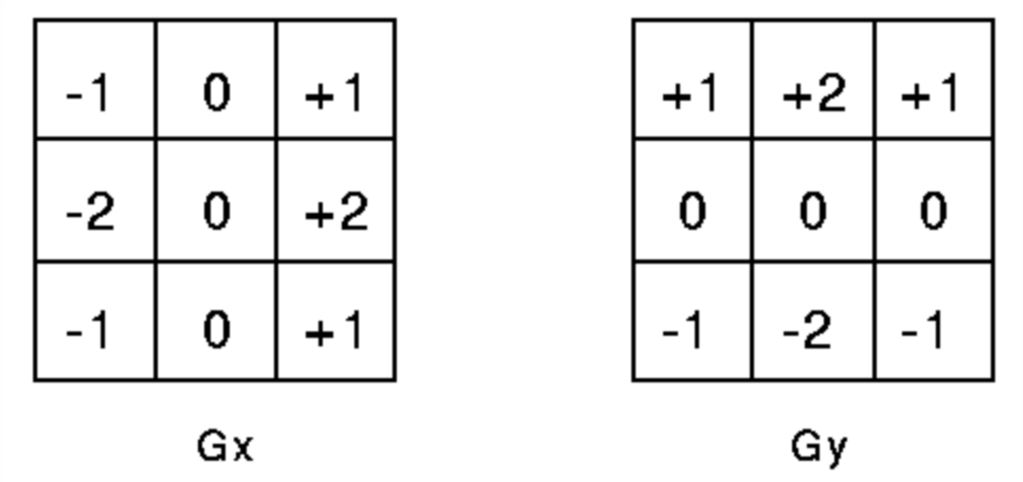
This kernel is used to apply the Laplacian operator to an image, highlighting the edges and details.

According to the OpenCV official documentation, `StandLaplacian()` provides the standard OpenCV operation `cv2.Laplacian`. You need to implement your own Laplacian operator in `CustomLaplacian()`, including the **kernel** and **filter**. The comparsion output will be in `EdgeDetection/output/Laplacian`.

1.2 Sobel Operator (15\%)

The Sobel operator is a first-order derivative edge detection operator. In the algorithm implementation, a 3×3 kernel is used to perform convolution with each pixel in the image, and then an appropriate threshold is selected to extract the edges.

This operator consists of two sets of 3×3 matrices, corresponding to the horizontal and vertical directions. By performing planar convolution between these matrices and the image, we can obtain approximate values of the brightness differences in the horizontal and vertical directions, respectively.



-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

If A represents the original image, and G_x and G_y represent the grayscale values of the image after horizontal and vertical edge detection, the formulas are as follows:

$$\mathbf{G_x} = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G_y} = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

The horizontal and vertical grayscale values of each pixel in the image are combined using $\mathbf{G} = \sqrt{\mathrm{G_x}^2 + \mathrm{G_y}^2}$ to calculate the gradient magnitude at that point. If the gradient G is greater than a certain threshold, the point (x, y) is considered an edge point.

According to the OpenCV official documentation, `StandSobel()` provides the standard OpenCV operation `cv2.Sobel`. You need to implement your own Laplacian operator in `CustomSobel()`, including the **kernel** and **filter**, and **combination**. The comparsion output will be in `EdgeDetection/output/Sobel`.

1.3 Canny Operator (20\%)

Laplacian edge detection and Sobel edge detection both calculate edges through convolution operations. Their algorithms are relatively simple, so the results may lose too much edge information or contain a lot of noise. In contrast, the Canny edge detection algorithm is more complex and consists of the following five steps:

1. Use Gaussian filtering to remove image noise.
2. Use the Sobel kernel for filtering to calculate the gradient.
3. Apply non-maximum suppression at the edges.
4. Use double thresholding on the detected edges to remove false positives.
5. Analyze the connectivity between edges to retain true edges and eliminate weak or insignificant edges.

You need to implement your own Laplacian operator in `CustomCanny()`, including at least the first four of the above five steps. The comparsion output will be in `EdgeDetection/output/Canny`.

Part 2: Image segmentation (50%)

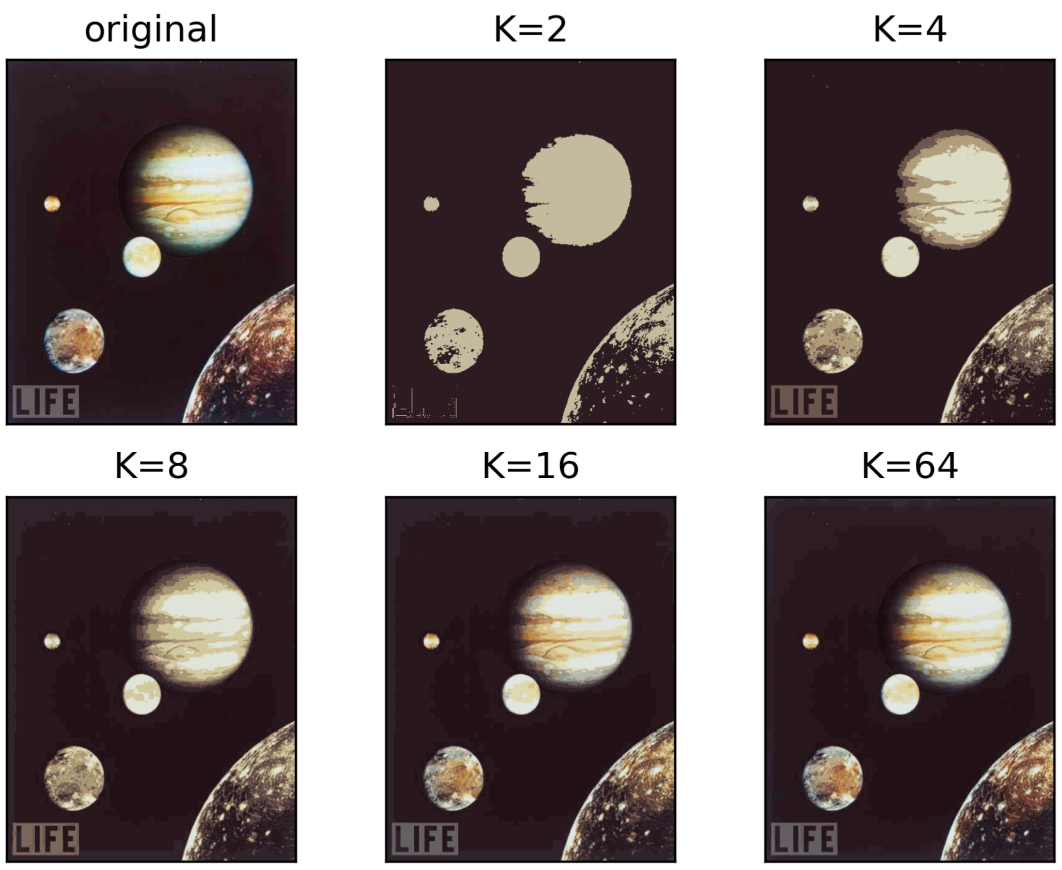
2.1 Image segmentation with k-means (25%)

K-Means clustering is the most commonly used clustering algorithm, originally derived from signal processing. Its goal is to partition data points into K clusters, find the center of each cluster, and minimize the metric. K-Means clustering is also widely used as an image segmentation clustering algorithm. In this method, pixels in the image are treated as data points, and the algorithm divides them into K clusters based on their similarity. Similarity is measured using distance metrics, such as Euclidean distance or Mahalanobis distance. The algorithm first randomly selects K initial centroids, then iteratively assigns each pixel to the nearest centroid and updates the centroids based on the mean of the assigned pixels. This process continues until the centroids converge to stable values.

You need to implement the k-means clustering algorithm according to the following steps:

1. Determine the positions and number of the K initial centers.
2. Determine the algorithm's termination conditions: maximum number of iterations or minimum movement precision.
3. Code the distance metric function.
4. Code the k-means clustering function.
5. Clustering input images using your k-means function.
6. Get the segmentation output according to the cluster result.

Input all images from `ImageSegmentation/data`, and save the comparison results into `ImageSegmentation/output/k-means` under **different termination conditions** and **different K values** like:



2.2 Image segmentation with Region Growth (25%)

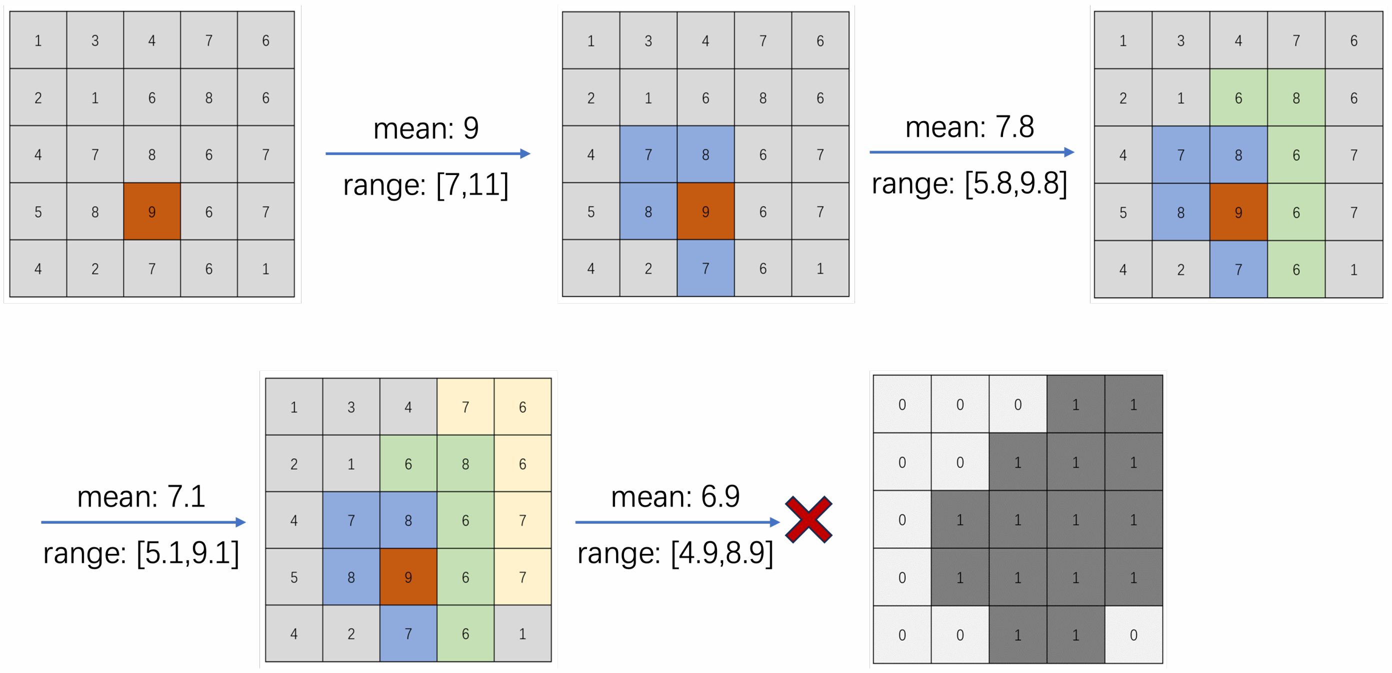
Region Growth is a traditional region-based image segmentation algorithm. Region Growth can be described as a process in which pixels or small regions are continuously merged into larger regions according to predefined growth rules. Specifically, Region Growth starts from a set of initial seed points, and by applying predefined Region Growth rules, neighboring pixels with similar properties to the seed points are continuously added to each seed point. When the termination condition of Region Growth is met, the final grown region is formed.

The steps of region growing are as follows:

1. Randomly or manually select a pixel that does not yet belong to any region, and set this pixel as (x_0, y_0);
2. Take (x_0, y_0) as the center, and consider its 8-connected neighboring pixels (x, y). If (x, y) satisfies the growth criterion with respect to (x_0, y_0), merge (x, y) with (x_0, y_0) (i.e., assign them to the same region), and push (x, y) onto the stack;
3. Pop a pixel from the stack and treat it as the new (x_0, y_0), then return to step 2;
4. When the stack is empty, return to step 1;

5. Repeat steps 1–4 until every pixel in the image has been assigned to a region. The region growing process is then complete.

The following image is a simple example of step 2~4 in region growing: the pixel with a value of 9 is selected as the initial seed point, the growth threshold is set to 2, and growth is performed using 8-connectivity. After each growth step, the mean value is updated, and then the next pixel to grow is determined. This process continues until there are no more pixels that can be grown.



You need to implement this Region Growth, and input all images from `ImageSegmentation/data`, and save the comparison results into `ImageSegmentation/output/Region-Growth` under **different growth criterions**.