Experiment No. 03:

Basics of Image Processing with Jetson Nano

By: Alan Palayil

Lab Partner: Zahra Tolideh and Lukas Klicker

Instructor: Dr. Jafar Saniie

ECE 501

Due Date: 06/08/2023

Acknowledgment: I acknowledge all the work (including figures and codes) belongs to me and/or the people who are referenced.

Electronic Signature: Alan Palayil A20447935 (Due Date: 6/8/2023)

Table of Contents

I. Introduction

A. Purpose

This technical paper presents the objectives and outcomes of a laboratory exercise focused on Python image processing programs executed on Jetson Nano. The lab aimed to familiarize students with the essential concepts of image processing, introduce them to Python-compatible image processing libraries like OpenCV, and explore the Jetson Nano-specific image processing toolkit known as CUDA.

The primary goal of this lab was to enable students to develop a comprehensive understanding of image processing fundamentals. By working through various Python programs and leveraging the functionalities offered by the image processing toolbox, students were expected to gain proficiency in programming image processing tasks. Additionally, they were encouraged to create their own image processing programs and execute them on Jetson Nano.

B. Background

The following summary provides a concise overview of the background information as outlined in the laboratory manual.

a) Description of Python Compatible Image Processing Library - OpenCV

OpenCV (Open-Source Computer Vision Library) is a versatile open-source software library designed for computer vision and machine learning applications. With a focus on providing a common infrastructure for computer vision tasks, OpenCV enables the efficient utilization of machine perception in commercial products. It boasts a BSD license, offering businesses the flexibility to utilize and modify the code to meet their specific requirements.

The library encompasses a rich collection of over 2500 optimized algorithms, encompassing both classic and state-of-the-art techniques in computer vision and machine learning. These algorithms empower developers to perform a wide range of tasks, such as face detection and recognition, object identification, human action classification in videos, camera movement tracking, object tracking, 3D object model extraction, generation of 3D point clouds from stereo cameras, image stitching for creating high-resolution panoramic images, image database similarity search, red-eye removal from flash photography, eye movement tracking, scene recognition, and marker establishment for augmented reality overlays, among many others.

OpenCV provides interfaces for several programming languages, including C++, Python, Java, and MATLAB, and supports popular operating systems such as Windows, Linux, Android, and macOS. Specifically for Jetson Nano, OpenCV offers GPU acceleration support, leveraging the device's capabilities for enhanced performance. While OpenCV primarily focuses on real-time vision applications, it takes advantage of MMX and SSE instructions when available, ensuring efficient processing. Furthermore, ongoing development efforts are dedicated to expanding the library's capabilities through a comprehensive CUDA and OpenCL interface. Currently, OpenCV provides over 500 algorithms and a multitude of supporting functions, offering extensive functionality for diverse computer vision tasks. The library is primarily written in C++ and features a templated interface that seamlessly integrates with STL containers.

In summary, OpenCV serves as a powerful and flexible tool for computer vision and machine learning tasks, equipped with a vast array of algorithms and functions. Its multi-language support, cross-platform compatibility, and ongoing development efforts make it a widely adopted choice for a variety of applications in the field of computer vision.

b) Description of NVIDIA CUDA Toolkit for Jetson Nano (CUDA 10.0)

The NVIDIA® CUDA® Toolkit, as stated on its official website [2], serves as a comprehensive development platform designed to facilitate the creation, optimization, and deployment of GPU-accelerated applications on NVIDIA GPU-powered devices, including Jetson Nano. This toolkit provides developers with a rich development environment for building high-performance applications that leverage the power of GPUs.

With the CUDA Toolkit, developers can harness the capabilities of GPU acceleration across a wide range of computing platforms, including embedded systems, desktop workstations, enterprise data centers, cloud-based platforms, and HPC supercomputers. It offers a suite of essential tools and resources, including GPU-accelerated libraries, debugging and optimization tools, a C/C++ compiler, and a runtime library for seamless application deployment. The GPU-accelerated CUDA libraries enable developers to easily incorporate accelerated computations into various domains, such as linear algebra, image and video processing, deep learning, and graph analytics. Additionally, the CUDA Toolkit provides integrations with commonly used languages and numerical packages, as well as well-documented development APIs, facilitating the development of custom algorithms tailored to specific requirements.

CUDA applications developed using the toolkit can be deployed across the entire range of NVIDIA GPU families, whether on local premises or on GPU instances in the cloud. The CUDA Toolkit includes built-in capabilities for distributing computations across multi-GPU configurations, empowering scientists and researchers to develop applications that seamlessly scale from single GPU workstations to cloud installations equipped with thousands of GPUs.

In summary, the NVIDIA CUDA Toolkit offers a comprehensive development ecosystem for creating, optimizing, and deploying GPU-accelerated applications. With its extensive range of tools, libraries, and compatibility across various computing platforms, the CUDA Toolkit empowers developers to unlock the full potential of GPU acceleration for a broad spectrum of computational tasks.

C. Preliminary Assignment

1. Familiarize yourself with OpenCV operations by thoroughly reviewing the course materials dedicated to this topic. Gain a comprehensive understanding of the various operations and functionalities offered by OpenCV for image processing tasks.
2. Deepen your knowledge of GPU acceleration using CUDA on Jetson Nano by studying the course materials that cover the basic concepts in detail. Ensure that you grasp the advantages and benefits of utilizing GPU acceleration for image processing tasks, including enhanced performance and computational efficiency.
3. Examine one of the provided scripts and provide a general description of its functionality. The script, which can be found in Appendix A, performs shape detection on an input image. It identifies the edges of the detected shapes by applying thresholding techniques and traces these edges using lines. Additionally, the script classifies each shape based on its characteristics, categorizing them as triangles, rectangles, pentagons, circles, or ellipses.
4. Prepare a set of diverse images specifically for image classification purposes. By having multiple images available, you will be able to generate more comprehensive results and gain a deeper understanding of the image classification process. Select images that cover a wide range of shapes, sizes, and complexities to ensure a thorough evaluation and analysis when examining them.

II. Lab Procedure and Equipment List

1. Equipment

- 1 x Jetson Nano single-board computer
- Internet connectivity
- I/O devices (USB keyboard/mouse, etc.)
- External Monitor with HDMI/DP port

2. Procedure

The following summary provides a concise overview of the procedure as outlined in the laboratory manual.

A. Run Image Segmentation Script and Edge Detection Script

In this section of the lab, you will be performing image segmentation using a provided program. It is essential to examine the results obtained with different operations and analyze their effects. OpenCV is required for running this program, which should already be preinstalled if you have the OS image from NVIDIA.

The following procedures need to be followed:

1. Execute the image segmentation source code with only the thresholding technique applied. The code provided below accomplishes this task:

```python
import sys

for image in sys.argv[1:]:
    print('Processing image:', image)

    img = cv2.imread(image, cv2.IMREAD_GRAYSCALE)
    ret, thresh = cv2.threshold(img, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
    out = image.rsplit('.', 1)[0] + '_thresh.png'
    cv2.imwrite(out, thresh)

    print('Finished processing')
    print('Output saved to:', out)
```

The program uses the demo image "coins.png" for image segmentation, which is the same as used in Experiment #1. The demo image and the resulting thresholding output (right) are displayed below.

After running the program, you will find the segmentation result in the "out.png" file. Examine the result and analyze the effectiveness of the image segmentation algorithm. Additionally, take note of any differences between the results obtained using Python and MATLAB.

2. Repeat the above step, but this time use your own image. Place the image you wish to perform image segmentation on in the same folder as the program and modify the `imread` variable to match the filename of your image. Execute the modified program and evaluate the results. Note that the portion of the image with lower intensity is considered the background.

3. Create and execute an edge detection script. For this procedure, you need to develop your own edge detection program. An example script is provided below:

```python
import cv2
import sys
from matplotlib import pyplot as plt

for image in sys.argv[1:]:
    print('Processing image:', image)

    img = cv2.imread(image, cv2.IMREAD_GRAYSCALE)
    edges = cv2.Canny(img, 100, 200)

    fig, axes = plt.subplots(1, 2, figsize=(10, 5))
    axes[0].imshow(img, cmap='gray')
    axes[0].set_title('Original Image')
    axes[0].axis('off')
    axes[1].imshow(edges, cmap='gray')
    axes[1].set_title('Edge Image')
    axes[1].axis('off')
    plt.tight_layout()
    plt.show()

    out = image.rsplit('.', 1)[0] + '_edges.png'
    cv2.imwrite(out, edges)

    print('Finished processing')
    print('Output saved to:', out)
```

Note that you can change the edge detection algorithm to obtain different results. Run your program and examine the outcomes, analyzing any differences compared to the results obtained from MATLAB.

B. Run Shape Extraction Script to Perform Object Recognition (OpenCV)

In this section of the lab, you will be working with a shape recognition program. Follow the procedures outlined below:

1. Execute the shape extraction program using the provided test image. Analyze the results obtained from the shape extraction program. The program, along with the test image, can be found in Appendix A. Note that the script is capable of recognizing rectangular shapes, ellipses, and circles.

```python
import cv2
import sys

if len(sys.argv) != 2:
    print('Usage: python3 shapes.py <image>')
    exit()

```

```
print('Image:', sys.argv[1])
font = cv2.FONT_HERSHEY_COMPLEX
img = cv2.imread(sys.argv[1], cv2.IMREAD_GRAYSCALE)
_, threshold = cv2.threshold(img, 192, 255, cv2.THRESH_BINARY_INV)
_, contours, _ = cv2.findContours(threshold, cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)

for cnt in contours:
    approx = cv2.approxPolyDP(cnt, 0.01 * cv2.arcLength(cnt, True), True)
    cv2.drawContours(img, [approx], 0, (0), 5)
    x = approx.ravel()[0]
    y = approx.ravel()[1]
    if len(approx) == 3:
        cv2.putText(img, "Triangle", (x, y), font, 1, (0))
    elif len(approx) == 4:
        cv2.putText(img, "Rectangle", (x, y), font, 1, (0))
    elif len(approx) == 5:
        cv2.putText(img, "Pentagon", (x, y), font, 1, (0))
    elif 6 < len(approx) < 15:
        cv2.putText(img, "Ellipse", (x, y), font, 1, (0))
    else:
        cv2.putText(img, "Circle", (x, y), font, 1, (0))

cv2.imshow("shapes", img)
cv2.imshow("Threshold", threshold)
cv2.waitKey(0)
cv2.destroyAllWindows()

out = sys.argv[1][:-4] + '_annotated.png'
cv2.imwrite(out, img)
```

2. Introduce noise to the test image by running the following script:

```
import numpy as np
import cv2
import sys
import os

mean = 0
variance1 = 0.005
variance2 = 0.01

for image in sys.argv[1:]:
    img = cv2.imread(image)
    row, col, ch = img.shape

    noise1 = np.random.randn(row, col, ch) * np.sqrt(variance1)
    noisy1 = img + noise1.astype(np.uint8)

    noise2 = np.random.randn(row, col, ch) * np.sqrt(variance2)
    noisy2 = img + noise2.astype(np.uint8)

    base_name = os.path.splitext(image)[0]
    out1 = base_name + '_noise_005.png'
    out2 = base_name + '_noise_01.png'

    cv2.imwrite(out1, noisy1)
    cv2.imwrite(out2, noisy2)
```

This program adds Gaussian noise to the image with a mean of 0 and a variance of 0.005. Pay attention to the difference in numbers.

3. Modify the shape extraction program to process the image with added noise. Evaluate the results obtained after running the modified program.

4. Repeat steps 2 and 3, but this time set the variance of the Gaussian noise to 0.01.

By following these steps, you will analyze the impact of noise on the shape recognition program and observe how varying levels of noise affect the results.

C. Image Morphological Operation and Histogram Expansion with Python (OpenCV)

OpenCV provides built-in functions for performing the four image morphological operations and histogram expansion. Follow the procedures outlined below:

1. Experiment with opening, closing, erosion, and dilation using Python (OpenCV):

You can use the following functions for each operation:

- Opening: `opening = cv2.morphologyEx(img, cv2.MORPH_OPEN, kernel)`

- Closing: `closing = cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel)`

- Erosion: `erosion = cv2.erode(img, kernel, iterations=1)`

- Dilation: `dilation = cv2.dilate(img, kernel, iterations=1)`

In the above code, `img` refers to the original image matrix, and `kernel` is a structural element/matrix that defines the direction and area of the function. You can generate the kernel using the following command:

`kernel = np.ones((5, 5), np.uint8)`

Refer to the corresponding help document [6] for more information. To perform these operations, use your own image and compare the results with the example effect image provided in the background section.

2. Perform histogram expansion using Python (OpenCV):

To apply histogram expansion in OpenCV, use the following function:

`equ = cv2.equalizeHist(img)`

In the above code, `img` refers to the original image. OpenCV automatically determines the histogram range. Perform histogram expansion using your own image and compare the result with the example effect image provided in the background section. Note that if your original image has balanced intensity, the result may be indistinguishable from the original. Below is a comparison of the image before and after histogram expansion.

[Include the comparison images here]

By following these procedures, you can explore the image morphological operations and histogram expansion using OpenCV's functions.

```python
import cv2
import sys
import numpy as np
from matplotlib import pyplot as plt

kernel = np.ones((5, 5), np.uint8)

for image in sys.argv[1:]:
    print('Processing image ' + image)
    img = cv2.imread(image)

    # Histogram Expansion
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    equ = cv2.equalizeHist(gray)
    plt.subplot(121), plt.imshow(gray, cmap='gray')
    plt.title('Original Image'), plt.xticks([]), plt.yticks([])
    plt.subplot(122), plt.imshow(equ, cmap='gray')
    plt.title('Histogram Expansion'), plt.xticks([]), plt.yticks([])
    plt.show()

    out_histogram = image[:-4] + '_histogram.png'
    cv2.imwrite(out_histogram, equ)

    # Morphological Operations
    opening = cv2.morphologyEx(img, cv2.MORPH_OPEN, kernel)
    closing = cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel)
    erosion = cv2.erode(img, kernel, iterations=1)
    dilation = cv2.dilate(img, kernel, iterations=1)

    image = image[:-4]
    cv2.imwrite(image + '_opening.png', opening)
    cv2.imwrite(image + '_closing.png', closing)
    cv2.imwrite(image + '_erosion.png', erosion)
    cv2.imwrite(image + '_dilation.png', dilation)
    print('Image processed')
```

D. Perform Video Processing with Python (OpenCV)

In this section of the lab, you will create and execute a Python program for video processing. The program will generate frames from a provided example video. You will then utilize the script from part B, which extracts shapes from the frames, to label the test video with shape markers. For more detailed information, please refer to [7].

Follow the procedures outlined below:

1. Create and execute the video processing program with the test video:

Create a video processing program that extracts frames from the test video. Run this program to generate separate frames from the video. Ensure that the generated frames are correctly stored as PNG images.

2. Develop a program to iteratively process the frames using the shape extraction script from part B:

Create a program that utilizes the shape extraction script from part B to process each frame iteratively. Label each frame with shape annotations and store the annotated frames.

3. Utilize the "generate video from frame" function to create a video processing program that generates a video with labeled shapes:

Develop a video processing program that uses the "generate video from frame" function. This program will take the annotated frames and generate a new video with labeled shapes.

By following these procedures, you will successfully create a video processing pipeline that extracts frames, applies shape extraction to label the frames, and generates a new video with labeled shapes.

III. Results and Analysis

Please refer to the attached JPG files for the images and python files for code.

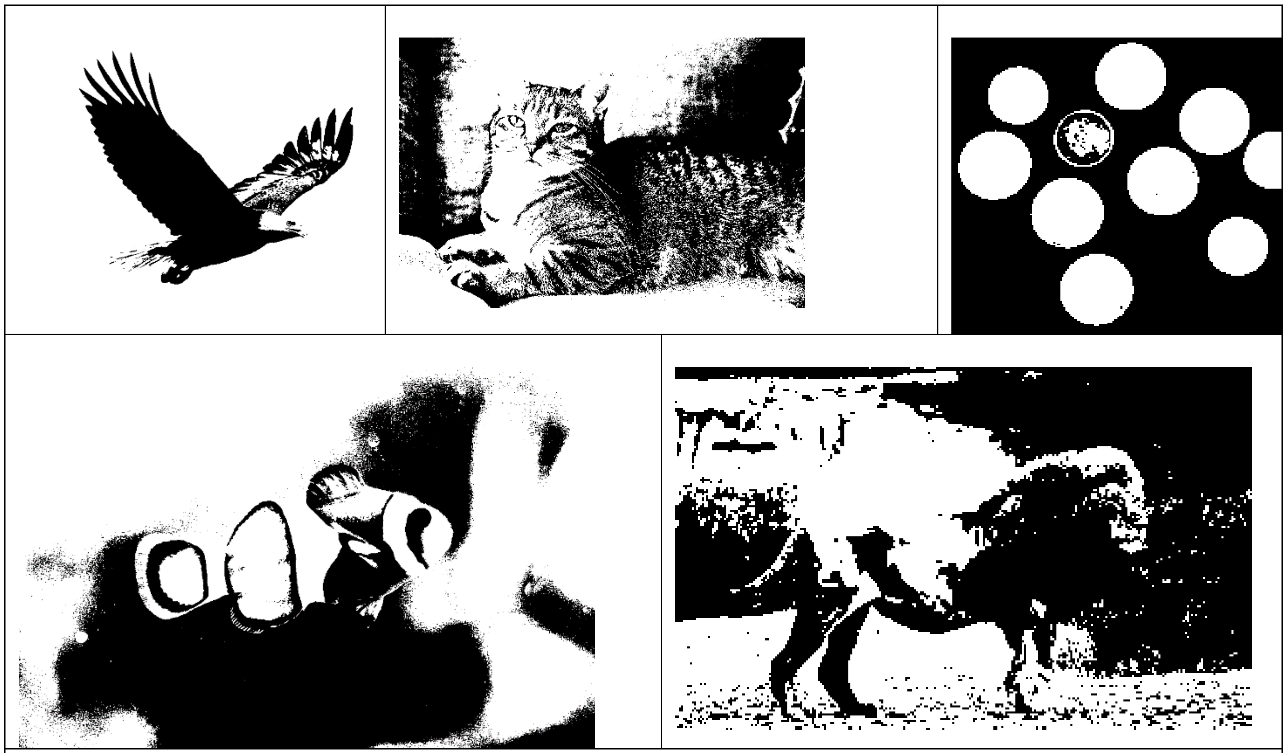


Figure 1: Image output when running Threshold

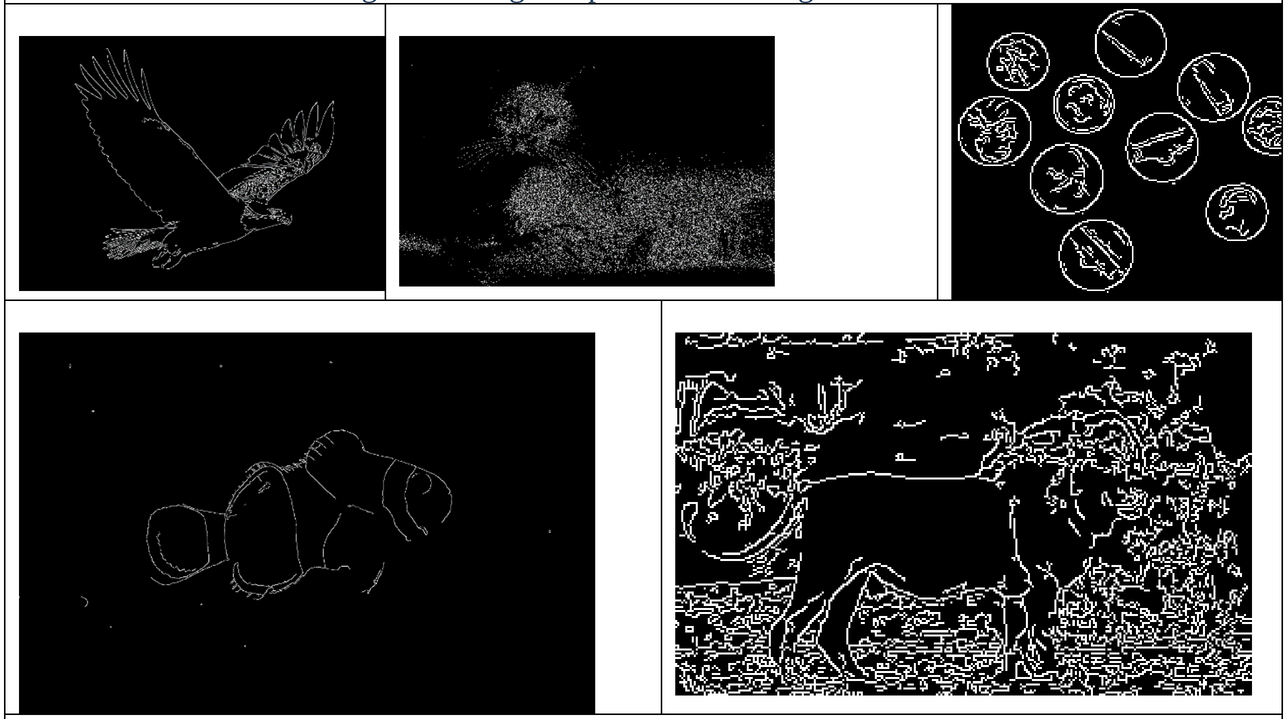


Figure 2: Image output when running Edge Detection

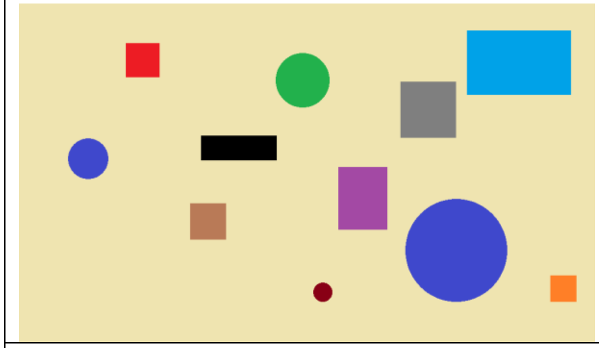
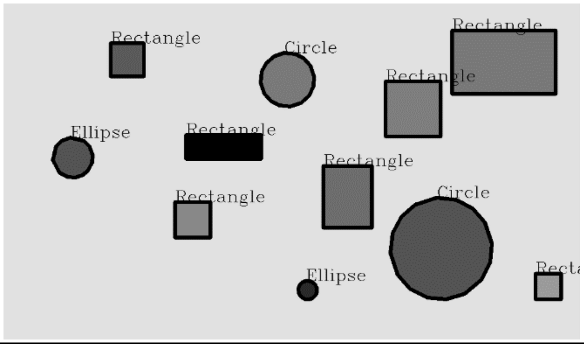
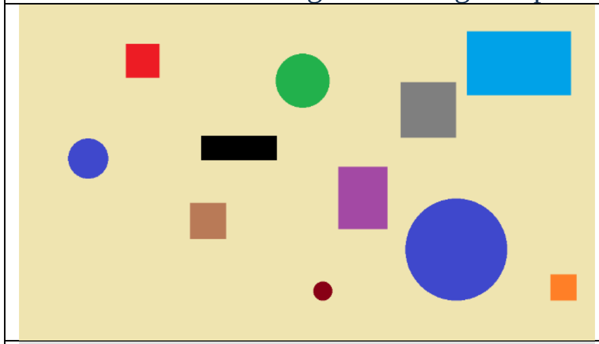
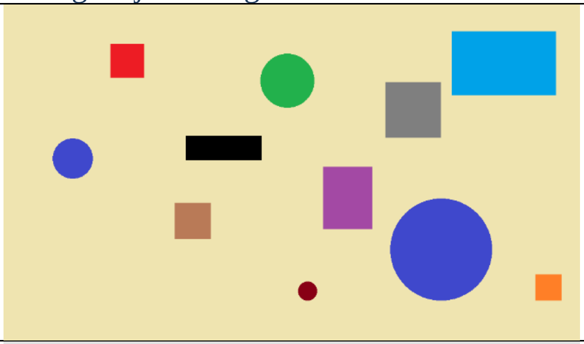
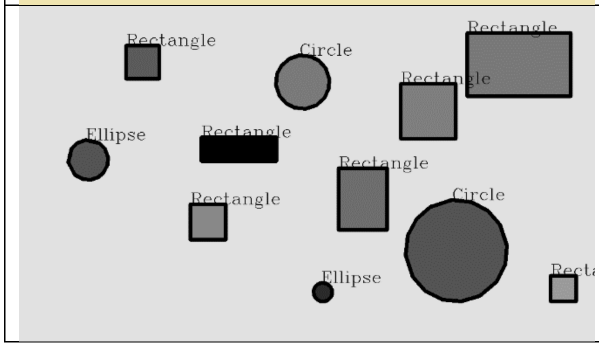
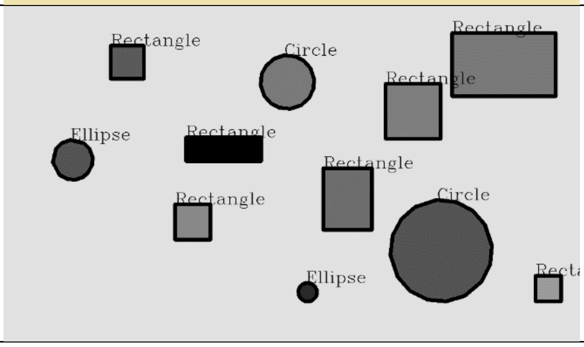
	 Rectangle Circle Rectangle Rectangle Ellipse Rectangle Rectangle Rectangle Circle Ellipse Recta
Figure 3: Image output when running Object recognition	
	
 Rectangle Circle Rectangle Rectangle Ellipse Rectangle Rectangle Rectangle Circle Ellipse Recta	 Rectangle Circle Rectangle Rectangle Ellipse Rectangle Rectangle Rectangle Circle Ellipse Recta

Figure 4: Image output with Gaussian noise 01 and 005

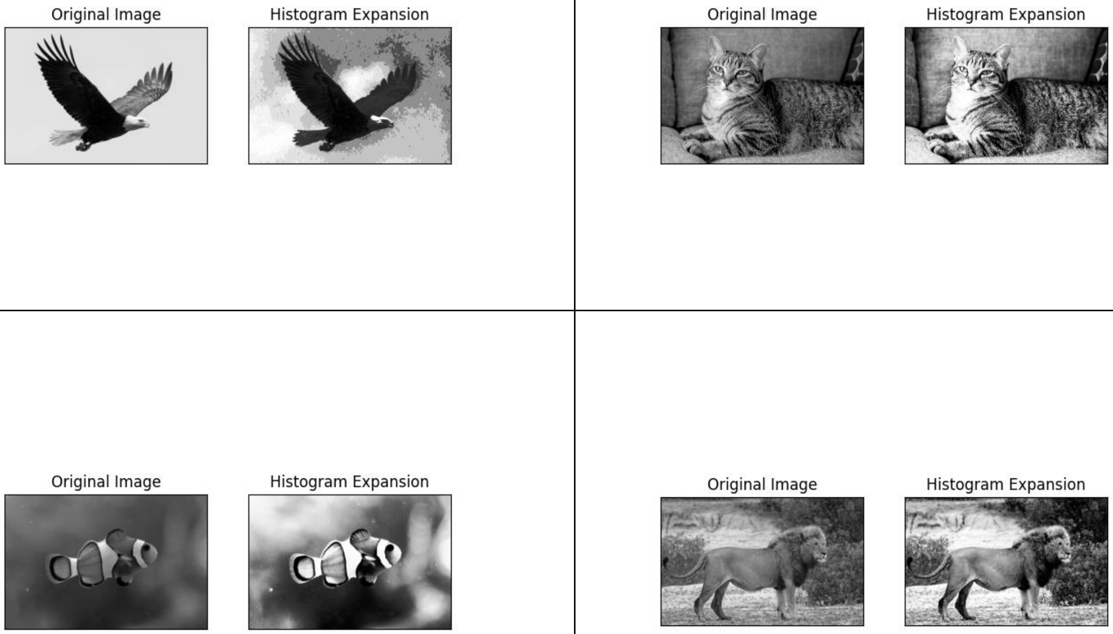
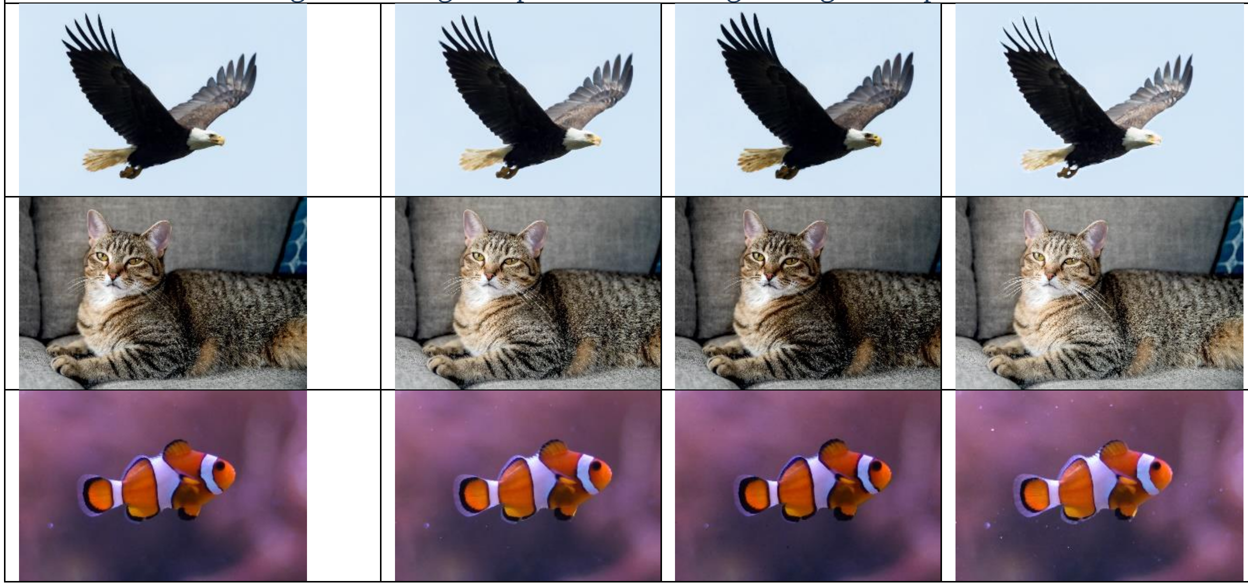


Figure 5: Image output when running Histogram expansion



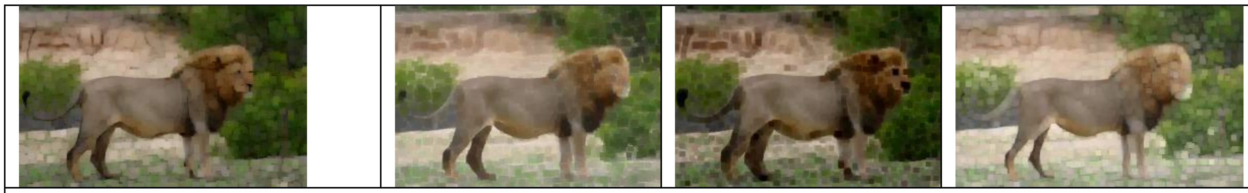


Figure 6: Image output when running Morphological operations

A. Discussion

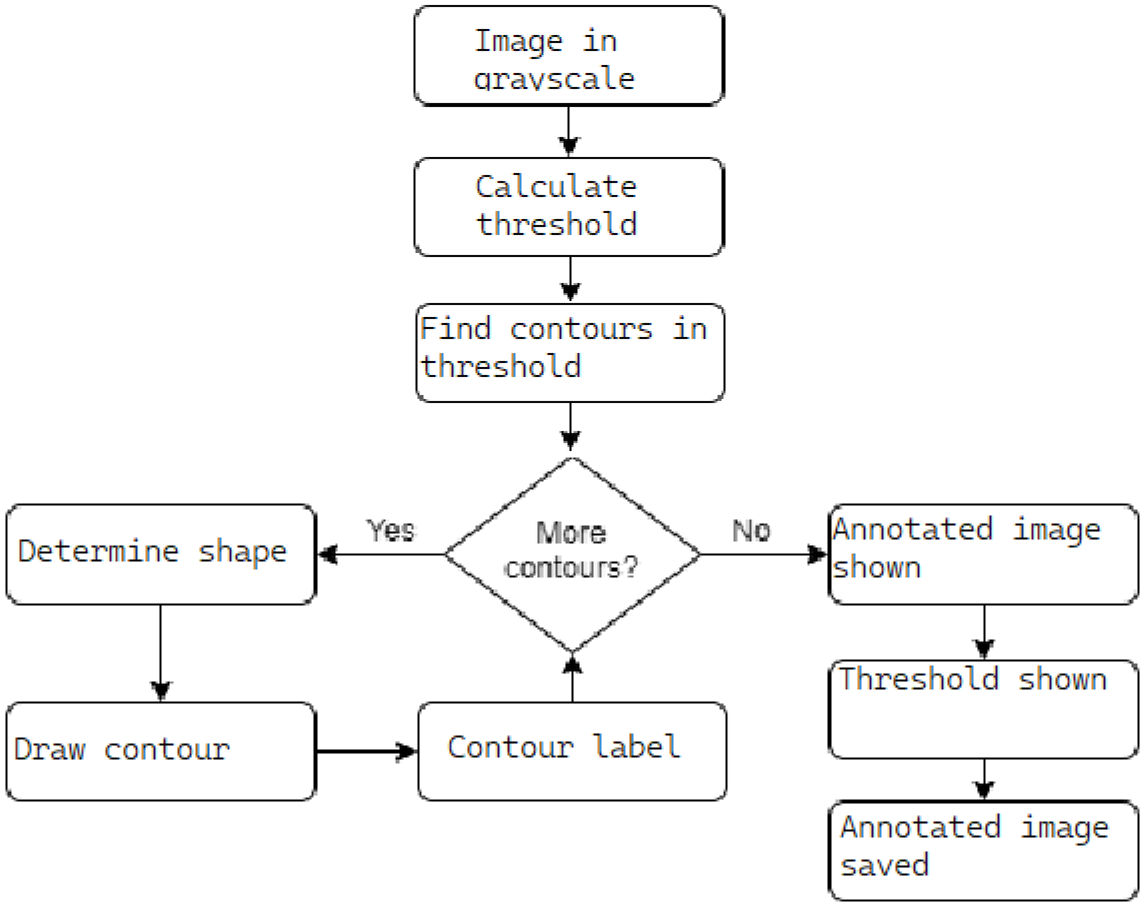
1. Comment on the advantages and disadvantages of programming with MATLAB and Python, specifically on the image segmentation script/program. (10 pts)

	MatLab	Python
Advantages	• MATLAB has a rich set of built-in functions and toolboxes specifically designed for image processing, making it easier to implement and experiment with various algorithms. • MATLAB provides a user-friendly and interactive environment with a visual interface for developing and debugging code. • MATLAB has excellent documentation and a large community of users, which makes it easier to find resources and get help when needed. • MATLAB's matrix operations and linear algebra capabilities make it well-suited for handling image data and performing mathematical operations.	• Python is an open-source language with a large and active community, providing access to numerous libraries and frameworks for image processing, such as OpenCV, NumPy, and scikit-image. • Python is a versatile language that can be used for a wide range of applications beyond image processing. • Python offers better performance optimization options, including the ability to utilize GPU acceleration effectively. • Python code is more portable and can be easily integrated with other programming languages and libraries. • Python has a simpler syntax and is easier to learn and read compared to MATLAB.
Disadvantages	• MATLAB is a commercial software and requires a license, which can be expensive for individual users or small projects. • MATLAB's performance is not always optimal, especially for computationally intensive tasks, and it may not leverage GPU acceleration efficiently.	• Python's image processing capabilities are not as comprehensive as MATLAB's, and some advanced features may require more manual implementation. • Python may require additional setup and configuration for certain libraries and dependencies. • Python's visual development environment is not as sophisticated

	• MATLAB code is not as portable as Python, as it relies on the MATLAB runtime environment. • MATLAB has limited integration with other programming languages and libraries compared to Python.	as MATLAB's, although tools like Jupyter Notebook provide interactive coding and visualization capabilities. • Python's documentation for specific image processing algorithms or techniques may not be as extensive as MATLAB's, but the active community often compensates for this limitation.

2. Draw a flow chart and explain how the shape extraction program in Appendix A works. (10 pts)

The shape extraction program in Appendix A typically follows the following flow:



The image processing program begins by loading the image and obtaining the threshold value. It then proceeds to analyze each contour present in the image to determine the corresponding shape. Each contour is drawn and labeled with its calculated shape. Once all contours have been appropriately processed and annotated, the program displays the original image along with the applied threshold. Furthermore, the annotated image is saved to a file for future reference. This process ensures that the shapes within the image are accurately identified and visually represented, allowing for easy interpretation and analysis.

3. Why are the numbers different between the MATLAB and Python noise generation script/program? (10 pts)

    The numbers used in the MATLAB and Python noise generation scripts/programs are different because they represent different statistical properties of the noise. Specifically, MATLAB uses the variance as a parameter to determine the intensity or magnitude of the noise, whereas Python uses the standard deviation, which is the square root of the variance.

    The variance measures the average squared deviation of each value from the mean, providing a measure of the spread or variability of the noise distribution. On the other hand, the standard deviation represents the average amount of deviation or dispersion from the mean, providing a measure of the spread of values in the noise distribution. Since the MATLAB script uses the variance, it directly controls the spread or intensity of the generated noise. In contrast, the Python script uses the standard deviation, which is derived from the variance. By using the standard deviation, Python ensures that the generated noise has a comparable level of intensity to the noise generated by the MATLAB script.

    Therefore, the numbers used in the MATLAB and Python scripts differ because they represent different statistical properties, but they are adjusted to achieve a similar level of noise intensity.

4. Why is GPU acceleration useful/beneficial to image/video processing in general? (10 pts)

    GPU acceleration is a highly beneficial and useful approach for image and video processing tasks. GPUs excel in parallel processing due to their large number of cores, enabling them to perform multiple calculations simultaneously. This parallel architecture results in significant speed improvements compared to traditional CPU-based processing, allowing for real-time or near-real-time processing of image and video data. GPUs are optimized for matrix operations, which are fundamental to image and video processing algorithms. They offer high memory bandwidth, efficient data transfer, and scalability by utilizing multiple GPUs in parallel.

    Furthermore, GPU acceleration is supported by specialized libraries and frameworks such as CUDA and OpenCL. These provide optimized algorithms and functions that leverage the power of GPUs, enhancing performance and simplifying development. GPUs' energy efficiency, particularly in parallel processing tasks, is advantageous for large-scale image and video processing, as they achieve higher performance per watt compared to CPUs. In summary, GPU acceleration offers faster processing times, real-time capabilities, and the ability to handle complex and demanding image and video processing tasks, making it an essential and valuable tool in this domain.

5. In the video processing part, explain why we need to generate separate frames to perform video processing. Assume the video is encoded. (10 pts)

   When working with encoded videos, the information about frame differences is stored in the video file rather than within individual frames. However, for effective video processing using image analysis programs, it is necessary to extract separate frames from the video. By converting the video into discrete images, each frame can be independently processed and analyzed. This allows for tasks such as object detection, image enhancement, and shape recognition to be performed on a frame-by-frame basis. The extracted frames provide the flexibility to apply various image processing techniques and leverage temporal information for tasks like motion detection and tracking.

   Converting encoded videos into separate frames enables the integration of video data with existing image processing libraries and algorithms. This process aligns with the standard image processing workflow, as these libraries are typically designed to operate on individual images. Additionally, by treating each frame as a standalone image, it becomes possible to leverage the parallel processing capabilities of multi-core systems or GPUs. This enables efficient and accelerated video analysis, as multiple frames can be processed simultaneously. By extracting and processing separate frames, video processing can be performed effectively, allowing for comprehensive analysis, integration with existing algorithms, and the utilization of parallel processing for enhanced efficiency.

IV. Conclusions

In conclusion, this experiment successfully fulfilled its purpose of imparting knowledge and practical skills in basic image manipulations using Python. By analyzing and modifying the given Python code snippets, students gained a comprehensive understanding of various image processing techniques, including image segmentation, object detection, morphological operations, histogram expansion, edge detection, and thresholding. Through hands-on execution of the Python code snippets, students were able to apply these techniques to manipulate images and videos effectively. This practical experience enabled them to comprehend the underlying algorithms and methods involved in each image manipulation task.

Additionally, by comparing the results obtained from Python frameworks to similar operations performed using MatLab, students were able to evaluate the advantages and disadvantages of Python for image processing. This comparison facilitated a deeper understanding of the performance, ease of use, and flexibility offered by Python frameworks in contrast to MatLab. The experiment provided valuable insights into the strengths and limitations of Python for image processing tasks. Students acquired a refined understanding of the capabilities and nuances of Python frameworks through direct implementation and evaluation.

Overall, this experiment was successful in achieving its objectives of enhancing students' understanding of basic image manipulations and enabling them to apply these techniques using Python. The hands-on nature of the experiment allowed students to gain practical experience and proficiency in image processing, making it a valuable learning exercise.

References

i. Experiment 3 Lab Manual

ii. OpenCV. About OpenCV. Retrieved from https://opencv.org/about/

iii. NVIDIA CUDA Toolkit. Retrieved from https://developer.nvidia.com/cuda-toolkit

iv. Image Segmentation with Watershed Algorithm from https://opencv-pythontutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_watershed/py_watershed.html

v. Image Thresholding from https://opencv-pythontutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_thresholding/py_thresholding.html

vi. Canny Edge Detection from https://opencv-pythontutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_canny/py_canny.html

vii. Morphological Transformations from https://opencv-pythontutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_morphological_ops/py_morphological_ops.html

viii. Getting Started with Videos from https://opencv-pythontutroals.readthedocs.io/en/latest/py_tutorials/py_gui/py_video_display/py_video_display.html

Attachments

1. *edge_detect.py*
2. *noise.py*
3. *shapes.py*
4. *morphology-histogram.py*
5. *thresh.py*
6. *video_process.py*
7. *video_generate.py*
8. *video_extract.py*
9. *bird.jpg*
10. *bird_closing.png*
11. *bird_dilation.png*
12. *bird_erosion.png*
13. *bird_histogram.png*
14. *bird_histogram_comparison.png*
15. *bird_opening.png*
16. *bird_thresh.png*

17. *cat.jpg*
18. *cat_closing.png*
19. *cat_dilation.png*
20. *cat_erosion.png*
21. *cat_histogram.png*
22. *cat_histogram_comparison.png*
23. *cat_opening.png*
24. *cat_thresh.png*
25. *coins.png*
26. *coins_edges.png*
27. *coins_thresh.png*
28. *fish.jpg*
29. *fish_closing.png*
30. *fish_dilation.png*
31. *fish_erosion.png*
32. *fish_histogram.png*
33. *fish_histogram_comparison.png*
34. *fish_opening.png*
35. *fish_thresh.png*
36. *lion.jpg*
37. *lion_closing.png*
38. *lion_dilation.png*
39. *lion_erosion.png*
40. *lion_histogram.png*
41. *lion_histogram_comparison.png*
42. *lion_opening.png*
43. *lion_thresh.png*
44. *shapes.png*
45. *shapes_annotated.png*
46. *shapes_annotated_noise_01.png*
47. *shapes_annotated_noise_005.png*
48. *shapes_noise_01.png*
49. *shapes_noise_005.png*
50. *video1.mp4*
51. *video2.avi*
52. *video_processed1.avi*
53. *video_processed2.avi*