

LAB EXPERIMENT 1:  
BASICS OF IMAGE PROCESSING WITH MATLAB

By: Gowtham Kumar Kamuni (A20549435)

Instructor: Dr. Jafar Saniie

ECE 501, Artificial Intelligence and Edge Computing

Lab Date: 05-23-2025

Due Date: 06-06-2025

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

Signature: Gowtham Kumar Kamuni

# **I. Introduction**

## **A. Purpose**

The objective of this lab was to develop a foundational understanding of image processing using MATLAB and its Image Processing Toolbox. Through a series of structured exercises, we explored key techniques such as image segmentation, edge detection, morphological operations, histogram expansion, and object classification based on shape. Additionally, the lab introduced basic video processing concepts by analyzing and manipulating video frames. The hands-on nature of the lab allowed for a practical understanding of how digital images are handled, processed, and interpreted in computational systems.

## **B. Background**

Image processing is a critical component in many modern technologies including computer vision, medical imaging, and autonomous systems. In this lab, we implemented and observed several common image processing operations using MATLAB scripts provided in three appendices. Appendix A focused on blob detection and basic object filtering using binary masks. Appendix B demonstrated shape classification through analysis of geometric properties such as bounding box and extent. Appendix C introduced video processing, where we extracted individual frames, calculated frame statistics, and generated a new video. These exercises provided valuable insight into how raw visual data can be transformed and analyzed programmatically, forming the basis for more advanced machine vision tasks.

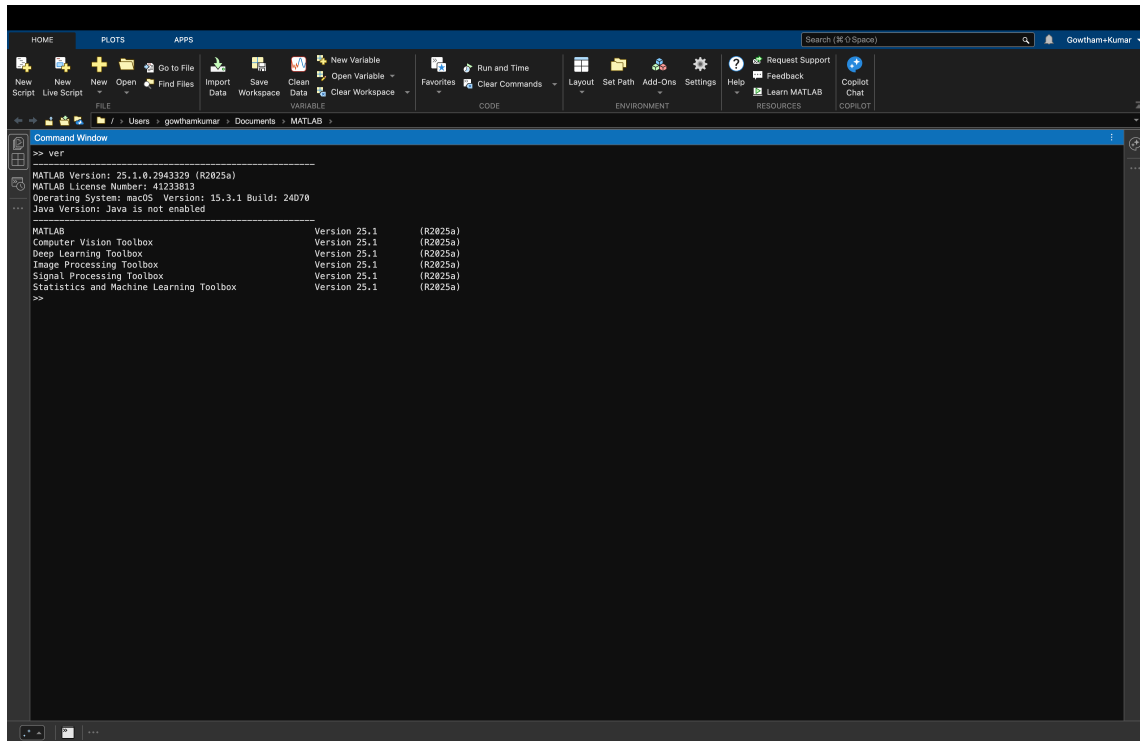
# **II. Lab Procedure and Equipment list**

## **A. Equipment**

- Laptop: Apple MacBook Air (M2, 8GB RAM, macOS 15.3.1)
- Software: MATLAB R2025a
- Toolboxes Installed:
  - Image Processing Toolbox
  - Computer Vision Toolbox
  - Deep Learning Toolbox
  - Signal Processing Toolbox
  - Statistics and Machine Learning Toolbox

## B. Procedure

MATLAB was installed and configured on a personal MacBook Air. Toolboxes were verified using the `ver` command to ensure all necessary modules were available for the lab. All provided .m files were organized into a dedicated MATLAB folder, and the test image files were moved accordingly to ensure correct file path access during script execution.



The image shows the MATLAB Command Window with the following output:

```
>> ver
-----
MATLAB Version: 25.1.0.2043320 (R2025a)
MATLAB License Number: 43233813
Operating System: macOS Version: 15.3.1 Build: 24D70
Java Version: Java is not enabled
-----
MATLAB                Version 25.1    (R2025a)
Computer Vision Toolbox Version 25.1    (R2025a)
Deep Learning Toolbox  Version 25.1    (R2025a)
Image Processing Toolbox Version 25.1    (R2025a)
Signal Processing Toolbox Version 25.1    (R2025a)
Statistics and Machine Learning Toolbox Version 25.1    (R2025a)
>>
```

*Fig: set up, MATLAB*

### III. Results and Analysis

#### A. Appendix A – Blob Detection and Filtering

This script demonstrated basic object detection in binary images using built-in MATLAB functions such as `imbinarize`, `bwboundaries`, and `regionprops`. The test image used was MATLAB's built-in 'coins.png', which contains scattered circular objects (coins) on a contrasting background.

Once executed, the script:

- Converted the grayscale image to binary
- Detected connected regions (blobs)
- Filtered these blobs based on area
- Outlined each detected object and labeled them numerically

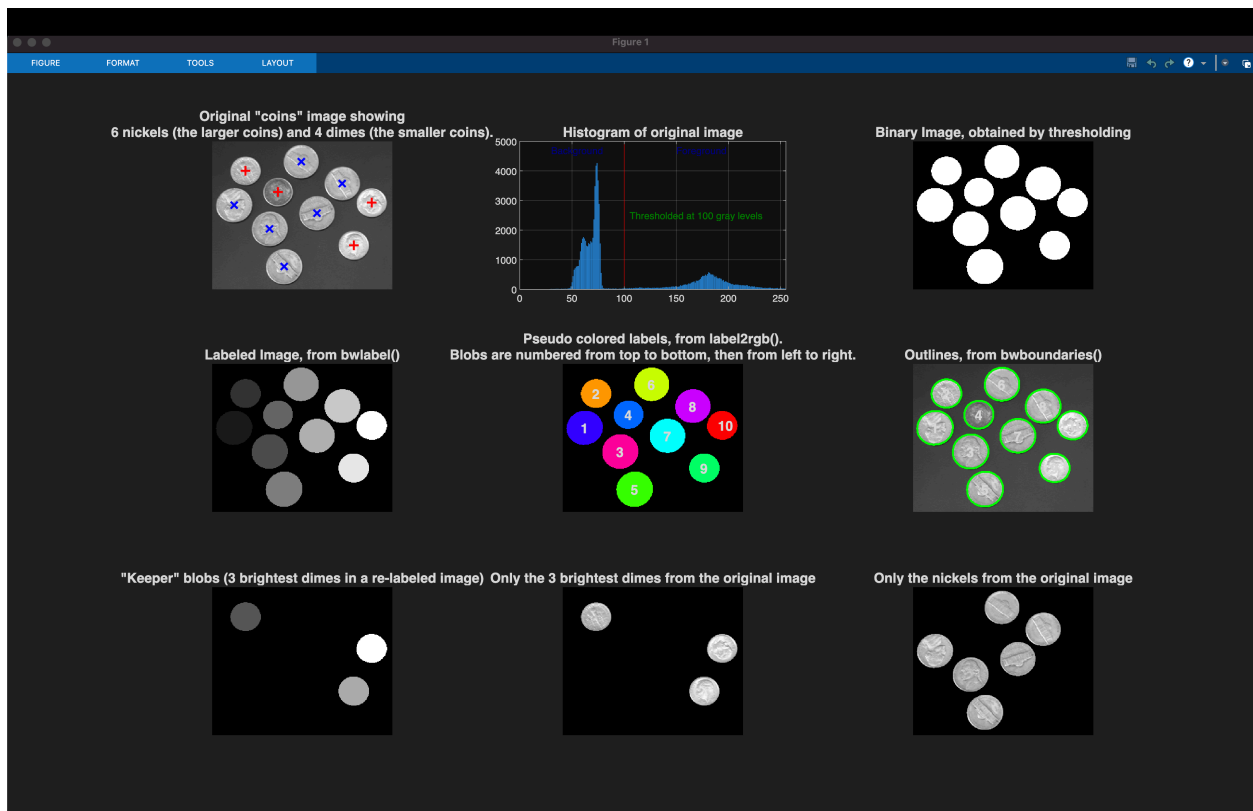
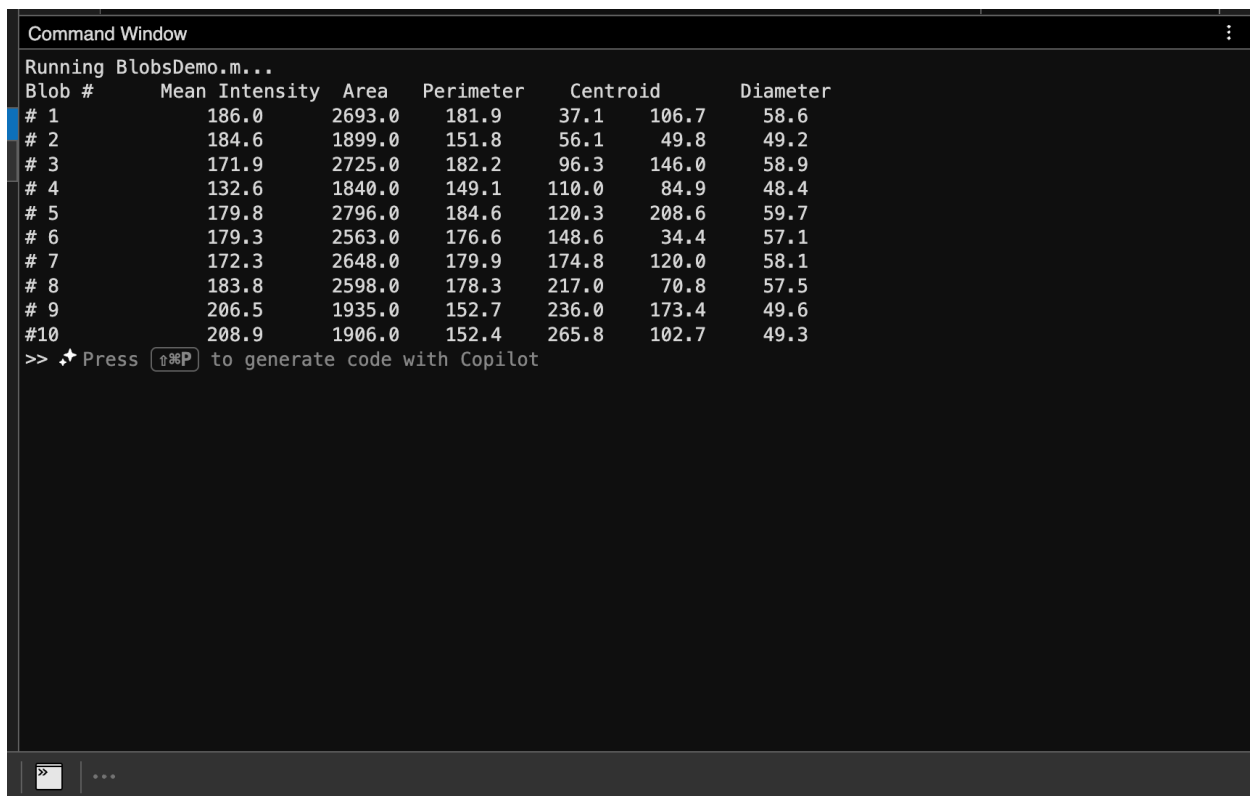


Fig 1.1: Blob Detection Pipeline Output (*BlobsDemo.m*)


Figure 1.1 shows the full processing pipeline applied to the coins.png image. It includes grayscale conversion, binary thresholding, labeling with `bwlabel`, pseudo-coloring with `label2rgb`, and boundary extraction using `bwboundaries`. The bottom row shows filtered blobs based on brightness and size, separating dimes from nickels for classification.



Command Window

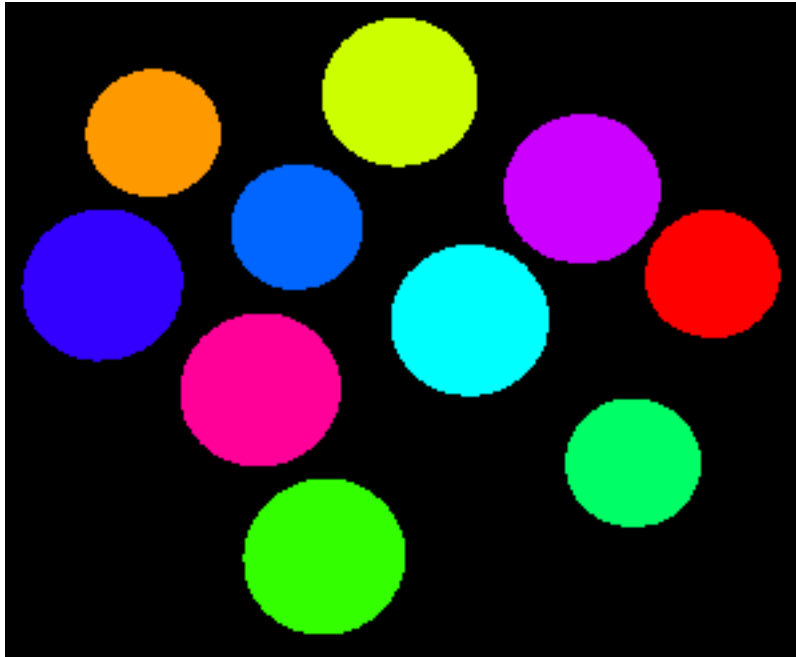
Running BlobsDemo.m...

Blob #	Mean Intensity	Area	Perimeter	Centroid		Diameter
# 1	186.0	2693.0	181.9	37.1	106.7	58.6
# 2	184.6	1899.0	151.8	56.1	49.8	49.2
# 3	171.9	2725.0	182.2	96.3	146.0	58.9
# 4	132.6	1840.0	149.1	110.0	84.9	48.4
# 5	179.8	2796.0	184.6	120.3	208.6	59.7
# 6	179.3	2563.0	176.6	148.6	34.4	57.1
# 7	172.3	2648.0	179.9	174.8	120.0	58.1
# 8	183.8	2598.0	178.3	217.0	70.8	57.5
# 9	206.5	1935.0	152.7	236.0	173.4	49.6
#10	208.9	1906.0	152.4	265.8	102.7	49.3

>> Press  to generate code with Copilot

*Fig 1.2: Blob Measurements*

The output from the MATLAB console listing the mean intensity, area, centroid, and estimated diameter of each detected blob. This provides measurable evidence for how the script distinguishes between dimes and nickels.



*Fig 1.3: Pseudo-Colored Blob Labeling Output*

This image displays all detected blobs color-coded using `label2rgb`, where each unique object is assigned a different color. This visualization makes it easy to distinguish separate regions identified in the binary mask, even before any numeric labels or filtering logic are applied.

#### **Observation:**

The blob detection and classification script successfully segmented and identified all visible coins in the image. It applied binary thresholding to isolate foreground objects, labeled them using `bwlabel`, and assigned each a unique index. Through `regionprops`, properties such as area, perimeter, centroid, and diameter were calculated.

The classification into dimes and nickels was based on area and mean intensity, with an intuitive visual output showing labeled blobs and a pseudo-colored mapping. The final grid-based display (Figure 1.3) made the results even clearer by isolating each coin and stating its classification, diameter, and area. This step-by-step output made it easier to understand how the filtering and classification logic works, even when several blobs had close sizes.

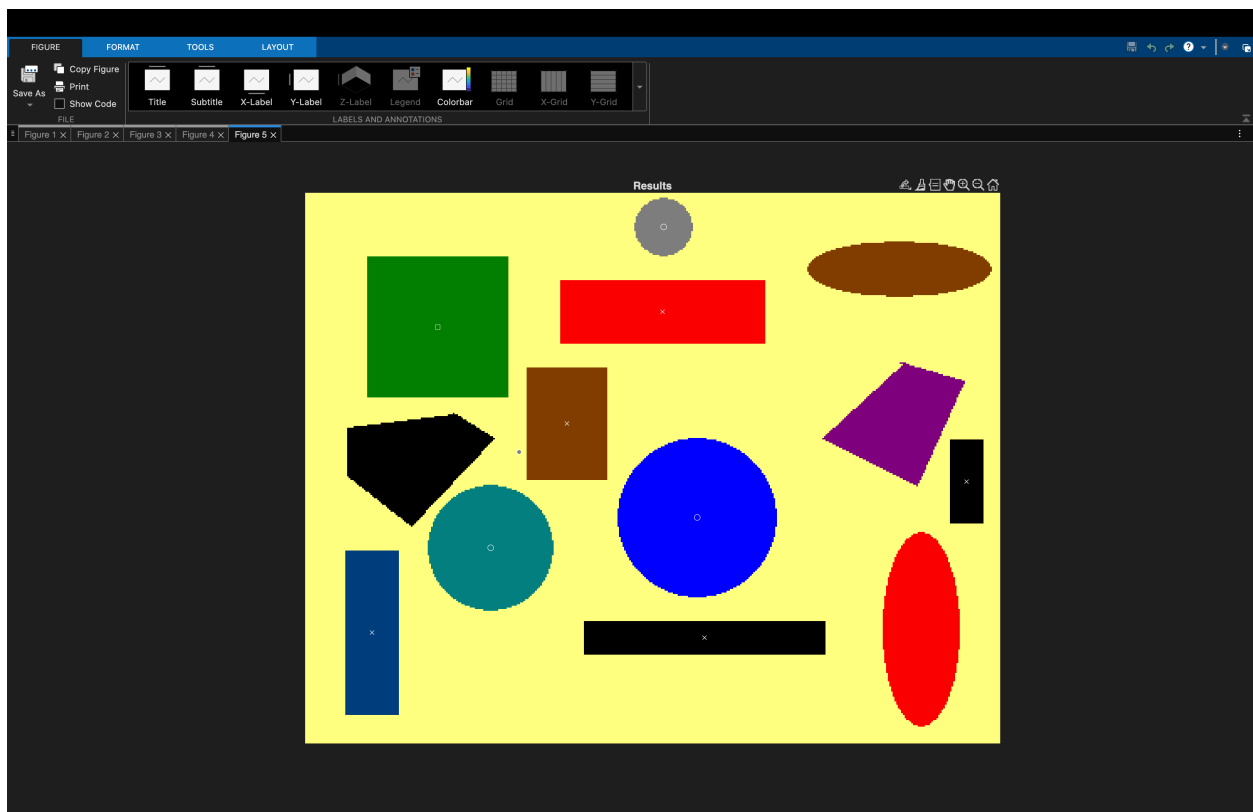
Overall, this exercise provided an excellent introduction to region-based segmentation and object classification using simple statistical properties.

## B. Appendix B – Shape Classification

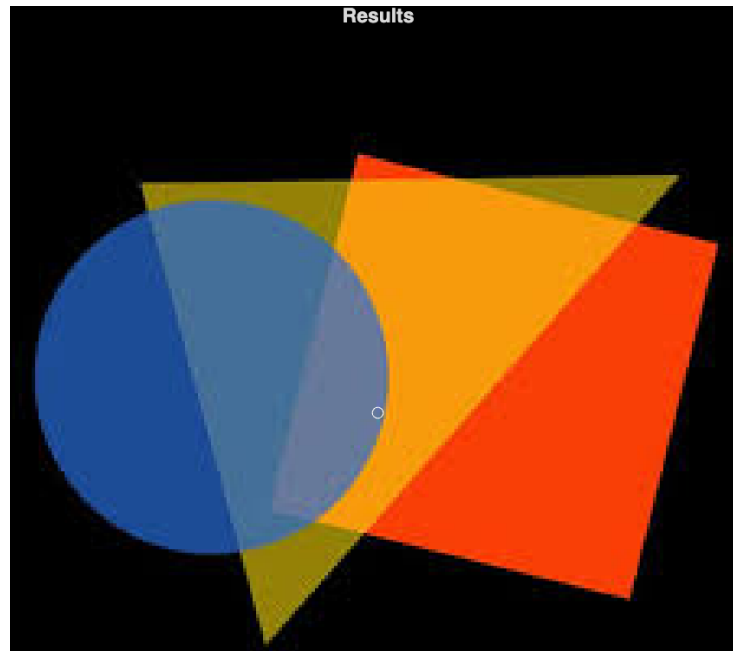
This script classified basic shapes (circle, square, rectangle) in a binary image based on their geometric properties. The classification logic relies on the BoundingBox dimensions and Extent values calculated using regionprops.

- Circle: Detected when width  $\approx$  height, and extent  $< 1$
- Square: width  $\approx$  height and extent = 1
- Rectangle: width  $\neq$  height and extent = 1
- Unknown: Anything that doesn't match above rules

We tested this script with a .bmp image provided by the professor, and additional shape images sourced online.



*Fig: Shape classification result with labeled markers.*



*Fig: Classifier output on noisy/complex shapes.*

### **Observation:**

The shape classification script worked as expected for clean, clearly separated geometric shapes. In the first test image, it correctly placed white markers (O, X, S) over basic shapes like circles, rectangles, and squares based on their bounding box dimensions and extent values. This confirmed that the classification logic was functioning correctly for typical examples.

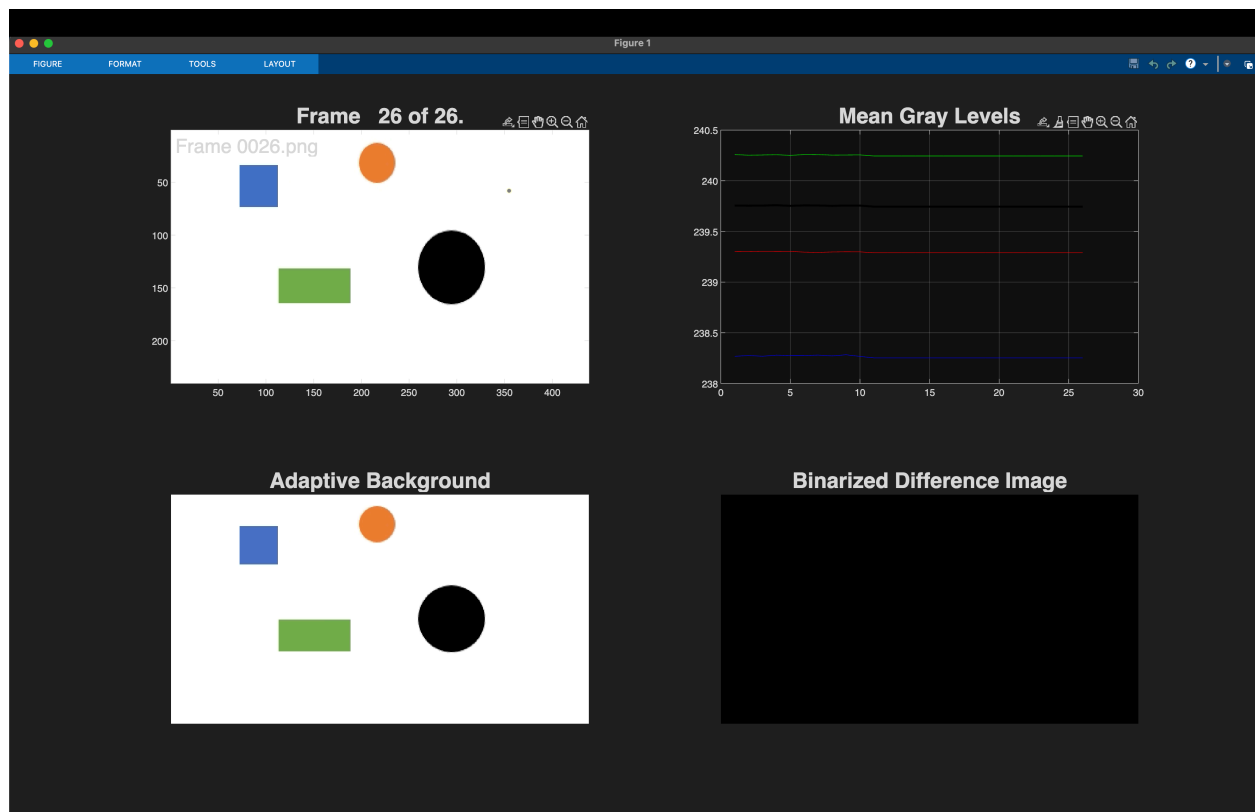
However, in the more complex and overlapping shape test, the results were more surprising. Despite significant overlap between the shapes, the script still detected a merged blob and classified it as a circle, placing a white 'O' on it. The centroid marker was slightly offset, which made sense upon realizing that the blob included merged pixels from multiple shapes. The classification logic only sees what exists in the binary mask, not the visual intention behind each shape. This reinforced the importance of using clean, well-separated inputs for shape-based classification and helped me understand the limitations of relying solely on geometry for object recognition.



## C. Appendix C – Video Frame Analysis and Background Subtraction

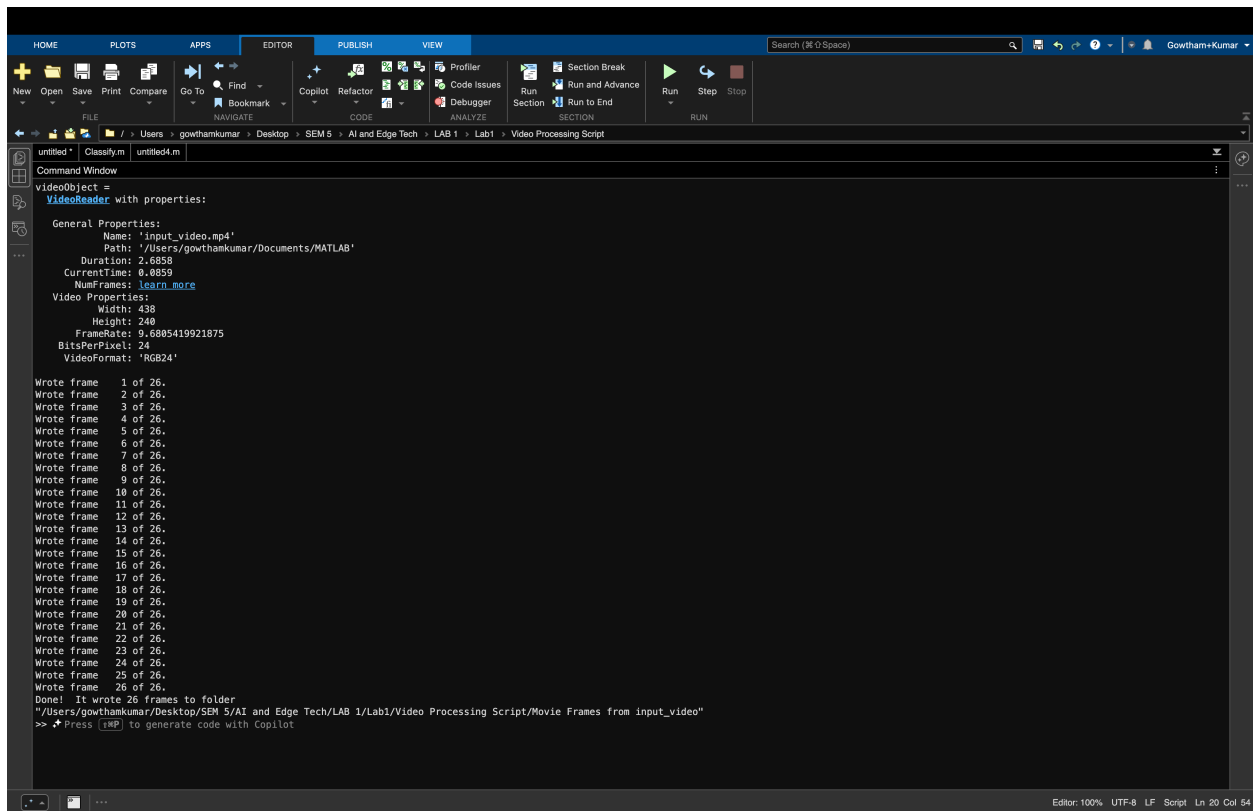
This script focused on video processing using MATLAB's VideoReader and VideoWriter functionalities. The input video was a short .mp4 file provided for the lab. The script was designed to:

- Extract individual frames from the video
- Calculate mean grayscale and RGB intensity values for each frame
- Apply an adaptive background estimation technique using a weighted moving average
- Detect moving objects by subtracting the background and applying binary thresholding
- Reconstruct and save the processed video frame-by-frame



*Fig: Live Video Processing Layout (Final Frame)*

Displays the last frame processed from the input video, alongside RGB intensity tracking, the adaptive background estimate, and the binarized difference image.

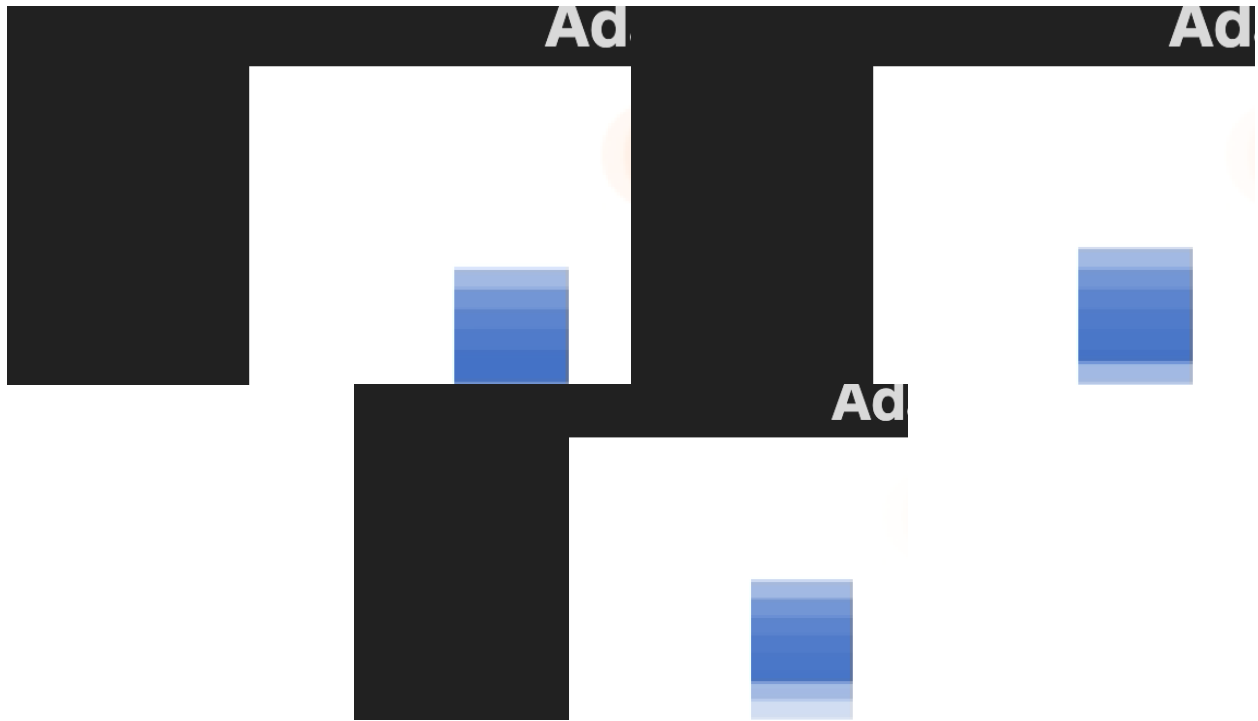
The image shows the MATLAB Command Window interface. At the top, the MATLAB toolbar is visible with tabs for HOME, PLOTS, APPS, EDITOR, PUBLISH, and VIEW. The Command Window title bar shows the file path: 'untitled' - Classify.m - untitled4.m. The Command Window content displays the following output:

```
VideoObject =  
VideoReader with properties:  
  General Properties:  
    Name: 'input_video.mp4'  
    Path: '/Users/gowthamkumar/Documents/MATLAB'  
    Duration: 2.6859  
    CurrentTime: 0.0859  
    NumFrames: learn more  
  Video Properties:  
    Width: 438  
    Height: 240  
    FrameRate: 9.6885419921875  
    BitsPerPixel: 24  
    VideoFormat: 'RGB24'  
  
Wrote frame 1 of 26.  
Wrote frame 2 of 26.  
Wrote frame 3 of 26.  
Wrote frame 4 of 26.  
Wrote frame 5 of 26.  
Wrote frame 6 of 26.  
Wrote frame 7 of 26.  
Wrote frame 8 of 26.  
Wrote frame 9 of 26.  
Wrote frame 10 of 26.  
Wrote frame 11 of 26.  
Wrote frame 12 of 26.  
Wrote frame 13 of 26.  
Wrote frame 14 of 26.  
Wrote frame 15 of 26.  
Wrote frame 16 of 26.  
Wrote frame 17 of 26.  
Wrote frame 18 of 26.  
Wrote frame 19 of 26.  
Wrote frame 20 of 26.  
Wrote frame 21 of 26.  
Wrote frame 22 of 26.  
Wrote frame 23 of 26.  
Wrote frame 24 of 26.  
Wrote frame 25 of 26.  
Wrote frame 26 of 26.  
Done! It wrote 26 frames to folder  
"/Users/gowthamkumar/Desktop/SEM 5/AI and Edge Tech/LAB 1/Lab1/Video Processing Script/Movie Frames from input_video"  
>> Press (F10) to generate code with Copilot
```

The status bar at the bottom indicates 'Editor: 100% UTF-8 LF Script Ln 20 Col 64'.

*Fig: Command Window Output and Frame Export Confirmation*

This figure displays the MATLAB command window log generated during the execution of the video processing script. It shows that the VideoReader object successfully accessed metadata from the input video file, including resolution (240x438), frame rate (29.68 fps), number of frames (26), and video format (RGB24). Following initialization, the script processes each video frame sequentially, printing a confirmation message for every frame written to disk. The final message confirms that all 26 frames were successfully extracted and saved into a specified folder. This output validates both the read and write operations involved in the frame processing pipeline.



*Fig: Frame-by-Frame Adaptive Background Evolution*

Three sequential frame snapshots that visually demonstrate how the background model gradually adapts across frames using the weighted average filter.

### **Observation:**

This part of the lab provided a practical introduction to video frame analysis using MATLAB. The script demonstrated how individual frames can be extracted from a video file, analyzed, and processed sequentially. It incorporated adaptive background modeling to estimate and update the background over time, and used frame differencing to highlight changes between frames.

Although the test video involved minimal motion, the overall pipeline showcased the core concepts of temporal analysis, frame-based statistics, and binary segmentation. The script also included functionality to save individual frames and reconstruct the output as a new video. Through this experiment, the process of handling videos programmatically, from reading and analyzing to writing results back to disk, became much clearer.

## Discussion

1. Comment on the advantages and disadvantages of thresholding and clustering as methods of image segmentation.

Thresholding is a pretty straightforward and fast method. You set a value, and everything above or below that gets segmented, it works great when the objects in the image have a clear contrast from the background. For example, in the coins image from Appendix A, thresholding helped separate the coins easily based on brightness. It's lightweight and super easy to implement.

But the downside is that it doesn't always hold up in complex or noisy images. If the lighting isn't even or if object intensities overlap, thresholding starts to fall apart. It's kind of rigid, it works well when conditions are ideal, but not when things get messy.

Clustering, like K-means, is more flexible. It groups pixels based on similarity, not just raw intensity. That helps when the image has subtle gradients or multiple regions with similar brightness. The trade-off is that it's heavier computationally, and the results can vary depending on parameters like the number of clusters. It's also a bit more sensitive to noise unless you clean the image beforehand.

So overall, thresholding is fast and simple but limited, while clustering is smarter and more adaptable, just needs more effort and resources.

---

3. Why are there differences between the results of two edge detection algorithms?

Different edge detection methods give different results because each one treats edges and noise differently. In our lab and lecture, we compared Canny and Prewitt, and the difference is very noticeable.

Canny is more sensitive, it's designed to pick up fine details and subtle changes in intensity. That's why it sometimes feels like it's picking up too much, especially inside objects like the apple we saw. But that's not just noise, it's actually capturing tiny texture differences that other filters ignore. If the image isn't blurred before running Canny, it can end up highlighting details we don't want. But overall, it's powerful because it gives thin, clean, and continuous edges, especially around complex shapes.

Prewitt is much simpler. It only looks at big intensity changes, so it's not as detailed. It's faster and less sensitive, but that also means it misses weak or subtle edges, like the bottom edge of the apple touching the table. It might look "cleaner," but only because it's ignoring a lot.

So, the differences in their outputs come down to how they balance detail vs. simplicity. Canny digs into every corner, while Prewitt keeps it surface-level. Depending on the image and what you're trying to detect, either one could be the better choice.

---

4. Explain the differences in shape extraction with different levels of noise. How can you reduce the effect of noise?

When there's noise in the image, shape extraction can go completely sideways. The cleaner the image, the easier it is to extract edges and boundaries. But once noise kicks in, especially random speckles or grain, the algorithms start picking up edges that don't even belong to real objects.

If you've got a clean image, something like a circle or rectangle comes out exactly as expected. But with high noise, you'll either lose parts of the shape or the algorithm starts detecting a bunch of false outlines around it. Some methods, like Prewitt or Sobel, don't really care, they just respond to intensity changes. So they end up catching noise as if it's part of the shape. On the other hand, methods like Canny are more precise but only work well if you help them out first by smoothing the image.

To reduce the effect of noise, the first thing I'd do is apply a Gaussian blur before edge detection. It softens out all those tiny distractions. After segmentation, morphological operations like erosion or dilation can clean up the boundaries and help connect or simplify shapes. Also, if you're working with color, sometimes one channel (like grayscale or red) gives a cleaner separation.

Bottom line, noise throws off everything, but with the right prep and cleanup, you can still get solid results. It's all about making sure the algorithm sees the shape, not the chaos around it.

---

5. In the video processing script, how can you reduce the number of generated frames while still keeping as much information as possible?

The key to reducing the number of frames without losing important stuff is all about efficiency and awareness. You don't want to blindly cut frames and miss motion or changes, you want to keep the ones that matter.

One solid way is to use frame differencing, compare each frame to the previous one. If there's barely any change between them, that frame is probably not adding much value. You skip or drop it. But if a new object enters the scene, or something moves, you keep that frame. That way, you're focusing on events, not just time.

Another way is motion detection or thresholding, keep frames where pixel differences cross a certain limit. You can even do region based checking (like background subtraction) and keep frames only when something enters or leaves a region of interest.

Also, you can downsample, for example, save every 3rd or 5th frame instead of all 30 per second, but only if the video is consistent and doesn't miss key moments. It's about balance.

End of the day, it's about being smart, keep the frames that add value, skip the ones that repeat. That saves memory, speeds up processing, and still lets you capture what matters.

---

2. Draw a flow chart and explain how the image segmentation script in part A works.

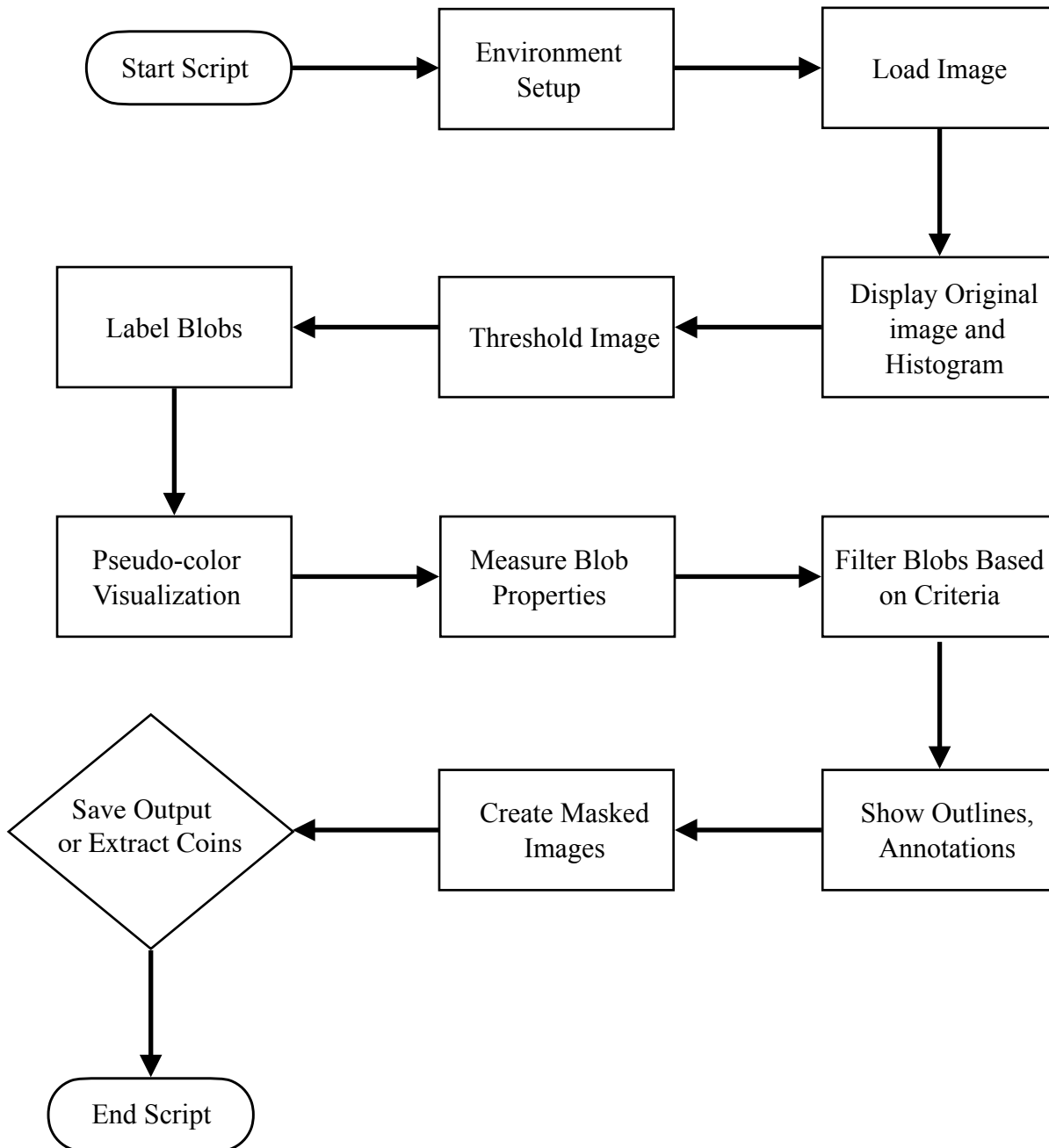
This script was pretty solid and straightforward. It's designed to take a grayscale image, in this case, the classic coins, and go through the steps of segmenting, labeling, analyzing, and filtering blobs (coins). It starts by making sure the environment is clean and everything needed is ready, like the toolbox and image file. Then it loads the image, and if it's not already grayscale, it converts it. After that, it thresholds the image to get a clean binary version, which is just a black and white version that separates foreground from background.

From there, it fills holes (in case some objects are incomplete), and labels each connected group of white pixels as individual blobs. Then it visualizes the blobs with different colors to make things clearer.

After that, it measures properties of each blob, things like size, shape, intensity, and filters out the ones we care about, based on how bright or large they are. That's how it separates dimes from nickels.

In the final part, it shows only the selected blobs, like the three brightest dimes or larger nickels, and gives the option to save the output or even crop each blob out as an individual image. (Shown in the demo video)

**Flow Chart:**



## IV. Conclusion

This lab gave me a solid, hands-on understanding of how image segmentation, shape classification, and video frame analysis work in practice. Each appendix tackled a unique challenge, from basic blob detection and property measurement to recognizing different geometric shapes and applying adaptive background subtraction in video streams.

MATLAB made it easy to visualize every stage of the process. The scripts weren't just black-box operations, they helped me see how the logic flowed, how grayscale conversion impacts segmentation, how extent and bounding box differences guide shape classification, and how motion detection benefits from filtering frame-wise differences. Honestly, it felt good to connect what we learn in theory to something I could actually run and understand visually.

All in all, this lab didn't just help me finish an assignment, it helped me learn something I'll probably use again, especially if I work more in image or video processing in the future.

## VII. References

1. MATLAB Documentation: <https://www.mathworks.com/help/>
2. Image Processing Toolbox – MathWorks
3. Computer Vision Toolbox – MathWorks
4. Lab 1 Manual and Appendix Files – Provided by Instructor
5. “BlobsDemo” and educational resources from MathWorks (used in Appendix A)