ARL-TN-1277 ● SEP 2025

Computer Automated Measurement System (CAMS)

by Gael Mota, Clara Mock, and Stephen Cluff

NOTICES

Disclaimers

The research reported in this document was performed in connection with contract no. W911NF21F0038 with the U.S. Army Combat Capabilities Development Command Army Research Laboratory. Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TN-1277 ● SEP 2025

Computer Automated Measurement System (CAMS)

Gael Mota
DoD HBCU/MI Summer Research Internship
Chitra Productions LLC
Virginia Beach, VA

Clara Mock and Stephen Cluff
DEVCOM Army Research Laboratory

REPORT DOCUMENTATION PAGE

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED	
		START DATE	END DATE
September 2025	Technical Note	6/2/2025	8/8/2025

4. TITLE AND SUBTITLE
Computer Automated Measurement System (CAMS)

5a. CONTRACT NUMBER	5b. GRANT NUMBER	5c. PROGRAM ELEMENT NUMBER
W911NF21F0038		
5d. PROJECT NUMBER	**5e. TASK NUMBER**	**5f. WORK UNIT NUMBER**

6. AUTHOR(S)
Gael Mota, Clara Mock, and Stephen Cluff

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
DEVCOM Army Research Laboratory ATTN: FCDD-RLA-MD Aberdeen Proving Ground, MD 21005	ARL-TN-1277

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited.

13. SUPPLEMENTARY NOTES
ORCID: Clara Mock, 0000-0001-8055-2371

14. ABSTRACT
Additive manufacturing (AM) provides unparalleled design freedom for valuable applications, from medical prosthetics to military equipment. However, the quality of AM products is critically dependent on processing parameters like layer thickness and scanning speed. While artificial intelligence and machine learning (AI/ML) are powerful tools for optimizing these parameters by learning from measured data, its application is hindered by the manual characterization of printed samples needed to build large datasets. This work presents the Computer Automated Measurement System (CAMS), a computer vision-based solution for high-throughput automated characterization of AM samples. Validation revealed that the difference between CAMS and manual measurements by FIJI was less than 2% for area measurements and less than 4% for aspect ratio measurements, while processing samples 198% faster than manual methods. This demonstrates the feasibility of fully automated high-throughput characterization, thereby enabling the rapid dataset generation required to accelerate AI/ML-guided refinement of AM processes.

15. SUBJECT TERMS
computer vision, additive manufacturing, image processing, dataset generation, automation, Sciences of Extreme Materials

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
a. REPORT	b. ABSTRACT	C. THIS PAGE		
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UU	46

19a. NAME OF RESPONSIBLE PERSON	19b. PHONE NUMBER *(Include area code)*
Clara Mock	(520) 691-5149

STANDARD FORM 298 (REV. 5/2020)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

List of Tables

1. Introduction

Additive manufacturing (AM) can rapidly manufacture unique tools, devices, and components using hybrid materials, which has led to its widespread adoption in the military, medical, aerospace, and other fields. Despite these advantages, each new AM application requires extensive parameter optimization—a process that currently relies on iterative experimentation to determine ideal settings for laser power, scanning speed, layer thickness, and other critical variables. To resolve this, artificial intelligence/machine learning (AI/ML) has been able to optimize these parameters without requiring extensive experimentation due to its ability to identify complex relationships between process parameters and product qualities.[1] Still, using AI/ML for optimizing AM process parameters requires that a large dataset of AM sample characteristics be created for the training process. Doing so with existing characterization methods that involve manual measurements of samples introduces a bottleneck, as the generation of these datasets would be similarly time-consuming. Thus, this project sets out to create a system that can automatically characterize AM samples.

Computer Automated Measurement System (*CAMS*) is a command line interface (CLI) program that aims to rapidly generate datasets of AM sample characteristics to be used to train AI/ML models for the optimization of AM process parameters. It is based on the Python programming language, utilizing the *OpenCV* library for image analysis, along with the *pandas* library to store the measured data into a CSV file for later analysis.

2. *CAMS* Script

CAMS works as a Python script that can rapidly characterize sample measurements from photos and videos. Using a set of arguments and flags, the user can change how the script stores debug image output, results, the scale used for analysis, and the minimum size of sample to be considered for analysis, among other parameters. When a file is loaded into the program, it detects its type (photo or video) and enables or disables the use of certain flags based on the operations those flags are based on. *CAMS* is divided into two Python files: *processing_functions.py*—which contains the image analysis code, and *CLI.py*—which contains the implementation of the CLI. Text copies of *processing_functions.py* and *CLI.py* can be found in Appendixes A and B, respectively.

2.1 Installation

CAMS' first dependency is Python. On macOS and Linux, it can be installed using the platform's package manager (e.g., "`brew install python`" or "`sudo apt install python`"). On Windows, users can either use a package manager like *Scoop* and run "`scoop install python`", or they can navigate to download the link to Python found here: https://www.python.org/downloads/. This project was developed using Python 3.12.2, but users should ensure that the Python version installed works with the following packages: *OpenCV* and *pandas*. To install these packages, the user must first create a Python *virtual environment (venv)*. To do this, run: `python -m venv CAMSvenv`. This creates a folder named `CAMSvenv` where the installed libraries will be stored. Then the virtual environment must be activated by either running `./CAMSvenv/Scripts/Activate.ps1` on Windows or `./CAMSvenv/bin/activate` on Linux/macOS. This step will make *CAMSvenv* the active virtual environment where packages can be installed. The last step in the installation process is to install the dependencies listed in *requirements.txt* using the command `pip install -r requirements.txt`. A text copy of *requirements.*txt can be found in Appendix C. The package manager, pip, will go through each listed package and install it into the *CAMSvenv* virtual environment—this installs the package locally in the *CAMS* directory, not globally across all Python environments on the host system.

2.2 File Structure

By default, the *CAMS* file structure contains the two Python files in the root directory (*CLI.py* and *processing_functions.py*). Additionally, *README.md* is a markdown file that contains detailed instructions for how to install and operate the *CAMS* CLI. A copy has been provided in Appendix D. When *CAMS* is run using default flags and arguments, a folder named *debug* is created to store the debug output images. Additionally, "contours_stats.csv" is created, which stores area and aspect ratio (AR) for each sample in each image or video.

2.3 Arguments and Flags

Table 1 showcases all the flags and arguments that can be passed to *CAMS*. Each of the flags extends the functionality of *CAMS*, enabling more finely tuned analysis when necessary.

Table 1. Flags and arguments for *CAMS*.

Flag	Long form	Type	Default	Description
Required				
input	N/A	string	N/A	Path to input video or image file
Output control				
-o	--output	string	CLI_test/	Output directory for images and debug files
N/A	--csv	string	contour_stats.csv	Path for CSV file with article measurements
Video processing				
-i	--interval	float	10	Frame interval in seconds (mutually exclusive with -t)
-t	--time-points	string	N/A	Specific time points as comma-separated values (e.g., "0,15,30.5")
Analysis parameters				
-m	--min-area	integer	850	Minimum contour area in pixels for detection
-s	--scale	integer	1	Pixel scale factor (pixels/inch) for measurements
N/A	--dark-background	flag	False	Ensures proper handling of images with dark backgrounds
Filtering options				
N/A	--exclude-edges	flag	False	Exclude particles touching on image borders

2.3.1 Input File Handling

- input (required argument)
    - Accepts video or image files
    - *CAMS* automatically determines operating mode based on file extension
    - Image files: video processing flags (--interval, --time-points) are disabled
    - Video files: additional time-based processing flags become available

2.3.2 Video Processing Flags

- interval
    - Specifies the time interval (in seconds) between frame captures
    - Default: 10 s
    - Use case: hardware systems where samples move through camera view at constant speed
    - Ensures unique data storage by capturing new sample sets at each interval
- time-points
    - Alternative to --interval flag

- Specify exact time points (in seconds) for frame extraction and analysis
- Use case: samples moving through camera view at nonconstant speeds
- Particularly useful for testing scenarios

2.3.3 Analysis Parameters

- min-area
  - Sets minimum sample area threshold (in pixels)
  - Default: 850 pixels
  - Purpose: filters out small artifacts (dust, debris) from dataset
- scale
  - Conversion factor from pixels to square units
  - Default: 259 (arbitrary value from testing setup)
  - Note: Users should measure and update this value for their specific setup
- exclude-edges
  - Optional flag
  - When set: excludes objects touching frame edges from analysis
  - User discretion is recommended based on sample positioning
- dark-background
  - Affects morphological-close processing step
  - When set: maintains white foreground objects on black background (no inversion after thresholding)
  - Use when samples appear lighter than background

Together, an example of the script's usage can be found in Figure 1. In this case, samples are analyzed in the video file "microscopy_video.avi" where the code selects a frame every 30 s, samples located on the edge of the video frame are excluded, and the scale is 300 pixels/inch.

```
# Process every 30 seconds, exclude edge particles, custom scale
python CLI.py microscopy_video.avi -i 30 --exclude-edges -s 300
```

Figure 1. Example of the *CAMS* script's usage on a video.

3. *CAMS* Image Analysis

When *CAMS* runs, it applies a series of transformations with the goal of deriving AR and area from each sample present in an image and storing the results. Figure 2 shows the process involved in analyzing frames with *CAMS*.

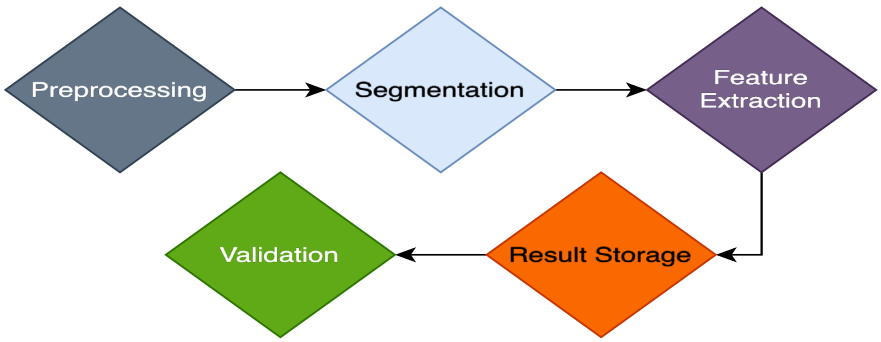

Figure 2. Flowchart outlining the image analysis process.

CAMS applies a total of five transformations to each frame to characterize the samples. Initially, each frame or image is in red, green, and blue (RGB) color, necessitating that it is converted to black and white to allow it to be processed by *OpenCV*. Figure 3 shows the original image and the image after being converted to grayscale using the *OpenCV* function `cvtColor()`.

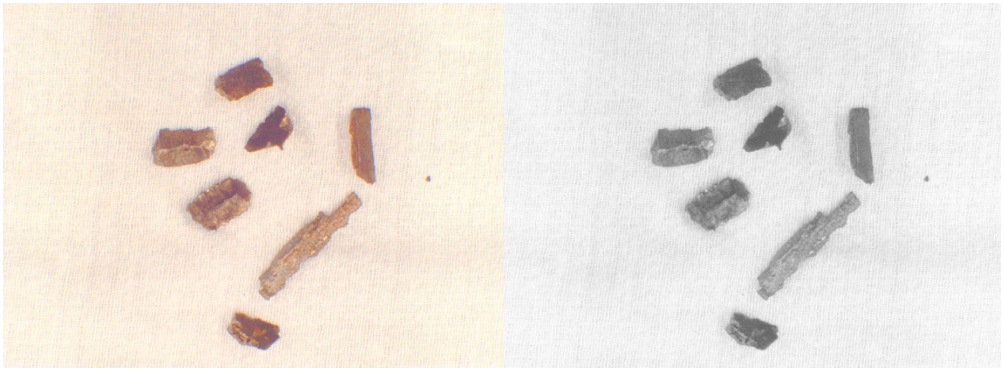

Figure 3. RGB image of samples (left) and grayscale (right).

Next, a Gaussian blur was applied to the image to filter out noise and smooth fine details that could interfere with sample detection. *CAMS* uses *OpenCV's* `GaussianBlur()` function with a 5 × 5 kernel to try and remove noise while retaining the structure of the particles. Figure 4 shows the image from Figure 3 (right) after the Gaussian blur was applied.

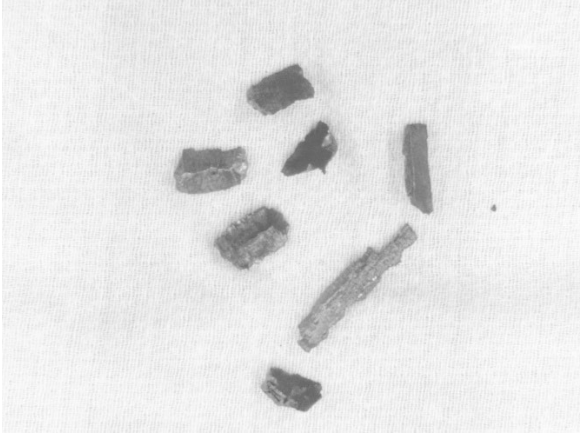

Figure 4. Samples after having Gaussian blur applied.

After the blur operation, objects were segmented from the background by applying a *threshold*. In essence, a grayscale value (0–255) was found such that all values darker than the threshold were made black, and any values above the value turned white. This part of the process distinguished between the background and the samples. To accomplish this, the `threshold()` function in *OpenCV* was used. Additionally, the argument `cv.THRESH_OTSU` was passed to tell the function to use Otsu's method to automatically determine the ideal threshold based on the histogram of the image (ensuring to preserve as much of the foreground objects as possible with the selected threshold value). Figure 5 (left) shows the histogram of Figure 4, which was automatically used to select an appropriate threshold value. The thresholded image is provided in Figure 5 (right).

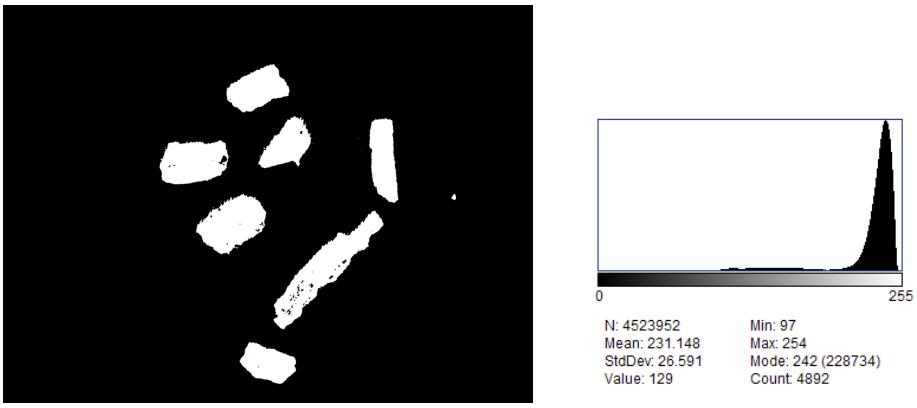

Figure 5. Otsu thresholded samples with histogram of base image.

Morphological closing was used to fill small gaps and holes within the boundaries of each sample. An elliptical structuring element (to accommodate the irregular shapes of the samples) was used with a 5 × 5 kernel to connect pieces of samples around the holes. This helps ensure that the samples appear as solid, continuous objects rather than fragmented regions (Figure 6).

Figure 6. Image after closing operation. Notice that some holes remain.

Finally, contour detection identified the boundaries around each sample in each image. To do this, *OpenCV's* `findContours()` function was used with the `RETR_EXTERNAL` parameter set to extract only the outermost contours, providing precise sample outlines that can be measured and analyzed. From these contours, *CAMS* extracted the area and AR values. An image of the contours drawn around samples in the original image is provided in Figure 7.

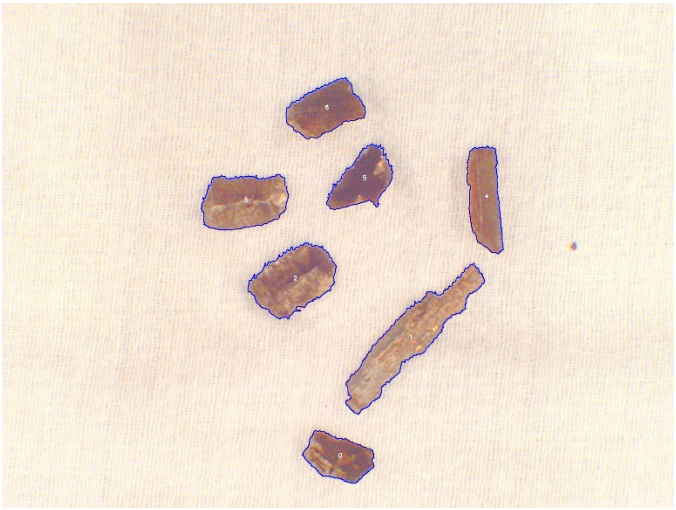

Figure 7. Original image with contours drawn around the samples.

From these contours, two *OpenCV* functions (`contourArea()` and `fitEllipse()`) were used to measure the area and AR of each sample's contour. Area was measured and converted to square inches using the *scale* parameter (either the default value of 1 or a user-specified value). The AR overlays the best fitting ellipse over each sample and divides the major by minor axis, giving the AR. Using *pandas*, the area and AR were stored for each sample in a "dataframe" that was

then converted into a CSV file and stored in the location specified with the *csv* parameter from the CLI.

4. Experimental Validation

To validate *CAMS*, its results were compared against manual measurements using *Fiji,* an open-source, batteries-included distribution of the particle analysis software *ImageJ*.[2] This experiment analyzed the differences in analysis between the two tools when used on the same images of samples. To take the pictures of these samples, a macrophotography camera was used on an adjustable stand. The samples were laid out on a solid-colored cloth. Each sample was laid onto the cloth ensuring that no samples were touching, and the exposure of the camera was adjusted to ensure sufficient contrast. Two images with 20 samples each were taken, with a ruler being used to set the scale.

To analyze with *Fiji*, a process analogous to the pipeline of transformations performed with *CAMS* was applied. First, the image is converted to 8-bit grayscale, followed by applying a blur, a manual threshold was applied and then the "analyze particles" option was used to characterize each sample. With *CAMS*, each image was passed as input to the CLI, and the other arguments and flags were left as default. Both results were stored as CSV files and an analysis was completed to compare the performance of the two techniques.

5. Comparison and Guidelines

5.1 Comparison

Figure 8 shows the two macrographs that were used for comparison.

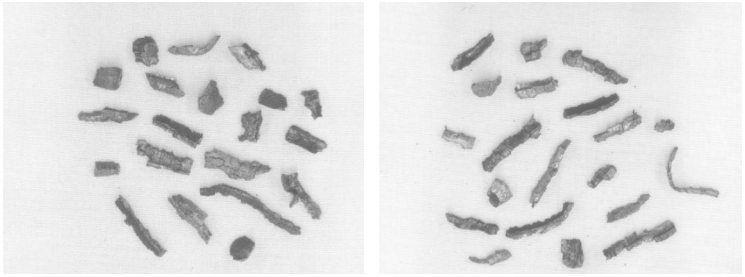

Figure 8. Two frames of 20 samples each used for comparison.

The five samples analyzed are provided in Figure 9.

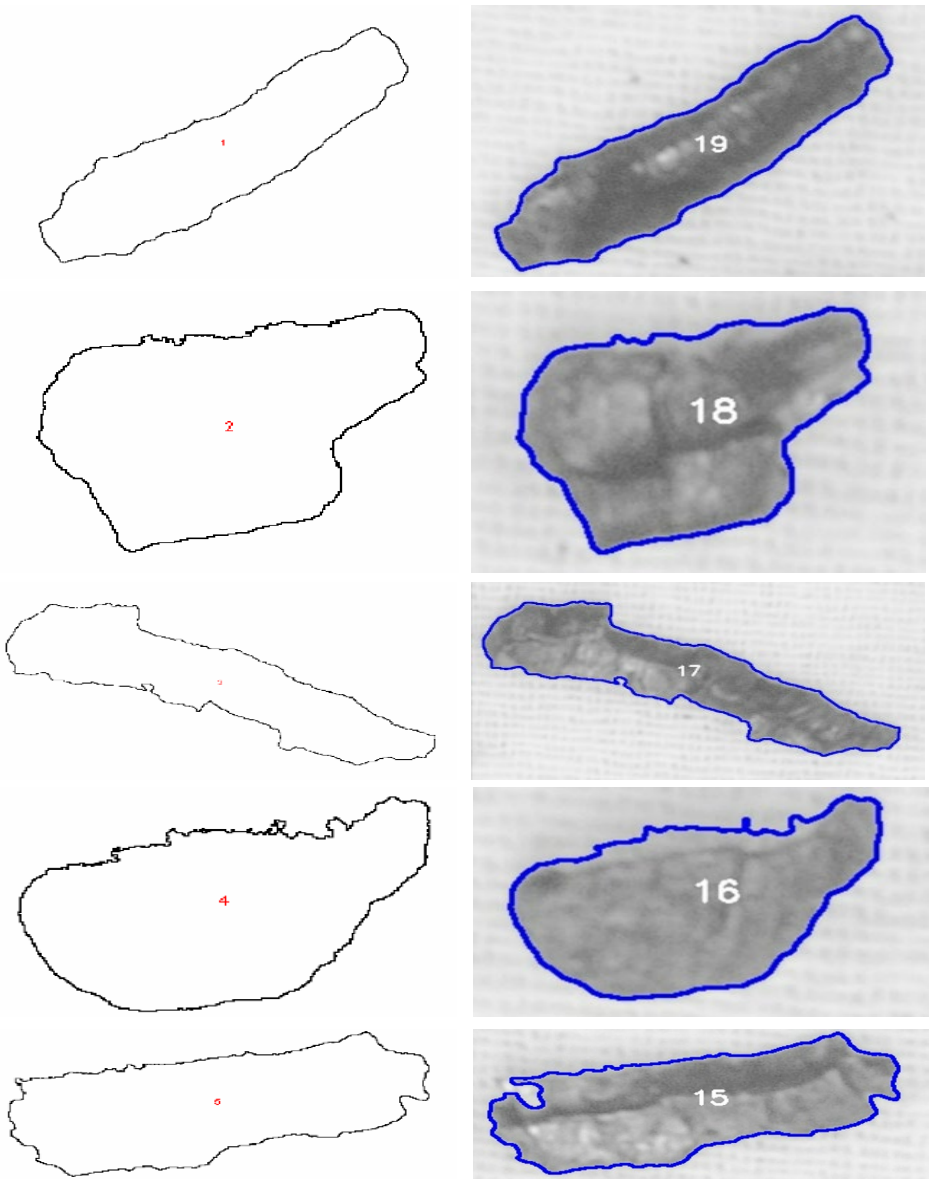

Figure 9. The contours of samples 1–5 drawn by *Fiji* (left) and *CAMS* (right). Note that the sample number assigned by *Fiji* differs from *CAMS*, so *Fiji's* designations were used for further discussion.

The differences between the results from *CAMS* and *Fiji* were minimal. Table 2 shows the percentage differences between the measurements for each sample.

Table 2. Results comparison between *CAMS* and *Fiji* for each of the selected five samples.

Sample no.	*Fiji* area ($inch^2$)	*Fiji* AR	*CAMS* area ($inch^2$)	*CAMS* AR	Area difference (%)	AR difference (%)
4	0.046	1.818	0.045	1.896	1.658	4.197
5	0.067	2.796	0.065	2.834	2.593	1.364
3	0.096	4.647	0.095	4.655	0.965	0.171
2	0.049	1.41	0.048	1.439	1.305	2.013
1	0.068	3.974	0.067	4.055	1.489	2.013

To validate *CAMS* further, a comparison of the analysis results for all 40 samples tested was performed. The average percentage difference and standard deviation for area and AR is shown in Table 3.

Table 3. Table showcasing the differences in results between *Fiji* and *CAMS* when analyzing forty AM samples from the same set.

Program	Average area	Difference (%)	Average AR	Difference (%)
Fiji	0.069 ± 0.025	1.587	3.112 ± 1.553	3.144
CAMS	0.068 ± 0.025		3.016 ± 1.511	

The results of the validation show that, on average, *CAMS* area measurements were within 2% of *Fiji* and 4% for AR measurements. This similarity is exemplary of the accuracy *CAMS* offers. Finally, a comparison was performed to measure how much faster *CAMS* is at measuring samples than Fiji. The time elapsed to analyze the samples in Figure 3 with Fiji was measured manually by an experienced user of the software, requiring 47.12 s to complete the analysis. The same Figure 3 image was then processed through *CAMS*, which completed the analysis in 0.2378 s, or a **198%** improvement over manual measurement.

5.2 Guidelines

Through testing, some of *CAMS*' shortcomings were revealed, necessitating that guidelines for its use are established. First, *samples being measured should not be touching.* This is because samples that are touching will likely be considered as one larger sample by *CAMS*. Figure 10 shows an example of how *CAMS* behaves when analyzing touching objects. This resulted in all samples having one contour drawn around them and inaccurate area and AR measurements.

Figure 10. Multiple samples having a single contour being drawn around them due to touching.

The second guideline is that *images should have sufficient contrast*. With improper contrast, *CAMS* struggled to segment the samples from the background correctly. This is illustrated in Figures 11 and 12.

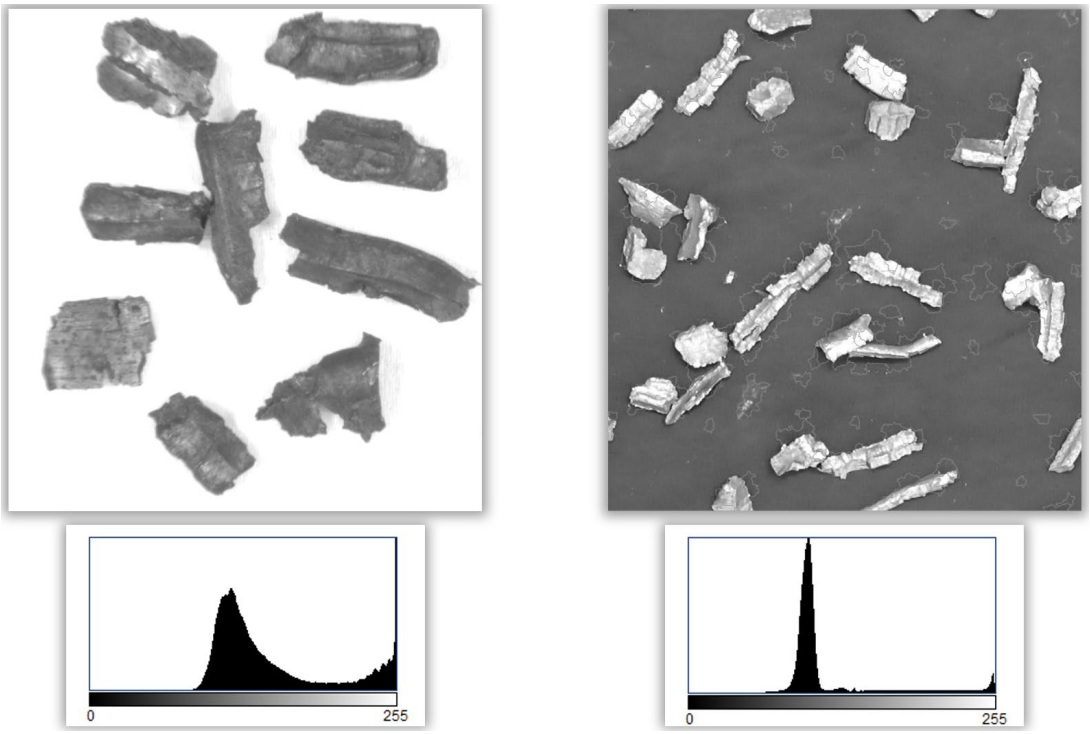

Figure 11. Ideal contrast (left) and poor contrast (right) with histograms. Note the difference in the shape of the histograms; the left has a clearer distinction between foreground and background made obvious by the distribution's more pronounced bimodality.

Figure 12. Result of thresholding with poor contrast.

It is important that the data is collected such that the histogram has a bimodal distribution so a clear threshold can be defined. Otherwise, parts of the background were included in the final contours resulting in inaccurate area and AR measurements.

6. Summary

CAMS enables researchers to automatically, rapidly, and accurately characterize large amounts of samples, dramatically reducing manual labor and human bias. *CAMS* measurements are similarly accurate to manual measurements while being able to measure more samples 198% faster. This rapid, robust high throughput of data can be used in AI/ML models to learn nuanced relationships between printing parameters and material properties, fueling more accurate predictions. Ultimately, *CAMS* lays the groundwork for rapid AM process optimization by delivering the rich, quality-assured datasets that advanced ML workflows require.

7. References

1. Equbal MA, Equbal A, Khan ZA, Badruddin IA. Machine learning in additive manufacturing: a comprehensive insight. Int J Lightweight Mater Manuf. 2025;8(2):264–284. https://doi.org/10.1016/j.ijlmm.2024.10.002

2. Schindelin J et al. Fiji: an open-source platform for biological-image analysis. Nat Meth. 2012;9(7):676–682. https://doi.org/10.1038/nmeth.2019

Appendix A. requirements.txt

This code contains the computer vision operations that allows the Computer Automated Measurement System (*CAMS*) to characterize samples in images. Six functions in this file work together to process images and video and store the results for later analysis. Copy the code below and save the file as "*processing_functions.py*". Ensure that the contents of the file are stored in the same directory as "*CLI.py"* and ensure that the file extension of the document that the code is stored is *.py*. The contents of the file are as follows:

```
import cv2 as cv
import pandas as pd
import os
import datetime


def process_image(
    image_path,
    output_path,
    min_cont_area,
    global_scale,
    csv_path,
    exclude_edges,
    dark_background=False,
):
    """
    Opens and analyzes a single image file to detect and measure
objects.

    Parameters:
    -----------
    image_path : str
        Path to the input image file
    output_path : str
        Directory where processed images will be saved
    min_cont_area : float
        Minimum contour area in pixels to be considered a valid
object
    global_scale : float
        Scale factor for converting pixels to inches (pixels per
inch)
    csv_path : str
        Path where the measurements CSV file will be saved
    exclude_edges : bool
        If True, objects touching the image edges will be excluded
    dark_background : bool
        If  True,  processes  light  objects  on  dark  background
(default is dark objects on light background)
    """
    # Validate that the image file exists
    if not os.path.exists(image_path):
        print(f"Error: Image file not found at '{image_path}'")
        return

    # Load the image using OpenCV
```

```
    frame = cv.imread(image_path)
    if frame is None:
        print(f"Error:   Could   not   read   image   file   at
'{image_path}'")
        return

    # Display processing parameters to the user
    print(f"Image Loaded: {os.path.basename(image_path)}")
    print(f"Minimum   contour   area   threshold:   {min_cont_area}
pixels")
    print(f"Measurement scale: {global_scale} pixels/inch")
    if exclude_edges:
        print("Excluding objects on edges.")
    if dark_background:
        print("Processing light objects on dark background.")
    print("Processing image...")

    # Call the main analysis function on the loaded image
    analyze_frame(
        frame=frame,
        output_directory=output_path,
        min_cont_area=min_cont_area,
        global_scale=global_scale,
        csv_path=csv_path,
        exclude_edges=exclude_edges,
        dark_background=dark_background,
    )
    print("Image processing complete.")


def process_video(
    video_path,
    interval_seconds,
    time_points,
    output_path,
    min_cont_area,
    global_scale,
    csv_path,
    exclude_edges,
    dark_background=False,
):
    """
    Opens a video file and processes frames either at regular
intervals or at specific time points.

    Parameters:
    -----------
    video_path : str
        Path to the input video file
    interval_seconds : float
        Time interval in seconds between processed frames (if
time_points is None)
    time_points : list or None
        List of specific time points (in seconds) to process frames
at
    output_path : str
        Directory where processed frames will be saved
```

```
    min_cont_area : float
        Minimum contour area in pixels to be considered a valid
object
    global_scale : float
        Scale factor for converting pixels to inches (pixels per
inch)
    csv_path : str
        Path where the measurements CSV file will be saved
    exclude_edges : bool
        If True, objects touching the frame edges will be excluded
    dark_background : bool
        If True, processes light objects on dark background
    """
    # Validate that the video file exists
    if not os.path.exists(video_path):
        print(f"Error: Video file not found at '{video_path}'")
        return

    # Open the video file using OpenCV's VideoCapture
    cap = cv.VideoCapture(video_path)
    if not cap.isOpened():
        print(f"Error:   Could   not   open   video   file   at
'{video_path}'")
        return

    # Extract video metadata
    fps = cap.get(cv.CAP_PROP_FPS)  # Frames per second
    frame_count = int(cap.get(cv.CAP_PROP_FRAME_COUNT))   # Total
number of frames

    # Validate video metadata
    if fps == 0 or frame_count == 0:
        print("Error: Could not retrieve valid FPS or frame count
from the video.")
        cap.release()
        return

    # Calculate video duration and display info
    total_seconds = round(frame_count / fps)
    video_time = datetime.timedelta(seconds=total_seconds)
    print(f"Video Loaded: {os.path.basename(video_path)}")
    print(f"Total duration: {video_time}")
    print(f"FPS: {fps:.2f}")

    # Choose processing method based on whether specific time points
are provided
    if time_points is not None:
        # Process specific time points provided by the user
        process_video_at_time_points(
            cap,
            fps,
            time_points,
            total_seconds,
            output_path,
            min_cont_area,
            global_scale,
            csv_path,
```

```
            exclude_edges,
            dark_background,
        )
    else:
        # Process at regular intervals
        process_video_at_intervals(
            cap,
            fps,
            interval_seconds,
            output_path,
            min_cont_area,
            global_scale,
            csv_path,
            exclude_edges,
            dark_background,
        )

    # Release video resources
    cap.release()
    print("Video processing complete.")


def process_video_at_time_points(
    cap,
    fps,
    time_points,
    total_seconds,
    output_path,
    min_cont_area,
    global_scale,
    csv_path,
    exclude_edges,
    dark_background=False,
):
    """
    Process video frames at specific user-defined time points.

    Parameters:
    -----------
    cap : cv2.VideoCapture
        OpenCV video capture object
    fps : float
        Frames per second of the video
    time_points : list
        List of time points in seconds where frames should be
processed
    total_seconds : int
        Total duration of the video in seconds
    output_path : str
        Directory for saving processed frames
    min_cont_area : float
        Minimum contour area threshold
    global_scale : float
        Pixel to inch conversion factor
    csv_path : str
        Path for saving measurements CSV
    exclude_edges : bool
```

```
        Whether to exclude edge-touching objects
    dark_background : bool
        Whether processing light objects on dark background
    """
    # Display processing information
    print(f"Processing frames at {len(time_points)} specific time
points:")
    for i, t in enumerate(time_points):
        print(f"  {i}: {t:.1f}s", end="")
    print()

    print(f"Minimum contour area threshold: {min_cont_area}
pixels")
    print(f"Measurement scale: {global_scale} pixels/inch")
    if exclude_edges:
        print("Excluding objects on edges.")
    if dark_background:
        print("Processing light objects on dark background.")

    # Filter out time points that exceed video duration
    valid_time_points = [t for t in time_points if t <=
total_seconds]
    if len(valid_time_points) < len(time_points):
        print(
            f"Warning: {len(time_points) - len(valid_time_points)}
time point(s) exceed video duration and will be skipped."
        )

    # Process each valid time point
    for interval_counter, time_point in
enumerate(valid_time_points):
        # Convert time to frame number
        frame_number = int(time_point * fps)

        # Seek to the specific frame in the video
        cap.set(cv.CAP_PROP_POS_FRAMES, frame_number)
        ret, frame = cap.read()

        if ret:
            print(f"Processing frame at {time_point:.1f}s (frame
#{frame_number})...")
            # Analyze the extracted frame
            analyze_frame(
                frame=frame,
                count=interval_counter,
                output_directory=output_path,
                min_cont_area=min_cont_area,
                global_scale=global_scale,
                csv_path=csv_path,
                exclude_edges=exclude_edges,
                dark_background=dark_background,
            )
        else:
            print(f"Warning:  Could  not  read  frame  at
{time_point:.1f}s")
```

```
def process_video_at_intervals(
    cap,
    fps,
    interval_seconds,
    output_path,
    min_cont_area,
    global_scale,
    csv_path,
    exclude_edges,
    dark_background=False,
):
    """
    Process video frames at regular time intervals.

    Parameters:
    -----------
    cap : cv2.VideoCapture
        OpenCV video capture object
    fps : float
        Frames per second of the video
    interval_seconds : float
        Time interval between processed frames in seconds
    output_path : str
        Directory for saving processed frames
    min_cont_area : float
        Minimum contour area threshold
    global_scale : float
        Pixel to inch conversion factor
    csv_path : str
        Path for saving measurements CSV
    exclude_edges : bool
        Whether to exclude edge-touching objects
    dark_background : bool
        Whether processing light objects on dark background
    """
    # Calculate how many frames to skip between processing
    interval_frames = int(interval_seconds * fps)

    # Display processing parameters
    print(
        f"Analyzing one frame every {interval_seconds} seconds
({interval_frames} frames)."
    )
    print(f"Minimum contour area threshold: {min_cont_area}
pixels")
    print(f"Measurement scale: {global_scale} pixels/inch")
    if exclude_edges:
        print("Excluding objects on edges.")
    if dark_background:
        print("Processing light objects on dark background.")

    # Initialize counters
    frame_iterator = 0  # Current frame number in the video
    interval_counter = 0  # Number of intervals processed

    # Process frames until end of video
    while True:
```

```
        ret, frame = cap.read()
        if not ret:
            print("\nEnd of video reached.")
            break

        # Check if current frame should be processed (at interval)
        if frame_iterator % interval_frames == 0:
            print(
                f"Processing frame at interval {interval_counter}
(frame #{frame_iterator})..."
            )
            # Analyze the frame
            analyze_frame(
                frame=frame,
                count=interval_counter,
                output_directory=output_path,
                min_cont_area=min_cont_area,
                global_scale=global_scale,
                csv_path=csv_path,
                exclude_edges=exclude_edges,
                dark_background=dark_background,
            )
            interval_counter += 1
        frame_iterator += 1


def   stats_to_CSV(large_contours,   frame_count,   global_scale,
output_csv_path):
    """
    Calculates statistics for detected contours and appends them to
a CSV file.

    Parameters:
    -----------
    large_contours : list
        List  of  OpenCV  contours  that  passed  the  minimum  area
threshold
    frame_count : int
        Frame number or identifier for labeling
    global_scale : float
        Pixel to inch conversion factor (pixels per inch)
    output_csv_path : str
        Path to the output CSV file

    The function calculates:
    - Area in square inches (converted from pixels)
    - Aspect ratio (ratio of longer dimension to shorter dimension)
    """
    data_for_df = []

    # Calculate conversion factor for area (pixels² to inches²)
    area_conversion_factor = global_scale**2

    # Calculate areas for all contours
    large_contours_areas   =   [cv.contourArea(cnt)   for   cnt   in
large_contours]
    large_contours_aspect_ratios = []
```

```
    # Calculate aspect ratios for all contours
    for cnt in large_contours:
        try:
            # Try to fit an ellipse (requires at least 5 points)
            ellipse = cv.fitEllipse(cnt)
            width, height = ellipse[1]
            aspect_ratio = (
                max(width, height) / min(width, height)
                if width > 0 and height > 0
                else 1.0
            )
        except:
            # Fallback to bounding rect for small contours or
fitting errors
            rect = cv.minAreaRect(cnt)
            width, height = rect[1]
            aspect_ratio = (
                max(width, height) / min(width, height)
                if width > 0 and height > 0
                else 1.0
            )

        large_contours_aspect_ratios.append(aspect_ratio)

    # Prepare data for DataFrame
    for contour_number, (area_in_pixels, aspect_ratio) in
enumerate(
        zip(large_contours_areas, large_contours_aspect_ratios)
    ):
        # Create unique label for each detected object
        label = f"sample_{contour_number}_frame_{frame_count}"

        # Convert pixel area to square inches
        area_in_sq_in = area_in_pixels / area_conversion_factor

        # Add to data list
        data_for_df.append(
            {"label": label, "area_sq_in": area_in_sq_in,
"aspect_ratio": aspect_ratio}
        )

    # Check if any contours were found
    if not data_for_df:
        print(f"No large contours found in frame {frame_count}.")
        return

    # Create DataFrame and prepare for CSV output
    df = pd.DataFrame(data_for_df)

    # Create output directory if it doesn't exist
    csv_dir = os.path.dirname(output_csv_path)
    if csv_dir and not os.path.exists(csv_dir):
        os.makedirs(csv_dir)

    # Determine if we need to write the header (first time writing
to file)
```

```
    write_header = not os.path.exists(output_csv_path)

    # Append data to CSV file
    try:
        df.to_csv(output_csv_path, mode="a", header=write_header,
index=False)
        print(
            f"Appended {len(df)} contours from frame {frame_count}
to {output_csv_path}"
        )
    except PermissionError:
        print(f"Error: Permission denied when trying to write to
'{output_csv_path}'.")
        print("Please make sure the file is not open in another
program.")


def analyze_frame(
    frame,
    output_directory,
    min_cont_area,
    global_scale,
    csv_path,
    exclude_edges,
    dark_background=False,
    count=0,
):
    """
    Main image analysis function that detects objects using contour
detection.

    Parameters:
    -----------
    frame : numpy.ndarray
        Input image/frame as a numpy array (BGR format)
    output_directory : str
        Directory where debug and result images will be saved
    min_cont_area : float
        Minimum contour area in pixels to be considered a valid
object
    global_scale : float
        Scale factor for converting pixels to inches
    csv_path : str
        Path where measurements will be saved
    exclude_edges : bool
        If True, contours touching image borders will be excluded
    dark_background : bool
        If True, uses thresholding for light objects on dark
background
    count : int
        Frame/image counter used for naming output files

    The function performs the following steps:
    1. Converts image to grayscale
    2. Applies Gaussian blur to reduce noise
    3. Performs adaptive thresholding (OTSU's method)
    4. Applies morphological closing to fill gaps
```

```
    5. Finds contours (object boundaries)
    6. Filters contours by size and edge proximity
    7. Saves measurements to CSV
    8. Saves debug images showing each processing step
    """
    img = frame
    assert img is not None, "File could not be read."

    # Create a copy for drawing results
    result = img.copy()

    # Convert to grayscale for processing
    gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

    # Apply Gaussian blur to reduce noise and improve contour
detection
    blurred = cv.GaussianBlur(gray, (5, 5), 0)

    # Apply adaptive thresholding based on background type
    if dark_background:
        # For light objects on dark background, use standard binary
threshold
        # White pixels (255) will represent objects
        _, thresh = cv.threshold(blurred, 0, 255, cv.THRESH_BINARY
+ cv.THRESH_OTSU)
    else:
        # For dark objects on light background, use inverted binary
threshold
        # Dark objects become white pixels (255) after inversion
        _,    thresh    =    cv.threshold(blurred,    0,    255,
cv.THRESH_BINARY_INV + cv.THRESH_OTSU)

    # Define morphological kernel for closing operation
    kernel = cv.getStructuringElement(cv.MORPH_ELLIPSE, (5, 5))

    # Apply morphological closing to fill small gaps in objects
    closed = cv.morphologyEx(thresh, cv.MORPH_CLOSE, kernel)

    # Find all external contours (object boundaries) in the
processed image
    contours,    _    =    cv.findContours(closed,    cv.RETR_EXTERNAL,
cv.CHAIN_APPROX_NONE)

    # Optional: Filter out contours touching the image border
    if exclude_edges:
        height, width = closed.shape[:2]
        original_contour_count = len(contours)

        # Check each contour for border contact
        non_edge_contours = []
        for cnt in contours:
            is_touching_edge = False
            # Check if any point in the contour touches the image
border
            for point in cnt:
                x, y = point[0]
```

```
                if x <= 0 or x >= width - 1 or y <= 0 or y >= height
- 1:
                    is_touching_edge = True
                    break
            # Keep only non-edge contours
            if not is_touching_edge:
                non_edge_contours.append(cnt)

        contours = non_edge_contours
        print(f"Filtered     out     {original_contour_count     -
len(contours)} edge contour(s).")

    # Filter contours by minimum area threshold
    large_contours    =    [cnt    for    cnt    in    contours    if
cv.contourArea(cnt) > min_cont_area]

    # Export contour statistics to CSV
    stats_to_CSV(
        large_contours,
        frame_count=count,
        global_scale=global_scale,
        output_csv_path=csv_path,
    )

    # Draw detected contours on the result image
    cont_image = cv.drawContours(
        result, large_contours, contourIdx=-1, color=(255, 0, 0),
thickness=2
    )

    # Label each contour with its index number
    for i, c in enumerate(large_contours):
        # Calculate centroid using image moments
        M = cv.moments(c)
        if M["m00"] != 0:
            # Standard centroid calculation
            cX = int(M["m10"] / M["m00"])
            cY = int(M["m01"] / M["m00"])
        else:
            # Fallback to bounding box corner if moments are zero
            x, y, w, h = cv.boundingRect(c)
            cX, cY = x, y

        # Draw the index number at the centroid
        cv.putText(
            cont_image,
            str(i),
            (cX, cY),
            cv.FONT_HERSHEY_SIMPLEX,
            0.7,
            (255, 255, 255),
            2,
        )

    # Create output directory if it doesn't exist
    if not os.path.exists(output_directory):
        os.makedirs(output_directory)
```

```
    # Save debug images showing each processing step
    cv.imwrite(f"{output_directory}grayscale_debug_{count}.jpg",
gray)
    cv.imwrite(f"{output_directory}blurred_debug_{count}.jpg",
blurred)
    cv.imwrite(f"{output_directory}closed_debug_{count}.jpg",
closed)

cv.imwrite(f"{output_directory}thresholded_debug_{count}.jpg",
thresh)
    cv.imwrite(f"{output_directory}final_co    ntour_{count}.jpg",
cont_image)
```

Appendix B. CLI.py

CLI.py is the Python source code file that describes the command line interface (CLI) interface for Computer Automated Measurement System (*CAMS*). It is the entry point of the program. Copy the code below and save the file as "*CLI.py"* to ensure the program operates as expected. Additionally, ensure it is in the same directory as "*processing_functions.py*". The contents of the file are as follows:

```
import argparse
import os

# Import both processing functions
from processing_functions import process_video, process_image


def parse_time_points(time_string):
    """
    Parse a comma-separated list of time points in seconds.
    Example: "0,15,30.5,67" -> [0.0, 15.0, 30.5, 67.0]
    """
    try:
        time_points = [float(t.strip()) for t in
time_string.split(",")]
        # Sort time points and remove duplicates
        time_points = sorted(list(set(time_points)))
        return time_points
    except ValueError:
        raise argparse.ArgumentTypeError(
            "Time points must be comma-separated numbers (e.g.,
'0,15,30.5,67')"
        )


def main():
    """
    Parses command-line arguments and initiates video or image
processing.
    """
    # Set up the argument parser with a description
    parser = argparse.ArgumentParser(
        description="Analyze particles in a video or image file.",
        formatter_class=argparse.RawTextHelpFormatter,
    )

    # --- Arguments ---
    parser.add_argument(
        "input", type=str, help="Path to the input video or image
file."
    )

    # Create a mutually exclusive group for interval vs time points
    timing_group = parser.add_mutually_exclusive_group()
    timing_group.add_argument(
        "-i",
        "--interval",
        type=float,
```

```
        default=None,
        help="For videos, the interval in seconds to capture a
frame. \n(Ignored for single images).",
    )
    timing_group.add_argument(
        "-t",
        "--time-points",
        type=parse_time_points,
        default=None,
        help="For videos, specific time points (in seconds) to
capture frames.\n"
        "Provide as comma-separated values, e.g., '0,15,30.5,67'\n"
        "(Ignored for single images).",
    )

    parser.add_argument(
        "-o",
        "--output",
        type=str,
        default="debug/",
        help="The output directory where resulting images are
stored. \nDefault is CLI_test/.",
    )
    parser.add_argument(
        "-m",
        "--min-area",
        type=int,
        default=850,
        help="Minimum contour area in pixels for analysis.
\nDefault is 850.",
    )
    parser.add_argument(
        "-s",
        "--scale",
        type=int,
        default=1,
        help="Pixel scale for measurements (pixels/inch). \nDefault
is 259.",
    )
    parser.add_argument(
        "--csv",
        type=str,
        default="contour_stats.csv",
        help="Path for the output CSV file. \nDefault is
'contour_stats.csv' in the working directory.",
    )

    # Add a flag to enable the "Exclude on Edges" feature
    parser.add_argument(
        "--exclude-edges",
        action="store_true",  # This makes it a boolean flag
        help="Enable this flag to exclude objects touching the
image edges.",
    )

    # Added a flag to account for morphological closing with dark
backgrounds.
```

```
    parser.add_argument(
        "--dark-background",
        action="store_true",
        help="Enable this flag if you have light-colored samples on
a dark background.",
    )
    # ============================

    args = parser.parse_args()

    # --- LOGIC TO DETECT FILE TYPE ---
    _, file_extension = os.path.splitext(args.input)
    image_extensions = [".tif", ".tiff", ".jpg", ".jpeg", ".png",
".bmp"]

    if file_extension.lower() in image_extensions:
        print("Image file detected.")
        process_image(
            image_path=args.input,
            output_path=args.output,
            min_cont_area=args.min_area,
            global_scale=args.scale,
            csv_path=args.csv,
            exclude_edges=args.exclude_edges,
            dark_background=args.dark_background,  # Pass the dark
background flag
        )
    else:
        print("Video file detected.")

        # Set default interval if neither interval nor time points
specified
        if args.interval is None and args.time_points is None:
            args.interval = 10.0

        process_video(
            video_path=args.input,
            interval_seconds=args.interval,
            time_points=args.time_points,
            output_path=args.output,
            min_cont_area=args.min_area,
            global_scale=args.scale,
            csv_path=args.csv,
            exclude_edges=args.exclude_edges,
            dark_background=args.dark_background,  # Pass the dark
background flag
        )


if __name__ == "__main__":
    main()
```

Appendix C. processing_functions.py

This text document lists all dependencies used during the Computer Automated Measurement System (*CAMS*) development process. Of note are *OpenCV* and *pandas* but included along with them are dependencies that they need to run. The process to install these packages (explained in the Installation section) uses the Python package manager to read from this text document and install the listed packages. Save this code as a text file named “requirements.txt”. The text is as follows:

```
contourpy==1.3.2
cycler==0.12.1
fonttools==4.58.4
imageio==2.37.0
kiwisolver==1.4.8
lazy_loader==0.4
matplotlib==3.10.3
networkx==3.5
numpy==2.3.0
opencv-python==4.11.0.86
packaging==25.0
pandas==2.3.0
pillow==11.2.1
pyparsing==3.2.3
python-dateutil==2.9.0.post0
pytz==2025.2
scikit-image==0.25.2
scipy==1.15.3
six==1.17.0
tifffile==2025.6.11
tzdata==2025.2
```

Appendix D. README.md

This contains brief instructions for the Computer Automated Measurement System (*CAMS*) installation and usage in markdown format. Copy the code below and save the file as "*README.md"*. It is recommended to view this file in a markdown viewer.

````
# CAMS - Computer Automated Measurement System
## User Documentation

### Overview
The Computer Automated Measurement System (CAMS) is a command-line
tool for automated characterization of Additive Manufacturing (AM)
samples using computer vision. CAMS processes images and videos to
detect, measure, and catalog sample characteristics including area
and aspect ratio, enabling rapid dataset generation for AI/ML model
training.

### System Requirements
- Python 3.12.2 or compatible version
- Operating System: Windows, macOS, or Linux
- Sufficient storage for output images and CSV data

### Installation

#### Step 1: Install Python
**Windows:**
- Option A: Download from https://www.python.org/downloads/
- Option B: Using Scoop package manager: `scoop install python`

**macOS:**
```bash
brew install python
```

**Linux:**
```bash
sudo apt install python
```

#### Step 2: Set Up Virtual Environment
Navigate to the CAMS directory and create a Python virtual
environment:

```bash
python -m venv CAMSvenv
```

#### Step 3: Activate Virtual Environment
**Windows:**
```powershell
./CAMSvenv/Scripts/Activate.ps1
```

**macOS/Linux:**
```bash
````
````

```
source ./CAMSvenv/bin/activate
```

Step 4: Install Dependencies
With the virtual environment activated, install required packages:

```bash
pip install -r requirements.txt
```

This installs:
- OpenCV (cv2) - Image processing and computer vision
- Pandas - Data management and CSV export
- NumPy, matplotlib, and other supporting libraries

File Structure
```
CAMS/
├── CLI.py                    # Command-line interface
├── processing_functions.py   # Core image analysis functions
├── requirements.txt         # Python dependencies
├── debug/                    # Output directory for debug images
(created on first run)
└── contour_stats.csv       # Measurements output (created on first
run)
```

Usage

Basic Syntax
```bash
python CLI.py [input_file] [options]
```

Command-Line Arguments

Required Arguments
- **`input`** - Path to the input video or image file
  - Supported image formats: `.tif`, `.tiff`, `.jpg`, `.jpeg`,
`.png`, `.bmp`
  - Supported video formats: Most common video formats (`.mp4`,
`.avi`, `.mov`, etc.)

Optional Arguments

Output Configuration
- **`-o, --output`**
  - Directory for saving processed images
  - Default: `debug/`
  - Example: `--output results/`

- **`--csv`**
  - Path for the output CSV file containing measurements
  - Default: `contour_stats.csv`
  - Example: `--csv measurements/data.csv`

````
###### Video Processing (Video files only)
- **`-i, --interval`**
  - Time interval in seconds between frame captures
  - Default: 10 seconds
  - Use case: Samples moving at constant speed
  - Example: `--interval 5.0`

- **`-t, --time-points`**
  - Specific time points for frame extraction
  - Format: Comma-separated values in seconds
  - Use case: Samples moving at variable speeds
  - Example: `--time-points 0,15,30.5,67`
  - Note: Cannot be used with `--interval`

###### Analysis Parameters
- **`-m, --min-area`**
  - Minimum contour area in pixels
  - Default: 850 pixels
  - Filters out dust, debris, and artifacts
  - Example: `--min-area 1000`

- **`-s, --scale`**
  - Pixel-to-inch conversion factor (pixels per inch)
  - Default: 1
  - Critical for accurate physical measurements
  - Example: `--scale 300`

###### Processing Flags
- **`--exclude-edges`**
  - Excludes objects touching image borders
  - Boolean flag (no value needed)
  - Use when edge samples may be partially visible

- **`--dark-background`**
  - For light samples on dark backgrounds
  - Boolean flag (no value needed)
  - Inverts the normal thresholding behavior

### Usage Examples

#### Example 1: Process a Single Image
```bash
python CLI.py sample_image.jpg
```

#### Example 2: Process Image with Custom Parameters
```bash
python CLI.py sample_image.tif --min-area 1200 --scale 350 --
exclude-edges
```

#### Example 3: Process Video at Regular Intervals
```bash
python CLI.py sample_video.mp4 --interval 15 --output video_frames/
--csv video_data.csv
```
````

Example 4: Process Video at Specific Time Points
```bash
python CLI.py sample_video.mp4 --time-points 0,10,25,30,45 --min-area 900
```

Example 5: Light Samples on Dark Background
```bash
python CLI.py dark_bg_image.png --dark-background --scale 400
```

Output Files

Debug Images (saved to output directory)
For each processed frame/image, CAMS generates:
- `grayscale_debug_[n].jpg` - Grayscale conversion
- `blurred_debug_[n].jpg` - After Gaussian blur
- `thresholded_debug_[n].jpg` - Binary threshold result
- `closed_debug_[n].jpg` - After morphological closing
- `final_contour_[n].jpg` - Original with detected contours and labels

CSV Data File
Contains measurements for each detected sample:
- **label**: Unique identifier (format: `sample_[n]_frame_[m]`)
- **area_sq_in**: Area in square inches
- **aspect_ratio**: Ratio of major to minor axis

Image Processing Pipeline

1. **Grayscale Conversion** - Converts RGB to single-channel image
2. **Gaussian Blur** - Reduces noise (5x5 kernel)
3. **OTSU Thresholding** - Automatic threshold selection for segmentation
4. **Morphological Closing** - Fills gaps in objects (5x5 elliptical kernel)
5. **Contour Detection** - Identifies object boundaries
6. **Filtering** - Applies size and edge criteria
7. **Measurement** - Calculates area and aspect ratio
8. **Export** - Saves data to CSV and debug images

Best Practices and Guidelines

Image Quality Requirements
1. **High Contrast**: Ensure clear distinction between samples and background
   - Bimodal histogram distribution is ideal
   - Adjust lighting/exposure as needed

2. **Sample Separation**: Samples must not be touching
   - Touching samples will be measured as single objects
   - Leave adequate spacing between samples

3. **Consistent Background**: Use uniform, non-textured backgrounds
   - Solid-colored cloth or paper recommended
   - Avoid reflective or patterned surfaces

```
#### Calibration
1. **Scale Calibration**:
   - Include a ruler or known reference in initial images
   - Measure pixels per inch for your setup
   - Update using `--scale` parameter

2. **Minimum Area Threshold**:
   - Start with default (850 pixels)
   - Adjust based on smallest valid sample size
   - Test with `--min-area` to filter noise

#### Processing Tips
- For videos with consistent sample movement, use `--interval`
- For videos with variable timing, use `--time-points`
- Always verify first few results before batch processing
- Review debug images to confirm proper segmentation

### Troubleshooting

#### Common Issues and Solutions

**Issue**: "Error: Image file not found"
- **Solution**: Verify file path and ensure file exists

**Issue**: Multiple samples detected as one
- **Solution**: Increase sample separation or adjust threshold

**Issue**: Samples not detected
- **Solutions**:
  - Reduce `--min-area` value
  - Improve image contrast
  - Try `--dark-background` flag if applicable

**Issue**: Incorrect measurements
- **Solution**: Recalibrate `--scale` parameter with known reference

**Issue**: "Permission denied" when writing CSV
- **Solution**: Close CSV file if open in another program

**Issue**: Poor segmentation results
- **Solutions**:
  - Improve lighting uniformity
  - Clean camera lens
  - Ensure consistent background

### Performance Specifications
- Processing speed: Hundreds of samples per minute
- Area measurement accuracy: Within 2% of manual methods
- Aspect ratio accuracy: Within 3-4% of manual methods
```

List of Symbols, Abbreviations, and Acronyms

AI	artificial intelligence
AM	additive manufacturing
AR	aspect ratio
ARL	Army Research Laboratory
CAMS	Computer Automated Measurement System
CLI	command line interface
CSV	comma-separated values
DEVCOM	U.S. Army Combat Capabilities Development Command
ML	machine learning
N/A	not applicable
RGB	red, green, and blue

1 (PDF) DEFENSE TECHNICAL INFORMATION CTR DTIC OCA

1 (PDF) DEVCOM ARL FCDD RLB CI TECH LIB