Slurry Image Analyzer Version 2

Image Analysis Software

Theory of Operation

|  |  |
| --- | --- |
| Original Author | Mark Polak |
| Company | Xanantec Technologies |
| Client | Syncrude Canada |
| Reviewers |  |
| Document Version | 1.0 |

# Purpose of Document

This document offers an in-depth understanding of how the software works, including its underlying principles, algorithms, and operational logic.

# Software Introduction and Purpose

The purpose of the Slurry Image Analyzer Version 2 (SIA V2) custom image analysis software is to efficiently process and analyze images of bitumen in oil sand slurry captured by the system with a 4 sensor JAI Line Scan camera. This includes tasks such as identifying bitumen droplets in the images, measuring droplet characteristics, and calculating droplet statistics. The primary focus is on providing a robust and easily maintained system that can deliver accurate results.

# System Architecture

This section describes the software's architecture, including high-level components, modules, and their interactions.

## Architecture Introduction

The image processing software is a command-line application that takes in a path to a folder of images, analyzes those images, measures the sizes of objects found in those images, and writes out statistics output in a CSV file. The main quality goals were accuracy, efficiency, and portability.

The software aims to analyze images, measure the sizes of objects, and generate statistics output in a CSV file. The architecture is based on the following user stories, described in the FUNCTIONAL DESIGN SPECIFICATION document:

1. User Story 1: Load Configuration File
2. User Story 2: Load Images from a Folder
3. User Story 3: Segment Bitumen Droplets
4. User Story 4: Measure Bitumen Droplet Characteristics and Save in File
5. User Story 5: Save Segmented Bitumen Images
6. User Story 6: Segment Sand and Air Objects
7. User Story 7: Measure Sand and Air Object Characteristics and Save in CSV
8. User Story 8: Save Segmented Sand and Air Object Images
9. User Story 9: Calculate Summary Statistics and Save in CSV
10. User Story 10: Error Logging

The SIA Image Analysis software is encapsulated within the **ProcessImagesSIA** class. The main functionality is orchestrated through the **main** function, calling methods of the **ProcessImagesSIA** class to load configuration from a JSON file, load image files to be processed, process images, and generate output files.

## Architectural Overview

This section provides a structured insight into the major components orchestrating the software's functionality. It outlines the primary roles of the ProcessImagesSIA class, acting as the central orchestrator, and describes the methods within. The architectural overview explains the software's organization and the interaction between key components.

## Main Function (main())

The **main()** function serves as the entry point for the program. It performs the following tasks:

1. Validates the command-line arguments, ensuring that the path to the input folder is provided.
2. Initializes an instance of the **ProcessImagesSIA** class with the input folder path.
3. Loads configuration parameters from a JSON file using the **load\_json\_config\_file** method.
4. Retrieves the list of image files and verifies their existence using the **get\_file\_list** method.
5. Segments the images using the **segment\_images** method.
6. Writes out segmented images to the output folder using the **write\_out\_segmented\_images** method.
7. Writes CSV files containing object properties using the **write\_csv\_files** method.

## Communication Between Methods

Methods within the **ProcessImagesSIA** class communicate with each other to execute the image processing workflow. Each method performs a specific step of the process, updating relevant parameters as needed.

## Data Flow

The data flow begins with the **main** function, which initializes an instance of the **ProcessImagesSIA** class. The class methods are then invoked sequentially to load configuration, load images, segment objects, save segmented images, and write CSV files. The results of each processing step are stored in class attributes.

## Class Details: ProcessImagesSIA

The "Class Details: ProcessImagesSIA" section offers a concise exploration of the core ProcessImagesSIA class in the software. From initialization to loading configurations, retrieving image files, conducting image segmentation, and writing results to CSV files, these methods collectively govern the image processing workflow.

### Attributes

* **experiment\_bitumen\_binary\_img**: Segmented binary image for bitumen objects.
* **experiment\_other\_binary\_img**: Segmented binary image for non-bitumen objects (air, sand, unknown).
* **output\_folder:** Path to the output folder for storing results.
* **image\_folder:** Path to the input folder containing images.
* Configuration Parameters for controlling image segmentation, downsampling, and other processing parameters: **image\_scale**, **line\_scan\_rate**, **save\_segmented\_images**, **calc\_summary\_stats**, etc.

### Methods

##### \_\_init\_\_(self, image\_folder)

* **Responsibility**: Constructor initializes the class instance with the provided image folder path. Initializes the **ProcessImagesSIA** object with necessary attributes.

##### load\_json\_config\_file(self) -> int

* **Responsibility**:
  + Loads configuration parameters from a JSON file specified in **SysConfigSIA.OPERATOR\_CONFIG\_FILE\_JSON** and assigns them to corresponding class attributes.
  + Checks for the presence of required parameters and logs errors if any are missing.
* **Return**: 0 if successful, error code otherwise.

##### get\_file\_list(self) -> int

* **Responsibility**: Retrieves and verifies lists of VIS (visual) and NIR (near infrared) image files from the specified image folder using the **get\_file\_list\_and\_verify\_correct\_files\_exist** function.
* **Return**: 0 if successful, error code otherwise.

##### segment\_images(self) -> int

* **Responsibility**: Segments bitumen, sand, and air objects in the images according to parameters set in configuration file.
  + Iterates through the loaded image sets, processing each image using specified segmentation algorithms.
  + Generates binary images for bitumen and non-bitumen objects.
  + Handles memory allocation errors and returns appropriate error codes.
* **Return**: 0 if successful, error code otherwise.

##### write\_out\_segmented\_images(self) -> int

* **Responsibility**: Writes segmented images to the output folder if specified.
  + Uses the **write\_segmented\_images** function for this purpose.
* **Return**: 0 if successful, error code otherwise.

##### write\_csv\_files(self) -> int

* **Responsibility**: Writes measurement results to CSV files using the object\_properties\_to\_csv function.
* **Return**: 0 if successful, error code otherwise.

## Error Logging

Error logging is handled by the Error Logger component. The code incorporates error handling mechanisms to log errors to a specified error log file (**SysConfigSIA.ERROR\_LOG\_FILE**). Errors related to configuration file reading, image loading, memory allocation, and other issues are logged with appropriate error codes.

## Deployment and Future Considerations

The software is designed to run on Windows systems. Configuration files and output folders should be set up based on user requirements. The modular design of the **ProcessImagesSIA** class allows for easy extension and integration of additional features or algorithms in the future.

This architecture documentation provides a detailed overview of the Simple Image Analysis software, encapsulated within the **ProcessImagesSIA** class. The modular design promotes maintainability and flexibility, allowing for easy adaptation to future requirements.

# Image Processing Algorithms

The image processing step of the software is performed by the **segment\_images** method of the **ProcessImagesSIA** class, which is stored in module “**ImageAnalysisSIA.py**”. The **segment\_images** method loads and setup up each image pair (Visual and NIR image) and in turn calls either **segment\_image\_set\_obj\_by\_nir** or **segment\_image\_set\_by\_vis\_img** function to segment the image pair. These image segmentation functions are stored in module “**ImageSegmentationSIA.py”** and they in turn call custom written image processing support functions stored in module “**ImageProcSupport.py”**.

Two image segmentation functions (**segment\_image\_set\_obj\_by\_nir** and **segment\_image\_set\_by\_vis\_img**) were written to segment each oil sand slurry image pair. Both functions are based on valid image processing steps; however, they take different approaches to determining where each object of interest is located in each image and in determining if the object is a dark object (bitumen) or a light object (sand or air bubble). The **segment\_image\_set\_obj\_by\_nir** function works by starting with the NIR image and finding each object in that image. The function then takes each of these found objects and retrieves the corresponding pixels from the visual colour images to determine if the object was bitumen or other (air or sand). On the other hand, the **segment\_image\_set\_by\_vis\_img** function starts with the visual colour image to find all dark bitumen objects and all light (air or sand) objects, and the uses the NIR image to validate each pixel of the segmented object. The **segment\_image\_set\_by\_vis\_img** is significantly more computationally efficient, especially in images with many objects, but it can have trouble segmenting objects in images where the background is dark and has similar intensity as the bitumen objects. Which image processing methodology works better may depend on the process location and resulting image characteristics. This procedure can be adjusted during instrument calibration and then the appropriate algorithm selected in the JSON configuration file.

Below is a description of the method and functions involved in image segmentation.

## Image Segmentation Method: segment\_images(self) -> int

The **segment\_images** method is a crucial component of the image analysis software. This method determines, sets up, and calls the image segmentation function to use based on configuration specified in the JSON file, either **segment\_image\_set\_obj\_by\_nir** or **segment\_image\_set\_by\_vis\_img**. It also handles the cutting of the Region of Interest (ROI) and potential downsampling of input images, providing flexibility and adaptability to different processing setups.

### Method Workflow:

1. **Load Image Information:**
   * The method starts by loading the image information from the first visual (VIS) image in the experiment. The **cv2.IMREAD\_UNCHANGED** flag is used to load the image without decoding it.
2. **Determine Image Dimensions:**
   * Extract the width and height of the image to set the dimensions for further processing.
3. **Adjust Image Dimensions for ROI and Downscaling:**
   * Calculate the width of the Region of Interest (ROI) after excluding the bad left and right edges from the image.
   * Adjust the image dimensions for possible downscaling based on the configured downscale factor.
4. **Memory Allocation for Result Images:**
   * Attempt to create large 8-bit images to store segmented objects for the entire experiment. Two large images are created: one for bitumen objects and another for non-bitumen objects. The latter is only created if the option to segment air and sand is set.
5. **Image Iteration and Segmentation:**

This is the main component of the image segmentation method. It involves the following steps.

* + Iterate through all the loaded image sets, loading corresponding VIS and NIR images.
  + Cut out the region of interest (ROI) from the loaded images.
  + If downsampling is enabled, resize the images accordingly.
  + Segment the image set using the specified segmentation algorithm, either **segment\_image\_set\_obj\_by\_nir** or **segment\_image\_set\_by\_vis\_img**.
  + Concatenate the segmented images into large binary images, updating the corresponding class attributes.

1. **Error Handling:**
   * Handle errors related to image loading, memory allocation, and unknown segmentation algorithms. In case of an error, an error code is returned.

### Parameters:

* **None:**
  + The method does not take any explicit parameters but relies on the class attributes set during the initialization of the object and loading of the JSON configuration file.

### Returns:

* **Error Code:**
  + An integer representing the success or failure of the segmentation process. Zero (0) indicates success, while specific error codes provide information about encountered issues.

### Note:

* The method efficiently handles memory allocation, image resizing, and concatenation of segmented images, providing a comprehensive solution for the segmentation of bitumen and non-bitumen objects in an experiment. Any encountered errors are appropriately logged, making it easy to diagnose and resolve issues during the image processing pipeline.

## Image Segmentation Function: segment\_image\_set\_obj\_by\_nir

The **segment\_image\_set\_obj\_by\_nir** function is designed to perform image segmentation by detecting objects using near-infrared (NIR) image and then using the RGB image to determine if each detected object is bitumen.

The primary objective is to identify and classify these objects based on whether they are bitumen or other materials such as air or sand. The method employs a series of image processing steps and custom functions from the **ImageProcSupport** module for background detection and corrections.

### Parameters

* **image\_vis (numpy.ndarray):** RGB color image.
* **image\_nir (numpy.ndarray):** Corresponding NIR image.
* **segment\_air\_and\_sand (Bool):** Choice for segmenting light objects (air and sand).
* **down\_sample\_factor (float):** Ratio by which the original image was downsampled.
* **k1 (float):** The threshold value for the first pass through the Gaussian background model.
* **k2 (float):** The threshold value for the second pass through the Gaussian background model.

### Algorithm Steps

1. **Input Validation:**
   * Check if the input images (**image\_vis** and **image\_nir**) are loaded successfully. If not, return None for both binary images.
2. **Background Detection:**
   * Apply a two-pass Gaussian background detection model to the NIR image to create an object mask. The object mask is refined to obtain a more accurate model of the background by analyzing only background pixels (objects are masked out).
3. **Image Preprocessing:**
   * Convert the RGB image section to a single-channel grayscale image.
   * Using the generated object mask in the previous step, calculate the average intensity of background pixels for each column in the RGB image.
   * Correct the RGB image intensity to make it uniform across columns.
4. **Object Mask Generation:**
   * Create a binary mask for detected objects using the second-pass Gaussian model.
   * Fill holes in the object mask to create a solid mask of detected objects.
5. **Morphological Operations:**
   * Perform morphological operations to eliminate small objects and artifacts.
6. **Connected Component Labeling:**
   * Label the connected components in the binary image.
7. **Object Analysis and Classification:**
   * Analyze each labeled object to filter out small and tall-thin artifacts.
   * Classify the objects based on the intensity of pixels in the RGB image.
8. **Binary Image Creation:**
   * Create a new binary image for bitumen objects.
   * Create a new binary image for non-bitumen objects.

### Return

The function returns two binary images:

* **numpy.ndarray:** Binary image with detected dark objects (likely bitumen).
* **numpy.ndarray:** Binary image with detected light objects (air and sand) if **segment\_air\_and\_sand** is True; otherwise, it is set to None.

### Note

The function utilizes custom functions from the **ImageProcSupport** module for background detection and corrections.

## Image Segmentation Function: segment\_image\_set\_by\_vis\_img

The **segment\_image\_set\_by\_vis\_img** function aims to segment an image set by detecting objects in a visual RGB (VIS) image (**image\_vis**). It further refines the segmentation using information from the corresponding near-infrared (NIR) image (**image\_nir**). This method utilizes a Gaussian background model on individual columns of the NIR image for background separation and custom thresholding to validate the objects detected in the RGB image.

**Parameters**

* **image\_vis (numpy.ndarray):** RGB color image.
* **image\_nir (numpy.ndarray):** Corresponding 8-bit NIR image.
* **segment\_air\_and\_sand (Bool):** Choice for segmenting light objects (air and sand).
* **k1 (float):** The threshold value for the first pass through the Gaussian background model.
* **k2 (float):** The threshold value for the second pass through the Gaussian background model.

**Algorithm Steps**

1. **Input Validation:**
   * Check if the input images (**image\_vis** and **image\_nir**) are loaded successfully. If not, return None for both binary images.
2. **Background Separation:**
   * Use a Gaussian background model on each column of the NIR image for background separation.
   * Apply custom thresholding using a Gaussian background model in two passes (**k1** and **k2**).
3. **Calculate Column-wise Statistics for Background:**
   * For each column, calculate the average background intensity in the RGB image separately for red, green, and blue channels, using only background pixels defined by the object mask created from the NIR image.
4. **Generate Difference Images:**
   * Find the difference in each column and channel between the average background and pixel values in the RGB image.
   * For dark objects, use the maximum difference to identify these objects.
   * For light objects, use the minimum difference because they are distinct.
5. **Thresholding:**
   * Threshold the difference images obtained in the previous step using a low threshold to ensure the identification of any pixels belonging to an object.
   * Apply a mask created from the background model to eliminate artifacts left over from the threshold operation.
6. **Fill Holes:**
   * Fill holes in the binary images to complete objects.
7. **Morphological Operations:**
   * Perform morphological "open" operation to eliminate small objects.
8. **Return Results:**
   * Return the binary images representing dark and light objects.

**Note**

* The function utilizes custom functions from the **ImageProcSupport** module for background detection and corrections.

## Image Processing Support Functions

The image segmentation methods (**segment\_image\_set\_obj\_by\_nir** and **segment\_image\_set\_by\_vis\_img**) utilize several custom written support functions to enhance the accuracy and efficiency of the segmentation process. Below are the descriptions of these support functions:

### find\_objects\_column\_gaussian

This function is designed to find objects within a Near-Infrared (NIR) image using a Gaussian model. It fits a Gaussian probability density function to individual columns of the image, creating a background model. The function then transforms each pixel into a standard scale, allowing the identification of objects based on a specified threshold (**k**). This function is crucial for creating an initial object mask from the NIR image.

* **Parameters:**
  + **image\_in (numpy.ndarray):** An image in NumPy array format loaded with OpenCV (cv2.imread) with dtype=uint8.
  + **k (float):** The number of standard deviations a pixel must differ from the mean to be classified as an object.
  + **mask (numpy.ndarray, optional):** An optional parameter. If provided, this parameter is an image with 0 for objects and 1 for the background. This mask image is used to "mask out" object pixels in the original image.
* **Returns:**
  + **numpy.ndarray:** A segmented binary image, where each pixel is identified as an object (0) or background (1).

### fill\_holes

This function fills holes in objects within an image using the cv2.floodFill method. It creates a flood-filled copy of the input image, inverts it to produce a mask for the holes, and combines it with the input image to fill the holes. The purpose is to ensure complete and accurate objects for further analysis.

* **Parameters:**
  + **input\_image (numpy.ndarray):** The input image with objects that may contain holes (object = 255, background = 0).
* **Returns:**
  + **numpy.ndarray:** An image with holes in objects filled.

### average\_column\_intensity

This function calculates the average intensity for each column of a grayscale image while excluding masked areas. It is used to obtain column-wise statistics, aiding in background correction and intensity normalization.

* **Parameters:**
  + **image\_gray (numpy.ndarray):** Grayscale image.
  + **mask (numpy.ndarray):** Binary mask where 1 indicates areas to include, 0 to exclude.
* **Returns:**
  + **list:** List of average intensities per column.

### background\_correct\_with\_clipping

This function adjusts the intensity of each column in an image with a correction factor while clipping values within a specified range. It is crucial for correcting lighting variations in RGB images and ensuring uniformity across columns.

* **Parameters:**
  + **image (numpy.ndarray):** The input image with pixel intensities. It should be of data type uint8.
  + **correction (numpy.ndarray):** The correction factor to be added to each column in the image. It should be of data type int16.
  + **clip\_min (int):** The minimum value to which pixel intensities are clipped.
  + **clip\_max (int):** The maximum value to which pixel intensities are clipped.
* **Returns:**
  + **numpy.ndarray:** An image with adjusted column-wise intensities, where values are clipped within the specified range, and the data type is uint8.

# Configuration and Parameters

The SIA V2 image analysis software has two configuration files:

1. “**SysConfig.py**” contains configuration used in the SIA image processing methods. It defines constants imported and used in various modules of the software. The constants defined in this configuration file should not need to be changed by the operator or control software that calls SIA image analysis. This configuration file will need to be modified if the software is ported to another computer, if the file path is changed, or if major modifications are made to the system. The current **SysConfig.py** file is shown below:

|  |
| --- |
| # # SysConfig.py # # This file contains configuration used in the SIA image processing methods. It defines constants imported and used # in various modules of the software. # These constants should not need to be changed by the operator or control software that calls SIA image analysis. # # Written by: Mark Polak # # Date Created: 2023-11-11 # Last Modified: 2023-12-29 #   # Defined absolute paths for: # 1) JSON configuration file that must be loaded in. A valid JSON configuration file for SIA must exist in the # specified location. # 2) SIA error log file. If file does not exist, it will be created, so just the path must be valid and have write # access. # # \*\*\* These paths will have to be modified for each machine that the SIA image analysis software is installed on. # \*\*\* Make sure the paths and filenames are correct before running the software.  OPERATOR\_CONFIG\_FILE\_JSON = "C:\\Users\\markn\\OneDrive\\Xanantec Work\\SIA\\ImageAnalysisSIA\\src\\OperatorConfigSIA.json" ERROR\_LOG\_FILE = "C:\\Users\\markn\\OneDrive\\Xanantec Work\\SIA\\ImageAnalysisSIA\\ErrorsSIA.log"  # Defined SIA system constants. FIRST\_IMAGE\_NAME\_FRAME\_CHARACTER = 13 # first character in image filename that specifies frame number. LAST\_IMAGE\_NAME\_FRAME\_CHARACTER = 16 # last character in image filename that specifies frame number. FIRST\_IMAGE\_NAME\_SAMPLE\_CHARACTER = 7 # first character in image filename that specifies frame number. LAST\_IMAGE\_NAME\_SAMPLE\_CHARACTER = 11 # last character in image filename that specifies frame number.  # Define frames and region of interest in image. # These parameters should only be changed during system calibration when SIA is first started. FIRST\_FRAME\_NUM = -1 # first frame number to process. -1 means whatever the first frame number is in the image set. LAST\_FRAME\_NUM = -1 # last frame number to process. -1 means whatever the last frame number is in the image set. BAD\_EDGE\_LEFT = 350 BAD\_EDGE\_RIGHT = 350  # String constants CSV\_FILE\_PREFIX = "LSCAN-Res-" # File name prefix for CSV files (object file and summary file). LS\_NIR\_PREFIX = "LS-NIR-" # File name prefix for LineScan NIR image files LS\_VIS\_PREFIX = "LS-VIS-" # File name prefix for LineScan visual image files  # Option to downscale the images to fraction of the original size in order to improve runtime performance. # When downscaling, results in CSV files are automatically adjusted to represent full size images. # DOWNSCALE\_FACTOR of 1 would mean no downscaling. DOWNSCALE\_FACTOR of 0.5 means width and height are reduced by half. # Whether DOWNSCALE\_FACTOR is used or not is specified in the OperatorConfigSIA.json file with flag "DOWN\_SAMPLE". # If "DOWN\_SAMPLE" is true, then the DOWNSCALE\_FACTOR specified in this configuration file will be used. # If "DOWN\_SAMPLE" is false, then DOWNSCALE\_FACTOR of 1.0 will be used, irrespective of what it is set here. # Note: DOWNSCALE\_FACTOR should either be 1 or 0.5, with possible 0.25 if absolutely needed for performance. # It should never be greater than 1. DOWNSCALE\_FACTOR = 0.5  # VIS\_THRESHOLD\_LEVEL is a threshold used when segmenting from visual image in function # "segment\_image\_set\_by\_vis\_img()". # This is a threshold that is applied after the background was subtracted out. When the background is effectively # removed, only a small threshold is needed. # We use a low threshold to make sure we identify any pixels belonging to an object. Any artifacts that are a # consequence of the low threshold are removed at later stage. VIS\_THRESHOLD\_LEVEL = 5   # In function segment\_image\_set\_obj\_by\_nir(), constant MAX\_HEIGHT\_TO\_WIDTH\_RATIO is used to filter # out objects of extreme proportions that are likely artifacts. MIN\_OBJECT\_WIDTH = 15 is used to improve # performance so that fewer objects need to be analyzed. MAX\_HEIGHT\_TO\_WIDTH\_RATIO = 25 MIN\_OBJECT\_WIDTH = 15  # Number of rows in each segmented image, if resulting segmented images are to be written out. ROWS\_PER\_SEG\_OUTPUT\_IMG = 4000  # Define error codes (these should never be changed) ERROR\_CODE\_BAD\_FRAME\_NUM = -200 ERROR\_CODE\_VIS\_NIR\_MISMATCH = -201 ERROR\_CODE\_MISSING\_IMAGES = -202 ERROR\_CODE\_IMAGE\_FOLDER\_MISSING = -203 ERROR\_CODE\_UNABLE\_TO\_OPEN\_IMAGE = -204 ERROR\_CODE\_UNABLE\_TO\_ALLOCATE\_MEMORY\_TO\_IMAGE = -205 ERROR\_CODE\_UNABLE\_TO\_LOAD\_IMAGE = -206 ERROR\_CODE\_UNABLE\_TO\_WRITE\_CSV\_FILE = -207 ERROR\_CODE\_UNABLE\_TO\_READ\_CONFIG\_FILE = -208 ERROR\_CODE\_UNKNOWN\_SEGMENT\_ALGO = -209 ERROR\_CODE\_UNABLE\_TO\_CREATE\_FOLDER = -210 ERROR\_CODE\_OUTPUT\_FOLDER\_INVALID = -211 ERROR\_CODE\_NOT\_ENOUGH\_SEG\_IMG\_ROWS = -212 # Not enough rows in segmented image to write out.  WARNING\_CODE\_FEWER\_THAN\_EXPECTED\_FRAMES = -101 |

1. “**OperatorConfigSIA.json**” is a JSON file that contain parameters that can be set by the operator through the HMI. The current **OperatorConfigSIA.json** file is shown below:

|  |
| --- |
| {   "IMAGE\_SCALE" : 7.797271,  "LINE\_SCAN\_RATE" : 4000,  "SYRINGE\_PUMP\_SPEED" : 13.0,  "BAD\_EDGE\_LEFT" : 350,  "BAD\_EDGE\_RIGHT" : 350,  "DOWN\_SAMPLE" : true,  "SAVE\_SEGMENTED\_IMAGES" : true,  "CALC\_SUMMARY\_STATS" : true,  "SEGMENT\_AIR\_AND\_SAND" : true,  "K1" : 1.5,  "K2" : 4.0,  "MIN\_OBJ\_DIAMETER\_UM" : 0.0,  "SEGMENTATION\_ALGO" : "VIS" } |

The above JSON file parameters are:

* "**IMAGE\_SCALE**" : This parameter is the calibrated scale of the image pixels in the camera image, given microns per pixel.
* "**LINE\_SCAN\_RATE**" : The scan speed of the LineScan camera in scans per second.
* "**SYRINGE\_PUMP\_SPEED**" : The calibrated speed at which the syringe pump moves the fluid up the glass viewing column, given in millimeters per second.
* "**BAD\_EDGE\_LEFT**" : How many pixels in each image are to be ignored from the left side. Pixels close to the left and right side of the image may not contain slurry, and also move at different speed than the centre of the image (as shown by Computational Fluid Dynamics analysis), and so should be cut out from the analysis.
* "**BAD\_EDGE\_RIGHT**" : How many pixels in each image are to be ignored from the right side.
* "**DOWN\_SAMPLE**" : An option to down-sample the image before processing it. Downsampling provides a significant performance boost, as shown in section (typically by half along each axis)
* "**SAVE\_SEGMENTED\_IMAGES**" : true,
* "**CALC\_SUMMARY\_STATS**" : true,
* "**SEGMENT\_AIR\_AND\_SAND**" : true,
* "**K1**" : 1.5,
* "**K2**" : 4.0,
* "**MIN\_OBJ\_DIAMETER\_UM**" : 0.0,
* "**SEGMENTATION\_ALGO**" : "VIS"
* Droplet Measurement Logic: Elaborate on how the SIA measures characteristics like

droplet diameter, speed, and axis lengths. Includes formulas and calculation

methods where applicable.

• Output Generation: Clarify how the software generates output, such as CSV files for

measurement data or segmented images, and how this data is formatted.

• Interactions with Control Software: Explains the communication and integration

between the image analysis software and the control software. This includes details

on how the control software runs the image analysis software, defines input

parameters and output data, and the mechanisms for returning any relevant results

or error codes to the control software.

• Performance Considerations: Comments on performance-related aspects, such as

processing speed, memory usage, and scalability.

• Results of testing and verification should they be valuable in understanding how the

system works.