Image Acquisition with Plenoptic Camera

Brenno FERREIRA
Student
M-IRIFS: 1
École Polytechnique de Bruxelles
ULB
Email: fr.brenno@gmail.com
brenno.ferreira.ribeiro@ulb.be

Sarah FACHADA
Doctorate
L.I.S.A.
École Polytechnique de Bruxelles
ULB
Email: sarah.fernandes.pinto.fachada@ulb.be

Mehrdad TERATANI
Professor
L.I.S.A.
École Polytechnique de Bruxelles
ULB
Email: mehrdad.teratani@ulb.be

I. Introduction

A. Project Overview

This project focuses on developing acquisition software for capturing, processing, and saving images taken by a plenoptic camera. Plenoptic cameras, unlike conventional cameras, capture multiple micro-images in a single shot. This unique capability enables three-dimensional (3D) reconstruction, making them highly valuable for various applications, including microscopy, industrial inspection, and biomedical imaging. Utilizing the camera's provided API, the project aims to create a tool for viewing, saving, and processing the captured images. Additionally, the project involves developing a micro-image detection algorithm, which provides essential data for image reconstruction.

B. Plenoptic Cameras

1) Introduction: Plenoptic Cameras, also known as light field cameras, are special cameras that capture information about the intensity and direction of light rays. Indeed, normal cameras only get information about intensity of the light of the scene, losing then, precious information. The additional information about the direction of light rays enables to reconstruct the scene in a 3D representation, which allows re-focus on post-processing, a parallax effect, depth map generation, etc. The capture of an extra information is made possible by the placing of array of micro-lenses between the camera lens and the image sensor. These micro-lenses capture a portion of the scene for a certain angle. As several consecutive micro-lenses would capture almost the same portion of the scene but for a different angle, it is possible to compute the light ray direction based on the same pixels positions on different micro-lenses.

2) Application & Advantages: This technique can transform a simple 2D camera into a device capable of capturing 3D information in a straightforward manner. Plenoptic cameras offer a wide range of applications across various industries. For instance, in industrial inspection, plenoptic cameras provide precise depth information, allowing for accurate measurements and quality control assessments of manufactured components.

3) Hardware Used: The camera used in the laboratory is a Shack-Hartmann Wavefront Sensor 20, a versatile CMOS-based sensor capable of capturing up to 1120 frames per second (fps). It comes with a kit of interchangeable Microlens Arrays (MLAs), making it highly adaptable for various applications. The high-speed sensor heads include a control box with a trigger input and a USB 2.0 port for connection to a PC. This setup allows the control box to analyze the camera image and transmit only the spot positions to the PC, reducing data volume and enabling measurement speeds exceeding 1000 fps. The MLAs included have a lenslet pitch of either 150 µm or 300 µm, are chrome masked or AR coated, and are mounted in exchangeable magnetic holders that ensure correct alignment with each installation. Additionally, each kit includes a removal tool for easy switching between MLAs.

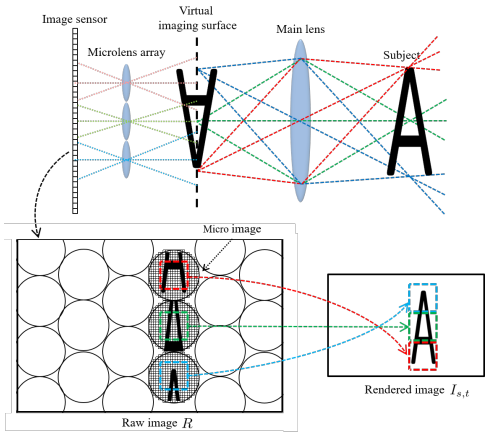


Fig. 1: Plenoptic Camera Diagram[1].

An API is available for users to develop their own custom software to exploit the camera's capabilities. The project initially started by utilizing the manual and the example program provided for debugging purposes. However, it was subsequently redesigned to meet our specific needs. While the manual includes a section on utilizing this API, it is not easily applicable and may be challenging to understand. Nonetheless, comprehensive API documentation is provided to assist users.

C. Calibration Algorithm

The calibration algorithm is essential in the process of image reconstruction. This algorithm operates on a calibration frame, which is a black-and-white image specifically designed to

highlight the micro-lenses against the background. To obtain this calibration frame, a light diffuser is placed in front of the camera and intentionally overexposed. Consequently, areas corresponding to the micro-lenses allow light to pass through, while spaces between the micro-lenses remain dark. This is foundation for the calibration algorithm's functionality which will be explained later.

D. Software Features

- Interface to display captured images.
- Functionality to save images and load images.
- User-friendly navigation and control options.
- Utilization of Thorlabs WFS API for image acquisition
- Real-time image preview
- **Micro-Image Detection/Calibration Process**: Identifies and extracts information about micro-lenses from calibration image.
- Compatibility with different operating systems
- Robust error handling and correction mechanisms

II. Software Presentation

When the software is launched, it is necessary to select the instrument from which the image will be acquired, which is useful when multiple sensors are connected. All data shown here are provided by the WFS API. After selecting the correct instrument, the software performs the necessary procedures to establish a session and prepare to acquire data from the sensor.

As shown in Figure 2, the software was initially built using the Test WFS API — a mock API designed for testing purposes without needing to connect to real hardware. This mock API supplies the app with the necessary data to operate.

Before compiling the software, ensure that the correct API is enabled in the `MyApp.cpp` class. To switch from the Test API to the real WFS API, comment out the Test API instantiation and uncomment the real WFS API instantiation. However, note that if the sensor is not connected to the computer, the software's functionality will be very limited, as it is primarily intended to exploit the capabilities of the API and the hardware.

The home screen, in Figure 3, of the software is designed for simplicity and ease of use, featuring a menu bar with essential functionalities and an image preview window with control buttons as the main frame. The preview window is a key feature, displaying real-time images at a 24-fps update rate, and integrates closely with the calibration algorithm. Users can control the content of the preview by starting real-time mode with the `Start Preview` button or by loading an image with the `Load` button. Additionally, users can save the current frame to disk by clicking the `Capture` button.

The File menu, in Figure 4a, includes the feature of selecting an instrument, allowing users to change the instrument after closing the initial window. Additionally, it replicates the save and load functions for convenience. Another key feature is the `Connect to API` option. This option makes API calls to verify the connection to the hardware. If the API becomes disconnected for any reason, users can use this option to attempt to reconnect.

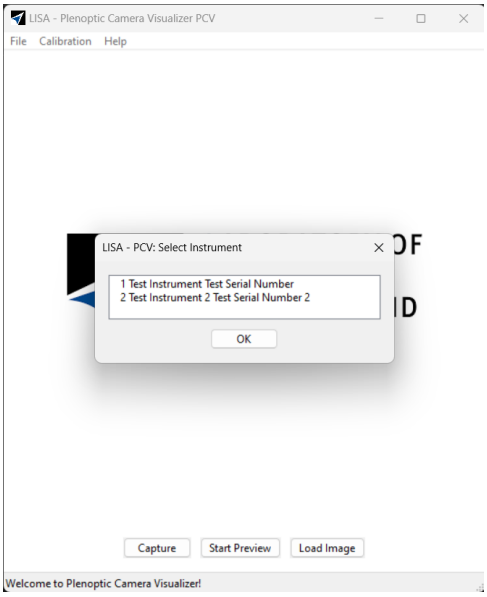


Fig. 2: Instrument Selection.

The Calibration menu, in Figure 4b, includes one single option which is the `Start Calibration` that opens the calibration dialog and debuts the calibration process.

The calibration screen, in Figure 5, is divided into four sections: File Options, Calibration Parameters, Results, and Preview. The File Options allow users to save a single calibration for a specific calibration frame. The Calibration Parameters section lists all the necessary parameters for the calibration algorithm to function effectively. Here is a list of the parameters and their effects on the calibration:

- **Invert Image**: Sometimes inverting the image helps achieve correctly aligned results. This adjustment can be particularly beneficial if the image is underexposed, as the algorithm works with white pixel counts, and inversion might yield better results.
- **Draw Circles, Draw Grid**: These options control whether circles and grid lines are drawn on the result frame.
- **Gaussian Blur Size**: Determines the size of the Gaussian blur kernel. Adjusting the smoothness of the image can improve microlens detection.
- **Block Size**: Sets the block size for adaptive thresholding. Larger block sizes result in rougher thresholding.
- **C**: This is the discount constant for adaptive thresholding.
- **Cluster Distance**: Defines how close white pixels must be to be considered part of the same cluster. This parameter is crucial for calculating the circle center and is very sensitive, often drastically changing results with adjustments of 20 units.

Below the parameters section, there are two buttons : `Default Parameters`, that restores the parameters to

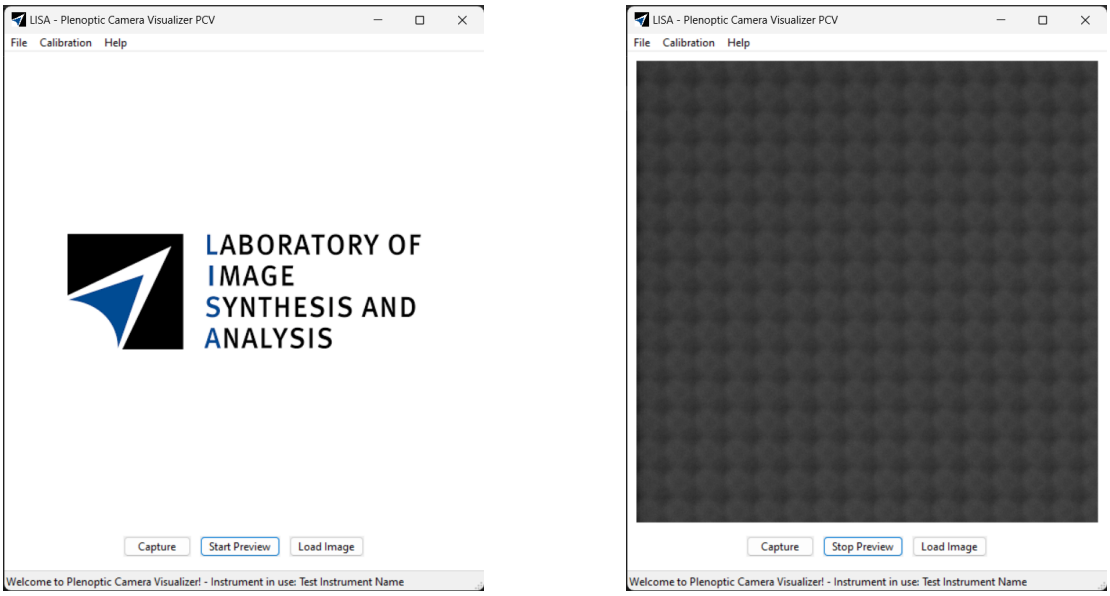


Fig. 3: Software home screen and Preview

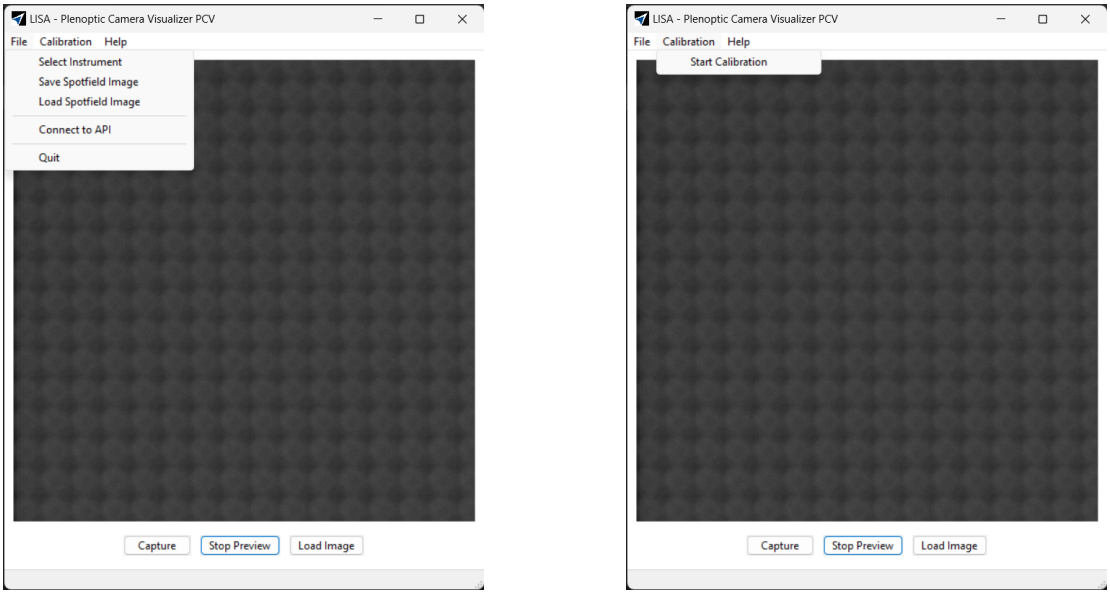


Fig. 4: Menu bar options

their default values; `Calibrate`, which executes the calibration algorithm on the last frame displayed by the preview.

The next window section is reserved to the results obtained from the last calibration. Users can view the coordinates of the reference circles, the grid spacing along the X and Y axes, as well as the mean error across the entire frame. The error for each micro-lens is calculated as the distance between the circle center and the grid cell center. Users can visualize the spatial distribution of errors by clicking the `Show Error Heatmap` button. Additionally, they can inspect the coordinates of the circle positions by clicking on the `Show Circles Position` button.

On the right side, users must name the current aperture before calibrating and they can observe the real-time image captured by the camera. Additionally, users can load a preset image onto the preview interface.

To enhance the accuracy of the calibration process, users must perform five calibrations with different apertures. After each calibration, users can proceed to the next by clicking the `Confirm` button. A calibration counter displayed on the same window keeps track of the number of calibrations performed. Once all five calibrations are completed, the software

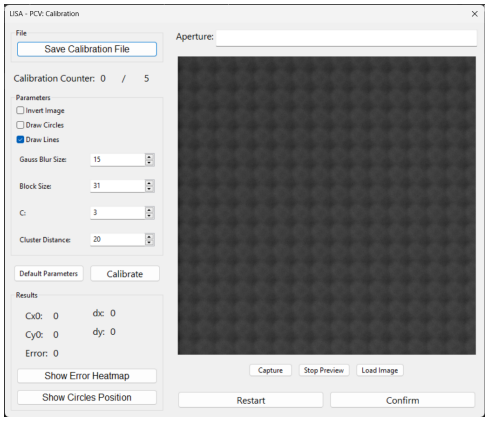


Fig. 5: Calibration Dialog.

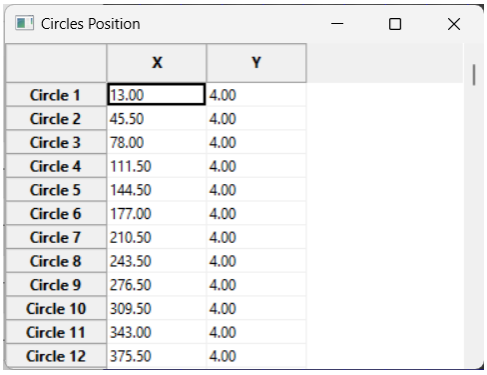
calculates the mean global calibration results and saves them to the designated location. Additionally, users have the option to restart the process using the `Restart` button.

The calibration result is saved as a JSON containing all data and six images, the five intermediate calibrations and the mean calibration image. The JSON has the following format :


```
{   "cx0": 12.569852941176485,
    "cy0": 3.2499999999998814,
    "dx": 32.98235294117649,
    "dy": 33.00000000000003,
    "error": 0.4623129490215421,
    "circles": [[13.0, 4.0]],
    "image": "C:\\calibration_files
        \\17-05.json_calibData_meanResult.
        png",
    "calibrationDataList":
    [{  "aperture": "1",
        "blockSize": 31,
        "c": 3.0,
        "circles": [[13.0, 4.0]],
        "clusterDistance": 20.0,
        "cx0": 12.569852941176485,
        "cy0": 3.2499999999998814,
        "drawCircles": false ,
        "drawGrid": true ,
        "dx": 32.98235294117649,
        "dy": 33.00000000000003,
        "error": 0.4623129490215421,
        "gaussKernel": [15, 15],
        "image": "C:\\calibration_files
            \\17-05.json_calibData_0.png",
        "useInvertImage": false
    }, ...]
}
```


III. Calibration Algorithm

The purpose of the calibration algorithm is to obtain the micro-images position, its diameters and their error if compared to a perfect regular grid of micro-lens array. These are



	X	Y
Circle 1	13.00	4.00
Circle 2	45.50	4.00
Circle 3	78.00	4.00
Circle 4	111.50	4.00
Circle 5	144.50	4.00
Circle 6	177.00	4.00
Circle 7	210.50	4.00
Circle 8	243.50	4.00
Circle 9	276.50	4.00
Circle 10	309.50	4.00
Circle 11	343.00	4.00
Circle 12	375.50	4.00

Fig. 6: Calibration generated data.

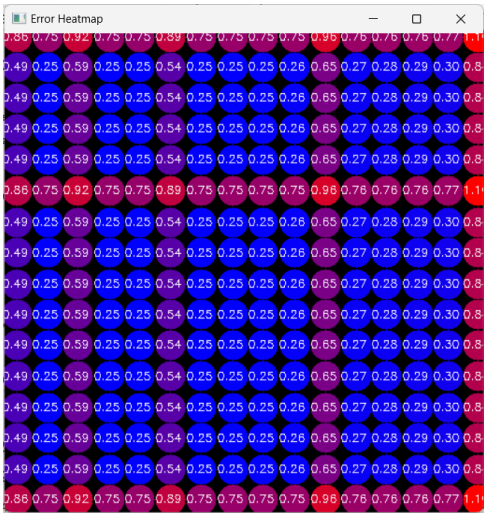


Fig. 7: Calibration generated data.

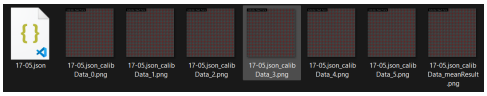


Fig. 8: Calibration generated data.

important information for the study of micro-lens and plenoptic image reconstruction. In order to extract this information, the software processes the calibration frame using specific camera settings. As mentioned in the introduction section, a light diffuser is placed in front of the camera and the scene is overexposed to enhance the contrast between the micro-lens and the background, making it easier for our algorithm to detect the micro-lenses.

The algorithm is managed by the `CalibrationEngine` class and it consists of several stages: data setting, image pre-processing, circle detection, grid computation, and error computation.

a) Data Setting: In this stage, the parameters and the frame to be used in the calibration are set through calls to the `CalibrationEngine` instance. Before setting these, the parameters should be validated to ensure they fall within the valid range. The `CalibrationEngine` class is initially

constructed with default parameters, which typically provide a good calibration.

b) Image Pre-processing: At the beginning, the class checks its parameters to determine if the image needs to be inverted, as sometimes working with the inverted image yields better results. If inversion is required, it employs OpenCV's `bitwise_not` function to accomplish this.

Next, the algorithm converts the source image into a binary image. Initially, it converts the image to black and white by retaining only one channel. Then, it applies a Gaussian blur filter, with the filter size defined by the `gaussKernel` parameter of the calibration engine.

During development, although a median blur filter was tested, the Gaussian blur produced superior results. This is because smoother images generate smoother intensity histograms, which are crucial for the subsequent stage — circle detection. Gaussian blur ensures a continuous and smooth histogram, without many peaks, which is essential for accurate circle detection, unlike median blur.

Finally, the algorithm applies an adaptive threshold to the smoothed image. In contrast to a simple threshold, which may only effectively process specific regions of the image, the adaptive thresholding algorithm operates on a block-based approach. This ensures accurate thresholding for every area of the image, thereby enhancing the contrast between the illuminated micro-lenses and the background. Moreover, this method prevents complete exclusion of certain regions due to uneven light exposure, resulting in a more precise thresholding procedure.

The block size and C constant are parameters for the adaptive threshold. The block size represents the kernel size used in the algorithm, while C is a discount constant that can further enhance the final result by adjusting the thresholding level based on local variations in intensity.

c) Circle Detection: Circle detection involves several steps, starting with generating an intensity histogram to identify peaks, which represent the circle center positions. This histogram is essentially a summation of white pixels along each row of the image. Since the circles exhibit a unique geometry, areas with the highest white pixel counts correspond to the centers. Detecting where these counts peak reveals the circle centers. By transposing the image, centers can also be detected along the other axis. The next step involves peak detection, achieved by computing the derivative of the histogram and clustering nearby values. Clustering is a critical aspect of the algorithm, and the parameter controlling cluster distance significantly influences results. In some cases, the center region may exhibit several consecutive nearby values that need to be merged. Consequently, values within the same cluster are replaced by their mean, which serves as the final output.

Using this method, it is possible to detect all coordinates. Given that the micro-lens array is regular, it is assumed that the circle positions are the combination of all acquired X- and Y-coordinates. The circle positions are then ordered by y-coordinates and subsequently by x-coordinates. This sorting ensures that the circle positions are arranged from the top-left, row by row, to the bottom.

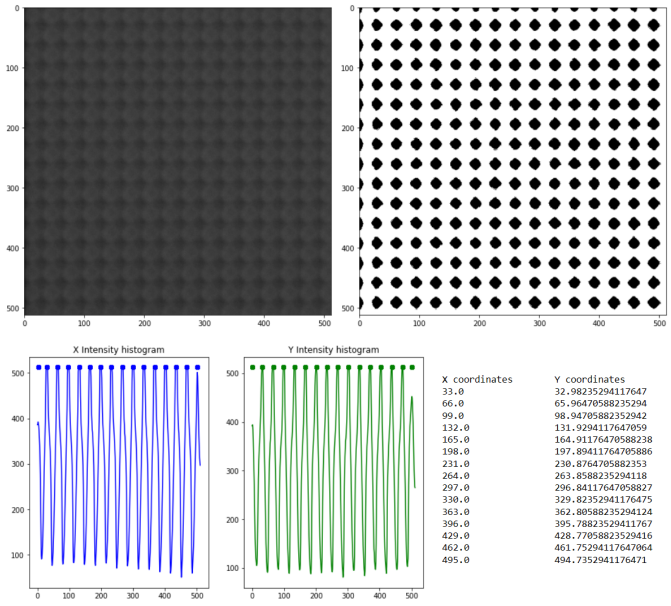


Fig. 9: Circle detection process. The original image is converted to a binary image through gaussian filtering and adaptive thresholding. Then, a histogram is generated and its peaks coordinates extracted. These peaks correspond to the coordinates of the circles center.

d) Grid Computation: Assuming that the micro-lens array is regular, it is possible to establish an equation for a grid where each cell fits one micro-lens. The equation forms a linear relationship between each previously found circle coordinate and the values needed to construct the grid, specifically the reference circle coordinates and the grid spacing in the x and y directions. Essentially, this relationship states that the x-coordinate of a circle is equal to the x-coordinate of the reference circle plus a certain displacement along the x-axis, denoted as dx. The same principle applies to the y-coordinates.

$$\begin{cases} Cx_i = Cx_0 + V_X \cdot dx \\ Cy_i = Cy_0 + V_Y \cdot dy \end{cases} \quad \text{where } i, V_X, V_Y \in N$$

Using this relationship, it is possible to construct a matrix A that represents the relationship between the circle coordinates and the variables of the problem, which are Cx_0, Cy_0, dx, and dy. The circle positions are represented as a single-column vector, where the first half of the vector contains the x-coordinates and the second half contains the y-coordinates. Here is the final equation:

$$B = AX \implies \begin{bmatrix} c_{x_1} \\ c_{x_2} \\ \cdots \\ c_{x_i} \\ \cdots \\ c_{x_n} \\ c_{y_1} \\ c_{y_2} \\ \cdots \\ c_{y_i} \\ \cdots \\ c_{y_n} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & 0 & V_X & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & 0 & n-1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 1 & 0 & V_Y \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 1 & 0 & n-1 \end{bmatrix} \cdot \begin{bmatrix} c_{x_0} \\ c_{y_0} \\ dx \\ dy \end{bmatrix}$$

$$\text{where } \begin{cases} V_X & = i \mod n_x \\ V_Y & = \left\lfloor \frac{i}{n_y} \right\rfloor \end{cases}$$

With n_x and n_y being the number of circles in the corresponding axes. The matrix is constructed once in the algorithm since the relationship is always the same if the circles are well-sorted, meaning by y-coordinates and then by x-coordinates. The algorithm checks if the solution is acceptable before continuing. For instance, if dx or dy is smaller than 0.001, it returns an error since it is not possible to have such small micro-lenses. The units are in millimeters.

This linear problem is solved with SVD method, available in the Eigen library. The solution is done by:

$$B = A \cdot X = USV \cdot X \implies X = V^T S^{-1} U^T \cdot B$$

With the results, the grid can be drawn over the image by starting at the position of the reference circle minus half of the spacing, and then drawing a line at each displacement for both axes. The calibration engine is only responsible for computing these values and providing a `CalibrationData` object containing the image, the reference circle coordinates, the x and y spacing, the mean error, the error heatmap, and the circle position list.

e) Error Computation: The error computation relies on the SVD resolution error. An error vector, $\text{Error} = A \cdot X - B$, represents the difference between the computed circle positions, assuming a perfectly regular grid, and the actual circle positions, B. However, as vector B is structured with x-coordinates preceding y-coordinates, the error vector for a specific cell is $E_i = (\text{Error}_i, \text{Error}_{i+n})$, where n is the number of circles. Thus, the mean error is computed by constructing the error vector for each cell, taking their norm, and then calculating the mean of all of them:

$$\overline{E} = \frac{1}{n} \sum_i \sqrt{E_i^2 + E_{i+n}^2}$$

The function that computes the error returns a pair: $\langle \text{meanError}, \text{errorVector} \rangle$, where the errorVector is the constructed vector E with $E_i = (E_i, E_{i+n})$.

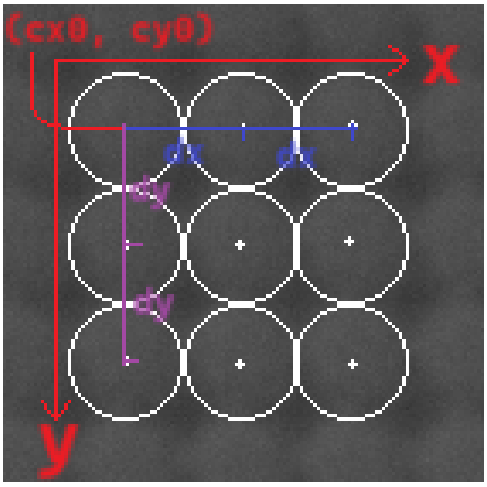


Fig. 10: Grid problem diagram. Each coordinate is composed of the reference circle coordinate and some displacement on x and y direction.

Subsequently, the error vector is used by another function to generate the error heatmap, which illustrates how error is distributed over the image. In the heatmap, the error vector is first normalized:

$$H_i = \frac{E_i - E_{\text{min}}}{E_{\text{max}} - E_{\text{min}}}$$

The region of each micro-lens is then colored from blue (low error) to red (high error), using the following relation:

$$\text{color} = \text{RGB}(255 \times (1 - H_i), 0, 255 \times H_i)$$

This heatmap is stored within the calibration data, and its RGB representation is included in the JSON file stored on disk.

f) Final result: After completing all five calibrations, the calibration controller computes the mean global result. This mean global result consists of several components: a list of circle coordinates where each coordinate represents the mean of all respective circle coordinates across all calibrations, the mean grid spacing for both the x and y directions, selection of the result image with the minimal error among all five calibrations, and finally, recomputation of the mean error and error vectors based on previous mean values. Using these means, a new calibration data is created, which is then saved as the main calibration data in the JSON file. This main calibration data is displayed as the calibration counter reaches 5/5, indicating completion, and the confirm button changes to "finish."

g) Conclusion & Discussion: At the conclusion of this process, the software generates a CalibrationData object containing all the necessary information: the coordinates of the circles and their diameter, determined by averaging the x and y grid spacings and multiplying the result by 2. With this data, it becomes feasible to capture each micro-image, as the algorithm has accurately identified each micro-lens region in

the image, as evidenced by the grid and circle drawings that correctly fit each micro-lens. Additionally, error information is obtained, which is crucial for verifying the correctness of the micro-lenses' manufacture and understanding any potential image deformations that may arise during later image reconstruction.

However, this algorithm is not yet optimal and could benefit from improvement. Several issues have been identified, such as certain parameter combinations not functioning properly, challenges with very bright or dark images where micro-lenses lack contrast, and some images requiring inversion to pass through the calibration process. Additionally, computations are resource-intensive and could be optimized; for example, the matrix A exhibits redundancy due to grid regularity. Moreover, the generated JSON file is excessively large and could be streamlined; for instance, by only including the error matrix rather than the error heatmap RGB matrix. These and other issues will be detailed and discussed in Section VI.

IV. Software Architecture

The code base is structured using the Model-View-Controller pattern. Essentially, the view classes set the user interface and interact with the controller classes to obtain data and respond to user interaction. On the other hand, the controller classes maintain data using model classes.

This software comprises one view class for each window and dialog box, along with their associated controllers. It also includes special controllers such as `WfsApiController`, `TestApiController`, `InstrumentController`, `ImageController`, and `CalibrationController`. Additionally, the code implements a special object called Preview, which is responsible for displaying images obtained from the camera. The software also features an event mechanism that allows different controllers to communicate and transmit data without having to directly call each other.

a) `WfsApiController` & `TestApiController`: These classes serve as interfaces to the WFS API. Any class requiring data from the API should have a reference to these classes and utilize the existing API calls to obtain data. This approach consolidates API calls into a single class and prevents scattered API calls throughout the codebase.

The `TestApiController` acts as a mock API, mimicking the methods of the `WfsApiController`. However, all data generated by this class is either fake or generated, allowing the software to execute correctly without requiring a connection to the camera. This class facilitated remote development of the software. Once a feature was implemented, it was tested in the laboratory with the real API.

Due to this setup, when building the software, users must ensure it is built with the correct API by modifying one line of code in the `MyApp` class.

b) `InstrumentController`: This class is responsible for controlling the `InstrumentSelectionDialog`, where users select the instrument they want to use. It retrieves data from the API and makes the necessary API calls to establish the connection to the API and utilize the camera sensor.

c) `ImageController`: This is a controller class that does not manage a view but instead serves as a detached controller interacting with the API to obtain an image. It is owned by the `MyApp` class, and its reference is provided to each controller managing a Preview. The Preview object requests images from the API through the ImageController. The ImageController needs to receive the reference of the Instrument model object, which is transmitted by the InstrumentController using event mechanisms. Whenever a new instrument is selected, the InstrumentController publishes an `InstrumentSelectedEvent`, which is in the subscription list of the ImageController. This event updates the instrument pointer to the new one. The instrument model contains the data used by the API to communicate with the camera.

d) `CalibrationController`: This class manages the calibration window and sets up the calibration process. It handles user interaction, parameter validation, preview management, and the calibration process flow. The `CalibrationController` interacts with the preview to obtain the image and uses an instance of the calibration engine to obtain calibration data. Additionally, this controller ensures that the user performs five calibrations before computing the mean global calibration result.

e) Preview: This object comprises several classes: `IPreview`, an interface implemented by other classes requiring a preview; `PreviewController`, responsible for managing the preview and its functionalities; and `PreviewPanel`, a UI element containing the image container and buttons.

To set up a preview in a window, the controller should implement the `PreviewHolder` interface and instantiate a `PreviewController` object. The view should then instantiate a `PreviewPanel` and append it to the current layout. During the controller construction, the holder class binds all classes together: the preview controller receives the preview panel through the holder's calls. Once the binding is complete, the preview controller can utilize its timer to update the image at a frame rate of 24 fps.

The image displayed in the preview is retrieved through the `ImageController`, which is passed as a reference to the `PreviewController`. This controller is responsible for handling the image source, whether it be captured from a camera or loaded from a file.

While this construction works, it may not be optimal and could be simplified. Exploring ways to streamline this process could lead to improved efficiency and code elegance.

A. Event Mechanism

The event mechanism is relatively straightforward. Each event type has its own class, which encapsulates its name and a pointer to some data. The `EventDispatcher` class is a singleton that provides the `subscribe` and `publish` methods.

If a class needs to receive data from a controller that is not directly related, it should subscribe to the event in its constructor. During subscription, the class designates the method it should call whenever the event occurs.

A class that needs to communicate data instantiates the event object of the correct class, fills its data if necessary, and uses the `publish` method of the `EventDispatcher`. This method traverses the subscription list of the `EventDispatcher`, checks for the event type, and sends the event to the corresponding classes.

For example, after the selection of a new instrument, the `InstrumentController` should communicate this change to all controllers that use the instrument model reference, to prevent errors. The `InstrumentController` instantiates the `InstrumentSelectedEvent` and sends it to the `EventDispatcher` using the `publish` method. The `ImageController` requires the current instrument to draw the image information, so it subscribes itself to the `InstrumentSelectedEvent` type. Whenever this event occurs, it calls the method `HandleNewInstrumentSelect(InstrumentSelectedEvent event)`, which sets the new pointer into the instrument attribute of the `ImageController` object.

There are different types of events: data-driven events, such as `InstrumentSelected`, and user-interaction-driven events, such as `CalibrationStartEvent` or `ExitEvent`.

V. Results

Here are the results obtained from the algorithm calibration. The aperture name is not known for the following image, but it is used as a counter.

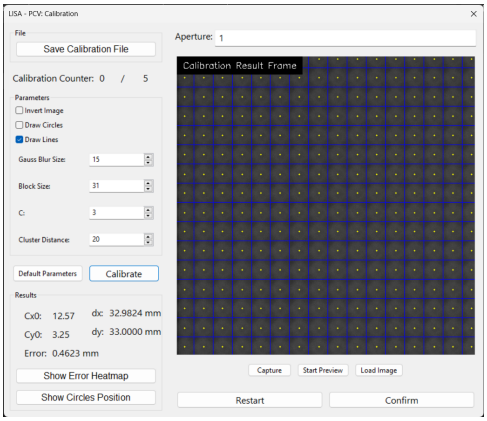


Fig. 11: Calibration 1.

The JSON generated at the end of this calibration is annexed to this document.

The correctness of the algorithm is verified because the specified diameter of the micro-lens, as presented by Thorlabs, is 33mm. The algorithm computes the diameter as the mean of dx and dy, which is $d = 33.4303$ mm. The error is recomputed for the mean solution found and the mean circle positions. However, this value seems too high to be correct, especially when compared with individual calibration errors. This discrepancy could indicate an error in the error computation or in the mean result construction.

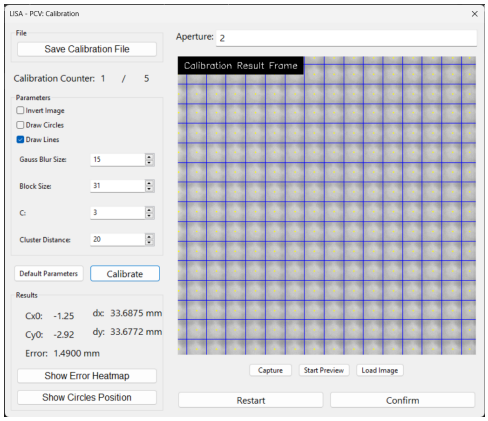


Fig. 12: Calibration 2.

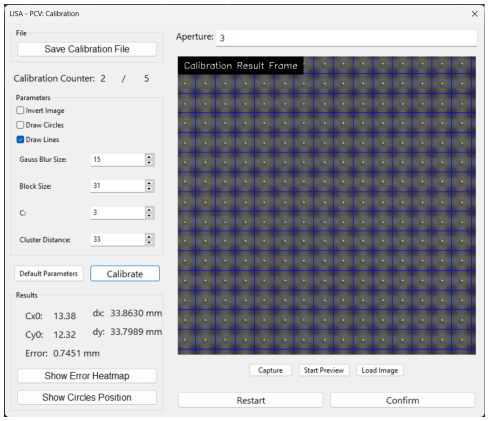


Fig. 13: Calibration 3.

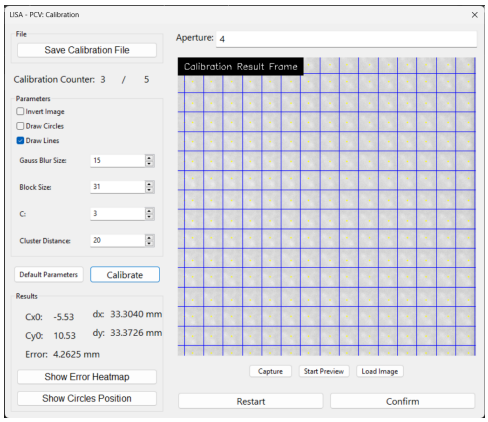


Fig. 14: Calibration 4.

VI. Future Works

The software has many improvements to be made, including code and process simplification, architecture cleanup, fixing computation bottlenecks and bugs, and performance enhancement. Additionally, new features need to be implemented to

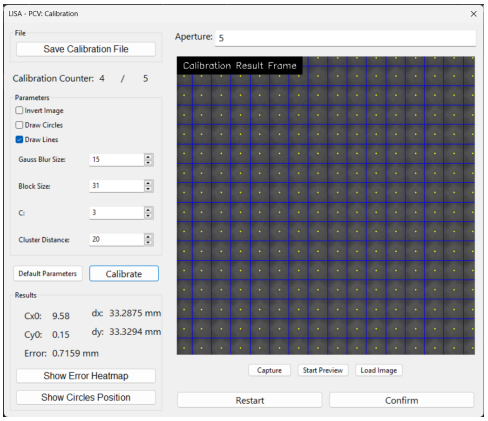


Fig. 15: Calibration 5.

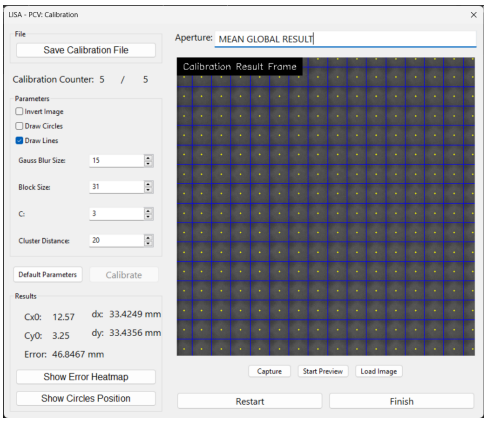


Fig. 16: Final Calibration Results.

make it more powerful. In this section, some of these problems and ideas are going to be discussed, and solutions will be proposed.

A. Problems, Bugs & Improvements

1) Preview Object Instantiation: Setting up a preview in a window is currently too complex, involving multiple instantiations and dependency injections. Simplifying this process by consolidating the instantiation steps and reducing dependencies can improve usability and maintainability. Implementing a factory pattern or a builder pattern might help streamline the setup.

2) Preview Frame Rate: Although the preview is set to 24 fps, it drops drastically when the frame changes significantly, despite the camera being capable of delivering up to 1000 fps. Investigating the cause of this bottleneck, whether it is in image processing or data transmission, and optimizing the relevant parts of the code could stabilize the frame rate.

3) Error Computation: There is a discrepancy between the final result error and the individual calibration errors. This issue suggests a potential problem in the error computation or the mean result construction. A thorough review and debugging of the error calculation algorithm are necessary to ensure accurate error representation.

4) Grid Computation Speed-up: The grid computation can be sped up by better understanding the underlying equation and leveraging the repetitive pattern of matrix A. Optimizing the matrix operations and potentially parallelizing some computations can lead to significant performance improvements.

5) Installation Process: The installation process may be too challenging for users without a technical background, especially concerning library installations and unresolved compilation errors due to linking issues. Creating a comprehensive installation guide, along with automated installation scripts, can make the process more user-friendly.

6) Large JSON Files: The generated JSON files are large due to the inclusion of extensive circle coordinate lists and the error heatmap RGB matrix. Encoding the circle coordinates and storing only the error vector instead of the full heatmap can reduce the file size. Using a more efficient data serialization format could also help.

7) Quantity of Files for a Single Calibration: Currently, a single calibration generates multiple files, which can be cumbersome to manage. Encoding the image data inside the JSON file or consolidating all calibration data into a single archive file could streamline data handling and storage.

B. Future Features

1) Hexagonal MLAs: Supporting hexagonal micro-lens arrays (MLAs) involves describing hexagonal positions and implementing them using linear transformations of the existing matrix. Since hexagonal positioning is a shift of some rows, it can be translated into a rotation of one of the axes, allowing support for both vertical and horizontal hexagonal arrangements.

2) Calibration Results Viewer: Users should be able to load a calibration file or individual calibration data and navigate through it within the software, without needing to inspect the JSON manually with a code editor. Implementing a dedicated viewer for calibration results can enhance usability and data accessibility.

3) Data Exportation & Custom JSON Results: Allowing users to select which data to export, such as only circle positions and grid spacing or only errors, can provide more flexibility. Customizable JSON export options can cater to different user needs and reduce unnecessary data storage.

4) Load a Calibration into the Calibration Process: Users should be able to load a previous calibration into the current calibration process, enabling incremental improvements. Adding a feature to load calibration files, similar to the existing save calibration functionality, can facilitate this process.

5) Exclude Micro-lens at Image Extremities from Calculations: Micro-lenses at the image extremities are often distorted and contribute to calculation errors. Excluding these micro-lenses from calculations could increase accuracy. This change would require modifying matrix A and reorganizing the system resolution to focus only on interior micro-lenses.

6) Drawing Capabilities in the Preview: Currently, the preview only displays the received image. Enhancing the preview to draw lines and circles on the image, based on

calibration data, would allow users to check grid regularity. This feature should be controlled by a display menu, enabling users to toggle the overlay drawings on and off.

7) Menu Displaying WFS API Configuration Parameters: A simple read-only menu where users can view all the configurations set for the WFS API would improve transparency and user understanding of the system settings. This feature would help in debugging and verifying the setup.

8) Error Vector Display on Preview: Adding an option to draw the error vector for each cell and overlay the error heatmap on the preview image can provide a visual representation of errors. This feature would help users quickly identify and address calibration issues.

9) Ability to Zoom and Move Image: Implementing zoom and pan functionalities for the preview image, with or without overlay drawings, can enhance the user experience. These features would allow for detailed inspection of specific areas and better visualization of calibration data.

VII. Conclusion

In conclusion, the development and implementation of the software have been guided by a meticulous approach, leveraging various algorithms and design patterns to achieve the desired functionality. The software's foundation lies in the Model-View-Controller architecture, providing a structured framework for handling user interface, data management, and interaction. Through the use of specialized controllers such as `WfsApiController`, `TestApiController`, `InstrumentController`, `ImageController`, and `CalibrationController`, the software seamlessly integrates with external APIs and hardware, facilitating the calibration process and image acquisition.

Critical components of the algorithmic pipeline, including image preprocessing, circle detection, and error computation, have been carefully crafted to ensure accuracy and efficiency. Techniques such as adaptive thresholding and peak detection play pivotal roles in enhancing image quality and extracting meaningful information from the data. Moreover, the software's ability to generate comprehensive calibration data, including circle coordinates, grid spacing, and error metrics, empowers users to assess the quality of micro-lens manufacturing and address any potential image deformations.

Looking ahead, there are several avenues for further improvement and expansion. Addressing issues such as code simplification, architecture cleanup, and performance optimization will enhance the software's usability and maintainability. Additionally, incorporating new features such as support for hexagonal micro-lens arrays, a calibration results viewer, and data export capabilities will broaden its utility and appeal to a wider range of users.

Overall, the software represents a significant step forward in micro-lens array calibration, offering a robust and versatile solution for image processing and analysis. By continuing to refine and innovate upon its existing capabilities, the software has the potential to make meaningful contributions to the field of plenoptic imaging.

Appendix A
Calibration Results JSON


```
{
    "cx0": 12.569852941176485,
    "cy0": 3.2499999999998814,
    "dx": 33.42485873195434,
    "dy": 33.43562825899592,
    "error": 47.83049397903899,
    "circles": [...]
    ],
    "image": "C:\\calibration_files\\
        results.json_calibData_meanResult.
        png",
    "calibrationDataList": [
        {
            "aperture": "1",
            "blockSize": 31,
            "c": 3.0,
            "circles": [...]
            "clusterDistance": 20.0,
            "cx0": 12.569852941176485,
            "cy0": 3.2499999999998814,
            "drawCircles": false,
            "drawGrid": true,
            "dx": 32.98235294117649,
            "dy": 33.00000000000003,
            "error": 0.4623129490215421,
            "gaussKernel": [
                15,
                15
            ],
            "image": "C:\\
                calibration_files\\results
                .json_calibData_0.png",
            "useInvertImage": false
        },
        {
            "aperture": "2",
            "blockSize": 31,
            "c": 3.0,
            "circles": [...]
            ],
            "clusterDistance": 20.0,
            "cx0": -1.2499999999999938,
            "cy0": -2.9227941176471472,
            "drawCircles": false,
            "drawGrid": true,
            "dx": 33.6875,
            "dy": 33.67720588235293,
            "error": 1.4899950842864227,
            "gaussKernel": [
                15,
                15
            ],
            "image": "C:\\
```

```
        calibration_files\\results
        .json_calibData_1.png",
    "useInvertImage": false
},
{
    "aperture": "3",
    "blockSize": 31,
    "c": 3.0,
    "circles": [...]
    ],
    "clusterDistance": 33.0,
    "cx0": 13.378952991453016,
    "cy0": 12.315384615384344,
    "drawCircles": false,
    "drawGrid": true,
    "dx": 33.86295787545788,
    "dy": 33.79890109890115,
    "error": 0.7451456600142117,
    "gaussKernel": [
        15,
        15
    ],
    "image": "C:\\
        calibration_files\\results
        .json_calibData_2.png",
    "useInvertImage": false
},
{
    "aperture": "4",
    "blockSize": 31,
    "c": 3.0,
    "circles": [...]
    ],
    "clusterDistance": 20.0,
    "cx0": -5.525183823529378,
    "cy0": 10.52928921568638,
    "drawCircles": false,
    "drawGrid": true,
    "dx": 33.30398284313729,
    "dy": 33.372622549019574,
    "error": 4.2625426260612365,
    "gaussKernel": [
        15,
        15
    ],
    "image": "C:\\
        calibration_files\\results
        .json_calibData_3.png",
    "useInvertImage": false
},
{
    "aperture": "5",
    "blockSize": 31,
    "c": 3.0,
    "circles": [...]
    ],
    "clusterDistance": 20.0,
    "cx0": 9.58333333333333,
    "cy0": 0.15441176470561357,
    "drawCircles": false,
    "drawGrid": true,
    "dx": 33.28750000000001,
    "dy": 33.32941176470589,
    "error": 0.7158920885810436,
    "gaussKernel": [
        15,
        15
    ],
    "image": "C:\\
        calibration_files\\results
        .json_calibData_4.png",
    "useInvertImage": false
},
{
    "aperture": "",
    "blockSize": 31,
    "c": 3.0,
    "circles": [...]
    ],
    "clusterDistance": 20.0,
    "cx0": 12.569852941176485,
    "cy0": 3.2499999999998814,
    "drawCircles": false,
    "drawGrid": true,
    "dx": 33.42485873195434,
    "dy": 33.43562825899592,
    "error": 47.83049397903899,
    "gaussKernel": [
        15,
        15
    ],
    "image": "C:\\
        calibration_files\\results
        .json_calibData_5.png",
    "useInvertImage": false
}
]
}
```


References

[1] "Extracting Multi-View Images from Multi-Focused Plenoptic Camera." Accessed: May 27, 2024. [Online]. Available: https://www.fujii.nuee.nagoya-u.ac.jp/~s.fujita/projects/RenderingMV/.