Plenoptic Imaging in Particle Tracking Velocimetry Experiments

by

Jake Hadfield

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Mechanical Engineering

Mechanical Engineering

University of Alberta

Abstract

The aim of this study is to develop a technique capable of performing three-dimensional particle tracking by using images from a plenoptic (or light field) camera. Use of a plenoptic camera enables the collection of three-dimensional positional data within a volume, which would normally require more complex, multi-camera setups. A plenoptic imaging system is simpler to set up, and can be scaled to micro-scale experiments easily. Calibration of the camera is necessary for performing particle tracking, and a major contribution of this work has been developing a Matlab code based on past works to allow for calibration. Two techniques for three-dimensional particle location were developed over the course of this study. One of these relies upon existing commercial software for processing the plenoptic images into refocused images and depth maps. As the commercial software is not specifically aimed at finding particles this approach is inefficient and hard to customize. A second method was developed to overcome these limitations. This method has been named the ETC method after its use of epipolar triangular connections, is capable of extracting 3D particle locations from the raw plenoptic images. Of the two, the refocusing-based approach was determined to be the more accurate method through the interrogation of a toroid vortex in water. To investigate the limitations of the ETC approach, an experimental system for quantifying its uncertainty was developed. A study of a matched-refractive index droplet moving through a slot was also conducted to test the viability of the technique for performing microscale experiments. Based on the results of these experiments, hypotheses regarding the limitations of the ETC approach and recommendations for improving it have been made.

Acknowledgements

I owe a debt of gratitude to Hirad Soltani for the use of a slightly modified version of his rising droplet rig and all his research into the refractive index matched fluids used. I would also like to thank Michael Bayans for his 3D-printing expertise which went into the design of both experiments. Most importantly I’d like to thank my supervisor Dr. David Nobes for all his expertise and support, without which none of this work would have been possible.

This work was carried out with financial support from the Natural Sciences and Engineering Research Council (NSERC) of Canada through their Discovery and Collaborative Research and Development (CRD) programs, the Canadian Foundation for Innovation (CFI), and an industrial sponsor, RGL Reservoir Management.

Table of Contents

List of Figures

List of Tables

CHAPTER 1 INTRODUCTION

When studying fluid dynamics, it can be important to consider a full 3D fluid field. Experimental analysis of fluid dynamics is modernly conducted through imaging-based fluid velocimetry methods, as in the best case, these can provide full time-resolved, three component, three dimensional velocity fields [1]. These can be further interrogated to obtain greater insight into the dynamics of the fluid flow phenomenon under investigation. Developing techniques which can overcome the challenges associated with imaging a 3D field of fluid is important to further improve the analysis of fluid dynamics. This thesis describes and analyzes an approach which uses single-camera plenoptic (also known as light field) imaging to obtain particle-seeded fluid images upon which time-resolved particle tracking velocimetry is performed. The single-camera nature of the technique allows it to be applied with equal ease in both macro-scale and micro-scale experiments.

1.1 Overview of General Fluid Image Velocimetry

There are two major classes of techniques used in image-based fluid velocity measurements. These are particle image velocimetry (PIV) [2] and particle tracking velocimetry (PTV) [3]. Both techniques involve seeding the fluid with particles of a similar density to the fluid, which trace the flow of the fluid so that the fluid motion can be inferred. Illumination is generally accomplished using high-intensity sources such as lasers. The particles are then imaged using a camera system at a frequency designed to make the movements of the particles between individual frames appropriate for the processing technique being used. PIV uses a windowed cross-correlation algorithm which detects local similarities in groups of pixels, producing an estimation of the fluid velocity in each group [2]. PTV focuses on tracking each individual particle in the image, usually by considering each particle in the context of nearby particles [3]. PTV has the potential for higher resolution, since every particle in the volume is considered separately. This comes at the expense of the consistency of the resolution inherent in the grid of vectors generated by PIV. Application of time-resolved versions of each of these techniques enables temporal filtering of the data [1], which can help to reduce uncertainty and enable the calculation of higher-order derivatives describing the fluid motion.

When using either PIV or PTV, it is important to implement a calibration method that will allow the results to be scaled from image-side, pixel-space to object-side, physical space while accounting for distortion effects associated with the system optics [4]. Different methods for doing so are available, but the most common make use of multiple images of known targets [5], [6] to determine the parameters of the optical system. The results can then be interpreted in the context of the physical distances involved.

Commonly, three-dimensional data in fluid experiments is obtained through the use of a multi-camera approach such as tomographic PIV [7]. Tomographic PIV requires that all the cameras are oriented such that they are all focused exactly on the same imaging volume, which generally involves the adjustment of nine significant degrees of freedom [7], [8] for each camera used – three positional, three rotational, two associated with a Scheimpflug adapter, and one for the lens focus. As a result, orienting all of the cameras correctly can be a time-consuming process that can be a large source of uncertainty [7], [8]. The angle of the cameras also can result in issues with refraction through any windows used to view the experiment. Further, the imaging volume in a tomographic imaging experiment is generally much smaller in the out-of-plane direction than in-plane [7], [8], which can be restrictive for some experiments. The computational power required to process tomographic imaging data is also substantial, leading to extended computation times [8].

It can be especially challenging to collect time-resolved 3D data in microscale experiments. Orienting the cameras to undertake a tomographic imaging experiment can be difficult if not impossible where optical access from multiple views is not available or would be at too low an angle for proper perspective viewing [7], [8]. Many alternative methods for obtaining 3D velocity data at the microscale have been investigated. One example method uses high-frequency scanning approaches, which sequentially obtain multiple images at different depths before allowing a larger time-step for the fluid to move [9]–[11]. The imaging frequency of this technique is reduced by a factor of the number of scans taken through the volume, which limits the velocity range for which the method is appropriate. Scattering-based methods [12] are also possible, but these have limited resolution due to how many pixels on the camera sensor must be dedicated to each particle and the extreme difficulty of handling particle overlap. Other methods have applied specially-designed stereo microscopes [13], but as a stereo method this is limited to a plane of 3D velocity measurements rather than a volume.

To summarize, tomographic PIV is the most widely used technique for obtaining 3D velocity measurements in a volume of fluid. Its major limitations include its small depth of field relative to its in-plane imaging region, its long processing times, and its complex setup requirements. The latter of these make it particularly difficult to implement in micro-scale experiments. Other micro-scale techniques lack the ability to make instantaneous volumetric velocity measurements. A technique capable of large-volume imaging with easy setup, scalability and faster processing would be useful to make fluid flow interrogation more straightforward. Plenoptic particle tracking velocimetry is presented as a potential solution.

1.2 Overview of Plenoptic Imaging

Plenoptic imaging [14], also known as light field imaging [15], is a technique whereby information about the angular components of light incident to the system is preserved by using multiple integrated optical systems. This allows three-dimensional data about a scene to be captured. Both multi-camera [16] and single-camera [14] variants of the technique have been developed. The multi-camera approach, while effective, has similar logistical issues as those discussed regarding tomographic imaging, amplified by the larger number of cameras required. This study focuses on the use of single-camera plenoptic imaging in fluid velocity measurement experiments.

Two parallel branches of single-camera plenoptic imaging technology have been developed. These can be referred to as ‘standard’ and ‘focused’ plenoptic imaging. The ‘standard’ technique was developed initially in 1903, first using pinholes [17] instead of lenses [18], [19] to provide multiple views of an imaging field. The viability of the technique was limited by the inability to interrogate the images to generate anything particularly useful. The digitization of imaging led to recent improvements in computer processing of the images [14], [20]–[22], which have made the technique viable for general imaging by enabling computationally refocused images and depth data to be generated from plenoptic images. ‘Focused’ plenoptic cameras were developed later [23] and further improvements of the resolution in the image reconstruction algorithms have been made as well [24]–[26]. The focused technique improves the in-plane spatial resolution of the camera by sacrificing some of the angular information about incident rays, which leads to reduced out-of-plane spatial resolution [23]–[26]. The focused plenoptic technique has been further expanded upon to create ‘multi-focus’ plenoptic cameras [27], which redistribute the depth regions over which angular information is preserved to improve out-of-plane resolution. Plenoptic cameras and associated image reconstruction software are now commercially available [28].

In all of these cameras, a micro-lens array is positioned between the main lens and the camera’s sensor, essentially creating thousands of individual optical systems. The exact position and features of the array define the type of the plenoptic imaging system. In a ‘standard’ plenoptic camera, the focal points of the micro-lens array are aligned with the camera sensor [21]. ‘Focused’ plenoptic cameras place the micro-lens array focal point at a different distance relative to the sensor [25]. ‘Multi-focus’ plenoptic camera have micro-lenses of multiple different focal lengths organized in a repeating pattern, with each micro-lens type’s focal points placed at different distances away from the sensor [27]. Standard plenoptic cameras have the advantage of having the maximum available amount of data about the angle of the incident light rays and thus generally have better depth-resolution capability [21]. Focused plenoptic cameras sacrifice some depth-resolution in favor of improved in-plane resolution [25]. Multi-focus plenoptic cameras improve depth resolution by having each of the different micro-lens types targeted at a different depth range, expanding the overall imaged depth-of-field while mostly maintaining resolution [27].

In contrast to tomographic imaging, only two degrees of freedom are significant in a single-camera plenoptic imaging experiment [27]. These are the adjustment of the main lens focus and the distance of the camera from the experiment. A plenoptic camera also only requires only a single route of optical access to the region of interest, which can be a significant advantage in mini- and micro-scale experiments. If adequate processing algorithms for plenoptic fluid velocimetry can be developed, plenoptic cameras could serve as a simpler alternative to multi-camera tomographic approaches that could be easily implemented at the microscale. Additionally, the imaging volume of a plenoptic camera can be adjusted through selection of the main lens focal length and focus distance, even to the extent that the out-of-plane distance in the imaging volume is similar to the in-plane distance.

Standard plenoptic cameras have been applied to experimental fluid dynamics, using tomographic reconstruction techniques such as MART to produce 3-dimensional images on which standard particle imaging velocimetry (PIV) approaches can be applied [29]. The main drawback of this approach is that it tends to elongate found particles in the out-of-plane direction [30], [31]. This has been hypothesized to be one of the major sources of error in the out-of-plane velocity calculation of the technique in studies which have aimed to quantify the resolution of the this technique [30], [31]. Some particle location, size measurement, and tracking has also been conducted, and the measurement uncertainty of this technique has been determined [32]. However, this technique has been aimed at detecting large-scale particles (greater in size than the images produced by individual micro-lenses) which would limit its resolution in a PTV experiment. A 3D calibration algorithm has also been developed for this technique [33].

Multi-focus plenoptic cameras have also been employed in PIV and PTV applications using a commercial software package (RxFlow 3.1, Raytrix GmbH). Fixed-field testing has been performed to examine the performance of the technique [34], which found higher uncertainties in out-of-plane measurements relative to in-plane measurements. Little discussion of the exact methodology undertaken by the commercial package seems to be available. Researchers have indicated that this approach can only be used for low seeding densities, as the approach's particle location method tends to generate ghost particles. Small particle displacements are also required, since the algorithm employs a nearest-neighbor tracking method [34].

1.3 Research Aims

Most multi-focus plenoptic systems are aimed at computationally reconstructing large, continuous surfaces [27]. The first goal of the presented research was to adapt such a method for locating particles in a 3D volume. This leads to the second goal of the research, which was to implement a calibration approach capable of dimensionally scaling the located particles from image-space to object-space. The third goal was to implement a time-resolved 3D particle tracking method capable of both tracking and temporally filtering the data. The results of implementing these methods led to the final goal, which was to develop a new approach capable of locating and tracking particles in raw multi-focus plenoptic images.

To provide context, Chapter 2 will feature a complete discussion of the methodology involved in multi-focus plenoptic imaging. The bulk of the contributions made by this work take place in chapters 3 and 4. Chapter 3 will cover the two methodologies that have been developed to extract 3D particle locations from the plenoptic images, and the calibration method used to scale the results to physical space. Chapter 4 will discuss the particle tracking approach that has been implemented. Quantitative and qualitative assessments of the performance of the approaches are undertaken through their application to three experiments. Chapters 5, 6 and 7 will cover the three experiments that have been undertaken in this research. In the first, measurements of fixed particle fields, moved in a known way, were undertaken using one of the particle location approaches. This enables comparison of the measured results to the known particle motions, which allows an estimate of the uncertainties associated the approach to be made. The second experiment tests the performance of this approach at the micro-scale by examining the motion of the fluid within and around a rising droplet as it passes through the imaging region. The final experiment involves the interrogation of a ring vortex using both of the methodologies, which is used to compare the performance of the two methodologies when undertaking macro-scale experiments. Chapter 8 will summarize and conclude this thesis. Appendices complete the thesis, which include the Matlab scripts developed to process the plenoptic images and supplementary figures.

CHAPTER 2 MEASUREMENT PRINCIPLE OF FOCUSED PLENOPTIC IMAGING

The focused plenoptic cameras used in this study are a pair of commercial plenoptic cameras (R5; Raytrix GmbH), that differ based on the positioning, focal length, and lens size of the micro-lens array. This produces different effective *f*/numbers for each system. An *f/#* 2.4 camera is used for macro-scale imaging and an *f*/# 26 system is used for micro-scale imaging. These are multi-focus plenoptic cameras (MFPC), meaning that they are each equipped with micro-lens arrays that have interlaced lenses of 3 different focal lengths. Information about the micro-lens focal lengths or the position of the array relative to the sensor has not been released by the manufacturer, and must be extracted through the calibration approach. The cameras are capable of acquiring 2048 by 2048 pixel images at frame rates up to 180 fps. To fully understand the optical system of these cameras, the following chapter will discuss the optical properties of a general MFPC system. The bulk of the understanding of MFPC systems in this section comes from [27], [35], [36].

A schematic of an MFPC camera configuration used for this study is given in Figure 1. Here, the micro-lens array is mounted at a specific distance B in front of the camera sensor. The term *virtual depth* (denoted v) is used to describe an image-space distance as a ratio of B. Physical depth-locations in image-space can then be described in terms of $v \cdot B$. This allows the relative positions of projected images to be comparable for a given micro-lens array and sensor, regardless of the main lens' focal length f_L or its imaging distance T_L.

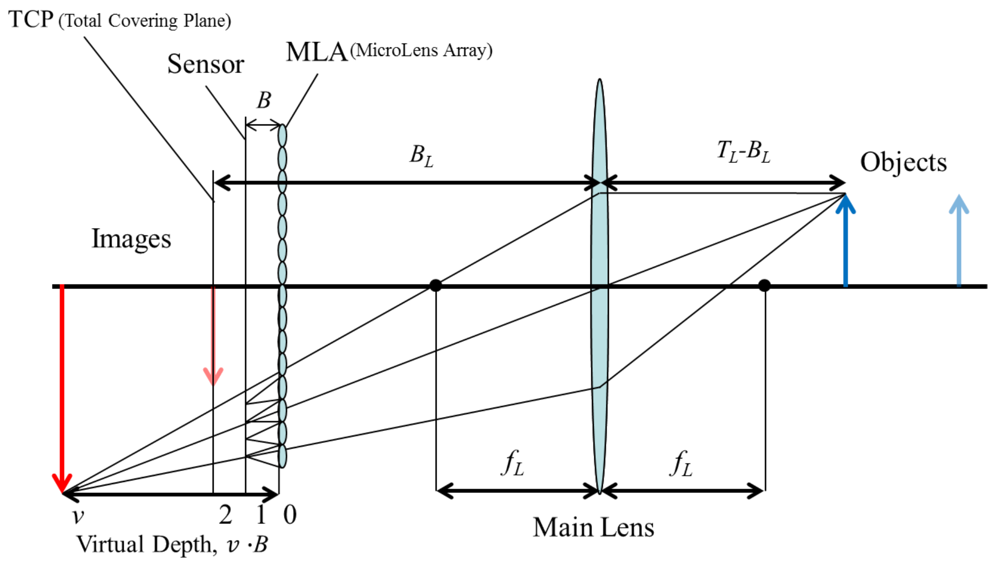


Figure 1 - Optical arrangement of a single-MLA focused plenoptic imaging system. A single set of ray-paths for one sample object is shown. Microlens focusing behavior assumed ideal for the object for simplicity.

In a plenoptic imaging system, the main lens of the camera is used to project an image of an object into the image-space of the camera. If the main image is formed behind the sensor and micro-lens array as shown in Figure 1, then the main image is said to be ‘virtual’. More micro-lenses will view the same point on an object and *v* will increase as the object being imaged gets closer to the main lens, with *v* approaching infinity as the object approaches the main lens’ focal point. It is also possible to form a real image in front of the micro-lens array with the main lens, in which case the virtual depths would be considered by this approach to take negative values. This configuration is not used here because theoretically, its imaging space projected into object-space will be between some minimum distance and infinity, which reduces the useful virtual depth space and the overall resolution of the technique.

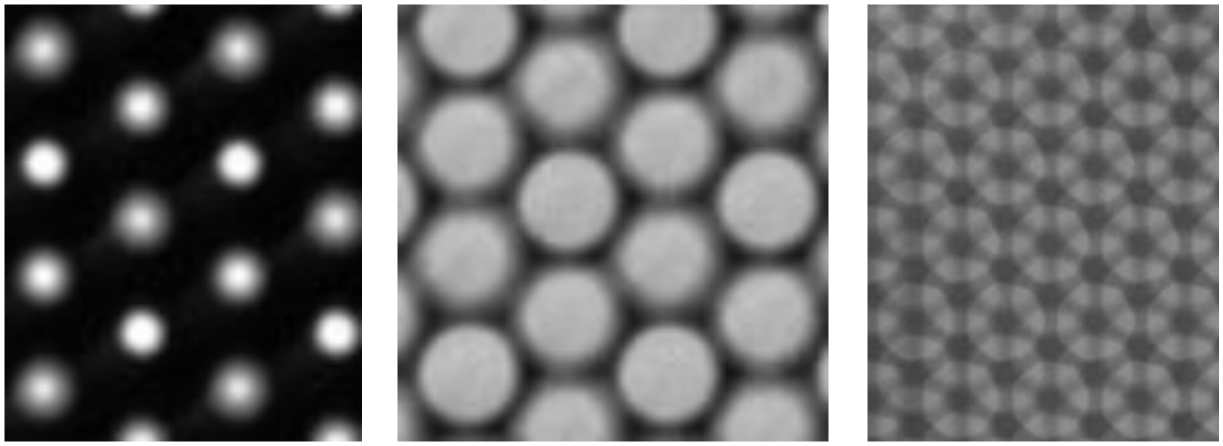

(a) (b) (c)

Figure 2 - Image of diffuse light through a focused plenoptic camera with (a) too small of an aperture, (b) properly adjusted aperture, and (c) too large of an aperture.

When the projected image of a point source in object space reaches the micro-lens array, the projection has not yet fully converged and is spread across multiple micro-lenses. Each exposed micro-lens will collect a portion of the light and focus it on the sensor to form ‘micro-images’, as shown in the side view schematic in Figure 1. To ensure that the images created by the micro-lenses do not overlap, it is important to adjust the effective *f/#* of the optical system to match the size of the micro-lenses. In most cases, this is simply done through imaging a diffuse white light source and adjusting the lens aperture until the formed microimages are just touching, as in Figure 2b. In some cases the maximum lens aperture can be too small to allow for this, such that the microimages have a greater separation, as shown in Figure 2a. This will reduce the overall resolution of the system, but not prevent it from working entirely. The opposite case shown in Figure 2c is an example where the aperture would be too large, since the regions illuminated by each lens overlap. This makes processing impossible since locations on the sensor cannot be attributed to a single micro-lens.

Up to a specific plane in object space represented by the faded blue arrow in Figure 1, every point is visible in only 2 micro-lenses. Beyond this point, some points will only be visible in a single micro-lens. The image-space projection of this location in the optical system is termed the total covering plane (TCP) [27], which is labeled on Figure 1. The distance between the main lens and the TCP is defined as B_L. For a multi-focus plenoptic camera with 3 micro-lens types arranged in a hex array as shown in Figure 3, the TCP will always be positioned at a virtual depth of 2 [27].

The location of the corresponding plane in object space depends on the focal length and relative position of the main lens.

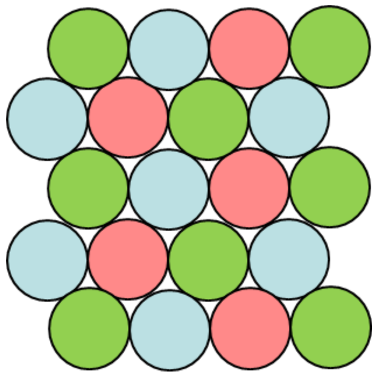


Figure 3 – A schematic of the multi-focus lens array orientation in top view showing the hex packing orientation. The three different lens types are encoded by color.

Theoretically, a plenoptic camera system in this configuration can image any point in its field of view between the object-space projection of its TCP and its main lens's focal point. However, nearer to the camera the object will become too out of focus in the micro-lens images to be resolved. This is somewhat mitigated here by using a multi-focus plenoptic camera, which incorporates micro-lenses of different focal lengths to target different ideal depth ranges. This effect is apparent in the sample raw image shown in Figure 4, which was collected using the *f/#*-2.4 camera with an 85 mm main lens and an approximate focus distance of 50 cm. The imaged particle here is more focused in one lens type than in the other two lens types, as annotated. This is a subsection of the full image, and spans only 7 micro-lenses where each micro-lens image is ~ 25 pixels wide.

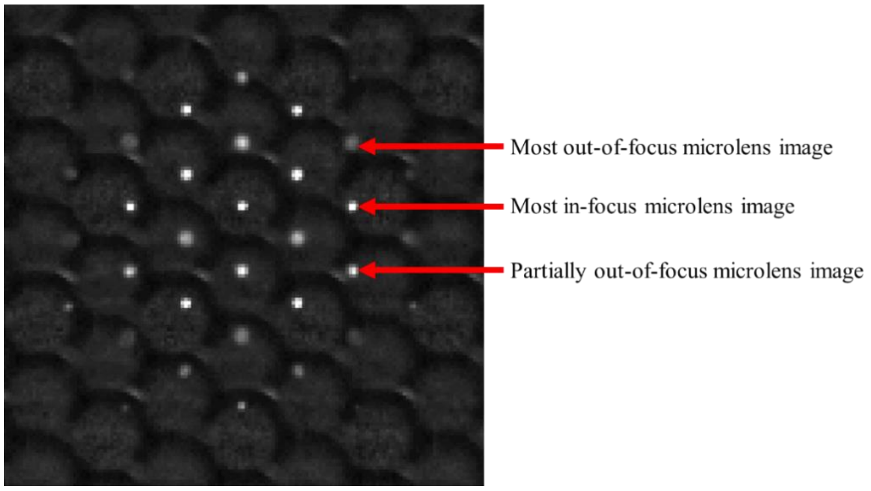

Most out-of-focus microlens image
Most in-focus microlens image
Partially out-of-focus microlens image

Figure 4 – Cropped multi-focus plenoptic image of a single particle. 4x gain has been applied for clarity.

The back of the micro-lens array in this case is on a flat plane [35], [36], and as such the different width of lens necessary to produce the three focal lengths will result in each set of lenses having its own distance to the sensor *B* as measured to the central plane of the lens, as shown in Figure 5. It is thus necessary to use three separate values for the distance from the sensor to the array (B_1, B_2, B_3) when processing multi-focus plenoptic images. Objects at corresponding virtual depths within a specific range, generally between 2 and 7.5 but depending on the micro-lenses' focal lengths and positions B_i, can be imaged with limited loss of resolution [27].

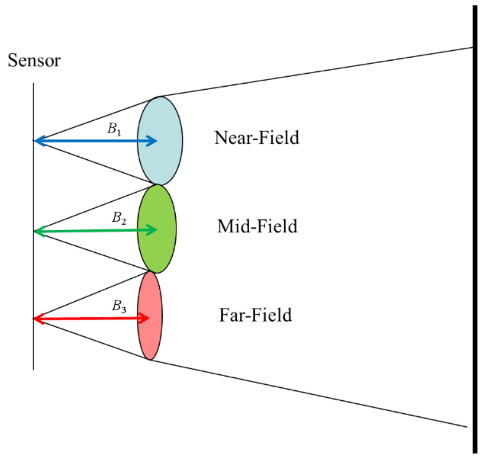


Figure 5 – A schematic of the multi-focus lens array orientation in side view showing different lens-sensor distances. Lens-sensor distance extended for clarity. Three lens types are encoded by color.

2.1 *Design of a focused plenoptic imaging system*

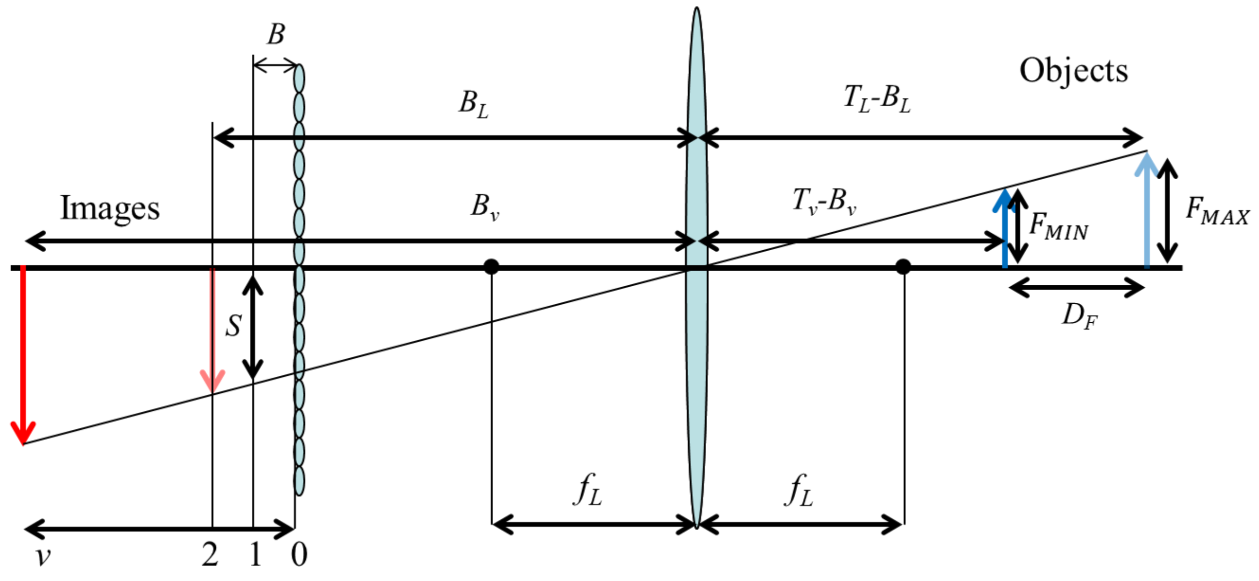


Figure 6 - Definition of parameters for determining imaging volume

When designing a plenoptic imaging system for use in a fluid imaging experiment, being able to achieve a desired imaging volume is usually the main concern. This section will assume the use of a given plenoptic camera with a fixed micro-lens array, and then discuss how to select and position a main lens for the camera given a desired imaging volume. How the imaging volume projects into image-space can be simply modeled using the thin lens [37] (1) and pinhole camera (2) equations. Application of the thin lens equation will be incorrect for many lenses, as most modern lenses need thick-lens modeling to be properly modeled. However, the thick lens model will only add a constant to the focus distance [35], so these equations will hold true after some slight repositioning to account for a thick main lens. The definition of virtual depth (3) is also important for the design. In the following equations, D_o represents an object-space point's distance from the main lens, D_i is the distance from the main lens to the image of the point, y_o is the distance of a point from the optical axis, y_i is the distance from the optical axis to the image of the point, f is the focal length of the main lens, v is the virtual depth, and B_L is the distance from the main lens to the TCP. These can be related by the thin-lens equation as:

$$\frac{1}{D_o} + \frac{1}{D_i} = \frac{1}{f} \tag{1}$$

$$y_i D_o = y_o D_i \tag{2}$$

Note that D_i can be defined in terms of the parameters of the plenoptic camera, as in (3) where *v* is the virtual depth, *B_L* is the distance from the main lens to the TCP, and *B* is the largest of the micro-lens-sensor distances *B_i*. It is important to use the largest *B_i* to avoid missing part of the imaging volume near *v* = 2.

$$D_i = (v - 2)B + B_L \tag{3}$$

This approach aims to take the TCP at *v* = 2 and project it into image space using (1). Then, the image sensor is projected to that plane using (2). This will give an estimate of the maximum field of view of the plenoptic camera. This is achieved through the substitutions given in Table 1. Here, T_L is the focus distance of the lens, F_{MAX} represents the desired maximum field of view in the far-field, and *S* is the size of the sensor. These are also defined on Figure 6. Note that performing these substitutions assumes that the sensor is located at *v* = 2, whereas it is actually located at *v* = 1. This is a simplifying assumption which is based on $B_L \gg B$, which should generally be true. It ensures that the actual FOV is slightly under-predicted rather than over-predicted, which is useful to ensure that the desired imaging volume is covered. Not applying this assumption produces a much less concise system of equations, the code for which is present in Appendix A.

Table 1 - Camera equation substitutions for *v* = 2

Variable	D_o	D_i	y_o	y_i
Substitution	$T_L - B_L$	B_L	$F_{MAX}/2$	$S/2$

Performing these substitutions into (1) and (2) and solving for T_L and B_L yields:

$$T_L = \frac{f}{F_{MAX} \cdot S}(F_{MAX} + S)^2 \tag{4}$$

$$B_L = \frac{f}{F_{MAX}}(F_{MAX} + S) \tag{5}$$

These parameters will allow a main lens with a given *f* to be positioned using T_L and B_L such that the field of view corresponding to *v* = 2 will be a desired F_{MAX}.

The next parameter that becomes important is the desired depth of field D_F within a specified maximum virtual depth *v* that results from the selected *f*. Finding D_F requires the definitions of new focus distance and image position T_v and B_v, respectively. B_v is found first from the definition of virtual depth:

$$B_v = B_L + (v - 2)B \tag{6}$$

The thin lens equation (1) can then be applied to get T_v (7), and then D_F is calculated from the difference between the object distances (8).

$$T_v = \frac{B_v^2}{B_v - f} \tag{7}$$

$$D_F = T_L - T_v - (v - 2)B \tag{8}$$

The field of view at position $T_v - B_v$, F_{MIN}, can then be calculated from the pinhole camera equation:

$$F_{MIN} = \frac{S(T_v - B_v)}{(B_L - B)} \tag{9}$$

Equations (4) - (9) allow the complete definition of the imaging volume of a focused plenoptic system with a known sensor size *S*, sensor-MLA distance *B*, and main lens of focal length *f*. It is also possible to determine *f* given a desired D_F, which can be useful for getting a general idea of which focal length of lens would be appropriate for a desired system. After applying (4) - (6), T_v is calculated from the desired D_F and *f* is calculated from the thin lens equation:

$$T_v = T_L - D_F + (v - 2)B \tag{10}$$

$$f = (B_v^{-1} + (T_v - B_v)^{-1})^{-1} \tag{11}$$

The expression for f given here is self-referencing; this is done for conciseness. The code to generate this full formula is present in Appendix A.

The final consideration in designing the imaging system is the required aperture size. For a micro-lens array with a manufacturer-specified working *f*/#, *N*, the required aperture diameter D_A is given by:

$$D_A = \frac{B_L - 2B}{N} \tag{12}$$

The suggested method for determining a suitable lens for a given imaging system is to apply equations (4) - (6) with (10) - (11) to get a predicted focal length, which is then used to select an appropriate main lens. Equations (4) - (9) and (12) are then used to fully define the imaging volume and required aperture diameter. Within this process it may become apparent that the desired imaging volume cannot be achieved due to the system requiring an unavailable focal length or diameter of lens. In this case, compromises on the selected in-plane or out-of-plane imaging volume will need to be made.

Given a specific main lens and micro-lens array, it is possible to determine the overall system field of view and depth of field as a function of B_L. Example plots for B=0.373 and 1.452 mm and f = 35, 50 and 85mm are given in Figure 7 and Figure 8, respective to B. These values are selected based on available cameras and lenses. As an example, if a similar field of view and depth of field are desired, the intersection between D and F_{MAX} or F_{MIN} can be considered. This intersection point is different for each main lens focal length and micro-lens distance B. To show this clearly, the plots have been zoomed to highlight the intersection points in Figure 9. Other ratios between the field of view and depth of field will correspond to different points along the *x*-axis in these plots. These plots are useful tools in the preliminary setup of an experiment to determine and define the important imaging field characteristics that set the overall range and resolution of the experiment.

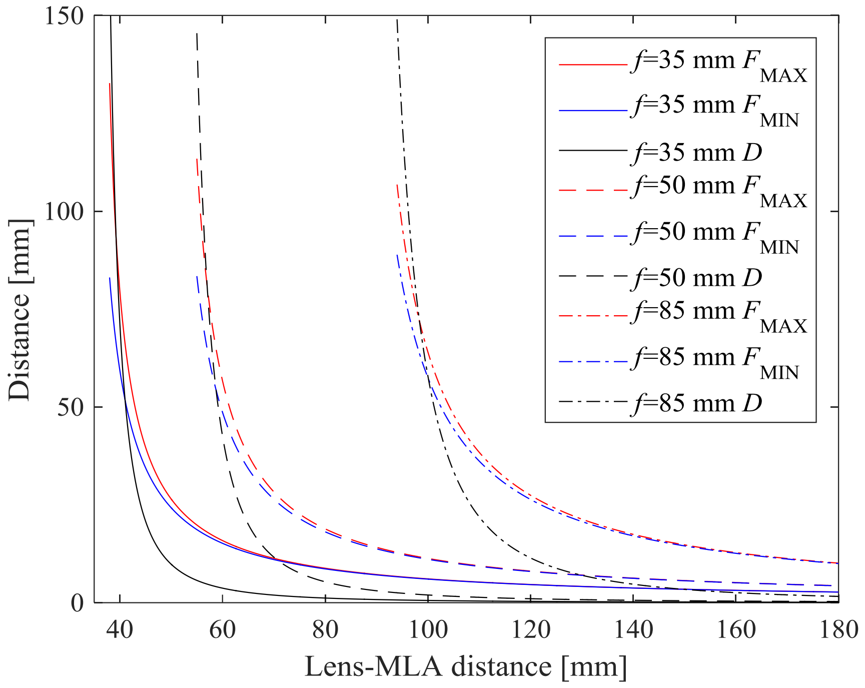


Figure 7 - Field of view and depth of field plots for B=0.313 mm

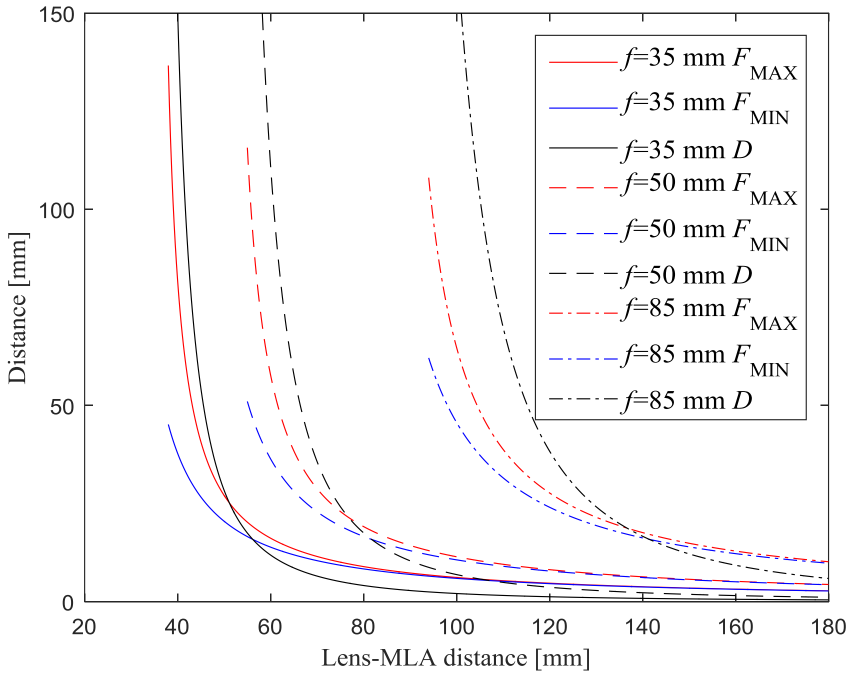
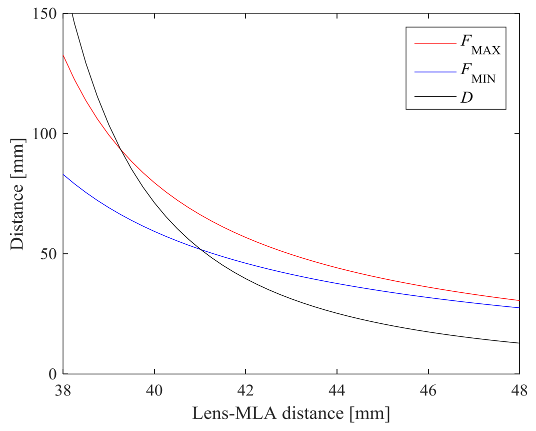
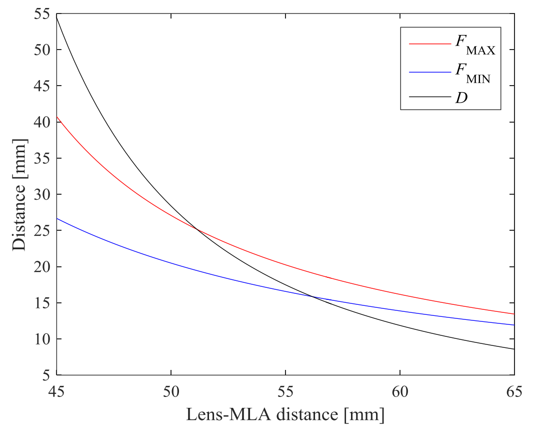


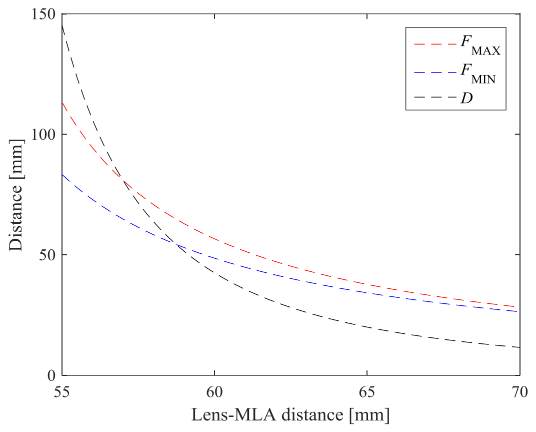
Figure 8 - Field of view and depth of field plots for B=1.452 mm



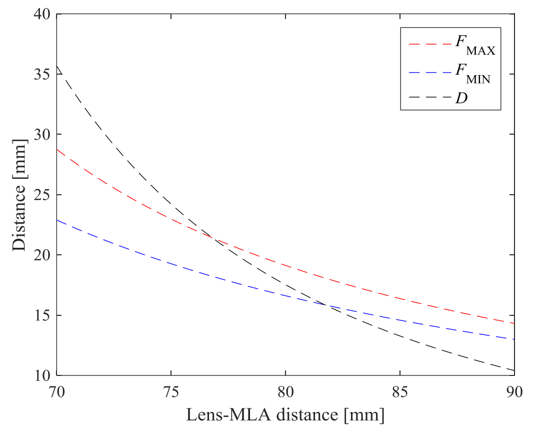
(a)



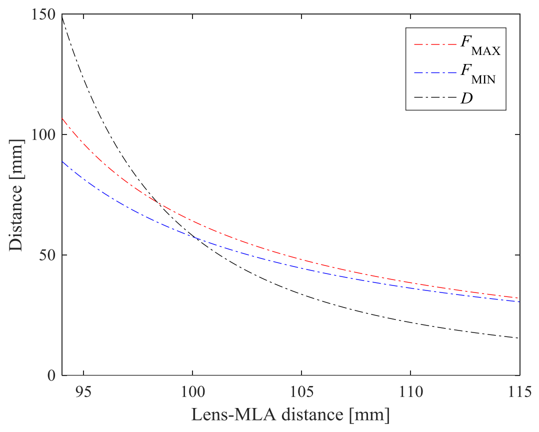
(d)



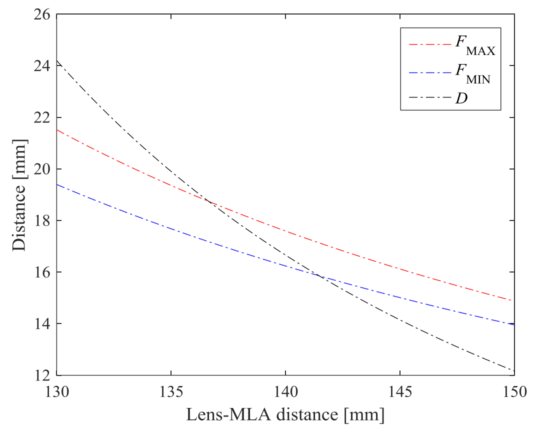
(b)



(e)



(c)



(f)

Figure 9 - Field of view and depth of field plots for (a-c) B=0.343 mm and (d-f) B=1.452 mm. (a) and (d) are plots for f =35 mm, (b) and (e) are plots for f =50 mm and (c) and (f) are plots for f =85mm. Note that the axes of these plots have been manipulated to focus on the intersection between the D and the two F plots in each case.

CHAPTER 3 PARTICLE LOCATION METHODS

Determining the three-dimensional location of particles in physical space from the collected imaging data is the first step in tracking particles. Two different methods for doing so have been developed as part of this work and both rely on first being able to find the 2D locations of the particles in images. This chapter will discuss the 2D particle location method used and the two methods developed for extracting the virtual depth of a particle. The plenoptic camera calibration algorithm, implemented for the first time in Matlab, that allows the particle data to be mapped from pixel and virtual coordinates in image space into physical coordinates in object space that uses these particle location methods will also be discussed.

3.1 Two-dimensional particle location method

Both of the 3D particle location approaches discussed here rely on being able to locate a particle in the projected 2D image to sub-pixel accuracy. Particle location methods have already been extensively reviewed by [9]. To summarize that review, particle location methods generally aim to find the 2D centroid of individual particles. Early methods used a center-of-area method [38]–[40] that calculates the centroid of the pixels containing the particle. Fitting methods which include Gaussian and polynomial-based methods [41]–[43] have also been attempted. The selected method [44] is an intensity-weighted center-of-area method which has been used extensively and is freely available as a Matlab open-source code under a GNU GPL [45]. This method was selected for its usefulness with tracking small particles with roughly Gaussian intensity distributions, and its ease of availability. The main steps in this particle location method are preprocessing, preliminary location, and then sub-pixel location, which will each be discussed in turn.

3.1.1 Image Preprocessing

The preprocessing approach generates a boxcar kernel based on a specified object size and performs 2D convolution on the raw image. This algorithm is representable as given in [44] by:

$$A_w(x, y) = \frac{1}{(2w+1)^2} \sum_{i,j=-w}^{w} A(x+i, y+j) \tag{13}$$

Here, A represents the image, A_w represents the convolved image, i and j are stepping variables for the summation, x and y represent pixel coordinates, and w is a distance in pixels which is set to the radius of the largest expected particle. The approach also suggests convolution with a Gaussian surface to reduce noise, representable by:

$$A_{\lambda_n}(x,y) = \frac{\sum_{i,j=-w}^{w} A(x+i, y+j) \exp\left(-\frac{i^2+j^2}{4\lambda_n^2}\right)}{\left[\sum_{i=-w}^{w} \exp\left(-\frac{i^2}{4\lambda_n^2}\right)\right]^2} \tag{14}$$

Applying this filter requires selecting the noise filtering radius λ_n such that λ_n is much smaller than the radius of a particle. The noise reduction of these operations is apparent in the sample images given in Figure 10. The structure of the ~25 pixel diameter circles of the micro-lens array visible in the background noise of the raw image in Figure 10a is completely eliminated by the boxcar convolution, the result of which is shown in Figure 10b. Adding the Gaussian filter ($\lambda_n = 1$ was used in this case) performs a blurring operation which normalizes the profiles of the particles as shown in Figure 10c, but has the risk of merging nearby particles or eliminating small particles, as shown by the red arrows in each image. When processing the images herein, only the boxcar filter has been used to avoid eliminating small particles.

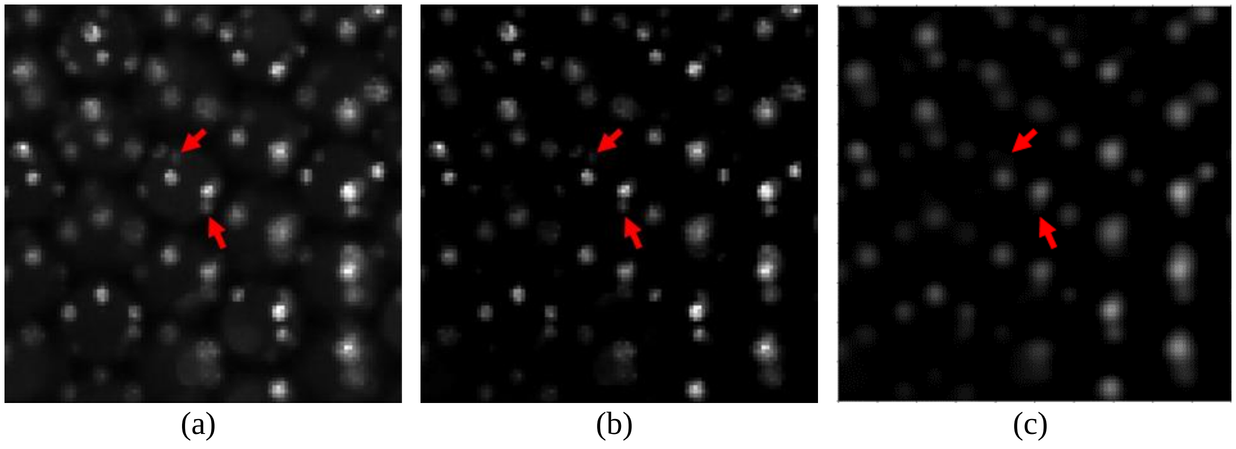


Figure 10 - Raw (a), boxcar filtered (b), and boxcar+Gauss-filtered (c) sample plenoptic images. Images have been cropped to 100x100 pixels for clarity. Red arrows identify sample particles which exist in (a), remain in (b), but are eliminated in (c).

3.1.2 Preliminary particle location

Preliminary particle location is conducted by applying a manual threshold to the preprocessed image to eliminate any remaining background noise, and then looking for local maxima within the image. This is accomplished by checking each pixel's 8-connected neighbors for higher intensity values. If multiple local maxima are found that are within the expected radius of a particle *w*, only the brightest is kept. Any local maxima which are within *w* of the edges of the image are ignored as it is expected that these particles would be truncated by the edges of the image. This approach is designed to select a pixel that is near to the geometric centre of the particle [44].

3.1.3 Sub-pixel particle location

The pixel-accurate centroids from the preliminary location algorithm are passed to the sub-pixel location algorithm, which takes a circular window around each location. The circular window is defined by a radius of *w'* around each centroid, which is manually set for the entire data set to ensure that the entire particle is enclosed. It is important that this radius be slightly larger than the particle to prevent the centroid estimation from being skewed by cutting off the edges of the particle. Intensity-weighted averaging is then performed to get a sub-pixel measurement of the particle centroid's location using:

$$x_c = \frac{\sum_{i=1}^{N} I(x_i, y_i) \cdot x_i}{\sum_{i=1}^{N} I(x_i, y_i)} \tag{15}$$

$$y_c = \frac{\sum_{i=1}^{N} I(x_i, y_i) \cdot y_i}{\sum_{i=1}^{N} I(x_i, y_i)} \tag{16}$$

Applying this operation for the sample images shown with *w'* = 3 yields the particle centres shown in Figure 11. This approach assumes that the intensity distribution of a particle will be roughly Gaussian, with the centre of the particle brightest and the edges dim. When these conditions are met, the error in estimation should remain under 0.15 pixels as long as the *w'* is larger than but no more than twice as large as the target particle [44]. If the distribution deviates from Gaussian, higher uncertainty may be expected. Particle overlap can cause deviation to occur. Assuming the adjacent particles each form a Gaussian intensity distribution, it is likely that the overlap between the Gaussians will result in the centres of the particles being shifted closer together than they should be. These issues will be returned to when discussing experimental results.

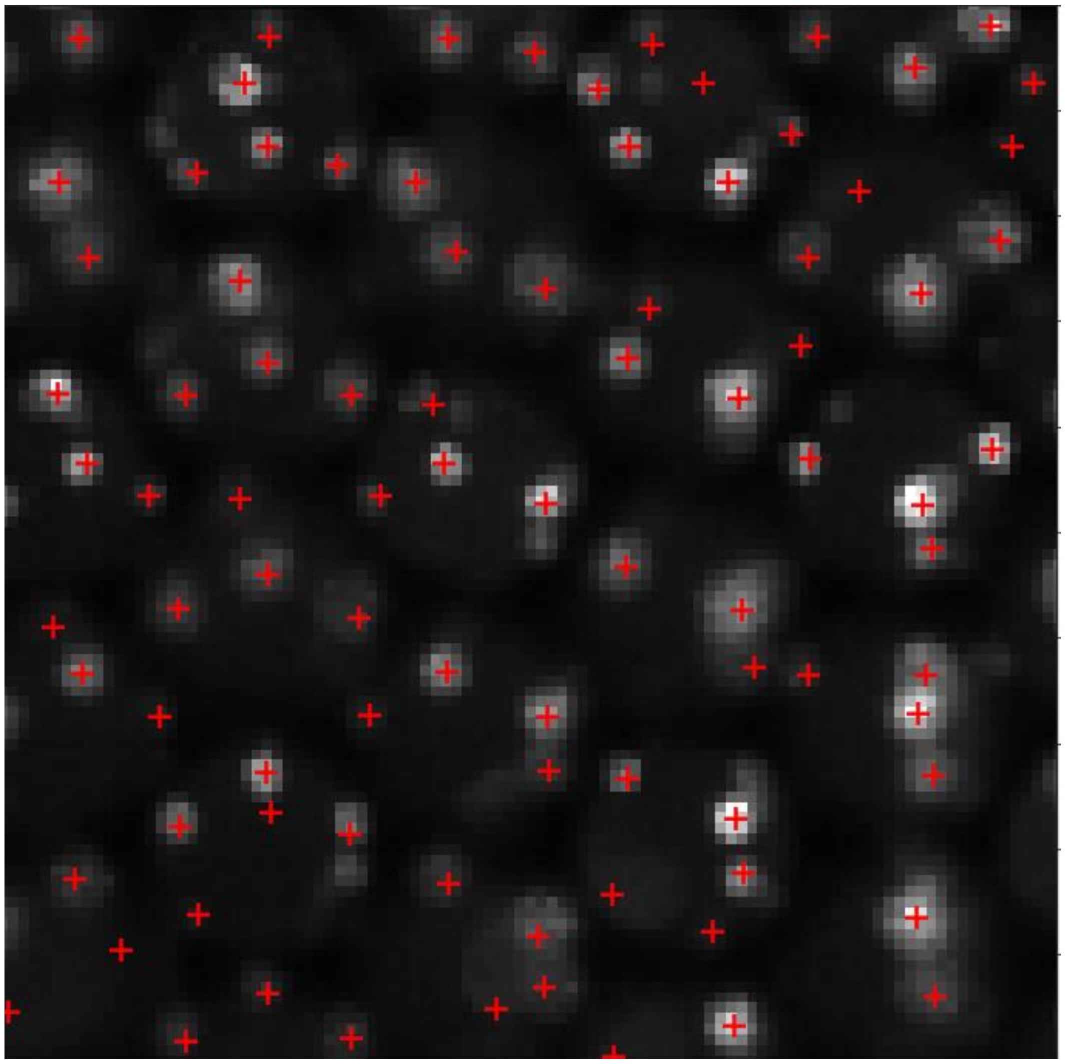
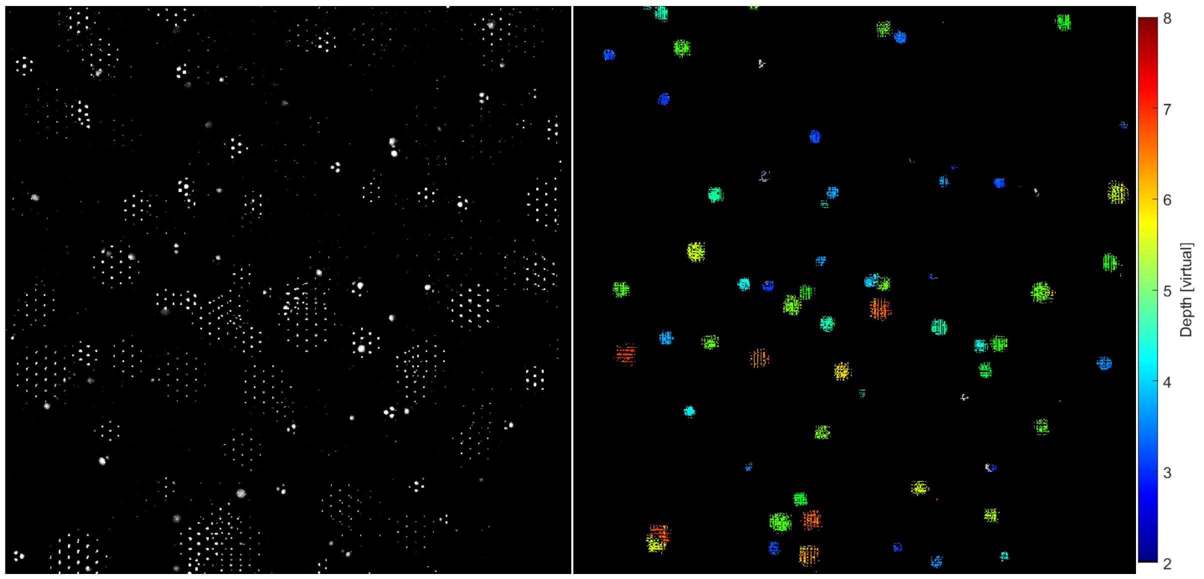


Figure 11 - Raw image with located particles shown as red crosses

3.2 *Refocusing-based three dimensional location method*

The initial method for determining the 3D location of particles in the plenoptic images involves using a commercial software package (RxLive 2.01, Raytrix GmbH). This software converts a plenoptic image into a conventional image by using cross-correlation on small patches of pixels along the epipolar lines of the micro-lens images to determine virtual depth. It uses interpolation and extrapolation methods to estimate the depth of nearby pixels, and then refocuses the image at these calculated depths [27]. Epipolar lines, in this case, are defined as the lines parallel to those passing through the centres of the micro-lenses of the same type. Any location which does not have an assigned depth is blacked out, resulting in focused images for which the background is noiseless. The software is able to generate one separate depth map for each micro-lens type as well as one computationally refocused image. A sample raw image, depth map, and refocused image are shown in Figure 12 and Figure 13. Note that particles for which fewer micro-images are present have been labeled with lower virtual depths, as should be expected from the discussion in Chapter 2. A custom Matlab code, available in Appendix B, was developed to extract the 3D particle locations from these refocused images and depth maps.



(a) (b)

Figure 12 - A subsection of a raw image of a particle field (a) ‘raw image’, cropped to show micro-lens images, and associated depth estimations (b) ‘depth image’.

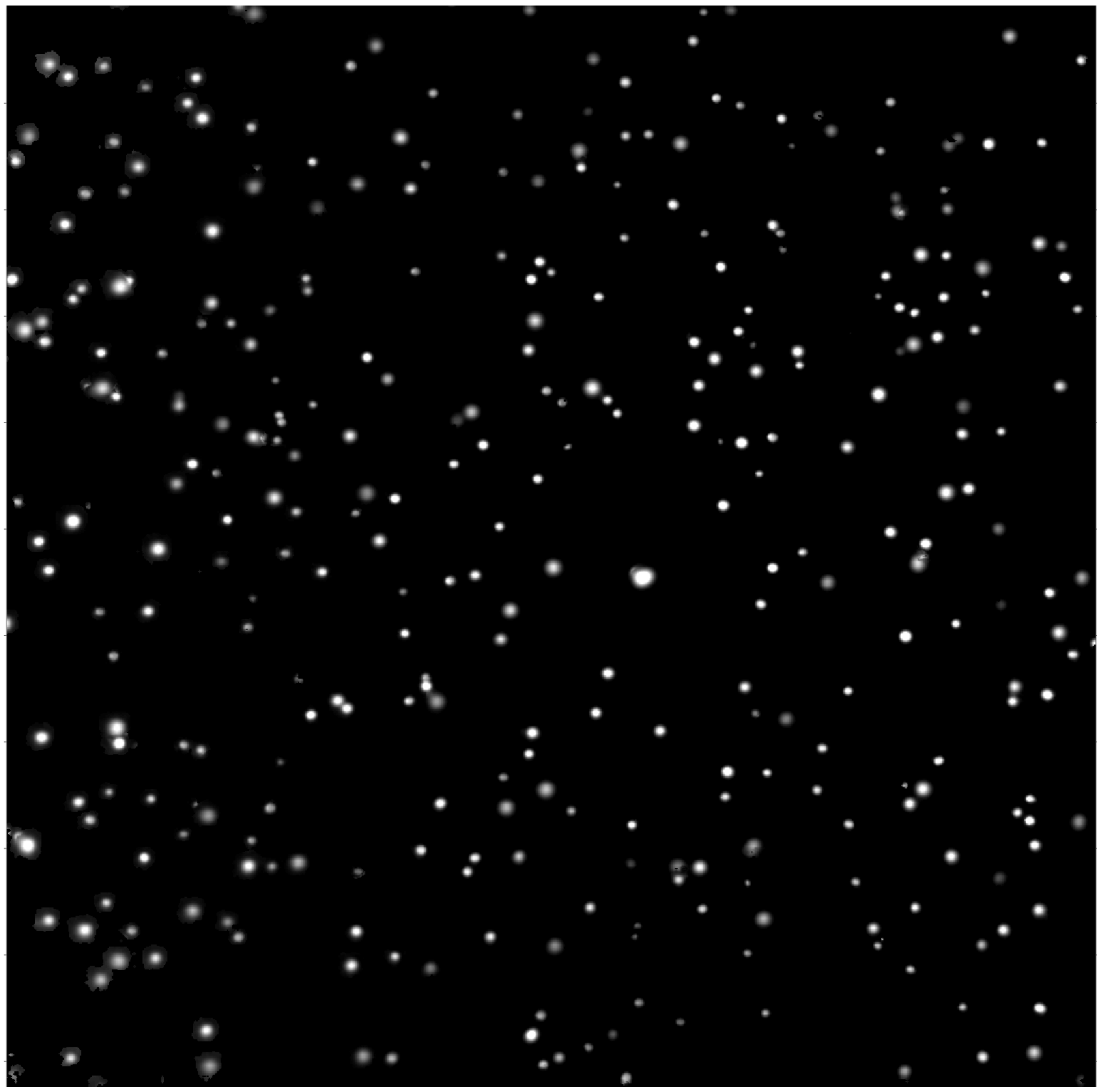


Figure 13 - Resultant total focus image

To obtain 2D particle location data, the feature-detection algorithm from section 3.1 is applied to the refocused images. This results in obtaining a list of 2D particle locations, which have been overlaid on the image in Figure 14.

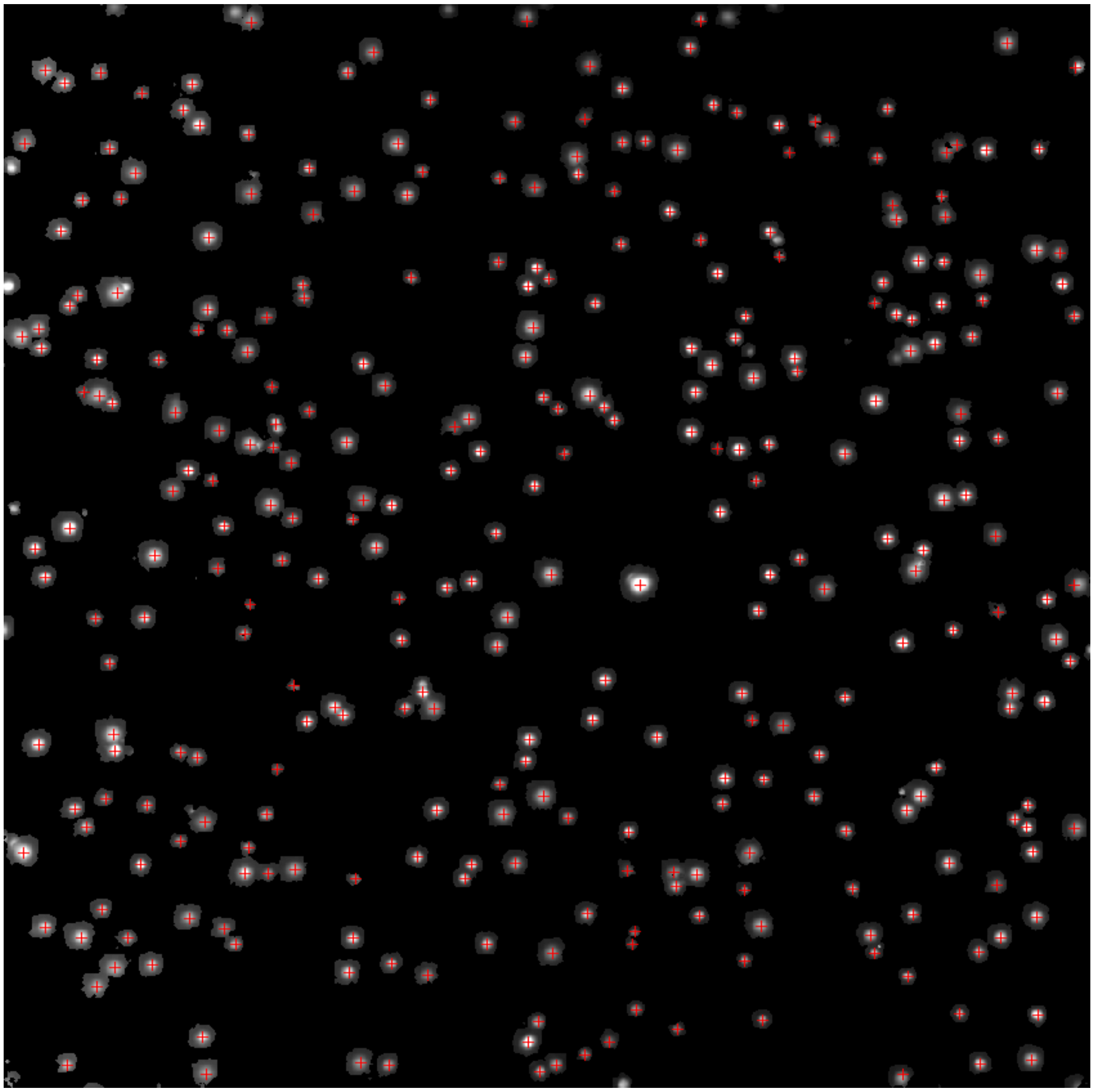


Figure 14 - 2D particle locations overlaid as red crosses on refocused image

Once the 2D locations of the particles have been found, the locations are projected onto the depth maps. A radius is taken around each location as shown in Figure 15, and all depth measurements within the radius are gathered. If a circle formed in this way does not overlap any of the other circles, a median of the gathered depth measurements is taken as the depth of the particle after repeated elimination of outlier measurements greater than 2 standard deviations from the median. This is performed for each of the 3 depth maps, and a weighted average virtual depth location is taken based on the number of non-outlier depth-locations found within the circles of the particle in each map to get the final virtual depth. In cases were circles overlap, the medians of the non-overlapping regions are considered first and any overlapping regions are passed to the particles with the highest virtual depth, which should lie in front. This operation results in the particles being defined in image-space in terms of pixels and virtual depth.

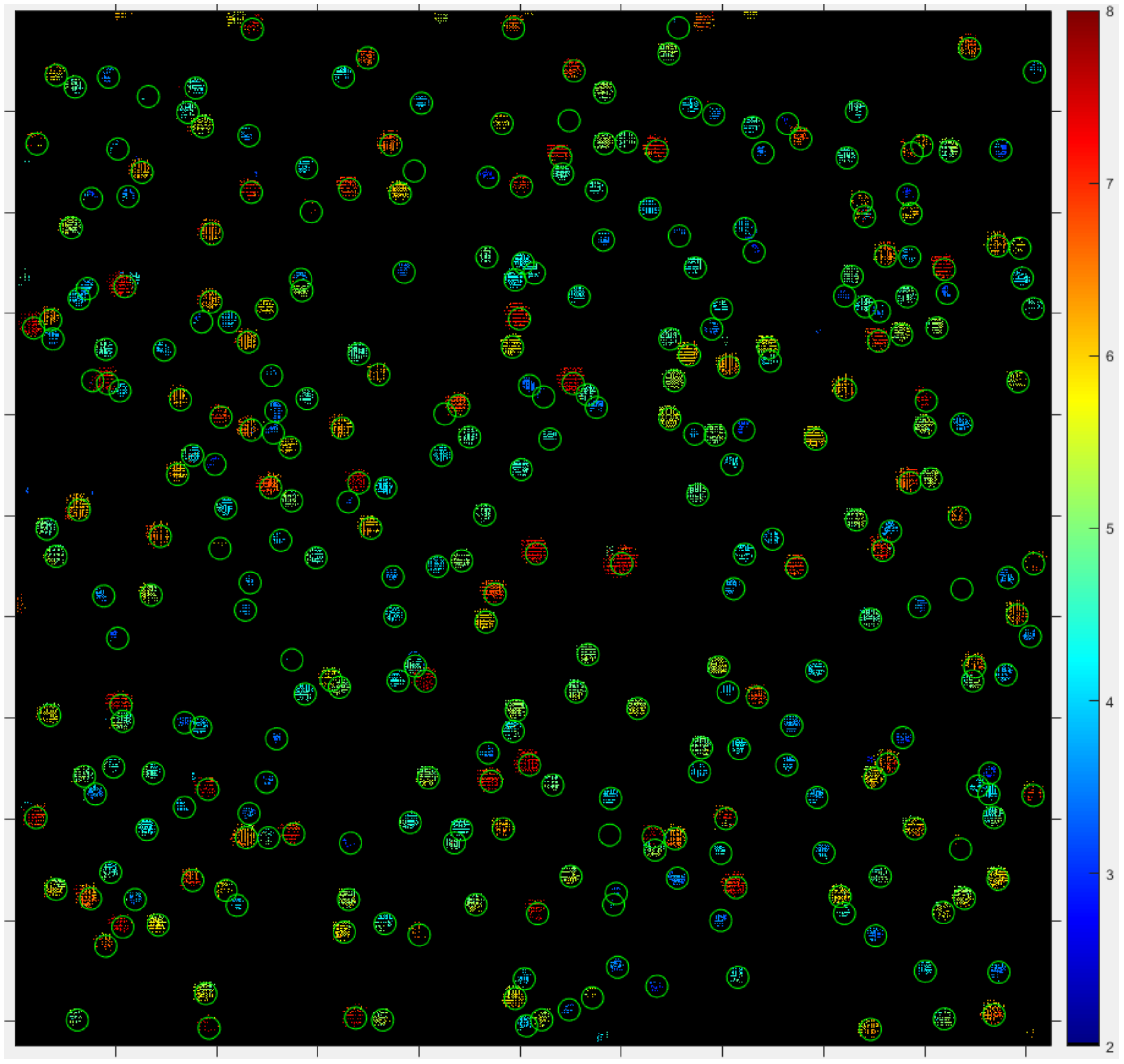


Figure 15 - Virtual depth map with 2D particle positions overlaid as green circles

While this approach has been found to be relatively effective for low-density particle fields, it begins to have issues as the density of the field increases. The depth map production algorithm is primarily designed for determining the depth of regularly-contoured, uniquely-featured objects [27] rather than discrete particles distributed randomly in the imaging volume. Its reliance on cross-correlation means that for higher-density particle fields, it is inevitable that the algorithm will try to calculate depths using two different, adjacent particles. The frequency of this failure will increase exponentially as the particle density increases. This will then lead to the production of ‘ghost’ particles in the refocused images, similar to how tomographic methods can produce ghost particles [7], [8]. Even where ghost particles are not generated, the reconstruction algorithm is unable to fully reconstruct objects with similar in-plane locations but different out-of-plane locations in the refocused images, meaning that it is impossible to discretize overlapping particles despite the multiple angles of view provided by the micro-lenses. All of these issues compounded result in this method being best used for low-density particle fields, where the chances of particles being nearby or overlapped are minimized. Use of the commercial algorithm makes customizing the approach for particle fields impossible, so an entirely different approach was developed.

3.3 Raw image 3D particle location

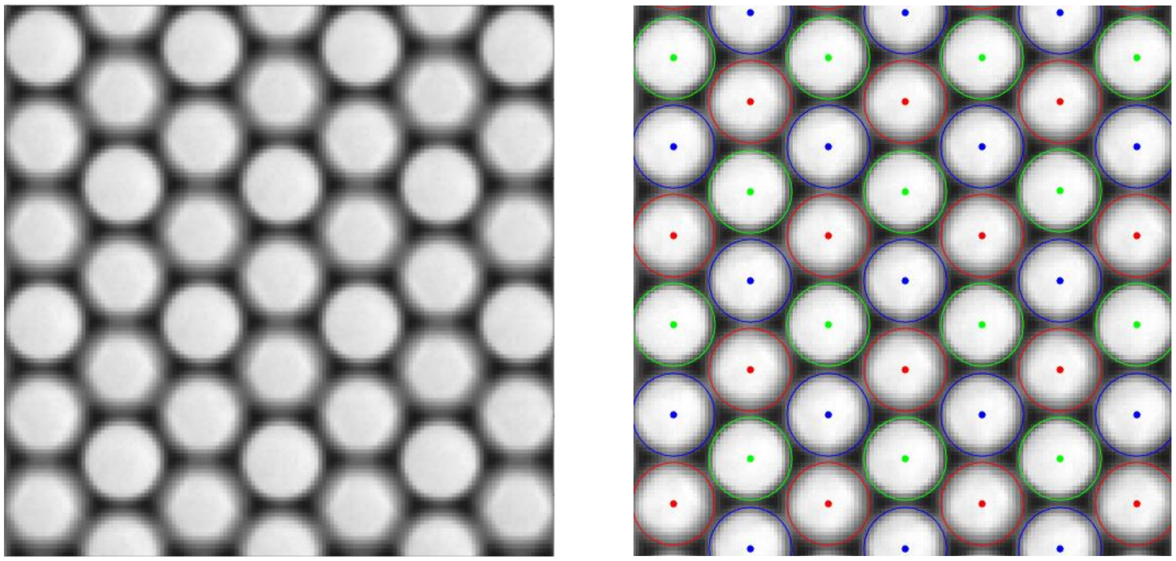
To overcome the limitations of the commercial (RxLive 2.01, Raytrix GmbH) algorithms, a new in-house code was conceived and developed to process the raw plenoptic images of particle fields to estimate 3D particle location. The relevant Matlab scripts are available in Appendix C. This approach draws much inspiration from the original algorithm [27] for its preprocessing, but diverges significantly in its feature-determination method as this is designed specifically for extracting 3D particle positions. The method has seven major steps, which will each be discussed in their own subsection. It proceeds as follows:

1. Determine the centre of each micro-lens and assign micro-lens types, using a ‘white image’ [27].
2. Plenoptic image preprocessing [27]
3. Determine the locations of individual particle-images in the raw plenoptic image
4. Determine which micro-lens each particle-image is in
5. Determine which same-type, adjacent micro-lenses contain particles-images along epipolar lines, and group all particle-images found to be epipolarly connected
6. Split the groups into subsets of particle-images that meet the Epipolar Triangular Connectivity (ETC) criterion

7. For each ETC group, determine the 3D location of the particle in pixel and virtual coordinates

3.3.1 Micro-lens location

To locate each micro-lens, a white image is obtained with the plenoptic imaging system by placing a diffuse light source nearer to the main lens than its focal point. With an exposure time set such that the maximum intensity count is just below the maximum of the sensor, many images can be acquired, and an average image can be generated to reduce any random noise present. An example image in which this has been done is shown Figure 16a. Here, the images produced by each micro-lens identify the area on the sensor that each lens will project images into. This image is then processed with the particle-location algorithm to approximate the centres of each micro-lens image, which should correspond to the centres of each micro-lens. Two vectors are then defined. A vector from the central micro-lens vertically to the next micro-lens in the same column and a vector from the central micro-lens to the nearest lens not in the same column. Each lens is then located as a combination of integer multiples of these two vectors. The vectors and the position of the central micro-lens are then manipulated by a least-squares fitting algorithm to minimize the differences between the found locations and the ones defined by the vectors. This step averages out any uncertainty created by inaccuracies in the particle location algorithm. The final grid produced by the vectors is then kept for use in the particle-image location algorithm. This grid, with micro-lens types coded by color, is overlaid on the original image in Figure 16b. The lens types are coded from the knowledge of the repetitive nature of the hexagonal grid shown in Figure 3. Exact knowledge of the focal lengths of individual lenses is not necessary for the processing approach. This must be performed for each unique lens orientation, and it is important to do so for each experiment conducted. The efficacy of this method will be discussed alongside the results of the experimental data which this method has been used for in Chapters 5-7.



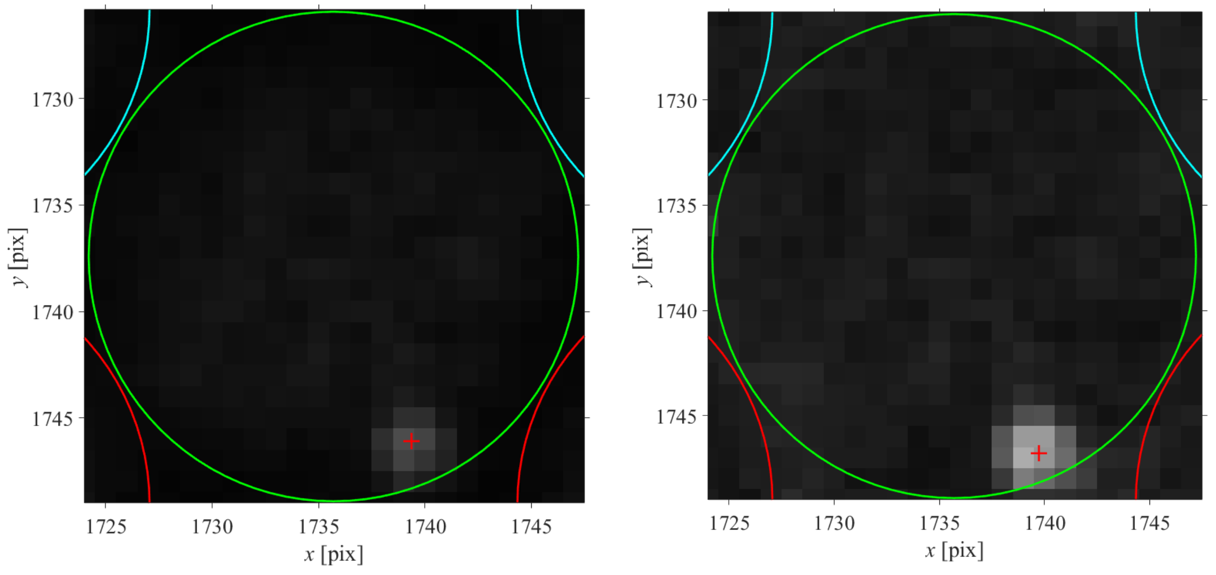
(a) (b)

Figure 16 - Micro-lens location method: averaged white image (a) is used to locate the centres of the micro-lenses (b). The different coloured markers in (b) represent the three different lens types.

3.3.2 Image preprocessing

The image preprocessing approach used assumes that the white image created from the averaging approach defines a uniform white light field. Ideally, each micro-lens image would be uniformly illuminated by this uniform light field. Typically, a gradient with a brighter center and darker edges is observed. This is an effect of the distortions produced by the individual micro-lenses and is more pronounced for lenses with a greater curvature and shorter focal length. In Figure 16b, the red-marked lenses have the shortest focal length and the green lenses have the longest. When observing a particle field, particles near to the edge of the lens can have their apparent centres shifted towards the centres of the micro-lens from this effect. To correct for this, the particle images are divided by the white images, which results in the particle location algorithm comparing the particles' intensities as a ratio of the white image as opposed to comparing the raw intensities. This helps correct for the centre-shifting of the intensity gradient. The centre-shifting effect can be observed in Figure 17. Here, a particle at the edge of the micro-lens in the raw image (a) has the intensities of the pixels on the outer edge of the lens amplified, while the pixels closer to the centre are slightly reduced. This has the effect of shifting the sub-pixel centre of the particle away from the centre of the lens, closer to where the actual position should be. The shift in this case is from

(1739.36, 1746.11) to (1739.73, 1746.80). This helps to highlight a potential source of error, as well: if the handling of particles near the edges of the micro-lenses is inadequate, it is likely that the 2D centroid-location approach will produce particle centroids closer to the centre of the micro-lens than the actual centre of the particle. A problem that cannot be corrected using this approach is the case where only part of a particle is visible in a lens, that is, the particle is cut off by the border of the lens. In this case, a centroid may be found that does not correspond to the particle's actual centre. The found centroid will be closer to the centre of the microlens than the true particle centroid. This is a major source of uncertainty in this approach. These particles need to be ignored, or they may affect results. The criteria for ignoring them is discussed in section 3.3.6.



(a) (b)

Figure 17 - Raw (a) and preprocessed (b) images of a single particle on the edge of a single micro-lens. Micro-lenses shown by colored circles; detected particle centre shown by red cross.

3.3.3 Particle-image location

Particle-image location is performed on the preprocessed images using the algorithm discussed in section 3.1. A result for a partial sample image, containing a single particle, is shown in Figure 18. This approach, while computationally efficient, can have its accuracy affected when handling cases where the radii of two particles overlap. However, because multiple micro-images of a single particle are obtained at slightly different angles of view, this issue will frequently only affect a few of the micro-lenses and 3D particle location can still proceed. This means that this technique is better suited to handling dense particle fields than the initial technique discussed in Section 3.2.

Having determined the particle-image locations, the nearest micro-lens centre is determined from the list of micro-lens locations and assigned to the particle-image. A single micro-lens can be assigned to any number of particles, as each micro-lens may contain multiple particles.

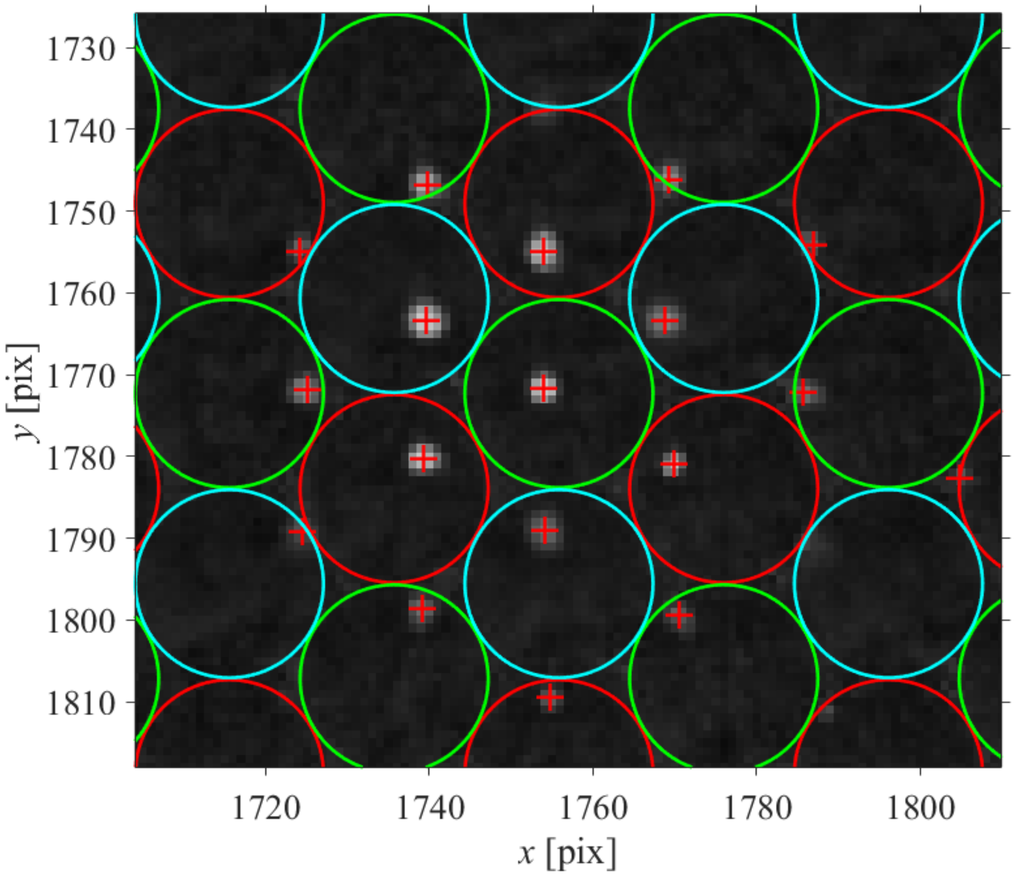


Figure 18 - Particle location result for a single point on a calibration target. Lens borders have been colored for clarity. Located particle centroids shown by red markers.

3.3.4 *Epipolar grouping*

Starting at any particle, the algorithm looks along a line defined by the vector between the micro-lens the particle is in and one of the nearest 6 micro-lenses of the same type. The epipolar plane generated by a given pair of lenses and a given particle imaged by both lenses intersects the camera sensor along this line. For simplicity and to be explicit, the line will be referred to as an "epipolar line" along the sensor. If a particle-image within the micro-lens is within a specified distance L_e of the epipolar line (typically <0.5 pixels), the particle-images are grouped together. The distance

L_e is easily calculable through application of the classical minimum point-to-line distance equation:

$$L_e = \frac{\begin{vmatrix} \vec{x}_i - \vec{x}_j \\ \vec{E}_i - \vec{E}_j \end{vmatrix}}{\left\| \vec{E}_i - \vec{E}_j \right\|} \tag{17}$$

Here, $\vec{x}_i$ refers to the location of particle i, $\vec{x}_j$ refers to the location of particle j in an adjacent lens, $\vec{E}_i$ is the position of the micro-lens that contains particle i, and $\vec{E}_j$ is the position of the micro-lens that contains particle j. The operators in the numerator and denominator are the determinant and 2D vector norm, respectively. If multiple particle-images in the same micro-lens are close to the epipolar line, only the particle with the smallest L_e is kept. This is repeated for each of the 6 micro-lenses and the whole process is repeated for each found particle-image until no further connected particle-images are found. Then, the algorithm continues picking particle-images from the set of all ungrouped particles until all particles that can be grouped have been assigned a group. The distances L_e between individual particles and the epipolar lines from adjacent-lens particles and these distances represent the error residuals of this step, and are preserved for later use.

3.3.5 Epipolar Triangular Connectivity (ETC) grouping

Within each of the epipolar groupings, each particle is checked to see if it has at least 2 neighboring particles along different epipolar lines. If so the neighboring particles, if they are also neighbors of each other, are checked for epipolar connectivity. If the three particles are all connected, they meet the epipolar triangular connectivity (ETC) criterion. Any particle-image for which the ETC criterion is not met for any group is removed. This further reduces the potential to produce incorrect matchings between particle images, which would lead to errors in location estimation. At the end of this process, any ETC groups that share a particle are grouped together. Each of these groups should each contain every particle-image location corresponding to a single particle. A graphical representation of a sample result of this method is shown in Figure 19a, where the groups of particles from Figure 18 which comply with this criterion have been linked together by magenta lines. Several particles which do not comply with the criterion are present as well. These are typically particles for which the centroid detection method has been unable to find the centroid accurately, frequently due to the particle-image being on the edge of a lens such that it is partially occluded. Figure 19b shows the same result for a particle that is much closer to the camera and has a higher virtual depth. Note how in Figure 19b, particles haven't been detected in one of the lens types because they are too far out of focus. A key thing to note here is that there are 3 outer particles in Figure 19a that have been rejected by the ETC, likely because their centres have been shifted due to being near the edge of the lens. Similar rejections are not seen at the higher virtual depth in Figure 19b, which may mean that some of the 2D centroid estimates in Figure 19b could be erroneous. The centre-shifting effect near the edge of the lens discussed earlier may be consistent enough to produce erroneous centroids for which the ETC criterion is still met.

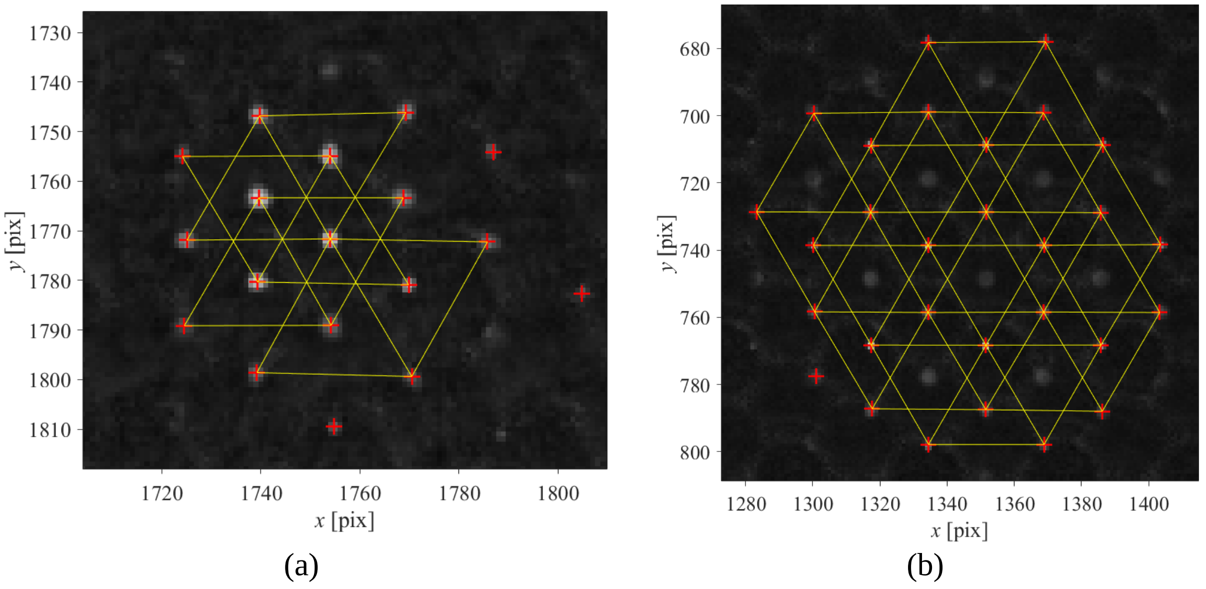


Figure 19 - Demonstration of the ETC criteria for a particle with (a) a small virtual depth and (b) large virtual depth. Images scaled differently, as demonstrated by axes.

3.3.6 Error-reduction filtering

To reduce potential errors in the final 3D location step, any particle-images which have associated epipolar connection errors exceeding a certain level relative to the other particle-images in the ETC group are eliminated from that group. This involves comparing the relative contribution $L_{e,r}$ of the ETC residuals L_e using:

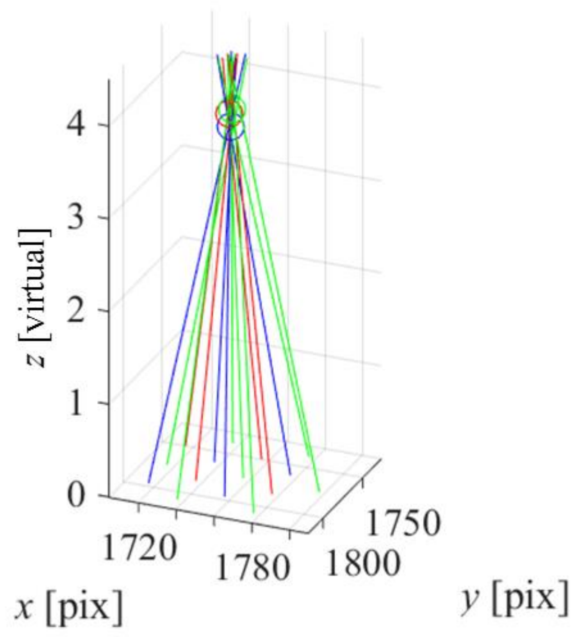
$$L_{e,i,r} = N \frac{L_{e,i}}{\sum_{j=1}^{N} L_{e,j}} < \varepsilon \qquad (18)$$

Particles with relatively large residuals, larger than the set limit ε, are then removed. This can help reduce the uncertainty in the data entering the 3D particle location algorithm, and is an attempt to address the issues that can arise when particles are partially occluded by the edge of a lens. Here, $L_{e,i,r}$ refers to the relative error contribution of particle i in a group of N particles j. Particles i can be eliminated based on if their relative error $L_{e,i,r}$ exceeds ε times the average error in the group. Typically, this parameter ε is set between 1.2 and 2 if it is used.

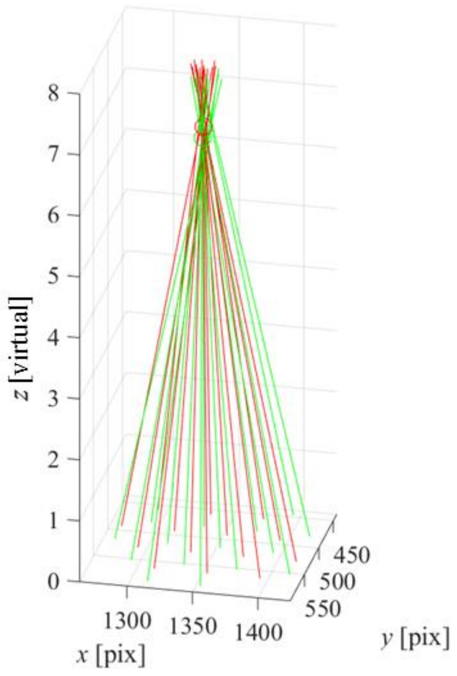
3.3.7 3D particle location

For each of the ETC-grouped particle-image sets, vectors are projected through the centre of each particle-image's micro-lens at a virtual depth of 0 to the particle-image locations on the CCD at a virtual depth of 1. The nearest-to-intersection point of the vectors is then determined by a least-squares fit of the minimum distance from a point to all the lines. The Levenberg-Marquardt method [46], [47] is used to perform the least-squares fit. The fitted point is taken as being the 3D location of the particle in pixel and virtual coordinates. The residual minimum point-to-line distances between the point and each of the vectors are preserved as a way to estimate uncertainty. The results of these operations applied to the images in Figure 19 are shown in Figure 20. All lenses have been collected into a single image for ease of comparison. Poor agreement in the virtual depths of the fitted points can be observed here. As each individual micro-lens-to-sensor distance B_i is different, the virtual depths of the located points are expected to be different. Implementation of a calibration algorithm capable of finding the distances B_i is thus necessary to align the z-values of the points predicted from each lens type in physical space.

Another key point is that the rejected points from Figure 19a are not present in the triangulation shown in Figure 20a, and good agreement can be seen between most of the lines and the detected centroids perhaps excluding the rightmost green line. However, in Figure 20b, the two green vectors to the back-right of the image can be seen to miss the main intersection point, and if this figure were rotated it is likely that other outer green vectors would miss the main intersection as well. This is an indication that an incorrect centroid has been found due to edge effects of the micro-lenses, and has then been preserved through the ETC method and error filter. For larger virtual depths, many particles can be expected to be found near the edges of the micro-lenses. Many triangles formed by adjacent incorrectly-located edge cases for high virtual depth particles appear to be complying with the ETC resulting in high average errors entering the error filter, which prevents the outer centroids from being rejected. It may be possible to improve the handling of particles near the edge of the micro-lenses by developing a custom 2D particle location algorithm, or it may be necessary to reject edge-case particles entirely to address this problem.



(a)



(b)

Figure 20 - Triangulation of particle position for the low virtual depth (a) and high virtual depth (b) particles shown in Figure 19. Vectors are colored by micro-lens type. Located points shown as large circles.

3.4 Calibration method

The camera calibration method used was originally developed as a standard single- or multi-camera calibration approach [5] and was more recently adapted for plenoptic cameras in C++ [35], [36]. This work re-implements the technique in Matlab. The Matlab scripts are given in Appendix D. A calibration target is generated by creating a square grid of equally-spaced dots appropriate for the imaging volume. For example, the macroscale experiments conducted involved a target field of view of a 60×60×60 mm cube, for which the selected dot spacing is 5.927 mm. In contrast, the microscale experiments take place in a 6×6×6 mm cube for which the selected dot spacing was 0.667 mm. These targets will give 8-10 dots across the field of view, which is adequate for the algorithm to numerically perform the calibration. After setting up the camera in the correct orientation for the experiment, this calibration target is held in the field of view and imaged at several orientations. A sample plenoptic image of the target is shown in Figure 21. The target in this image is angled with the bottom of the target nearer to the camera, and each dot of the target appears in multiple micro-lens images. As a result, dots at the bottom of the image are closer to the camera and appear in more micro-images than dots at the top. The dots also spread apart moving from the top of the image to the bottom, which is a result of the change in magnification reducing the in-plane field of view in the near-field of the camera. Other research has indicated that 4 images acquired in this way should be sufficient for this type of calibration algorithm if the target positions have good coverage of the imaging volume [35], [36]. The target used in this case had a lower dot density, so a minimum of 8 calibration images have been used to ensure that the imaging volume was well-covered.

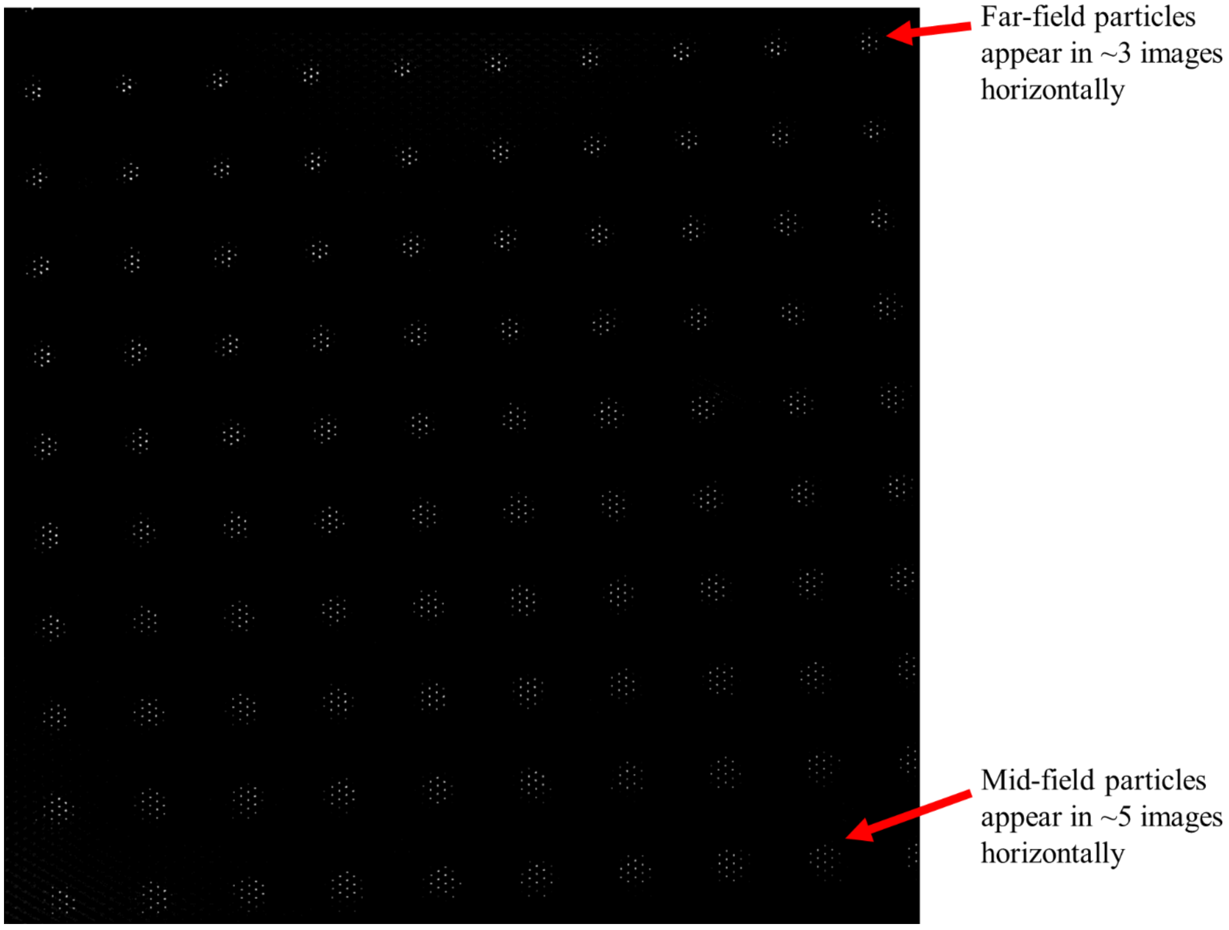

Far-field particles appear in ~3 images horizontally
Mid-field particles appear in ~5 images horizontally

Figure 21 – A sample calibration image. Individual calibration points have been split into multiple micro-images.

Once acquired, each of the calibration images can be processed using either of the 3D location algorithms discussed in sections 3.2 and 3.3. The relative 2D positions of the dots are then identified by considering the vector distances between them, beginning in the center of the image [35], [36]. This approach ignores any centroids that do not align with the grid that could have been created by noise in the image. A model for the target is generated for each image from the known vector distances between the grid points, containing only the points on the calibration target that have been identified in the image.

To convert from pixel and virtual coordinates to object-side positions, the pixel and virtual coordinates are first converted into metric locations in image-space. Then, Brown's model for radial distortion [48] and the Petzval model for field curvature [49] are applied to remove the 2D and 3D effects of main lens aberration, respectively. Finally, the thin lens model [37] is applied to re-project the points into object space. The locations of the points in object-space are then compared to the model target points, which are held in a fixed grid that is positioned by controlling the extrinsic positional parameters. Equations (19) - (23) are associated with these operations, using the variables in Table 2. Figure 1 contains some of the key associated variables, so it has been repeated here as Figure 22 for convenience.

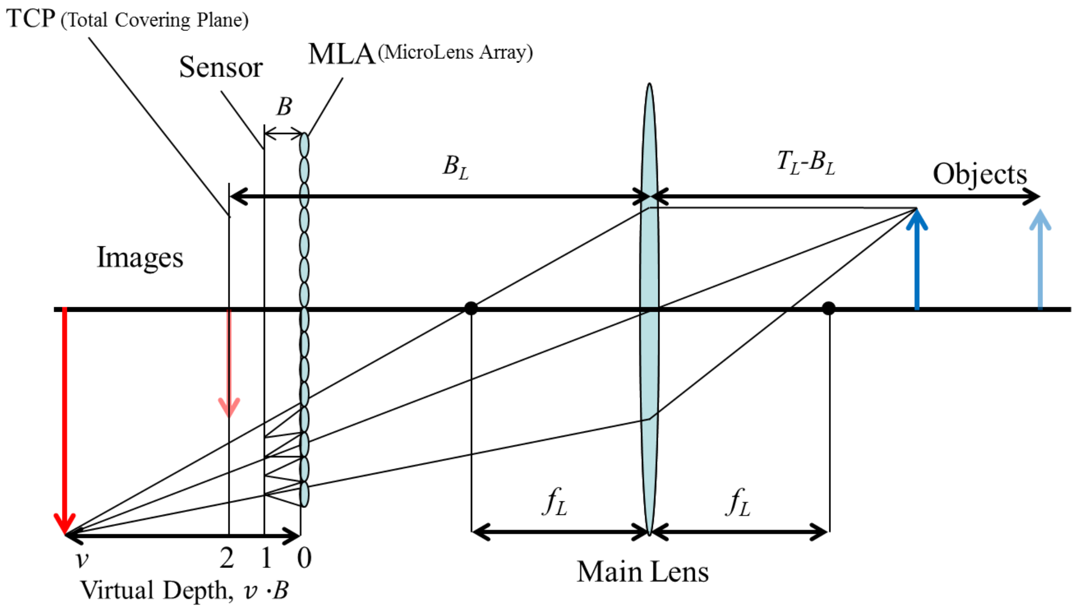


Figure 22 - Optical arrangement of a focused plenoptic imaging system. A single set of ray-paths for one sample object is shown. Microlens focusing behavior assumed ideal for the object for simplicity.

Table 2 - Variable list for metric conversion

x	Particle sensor-side in-plane x-position, in pixels	f_L	Main lens focal length, in mm
y	Particle sensor-side in-plane y-position, in pixels	T_L	Main lens focus distance; the physical distance from the total covering plane at $z = 2$ to its projected conjugate in object-space, in mm
z	Particle sensor-side out-of-plane position, in virtual units	B_L	Distance from main lens to the TCP*
$'$	Prime, indicator of a transformation applied to one of x, y, or z. Not related to the common convention of differentials.	r	Radial distance from the centre of distortion to each point
X	Particle object-side in-plane x-position, in mm	x_0	x-distortion centre
Y	Particle object-side in-plane y-position, in mm	y_0	y-distortion centre
Z	Particle object-side out-of-plane position, in mm	k_1	Second-order radial distortion coefficient
s_p	Sensor pixel size, in mm	k_2	Fourth-order radial distortion coefficient
R_x	Sensor pixel count in x direction	d_1	Second-order radial distortion coefficient
R_y	Sensor pixel count in y direction	d_2	Fourth-order radial distortion coefficient
B_i	Micro-lens-sensor distance for each micro-lens type i	d_D	Radial depth distortion coefficient

B_L is calculated here as:

$$B_L = \frac{T_L}{2} \times \left(1 - \sqrt{1 - \frac{4f_L}{T_L}}\right) \tag{19}$$

The first step is to convert the points from pixel and virtual coordinates to physical coordinates, relative to the centre of the main lens:

$$x' = \left(x - \frac{1}{2}R_x\right) \times s_p \tag{20a}$$

$$y' = \left(y - \frac{1}{2}R_y\right) \times s_p \tag{20b}$$

$$z' = B_L + (z - 2) \times B_i \tag{20c}$$

Here, $(z - 2)$ is used because B_L is calculated relative to the TCP, so this operation is applying the -2 shifts to the virtual depths to make them relative to the TCP.

Distortions are removed through the application of Brown's model and another model for Petzval field curvature (22). These are radially dependent, so the distance from the points to the distortion centre must also be determined (21).

$$r = \sqrt{(x' - x_0)^2 + (y' - y_0)^2} \tag{21}$$

$$x'' = x'(1 + k_1 r^2 + k_2 r^4) \tag{22a}$$

$$y'' = y'(1 + k_1 r^2 + k_2 r^4) \tag{22b}$$

$$z'' = z' + (1 + d_D z')(1 + d_1 r^2 + d_2 r^4) \tag{22c}$$

Finally, the locations are projected through the main lens using the thin lens equation and simple trigonometry:

$$Z = \frac{z'' f_L}{z'' - f_L} \tag{23a}$$

$$X = -x'' \frac{Z}{z''} \tag{23b}$$

$$Y = -y'' \frac{Z}{z''} \tag{23c}$$

These operations require known values for the influencing parameters. These parameters, along with the parameters which control the position and angle of the target positions, are then iteratively modified using the Levenberg-Marquardt method [46], [47] until the sum-of-squares of the distances between the calculated and fixed grid points is minimized. As it is expected that the out-of-plane locations will be less accurately determined than the in-plane locations, the weight of the out-of-plane distances is reduced, generally by between 2 and 4 times. This calibration must be performed for each new lens configuration, and the results for each of the calibrations performed in this study have been included in their respective chapters 5-7.

3.5 Particle selection

A final step after the mapping must be applied for the ETC-based method. Given that three different lens types are used with this type of camera, three different estimates of the location of a single particle are frequently obtained. This will be the case for the particle in Figure 19a. In Figure 19b, only two estimates of the particle's location are obtained because the particle is too dim in the other lens sets to be found by the particle location algorithm. In either case it is generally useful to select one of these as the 'best estimate'. Identifying that these are three estimates of a single particle is the first step in this process. This is accomplished by using one of the particles as a starting point and taking a radius around it, within which the nearest particle from the other two lens types will be considered to be the same particle. A larger radius is used in the *z*-direction than the *x*- or *y*-directions to ensure that the estimates will be grouped. It is easier to do this after the calibration step, as the calibration step will usually reduce the disparity in the measured depth of the particles caused by the differences in the MLA location parameters B_i.

Two methods are proposed for determining which measured location is the "best estimate" of the particle position. The per-vector residual of each particle can be calculated by simple division of the residuals of the least-squares operation in Section 3.3.7 by the number of vectors, similar to (17). The particle with the lowest per-vector residual is then considered the "best estimate". The second method assumes that the 2D location step will be most accurate for the most in-focus particle images and that the most in-focus particle images will have the highest average intensity. The average intensity for the particle-images of each lens type is calculated, and the particle derived from the lens type with the highest average intensity is selected as the "best estimate".

CHAPTER 4 PARTICLE TRACKING VELOCIMETRY

The particle tracking approach developed for this work draws on several past works, but has been specifically adapted and implemented in Matlab for tracking particles resultant from the plenoptic images. The Matlab scripts are given in Appendix E. The particle tracking approach initializes the particle tracks using a two-frame approach, and continues with forward-predictive tracking after a particle is tracked for a few frames. The two-frame approach was previously developed in-house by another researcher [9]. Acceleration-based track rejection is then applied to remove spurious tracks, followed by a combined forward- and backward-prediction approach that links together track segments. Lastly, a temporal filter is applied to smooth the data. The equations involved in the particle tracking approach used will make use of the variables defined in Table 3.

Table 3 - List of PTV variables

t	Time	i	Current particle index (frame or track)
n	Last particle in set	j	Secondary particle index
m	Time-indexing variable	ψ	Particle index for uncertain, potential-match particles
l	Track length	N	Neighborhood particle index
$\vec{x}$	Particle location vector	P	Polynomial-predicted particle label (subscripted); or polynomial coefficient
$\vec{v}$	Particle velocity vector	A_z	Noise ratio of z data relative to *x*,*y* data
$\vec{a}$	Particle acceleration vector	B_z	Noise ratio of z data relative to *x*,*y* data, adjusted to apply to velocity term
r_N	Neighborhood radius	r_T	Two-frame tracking acceptance radius
r_q	Neighborhood particle acceptance radius	r_p	Velocity-prediction particle position acceptance radius
r_v	Track-linkage velocity deviation acceptance term	a_T	Acceleration threshold

The challenge here is that the particle location algorithm typically has a high uncertainty in calculating the out-of-plane position of a particle. Many of the steps in the algorithm account for this by allowing for a much higher error range in the out-of-plane direction than the in-plane direction. Thus, when a particle is predicted to be within a certain radius by either the initialization step or the forward-prediction step, the allowable range of positions will be represented by an ellipsoid extended into the out-of-plane direction rather than a sphere.

4.1 Track initialization

Track initialization is achieved by using a two-frame neighborhood tracking approach based on a previously developed in-house algorithm [9]. The approach outlined in Figure 23 assumes that, within a given region around the tracked particle, all other particles should move at a similar velocity. It looks at the second-frame potential matches for a first-frame particle, within a specified 3-D radius r_T (see Figure 23a-b), and projects each of the resulting vectors from the other first-frame particles within a 3-D neighborhood radius r_n (see Figure 23c-d). The particle kept as the match for the first-frame particle is the particle for which the most projected vectors point to second-frame particles, within a specified trust radius r_q. In the example Figure 23, x_2 is selected as the vector for the central particle. Mathematically, this algorithm can be represented by (24) through (28).

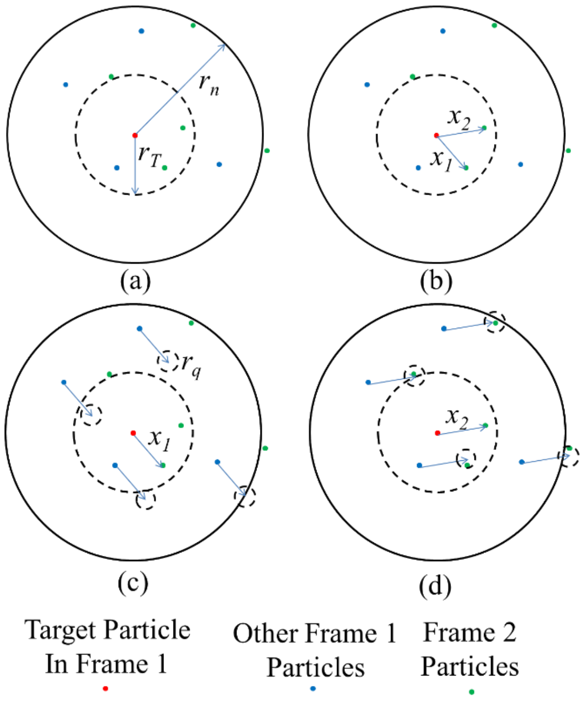


Figure 23 - Neighborhood tracking algorithm. Particle search radius r_T, neighborhood radius r_n, vectors x_1 and x_2.

In neighborhood tracking, neighboring particles $\vec{x}_{t,i_N}$ around the central particle $\vec{x}_{t,i}$ are selected from the set of all other particles in the same frame $\vec{x}_{t,j}, j \neq i$:

$$\vec{x}_{t,i_N} = \{\vec{x}_{t,j} \in \mathbb{R}^3 \mid (\|\vec{x}_{t,j} - \vec{x}_{t,i}\| < r_n)\} \qquad (24)$$

Here, r_N refers to the radius of the neighborhood that is considered around particle $\bar{x}_{t,i}$. All potential second-frame particles $\vec{x}_{t+1,i_\psi}$ for the central particle $\vec{x}_{t,i}$ are then found from all possible second frame particles $\vec{x}_{t+1,j}$, as well as the vector distances to these points:

$$\vec{x}_{t+1,i_\psi} = \{\vec{x}_{t+1,j} \in \mathbb{R}^3 \mid \|[\vec{x}_{t+1,j} - \vec{x}_{t,i}]\| < r_\mathrm{T}\} \tag{25}$$

$$\vec{v}_{t,i_\psi} = \vec{x}_{t+1,i_\psi} - \vec{x}_{t,i} \tag{26}$$

These vector distances are then applied to the neighborhood particles:

$$\vec{x}_{t+1,i_{\psi,N}} = \vec{x}_{t,i_N} + \vec{v}_{t,i_u} \tag{27}$$

A maximum deviation radius r_q is taken around these locations $\vec{x}_{t+1,i_{u,N}}$ and second-frame matching particles $\vec{x}_{t+1,j}$ are identified:

$$\left\|\vec{x}_{t+1,i_{\psi,N}} - \vec{x}_{t+1,j}\right\| < r_q \tag{28}$$

The second-frame particle $\vec{x}_{t+1,i}$ with the greatest number of matching neighbors satisfying this equation is taken as being the correctly tracked particle. If this algorithm fails, generally due to not having enough neighbors for the algorithm to work, nearest-neighbor tracking is employed, represented by (29). Nearest neighbor tracking is implemented by, for each of the particles *i* in the frame at time *t*, $\vec{x}_{t,i}$, finding a particle in the set of particles *j* in the frame at time *t*+1, $\vec{x}_{t+1,j}$ that satisfies:

$$\vec{x}_{t+1,i} = \{\vec{x}_{t+1,j} \in \mathbb{R}^3 \mid \min(\|[\vec{x}_{t+1,j} - \vec{x}_{t,i}]\|) < r_\mathrm{T}\} \tag{29}$$

Here, r_T refers to the radius around particle $\bar{x}_{t,i}$ within which a particle $\bar{x}_{t+1,j}$ is accepted.

When selecting the neighborhood size, the local spatial velocity gradients must be accounted for. Flows with larger velocity gradients will need smaller neighborhoods for this method to be successful. This approach is more than adequate for tracking the fixed field, as there are no spatial gradients in the velocity field. Particles which are tracked continuously through 4 frames by this method are sent into the time-resolved tracking algorithm for tracking in future time-steps.

These equations describe the classical implementation of the 2-frame tracking algorithm. The algorithm has been modified for the plenoptic particle data by weighting of the vector residuals using a settable *z*-error weighting term when looking for a particle using (28) or (29). This allows the algorithm to capture particles with higher *z*-deviations produced by the particle location method.

4.2 Time-resolved tracking

The time-resolved tracking algorithm collects several of the most recent positions of a continuous series of particles and fits three O(t^3) polynomial functions $x(t)$, $y(t)$ and $z(t)$, which are then used to predict the particle position in the next frame [50]. If a particle is found within a certain specified radius of that position, it is attached to the track that predicted it would be there. The number of positions used to create the polynomial function is another selected parameter. Typically, up to 10 positions are used, but this selection needs to be made based on the particle velocity gradients. Larger gradients in the particle velocity over these positions can make it difficult for the polynomial fitting function to predict particle position accurately. Equations (30)-(31) are associated with these operations.

The forward-predictive tracking algorithm attempts to fit a second-order polynomial to the last *n* particles in each track *i* ($\vec{x}_{t-(n-1),i}$ through $\vec{x}_{t,i}$), searching the particles $\vec{x}_{t+m,j}$ in frame *j* at the next *m* time steps (typically *m*=1,2,3) to find a potential particle $\vec{x}_{t+m,j_{\psi}}$. Due to the large errors in calculating the *z*-position of the particles inherent in the plenoptic reconstruction, a larger tolerance in the *z*-direction is allowed, accounted for by the term A_z. This is a key feature of the particle tracking approach implemented by this work. An overall acceptance region around the predicted point is defined by r_p.

The polynomial is found by least squares fitting. Then, $\vec{x}_{t,i_P}$ can be defined in terms of the polynomial fit:

$$\vec{x}_{t,i_P} = \begin{bmatrix} P_{2_x}t^2 + P_{1_x}t + P_{0_x} \\ P_{2_y}t^2 + P_{1_y}t + P_{0_y} \\ P_{2_z}t^2 + P_{1_z}t + P_{0_z} \end{bmatrix}_{t,i} \qquad (30)$$

To determine if a valid particle is found, a similar matching algorithm to the two-frame tracking algorithm is applied:

$$\vec{x}_{t+m,i_\psi} = \left\{ \vec{x}_{t+m,i_P} \in \mathbb{R}^3 \middle| \min \left(\left\| \begin{matrix} (x_{t+m,i_P} - x_{t,i}) \\ (y_{t+m,i_P} - y_{t,i}) \\ A_z(z_{t+m,i_P} - z_{t,i}) \end{matrix} \right\| \right) < r_p \right\} \tag{31}$$

Forward-predictive tracking is applied first where tracks of $l \geq n$ exist. On particles for which forward-predictive tracking fails, neighborhood tracking is applied next, and nearest-neighbor tracking is applied last to capture any remaining particles. This approach attempts to create as many particle tracks as possible and as such can result in bad particle tracks being formed. A filtering algorithm is applied after these preliminary tracking steps, removing these bad particles and completing linkages that the tracking algorithm was unable to.

4.3 Temporal filtering approach

Once all frames have been completed, the algorithm looks for accelerations that exceed a specified threshold and breaks the tracks at these locations. This helps remove spurious tracks that may have been a result of the two-frame tracking algorithm. The algorithm then looks at all the track segments and performs an iteration of forward- and backward-predictive tracking, which links together tracks based on the positions, velocities, and accelerations of their endpoints through further polynomial fitting. Finally, a temporal filter is applied by fitting polynomials over moving windows with limits up to 10 particles to either side of a given particle.

4.3.1 Track Removal

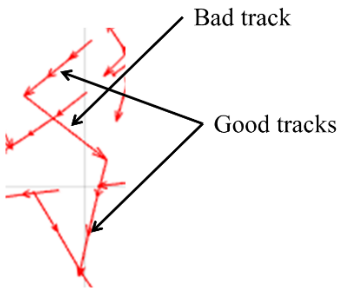


Figure 24 - Application of the acceleration-based track removal algorithm.

The track removal algorithm attempts to detect and remove track segments where a high acceleration is detected. An example of this case is shown in Figure 24. A track of length l is deemed a bad track when a local acceleration along the track is larger than a threshold a_T. For each track i, the highest acceleration is determined and compared to the threshold:

$$a_{\max} = \max\left(a_{t,i} = \left\|\vec{x}_{t+1,i} - 2\vec{x}_{t,i} + \vec{x}_{t-1,i}\right\|\right) > a_T, \qquad t = 2 \ldots l-1 \tag{32}$$

There are several conditions for which a bad acceleration can result. Table 4 outlines how the algorithm handles each case.

Table 4 - Track removal acceleration cases

Case	Result
1. Track is l = 3, acceleration found is bad	Points should not be linked, remove track.
2. Bad acceleration is at t = 2 or t = l-1	Remove first or last point
3. Bad acceleration is in middle of track, 2 < t < l-1	Calculate accelerations on either side of bad point. Track is split between bad point and second-worst acceleration.

Iterations of this algorithm are applied until no accelerations exceeding the threshold are present, as tracks may need to be split multiple times using case 3, or use of case 3 may produce tracks for which cases 1 or 2 apply.

4.3.2 Track Linkage

The track linkage algorithm aims to 'bridge' the gaps between existing track segments that should be a single, continuous track. An example of this type of track data is shown in Figure 25, where these gaps are circled. These can be caused where a particle is lost between frames, where the acceleration-based rejection improperly broke a track, or where a track wasn't linked because the depth errors were too high for the three primary methods to create a track.

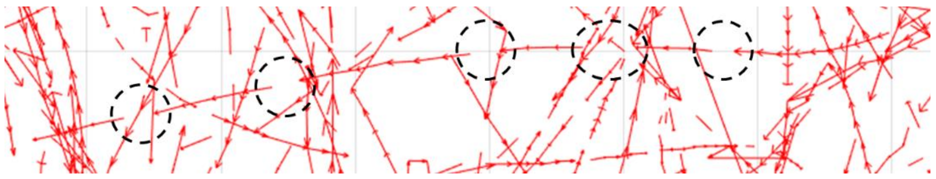


Figure 25 - Example of 'gaps' in a track which should be continuous.

The track linkage algorithm links tracks together by considering the time, position and velocity at the endpoint and starting point of nearby tracks. How it does so depends on the length l of the earlier track. If the track is too short, there is not enough z information to predict future z positions accurately due to the noise inherent in the z-data. Also, tracks of length 2 can't be fit by a second-order polynomial. The algorithm is thus applied with slight variations depending on the length of the track, as presented in Table 5.

Table 5 - Polynomial fitting cases

Track Length	Response	Variables
l=2	First-order polynomial predicts *x*-*y* positions *z*-positions averaged	$P_{2_{x_i}}, P_{2_{y_i}}, P_{2_{z_i}}, P_{1_{z_i}} = 0$
2<*l*<5	Second-order polynomial predicts *x*-*y* positions *z*-positions averaged	$P_{2_{z_i}}, P_{1_{z_i}} = 0$
l≥5	Second-order polynomial predicts *x*-*y* positions First-order polynomial predicts *z*-positions Only last 5 points used.	$P_{2_{z_i}} = 0$

Particle positions $\vec{x}_{t,i_P}$ are predicted based on track *i* using the calculated polynomials:

$$\vec{x}_{t,i_P} = \begin{bmatrix} P_{2_x}t^2 + P_{1_x}t + P_{0_x} \\ P_{2_y}t^2 + P_{1_y}t + P_{0_y} \\ P_{2_z}t^2 + P_{1_z}t + P_{0_z} \end{bmatrix}_{t,i} \qquad (33)$$

Calculating the derivative of this polynomial gives a predicted velocity of the points as well:

$$\vec{v}_{t,i_P} = \begin{bmatrix} 2P_{2_x}t + P_{1_x} \\ 2P_{2_y}t + P_{1_y} \\ 2P_{2_z}t + P_{1_z} \end{bmatrix}_{t,i} \qquad (34)$$

The matching algorithm from the forward-predictive tracking is applied for other tracks j, but now includes velocity information. Again, a position-acceptance region r_p applies, as does a velocity-acceptance region r_v:

$$\vec{x}_{t+m,i_\psi} = \left\{ \vec{x}_{t+m,i_P} \in \mathbb{R}^3 \;\middle|\; \begin{matrix} \min\left(\left\| \begin{matrix} \left(x_{t+m,i_P} - x_{t,j}\right) \\ \left(y_{t+m,i_P} - y_{t,j}\right) \\ A_z\left(z_{t+m,i_P} - z_{t,j}\right) \end{matrix} \right\| \right) < r_p \\ \min\left(\left\| \begin{matrix} v_{x,t+m,i_P} - \left(x_{t+m+1,j} - x_{t+m,j}\right) \\ v_{y,t+m,i_P} - \left(y_{t+m+1,j} - y_{t+m,j}\right) \\ B_z\left(v_{z,t+m,i_P} - \left(z_{t+m+1,j} - z_{t+m,j}\right)\right) \end{matrix} \right\| \right) < r_v \end{matrix} \right\} \qquad (35)$$

Typically, this algorithm is applied for *m*=1…5 with *t* defined by the end of the track. The algorithm thus has the potential to link track *i* to adjacent tracks multiple times and this is harnessed to increase the confidence of the algorithm.

4.3.3 Track Smoothing

Track smoothing is performed as a final step. This is a key feature of this tracking algorithm, implemented to reduce the level of noise inherent in the particle location measurements. Tracks of $l \geq 5$ from the linkage algorithm have the smoothing algorithm applied and any smaller tracks are too small to smooth and are eliminated. The smoothing algorithm looks at each time-step within the track and this can include time-steps for which a particle was not found, but for which a gap was bridged with the linking algorithm. The smoothing algorithm fits a second-order polynomial to all time-steps *t* in the track using all valid points within *m* time-steps $\vec{x}_{t-m,i} .. \vec{x}_{t+m,i}$ in the track, determining the position of the point at *t* using the polynomials:

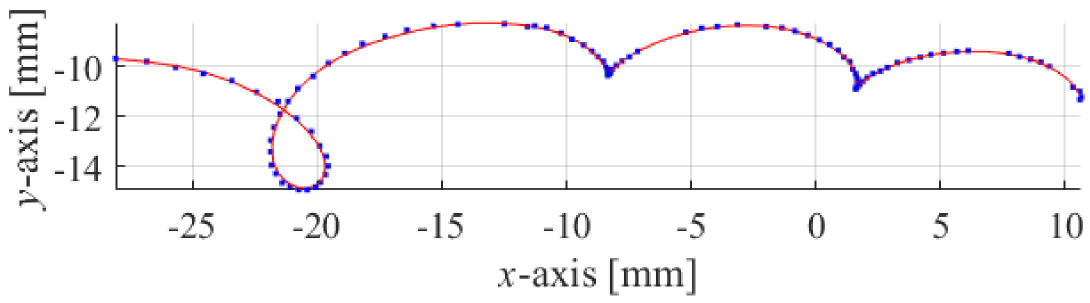
$$\vec{x}_{t,i_P} = \begin{bmatrix} P_{2_x}t^2 + P_{1_x}t + P_{0_x} \\ P_{2_y}t^2 + P_{1_y}t + P_{0_y} \\ P_{2_z}t^2 + P_{1_z}t + P_{0_z} \end{bmatrix}_{t,i} \tag{36}$$

The velocity and acceleration of the point can also be determined by taking the derivatives of the polynomials:

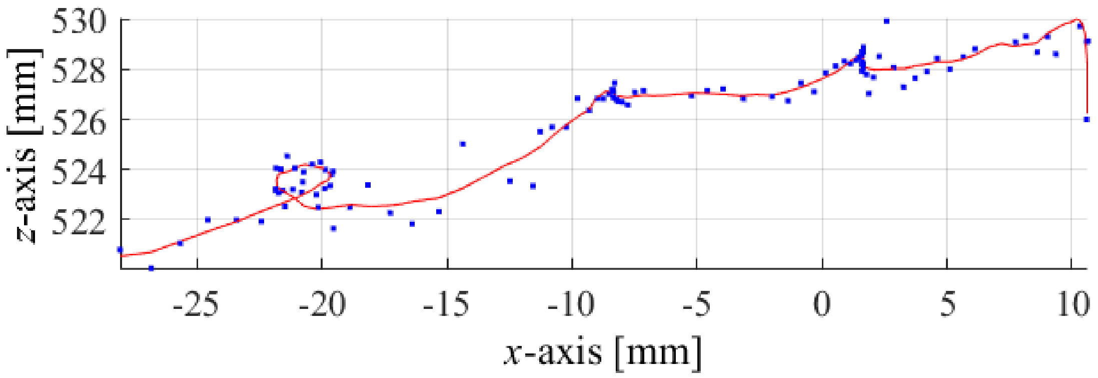
$$\vec{v}_{t,i_P} = \begin{bmatrix} 2P_{2_x}t + P_{1_x} \\ 2P_{2_y}t + P_{1_y} \\ 2P_{2_z}t + P_{1_z} \end{bmatrix}_{t,i} \tag{37}$$

$$\vec{a}_{t,i_P} = \begin{bmatrix} 2P_{2_x} \\ 2P_{2_y} \\ 2P_{2_z} \end{bmatrix}_{t,i} \tag{38}$$

This algorithm temporally filters the path taken by a single particle, which can effectively average the errors associated with the position of the particle. This is especially important in the out-of-plane direction. An example of this algorithm applied to a single track is shown in Figure 26. This figure shows that the tracking performance is stronger in-plane than out-of-plane due to the lower level of uncertainty in determining the particle position.



(a)



(b)

Figure 26 - Unfiltered points (blue) and filtered path (red) associated with a single, 112-frame long particle track. (a) shows the track in the x-y plane, and (b) shows the track in the x-z plane.

CHAPTER 5 EXPERIMENTAL MEASUREMENT OF UNCERTAINTY

In order to analyze the ETC technique's performance when being used to track particles, an experiment was designed and performed that aimed to evaluate the performance of the technique when observing known, constant velocities. The following will discuss the design of this experiment and the results of applying the ETC technique. This allowed an experimental evaluation of the uncertainty associated with the technique's use in the optical configuration of the experiment to be conducted.

5.1 Experimental setup and data acquisition

The focused plenoptic camera used in this study was a commercially available *f/#* 2.4 camera (R5, Raytrix GmbH) with an 85 mm main lens. This is a multi-focus plenoptic camera (MFPC); the micro-lens array has lenses of 3 different focal lengths. It is capable of acquiring 2048 by 2048 pixel images at frame rates up to 180 fps. The camera was used to image a calibration target with white dots on a black background in a grid as shown in Figure 27. The dots were spaced apart 5.927 mm and have a diameter-to-spacing ratio of 1/20. These dimensions were selected to ensure that 100 dots were visible at the far-field range of the camera, 64 dots were visible at the near-field range, and that the size of the dots was appropriate for the particle location algorithm through all ranges of depth. The target was generated using a 1200 DPI laser printer (M401dne, HP), then laminated, glued to a piece of flat glass, and fixed to a motorized traverse (500 mm BiSlide, Velmax Inc.). The traverse allows for controlled movements in steps as small as 5 micron.

Data is then generated by moving the traverse parallel to the camera for 100 steps of 0.5mm, followed by a 0.1 mm displacement towards the camera, followed by 99 further iterations of the same process. The traverse is held fixed at each position and an image of the target is taken. This generated a total of 10,200 data points which can be processed to either measure the in-plane displacements at each depth location, or to measure the out-of-plane displacements independently. 101 time-series containing 101 images can be constructed for both the in-plane and out-of-plane motion cases, for a total of 202 data sets. At least 64 dots should always be visible on the calibration target, which ensures that over 6,000 observations can be made at each depth.

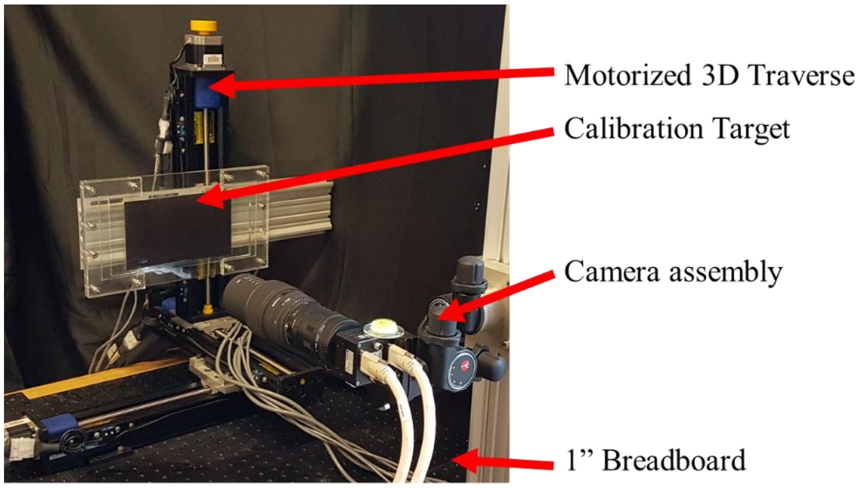


Figure 27 - Experimental setup of the fixed-field test

A pair of raw sample images of the target taken with the plenoptic camera are given in Figure 28. These have been cropped to show only the 3 central dots in the image for clarity. As expected, the near-field image contains fewer, but larger, sub-images of each dot when compared to the far field image.

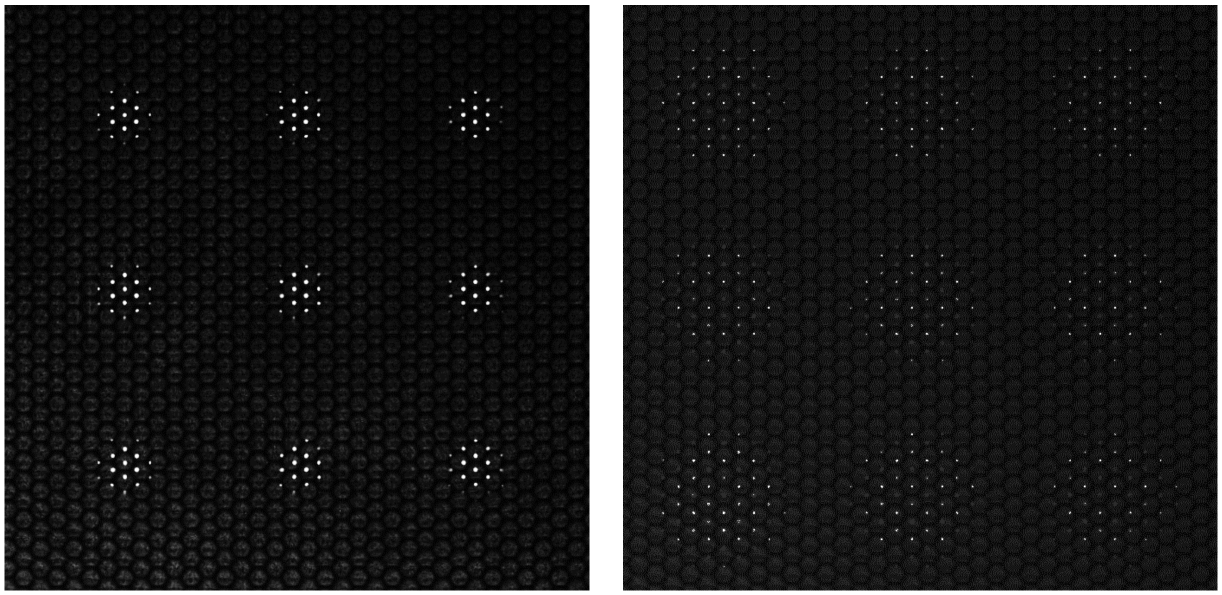


Figure 28 – Raw plenoptic images of the calibration target placed at the (a) far-field and (b) near-field of the plenoptic camera. Brightness and contrast have been adjusted to make the dots easily visible.

5.2 Calibration

The micro-lens location approach discussed in Section 3.3.1 was applied first. Taking a histogram of the residuals resultant from the least-squares determination of the micro-lens location vectors yields Figure 29. Percentile-based statistical analysis indicates that 95% of the data exhibit a residual of less than 0.133 pixels. The resultant vectors that define the positions of neighboring same-type lenses in this case are (40.28, -0.003) and (20.14, 34.88).

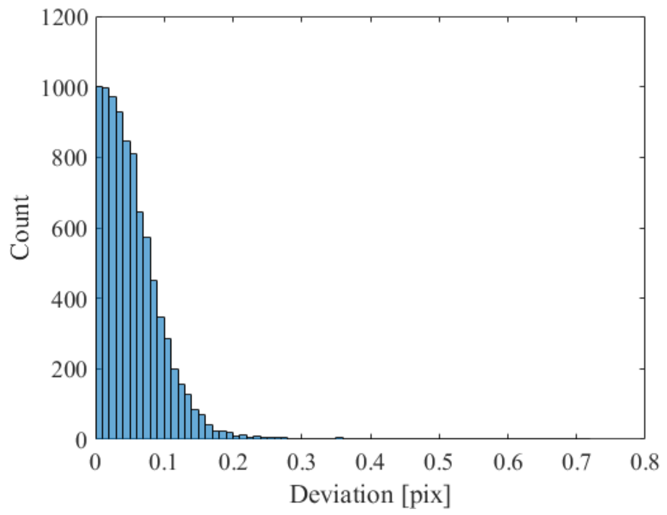


Figure 29 - Histogram of the residuals of the 2D vector norm of the difference between the measured micro-lens locations and the gridded locations for the flat field tests.

The physical-space calibration is performed as described in Section 3.4, using the ETC method described in Section 3.3 to find the target dots. The resultant calibration variables are given in Table 6. Figure 30 shows the detected points in red overlaid with the predictive calibration points in blue, for two different images of the target. The target in (a-b) was parallel to the imaging field of the camera, while the target in (c-d) was at an angle relative to the imaging field of the camera. Excellent agreement between the *x* and *y* positions is observed with 95% of the residuals in (-0.18, 0.18) and (-0.17, 0.16) mm, respectively. The *z* residuals are much higher, with 95% of the residuals in (-0.77, 0.80) mm which is indicative of the higher uncertainty associated with finding the out-of-plane locations relative to the in-plane locations. This is highlighted further in the histograms with overlaid normal distributions as shown in Figure 31.

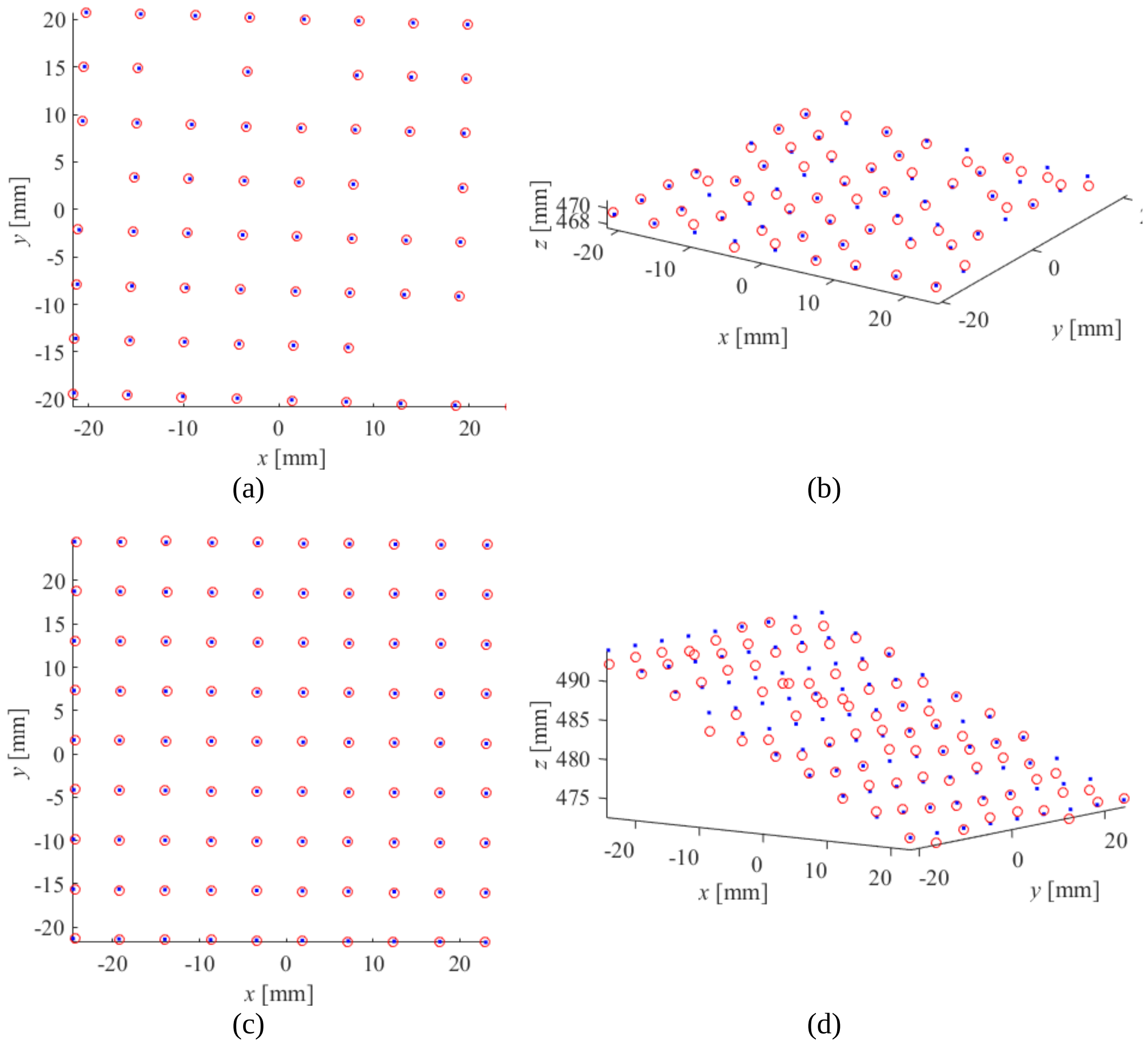
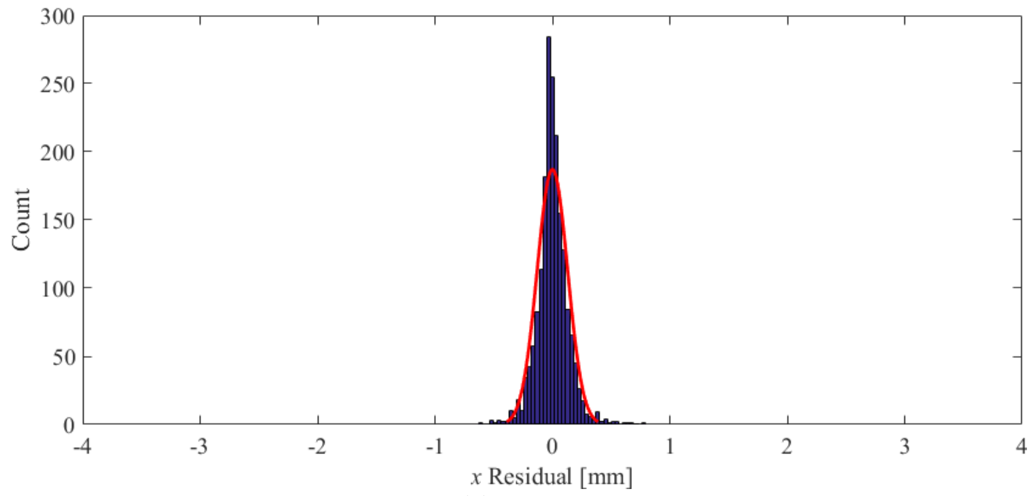
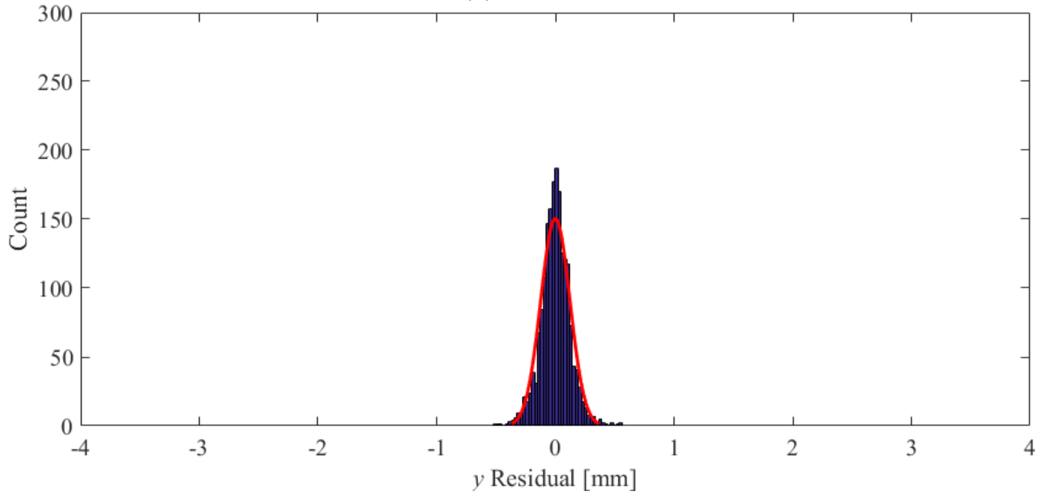


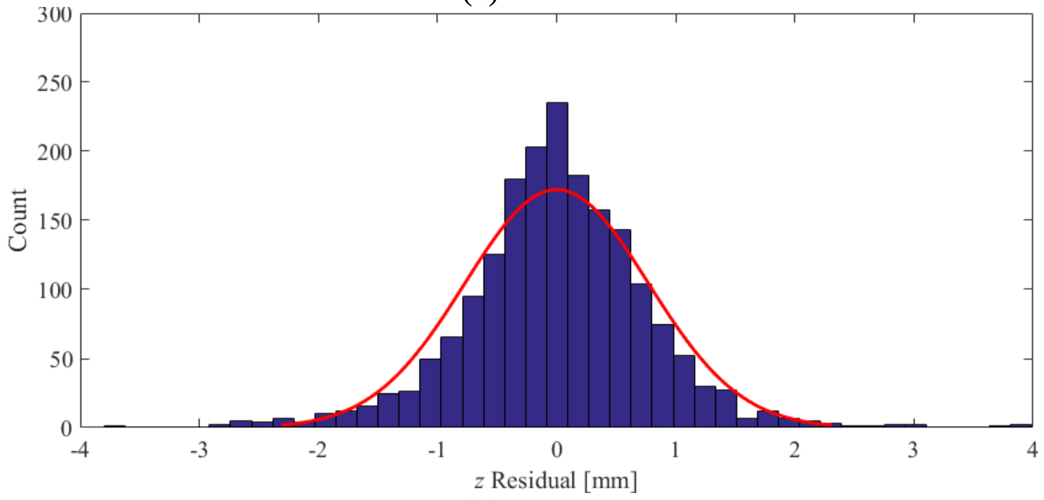
Figure 30 – Detected calibration points in red overlaid on blue grid of predicted points in the fixed-field test. In (a-b), the calibration target is aligned with the camera while in (c-d) the target is angled relative to the camera. (a) and (c) are in-plane views of the points, and (b) and (d) are positioned to show the angle of the target.



(a)



(b)



(c)

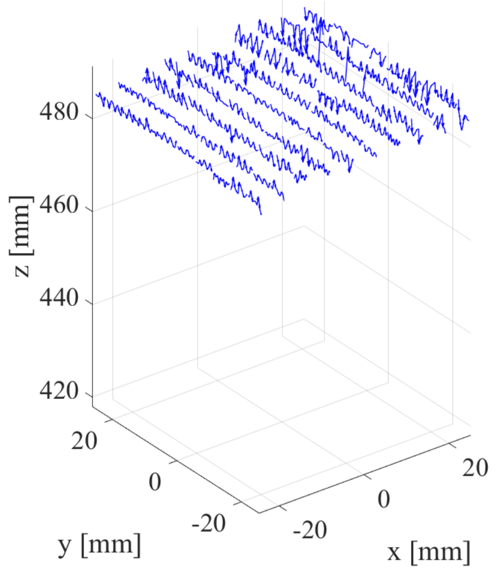
Figure 31 - Histograms of flat-field test micro-lens calibration residuals in (a) *x*, (b) *y*, and (c) *z* directions, overlaid with normal distributions.

Table 6 - Calibration variable results for flat field tests

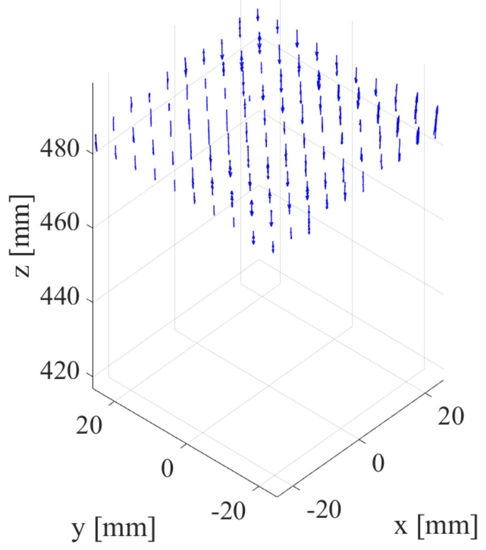
T_L	624.82 mm	k_1	5.54e-5
B_1	0.334 mm	k_2	5.83e-7
B_2	0.336 mm	d_1	4.81e-6
B_3	0.347 mm	d_2	1.52e-9
x_0	2.91 mm	d_D	3.88
y_0	1.54 mm		

5.3 Data Processing

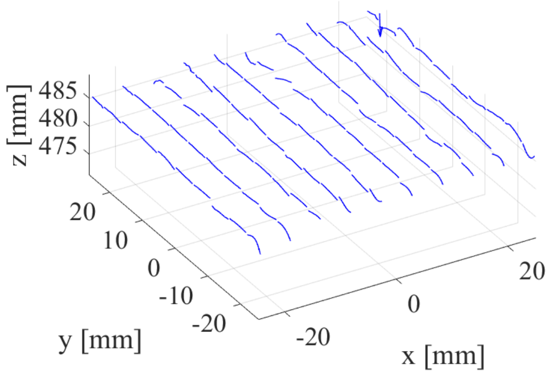
The processing methods discussed in Sections 3.3 and 4 are applied, producing particle tracks for each of the 202 time-series. Samples of the resultant raw particle tracks are shown in Figure 32a-b, and the particle tracks after smoothing are shown in Figure 32c-d. These figures show the vector displacements of the dots through 5 consecutive frames. The level of noise in the out-of-plane direction is apparent in Figure 32a, where the target was moved parallel to the camera, through the simple observation that the tracks produced have significant random deviations in the *z*-direction when only *y*-directional motion was expected. Moving the particles directly towards the camera as in Figure 32b produces a fairly consistent lack of movement in the *x* and *y* directions, but some level of random variation in the *z*-direction where consistent motion would be expected. Application of the smoothing algorithm appears to have significantly reduced the random variations, with smooth tracks being observed in both Figure 32c and d. These images also appear to indicate that the approach is more prone to errors near the edges of the image, as evidenced by a vector that deviates sharply in the *z*-direction in the smoothed set, observable in Figure 32c near (20,20). This is likely caused by there being fewer micro-lenses available to image the particle near the edge of the sensor. Curvature of the paths near the edge of the field of view is also notable, which is likely a result of there being no more tracks beyond the edge of the field of view combined with these edge effects producing poor estimations that are influencing the fitted tracks.



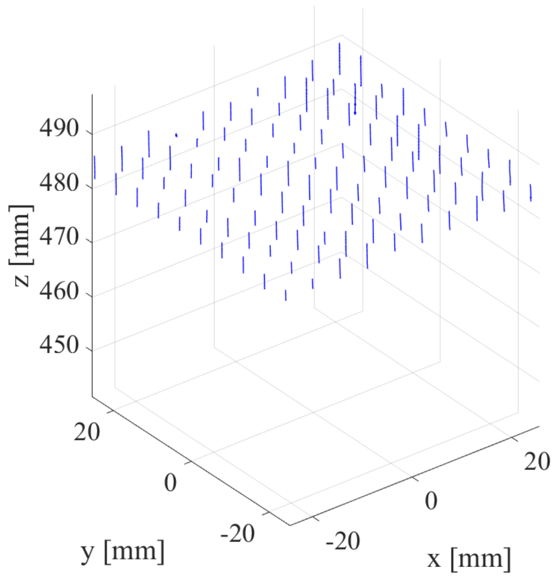
(a)



(b)



(c)



(d)

Figure 32 –Tracking results, showing vectors corresponding to 10 movements for each dot on the target. (a)-(b) show raw tracks for in-plane and out-of-plane motion, respectively. (c)-(d) show the same tracks after applying the smoothing algorithm.

5.4 Evaluation

To reinforce some of the visual observations that can be made regarding the plots of the particle tracks, histograms were plotted for the overall data set. The first pair, given in Figure 33 and Figure 34, show the histograms for sequential 0.5 mm in-plane *y*-directional movements of the traverse before and after smoothing. The next pair in Figure 35 and Figure 36 show the plots for sequential 0.5 mm out-of-plane *z*-directional movements of the traverse. Table 7 gives the median and 95% confidence intervals for the measured values. In all cases, outlier removal has been conducted using the generalized extreme Studentized deviate test [51] in order to remove any spurious data points generated by incorrect found point locations and subsequent track generation.

Table 7 - Medians and confidence intervals of in-plane displacements from fixed-field test for a 0.5 mm movement in the *y*-direction

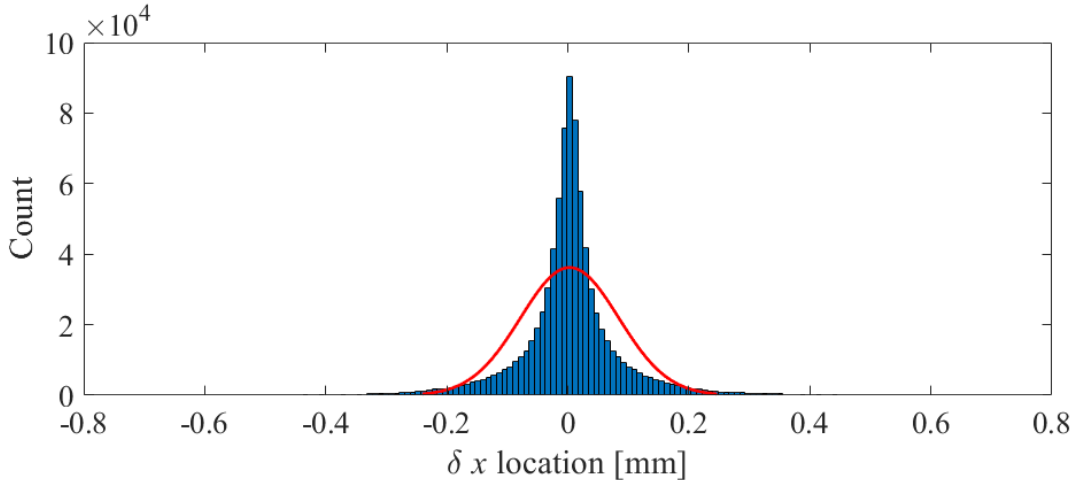
[mm]	Unsmoothed		Smoothed	
Direction	Median	95% Confidence	Median	95% Confidence
x	3.11e-3	(-0.181, 0.186)	3.31e-3	(-9.21e-3,1.60e-2)
y	0.464	(0.287, 0.640)	0.464	(0.450, 0.481)
z	1.93e-3	(-4.47,4.48)	1.00e-2	(-0.332, 0.354)

Table 7 allows for the first conclusions to be drawn on the effectiveness of the calibration. A displacement of 0.5 mm in the *y*-direction is expected based on the known motion of the traverse, but instead a median displacement of 0.464 mm is observed. It is most likely that this is an indication that the calibration method is imperfect, such that all in-plane distances have been under-predicted by the model. Another key observation is that the smoothing algorithm has reduced the confidence intervals by more than an order of magnitude in all directions. This is an indicator of the effectiveness of the smoothing approach at cleaning up the noise in the raw tracking results. Applying the smoothing method has allowed the straight tracks to be determined to within 0.017 mm in-plane (0.034% of the horizontal FOV), and to within 0.34 mm (0.68% of the horizontal FOV) out-of-plane. These statistics also quantify the disparity in the uncertainty of the technique in-plane compared to out-of-plane, with the out-of-plane uncertainty being roughly 20 times larger.

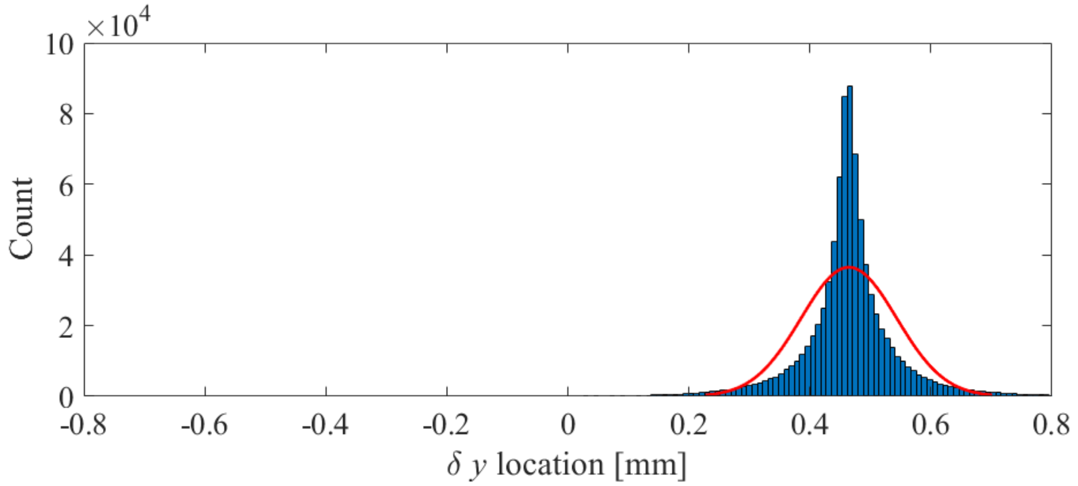
Table 8 shows similar results for the out-of-plane tracking when the calibration target is moved in the *z*-direction. Similar near order-of-magnitude reductions in uncertainty from the smoothing. However, these reductions are less substantial than for the in-plane case. The smoothing algorithm is thus expected to work best when at least some in-plane component of velocity is present. Again, the median *z*-displacement of 0.347 mm under-predicts the expected out-of-plane displacement of 0.5 mm. Together, these errors indicate that the calibration method has likely under-predicted the focus distance T_L.

Table 8 - Medians and confidence intervals of out-of-plane displacements from fixed-field test for a 0.5 mm movement in the *z*-direction

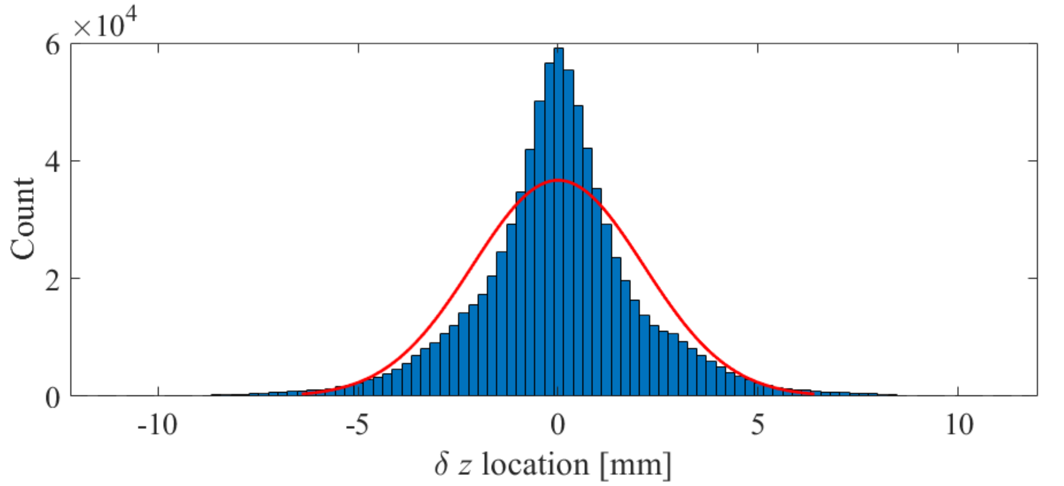
[mm]	Unsmoothed		Smoothed	
Direction	Median	Confidence	Median	Confidence
x	4.90e-3	(-0.132, 0.136)	4.74e-3	(-1.13e-2,2.17e-2)
y	-4.40e-3	(-0.137, 0.126)	-5.01e-3	(-2.03e-2,1.24e-2)
z	-0.281	(-4.28, 3.49)	-0.347	(-0.721,0.131)



(a)

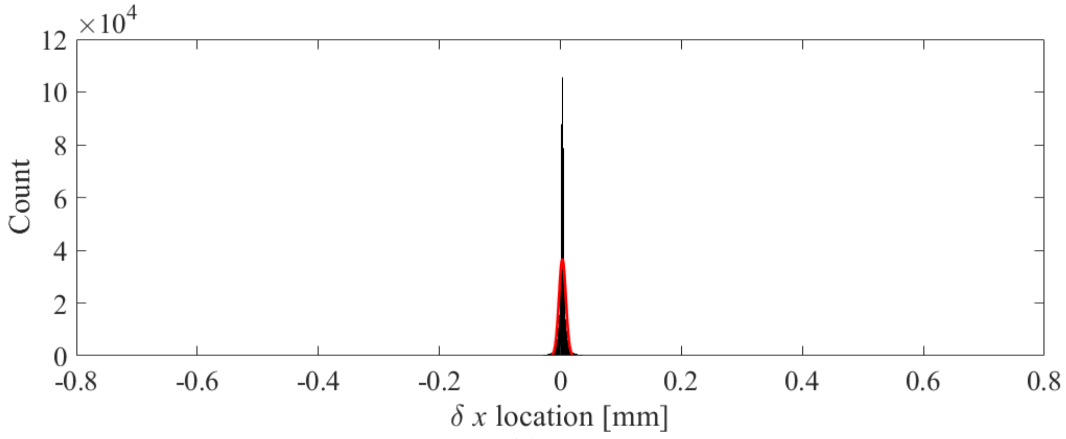


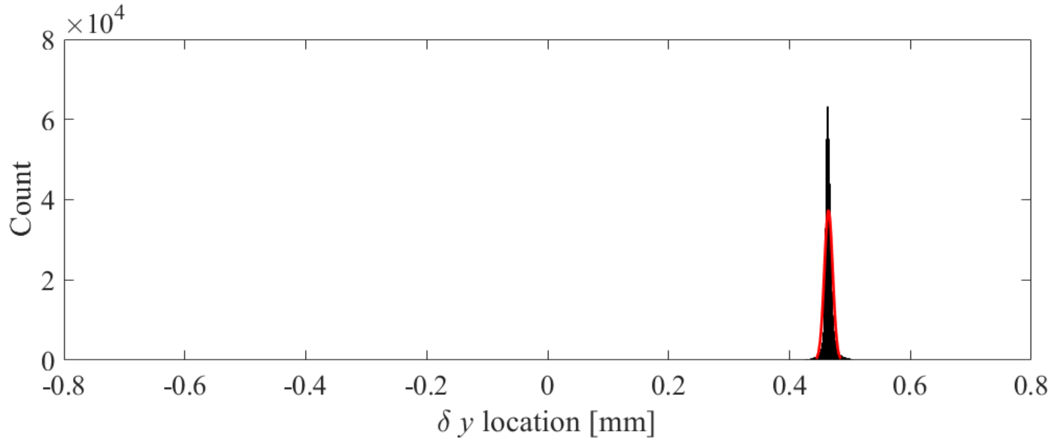
(b)



(c)

Figure 33 – Histograms of the raw measured velocities of the target dots for the in-plane tracking case. Note that *z* plot has been scaled differently for this case only, between this and the next figure. This is made necessary by the massive distribution in the *z*-movements.


(a)


(b)

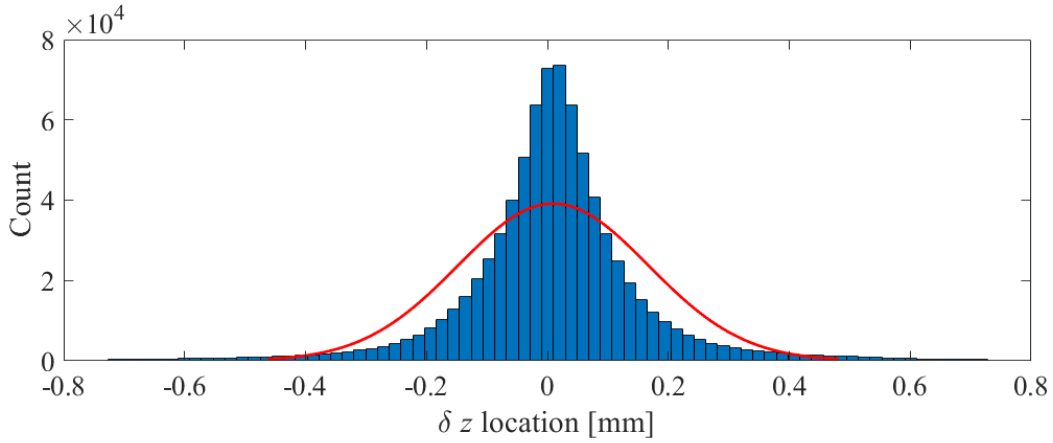
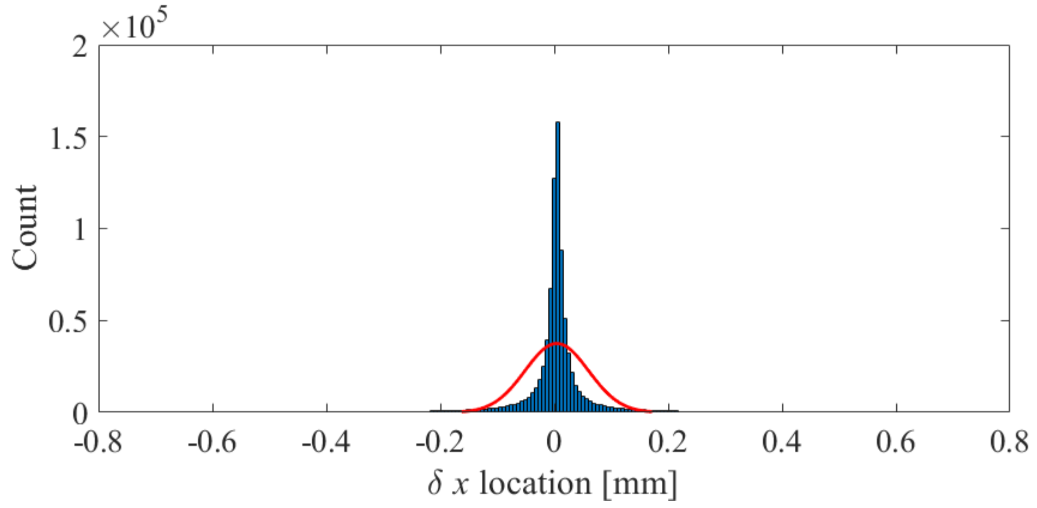
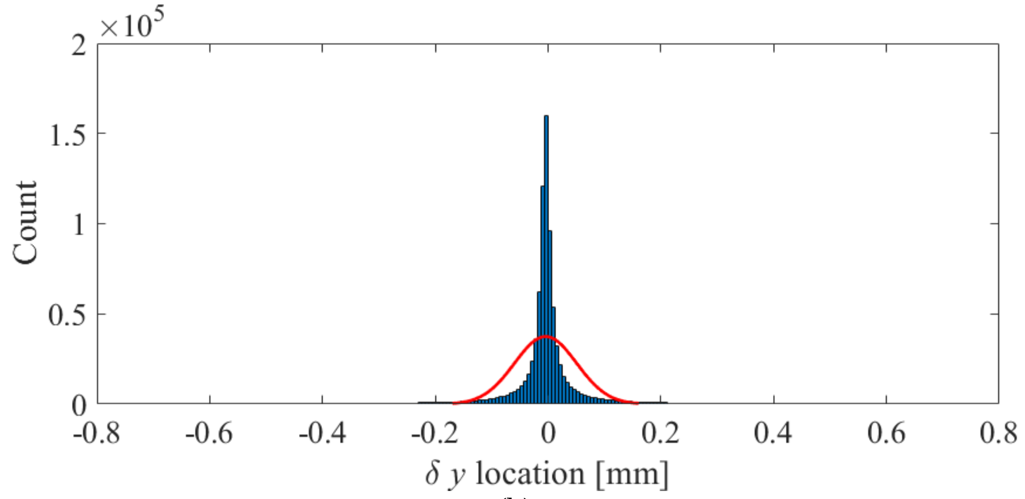

(c)

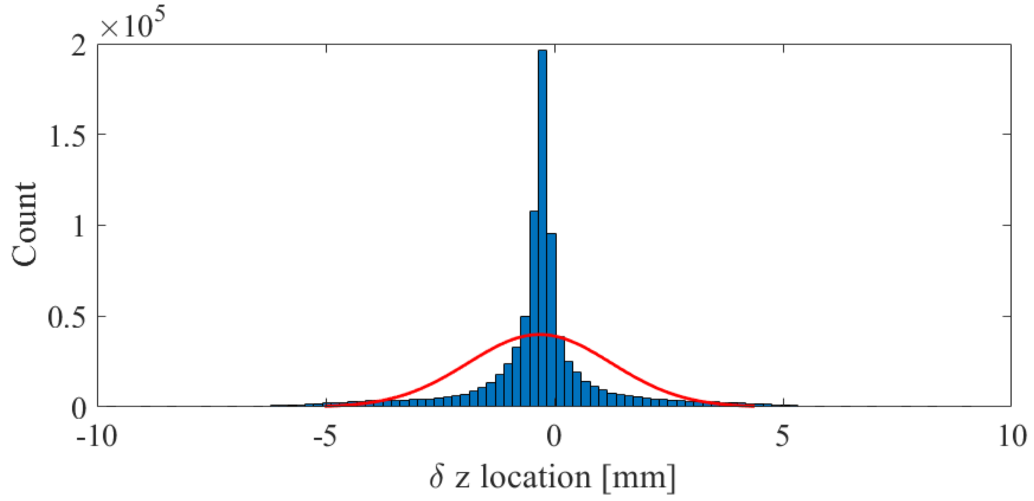
Figure 34 – Histograms of the smoothed measured velocities of the target dots for the in-plane tracking case. The *x*, *y*, and *z* movements are directly comparable with the *x* and *y* movements in the previous figure.



(a)

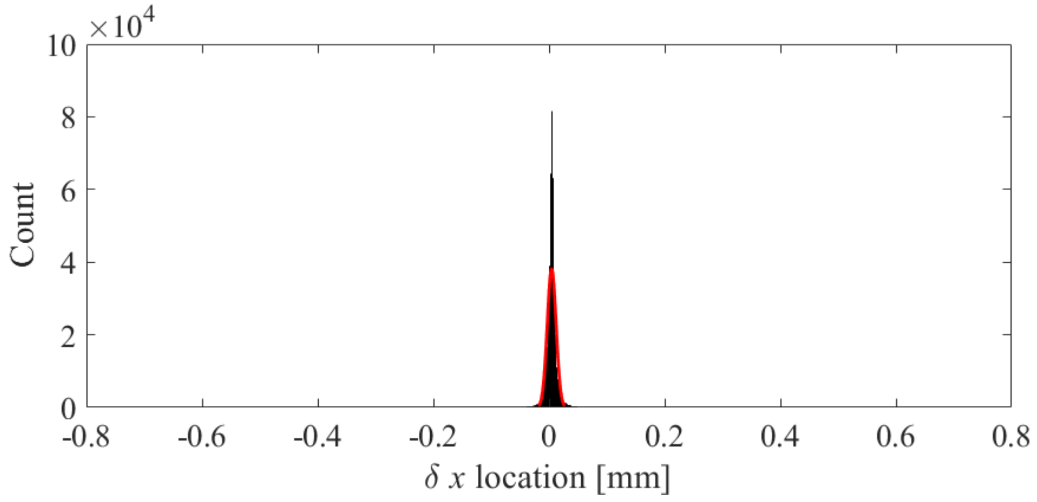


(b)

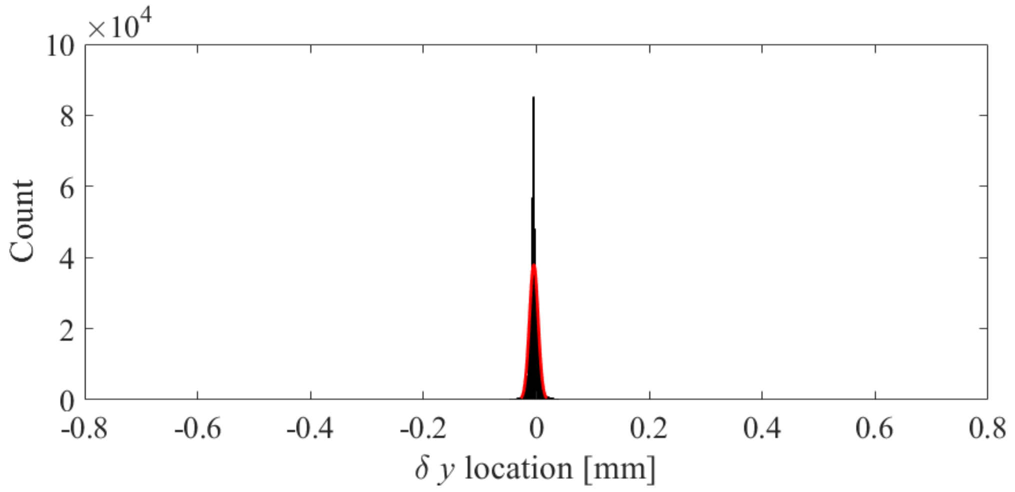


(c)

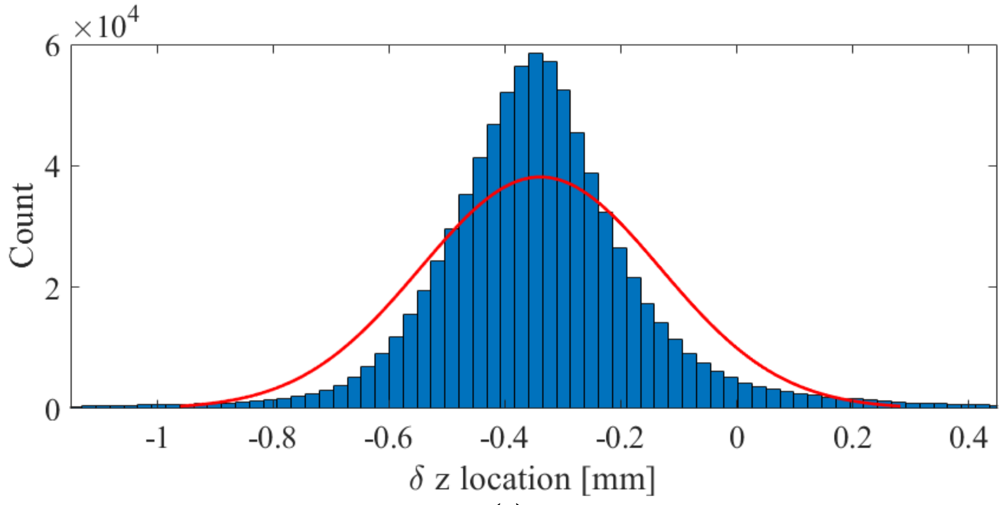
Figure 35 – Histograms of the raw measured velocities of the target dots for the in-plane tracking case. Note that *z* plot has been scaled differently for this case.



(a)



(b)



(c)

Figure 36 – Histograms of the smoothed measured velocities of the target dots for the in-plane tracking case. Note that the *z* plot is scaled the same, but has been shifted horizontally.

An immediately apparent feature of the plots is that the distributions are non-normal. This is made obvious by the red lines showing a normal distribution with the same mean and standard deviation of the data not sharing the same trend as the histogram bars. Instead, the data has in all cases a larger central peak and wider tails than a standard normal distribution. This type of data could be consistent with two overlapping standard normal distributions, one with a lower standard deviation to create the large central peak, and one with a high standard deviation to create the wide tails. This indicates that there is some statistically significant portion of the data for which the technique is performing poorly in comparison to the rest of the data. Another observation that can be made is that the uncertainty is substantially reduced by the smoothing method.

To determine the portion of the data for which the technique is performing poorly, 90% confidence intervals were generated for each of the individual data sets of the in-plane tracking case. These 90% intervals were selected as opposed to the 95% intervals used in all other cases in this document to reduce any bias generated by the outliers that are not eliminated when using this method. These intervals are shown in Figure 37 and Figure 38 as a function of the *z*-distance the traverse moved from its starting point. Figure 37 features the data points all plotted with similar *y*-axes, for direct comparison, while Figure 38 shows each set of points on its own axis to allow for comparison within each set. One set of three data points is plotted in each of these plots for each of the 101 data sets that were collected at decreasing distances to the camera and therefore increasing virtual depths. The red dots represent a median displacement, while the blue dots represent the upper and lower bounds for a 90% confidence interval for the displacement. As each in-plane data set was collected at a different *z*-location of the traverse, plotting in this way allows the displacements and their associated uncertainties to be compared as a function of the average depth of each data set.

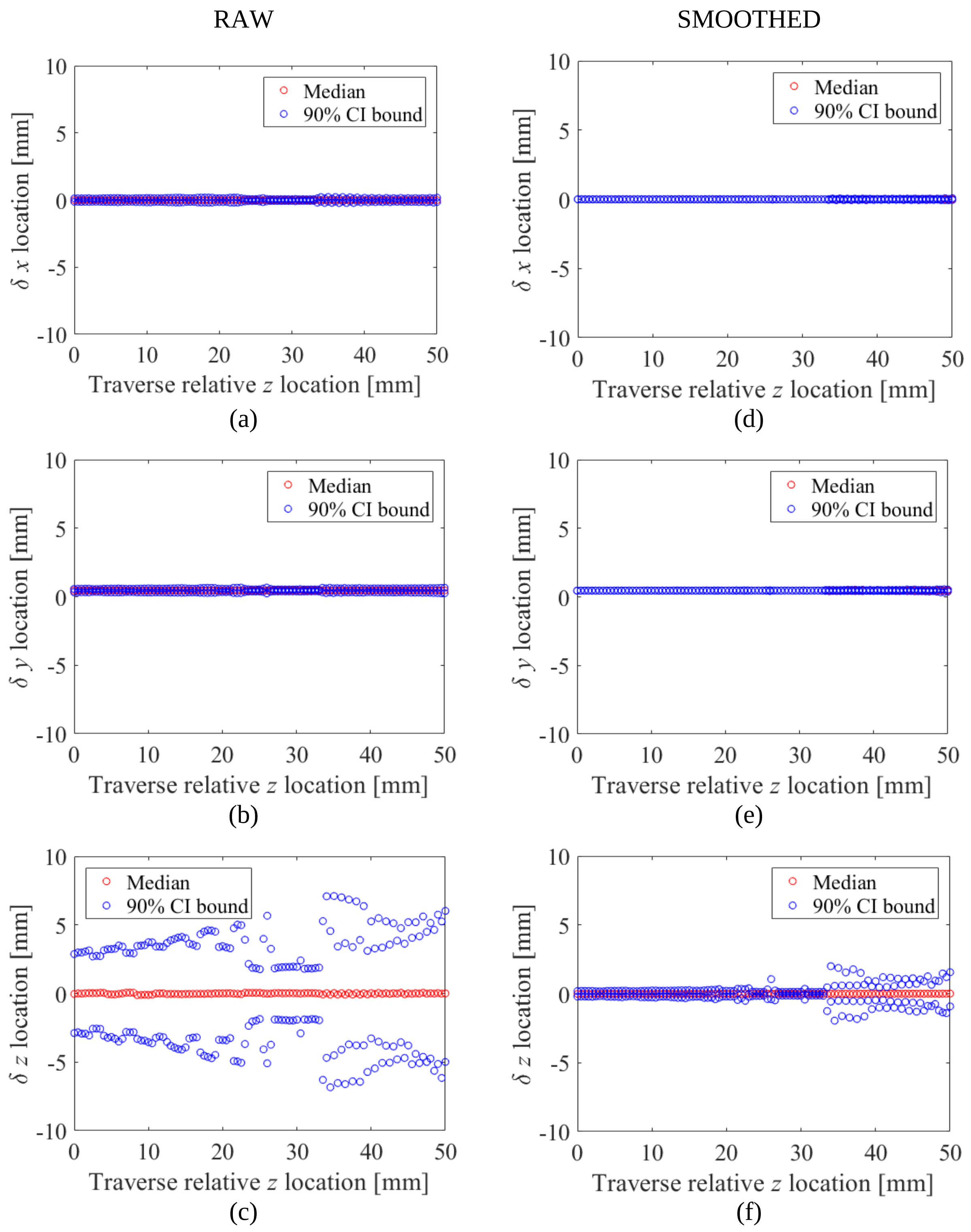
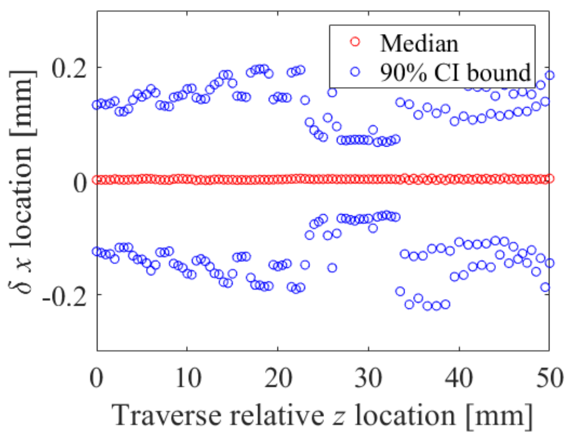
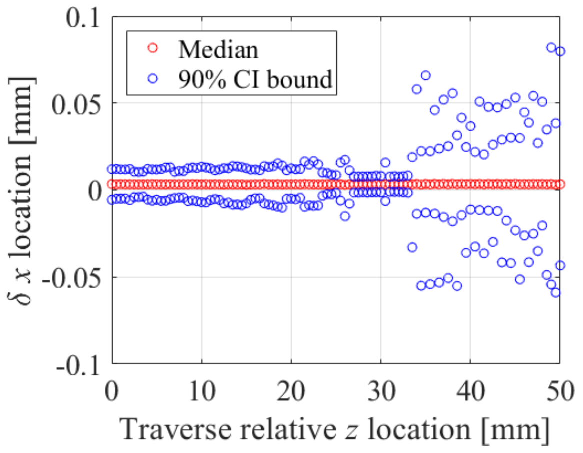


Figure 37 – Plots of the raw (a-c) and smoothed (d-f) measured displacements as a function of traverse position for the in-plane case. *y*-axes are scaled the same.

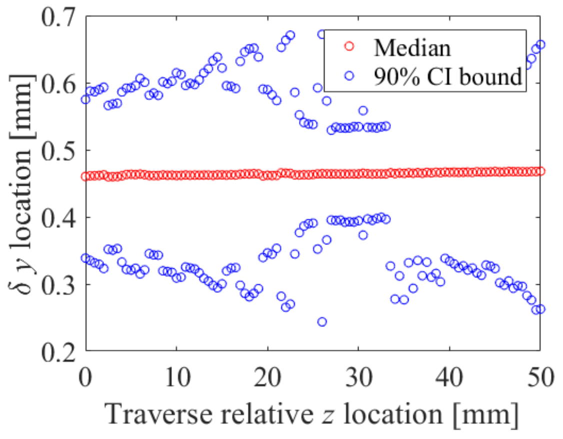
RAW SMOOTHED



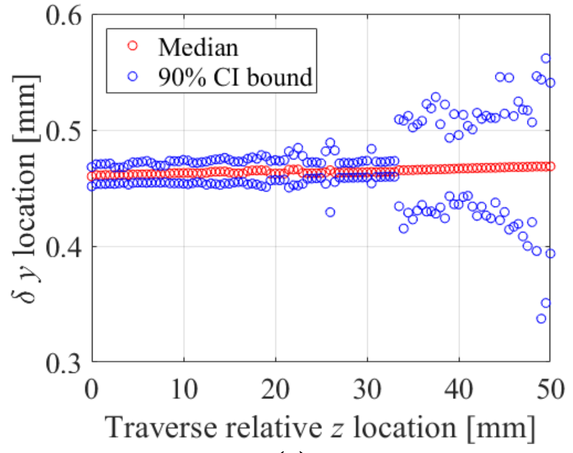
(a)



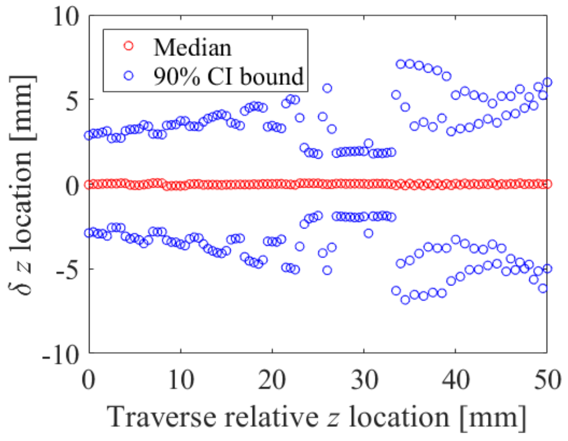
(d)



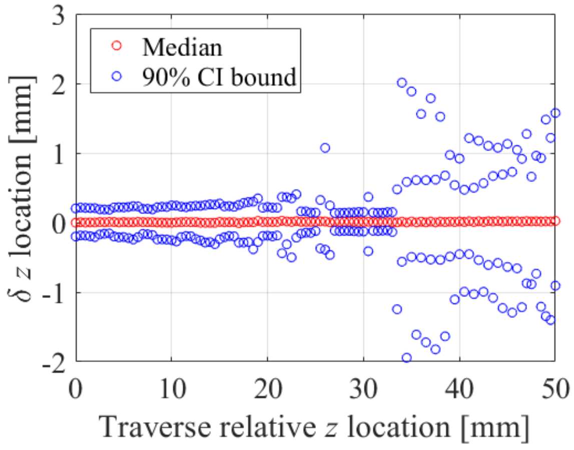
(b)



(e)



(c)



(f)

Figure 38 – Plots of the raw (a-c) and smoothed (d-f) measured displacements as a function of traverse position for the in-plane case. *y*-axes have been scaled differently to show uncertainty.

The main observation to be made here is that there is a large increase in the uncertainty of this method after the target reaches a certain *z*-location, ~35mm from its starting point. This is consistent with the reduction in resolution after a certain virtual depth discussed in [27]. Another interesting point is that the median *y*-displacements are a weak function of the traverse position. This is an indication that the calibration is imperfect, which warrants further investigation.

To investigate further, the results of this method are considered from purely a particle- location standpoint. The median *z*-locations of each data set within the plane were taken and plotted against the depth-steps of the traverse in Figure 39. Theoretically, this relationship should be exactly 1:1 linear, forming a 45° angle with the *x* and *y* axes if the calibration method has worked perfectly. This is not the case as the actual linear trend drops more steeply. This indicates that the calibration method has under-predicted the focus distance T_L. This is consistent with the earlier observation that the median measured in-plane displacements were smaller than the expected in-plane displacements, and the weak function of *z*-position exhibited by the *y*-displacements.

The volatility of the depths in Figure 39 between about 16 and 26 mm on the *x*-axis is likely caused by the intensity-selection algorithm from Section 3.5 having a difficult time of deciding which lens is the correct lens to use, compounded with the B_i values from the calibration not causing the virtual depths to re-project properly. This produces three different linear behaviors, all interlaced in the same plot. It is possible that a better calibration could align these by predicting the B_i values correctly. The poorly predicted B_i values are directly related to the under-prediction of T_L observed previously. This is consistent with the three different linear progressions observed between traverse positions 0-16, 27-33, and 34-50. The large increase in variance observed in Figure 38 is consistent with the transition to the progression of the 34-50 range in Figure 39. The calculated median depth at the start of this range is 464.6 mm, which corresponds to a virtual depth between 9.35 and 9.64, depending on which lens type is used.

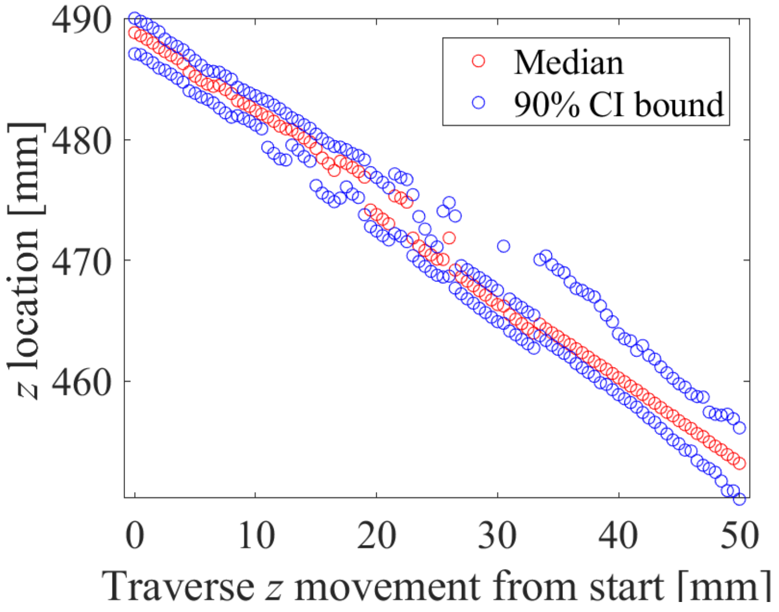


Figure 39 – Median and confidence intervals of *z*-locations of the calibration plate as a function of the known relative depth locations of the plate.

The main success of this section has been the smoothing method implemented, which managed to reduce the uncertainty in the displacements of the tracked particles by more than an order of magnitude. Other results indicate that the ETC method from Sections 3.3 and 3.5 produces higher variances in all particle location data after the targeted particles pass beyond a virtual depth of 9.35. Care should be taken to avoid exceeding this limit in future experiments when using the R5 *f*/# 2.4 camera. A key point to note here is that this can be algorithmically enforced by disallowing particles that are found beyond a certain virtual depth within the location step. The other conclusion that has been reached is that the calibration method has not been able to perfectly predict the optical system's parameters, mostly related to an under-prediction of T_L. It is possible that this is also related to the high uncertainties associated with finding the depth-locations of individual points on the target, which is likely caused by uncertainties in the 2D particle location method. This experiment was not undertaken for the refocusing method discussed in Section 3.2, as preliminary investigations [52], [53] had indicated that this technique had a high uncertainty. In retrospect, this uncertainty may be comparable to that observed with the ETC technique. Returning to the refocusing technique and repeating this experiment to allow for a direct comparison is a recommended follow-up to this investigation.

CHAPTER 6 APPLICATION OF THE ETC METHOD: THE RISING OIL DROP EXPERIMENT

In order to test the approach in a scenario in which a typical tomographic imaging setup would be unviable, a micro-scale flow scenario was desired. The flow of oil droplets through micro-scale slots is of particular interest to the sponsoring company (RGL Reservoir Management) and has been under study by other researchers using two-dimensional techniques. Expanding the knowledge of these flows to include three-dimensional flow patterns may be useful, as the surrounding fluid is capable of flowing in front of or behind the droplet relative to the camera. An experiment suited to the plenoptic camera was thus developed for study.

6.1 Experimental setup and data acquisition

The rising oil drop experiment is conducted by filling an acrylic flow channel with glycerol and then introducing a droplet of vegetable oil into the bottom of the channel using a glass syringe, as shown in Figure 40. The oil droplet, being less dense than the glycerol, will then float up through the channel. These two fluids have similar refractive indices, so the refraction at the interface between the two is minimal. The flow section observed by the camera, shown in Figure 41, has a 6×6 mm cross section. The camera used in this case is the *f/#* 26 (R5, Raytrix GmbH) with a 105 mm main lens. A sample image obtained as the droplet passed in front of the camera is shown in Figure 42. Due to the collection method, which precluded the use of commercial software (RxLive, Raytrix GmbH) the data collected is specific to the ETC method. Illumination is achieved through forward-scattering a diffuse LED light source (H528S, Amaran) of 40-micron particles (Dynoseeds TS 40, Microbeads®) uniformly distributed in the two fluids. Here, the individual particle-images are visible and a circle has been overlaid to show the oil droplet. The edges of the oil droplet are otherwise indistinguishable due to the refractive index matching of the two fluids. This also eliminates refraction through the droplet, allowing the inner flow structures of the droplet to be seen. In this case, due to the low velocities and high viscosities involved, little to no motion within the droplet is expected within the imaged range of motion.

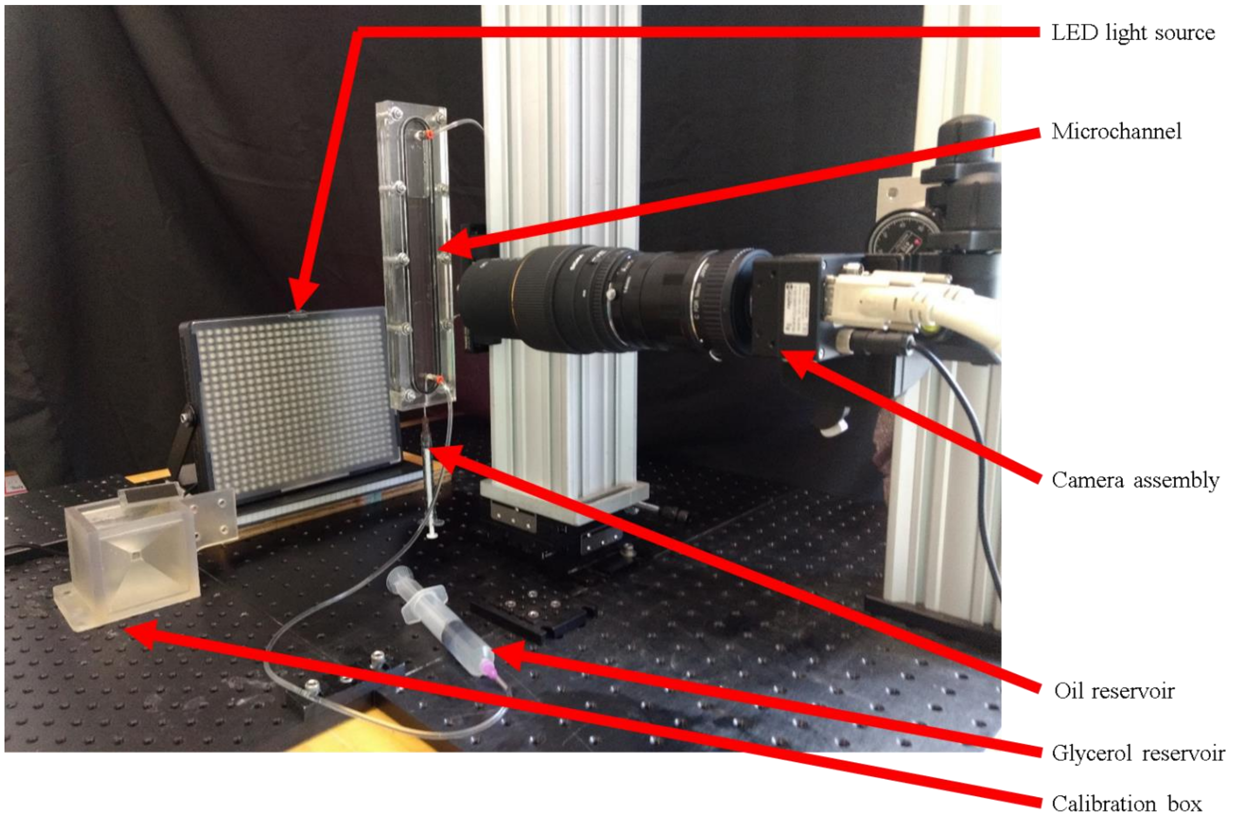


Figure 40 – Experimental setup of rising oil drop experiment.

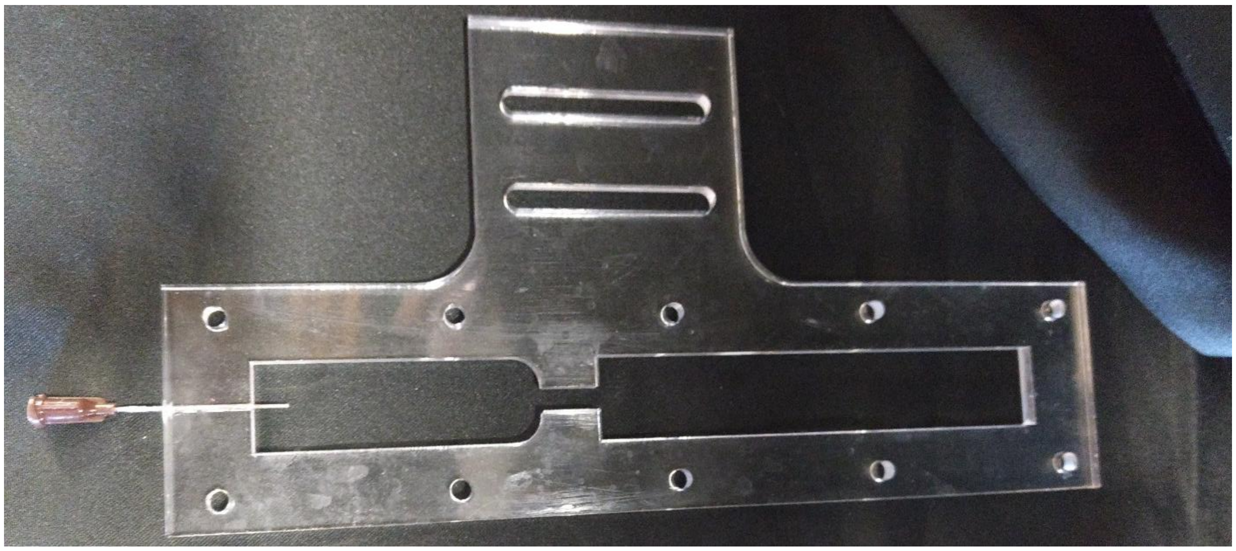


Figure 41 – Flow channel used for rising droplet experiment.

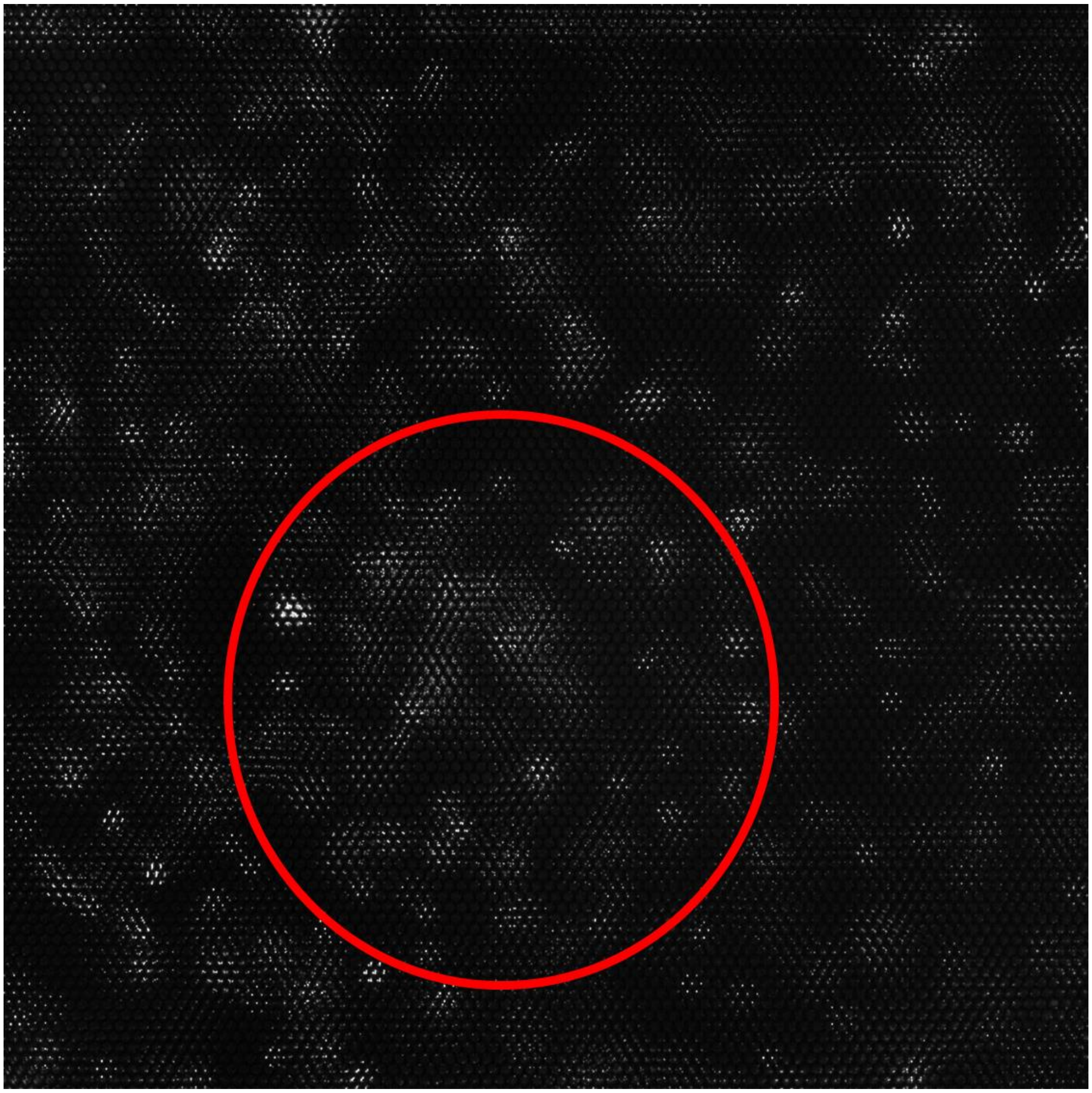


Figure 42 – Raw sample image from the oil drop experiment. Oil droplet is given by the red circle. Refractive index matching makes it all but indistinguishable from the surrounding fluid.

6.2 Calibration

The micro-lens location approach discussed in Section 3.3.1 was applied first. Taking a histogram of the residuals resultant from the least-squares determination of the micro-lens location vectors yields Figure 43. Percentile-based statistical analysis indicates that 95% of the data exhibit a residual of less than 0.38 pixels. The resultant vectors that define the positions of neighboring same-type lenses in this case are (0.011, 40.73) and (35.26, 20.35). These are similar to the results from the other camera in Chapters 5 and 6, with slight differences that can be attributed to the differences in the micro-lens locations between the two plenoptic cameras and the quality of the obtained white images.

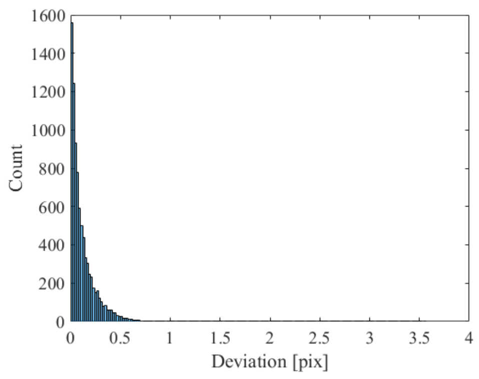
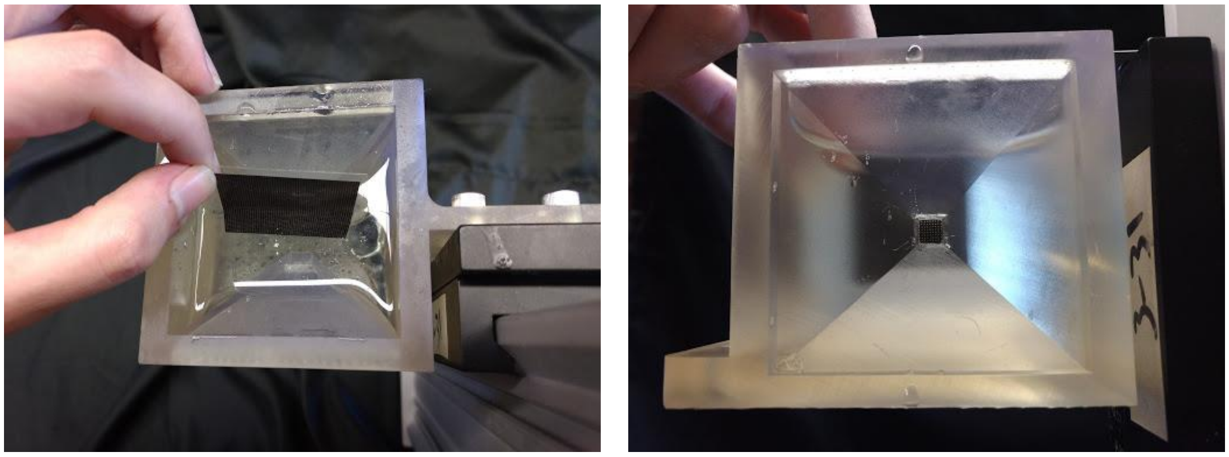


Figure 43 - Histogram of the residuals of the 2D vector norm of the difference between the measured micro-lens locations and the gridded locations for the rising oil drop experiment.

To perform the metric calibration, a target was created from a 5 cm square of paper upon which a layer of black ink was laid down using a laser printer (M401dne, HP). A laser cutter (VLS3.60, VersaLaser®) was then used to remove the ink to create dots as small as the focal point of the laser beam, separated by 0.677 mm. This target was laminated and glued to a piece of acrylic. To place the target within the imaging volume, a 'calibration box' (see Figure 44) was designed with the same mounting configuration and front window position as the flow channel, but with ample space to fit the target. This box is designed to have a 6x6 mm frontal acrylic window with the same optical properties as the flow channel, and a mount identical to the flow channel to allow for simple positioning. The target can be placed inside at any angle up to 30° relative to the camera. The calibration box was filled with glycerol to ensure it had the same optical properties as the flow channel, and calibration images were obtained. The process described in section 3.4 was implemented using these calibration images.



(a) (b)

Figure 44 – Use of the calibration box. A 6mm deep imaging section behind a small 6x6 mm piece of acrylic was created by opposing tetrahedral walls (a), allowing ample space for the calibration target to be placed. Imaging is then conducted through the window (b).

The resultant calibration variables are given in Table 9. Figure 45 shows the detected points in red overlaid with the predictive calibration points in blue. The target in (a-b) was parallel to the imaging field of the camera, while the target in (c-d) was at an angle relative to the imaging field of the camera. The *x* and *y* positions are observed to have 95% of the residuals in (-0.040, 0.043) and (-0.043, 0.044) mm, respectively. Contrary to the previous macroscale calibration, the *z* residuals are similar, with 95% of the residuals in (-0.043, 0.038) mm. Relative to the overall field of view, the in-plane residuals of this calibration are ~3 times as large as the in-plane residuals for the flat-field experiment, but the out-of-plane residuals are comparatively about half as large. This is an effect of the virtual depth range being less spread out in this case than in the previous case; the overall field of view here is 6×6×2 mm, a 3:1 ratio of in-plane to out-of-plane range compared to the 1:1 ratio in the previous experiment. The full depth of the channel could not be imaged, but the resolution within the available range is higher. Histograms with overlaid normal distributions for the residuals of the calibration points are given in Figure 46.

Table 9 - Calibration variable results for micro-scale tests

T_L	485.69 mm	k_1	5.94e-5
B_1	2.02 mm	k_2	-1.72e-6
B_2	2.00 mm	d_1	1.60e-4
B_3	1.98 mm	d_2	5.64e-5
x_0	-1.15e-3 mm	d_D	1.82e-3
y_0	5.09e-4 mm		

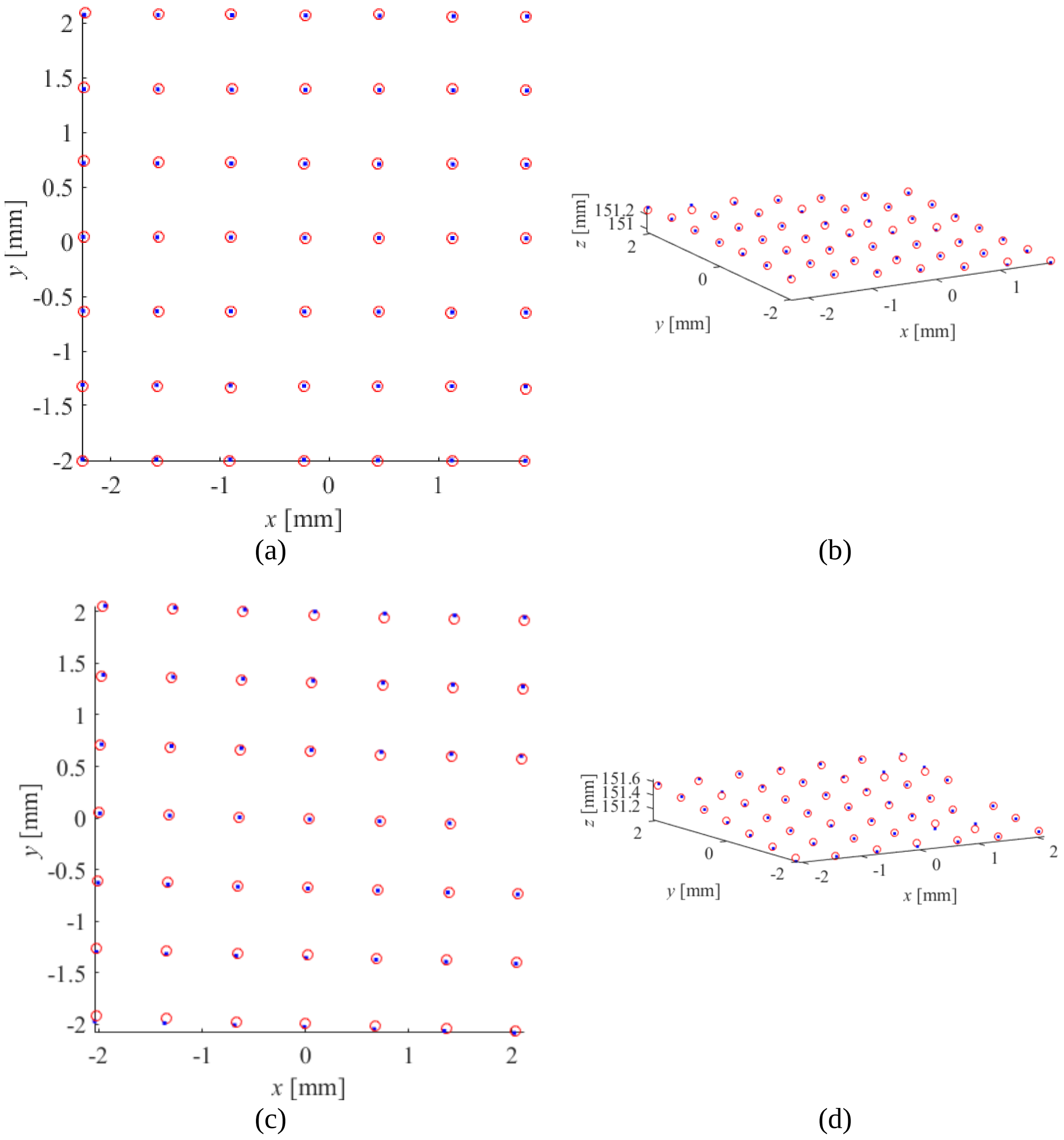
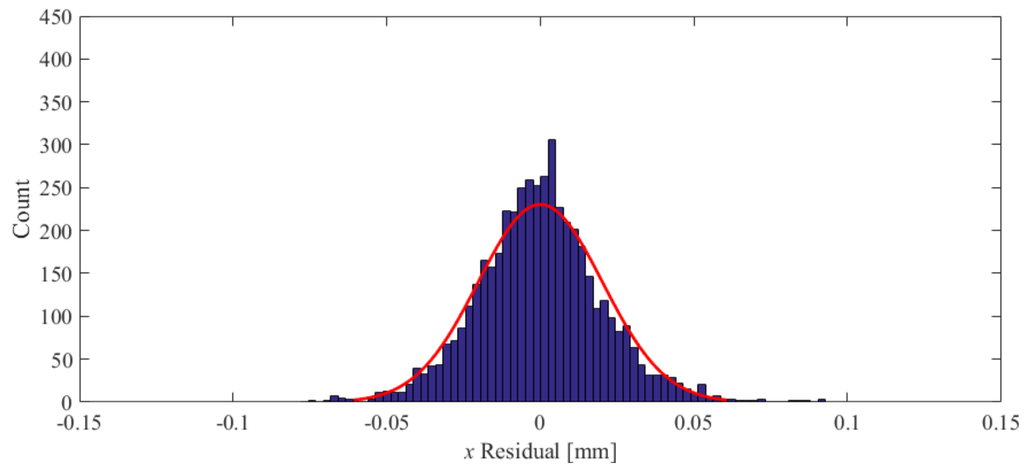
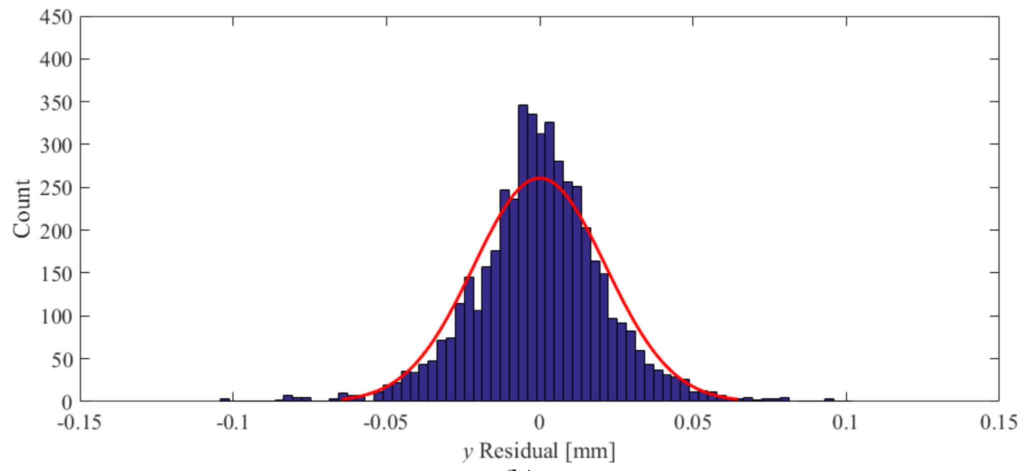


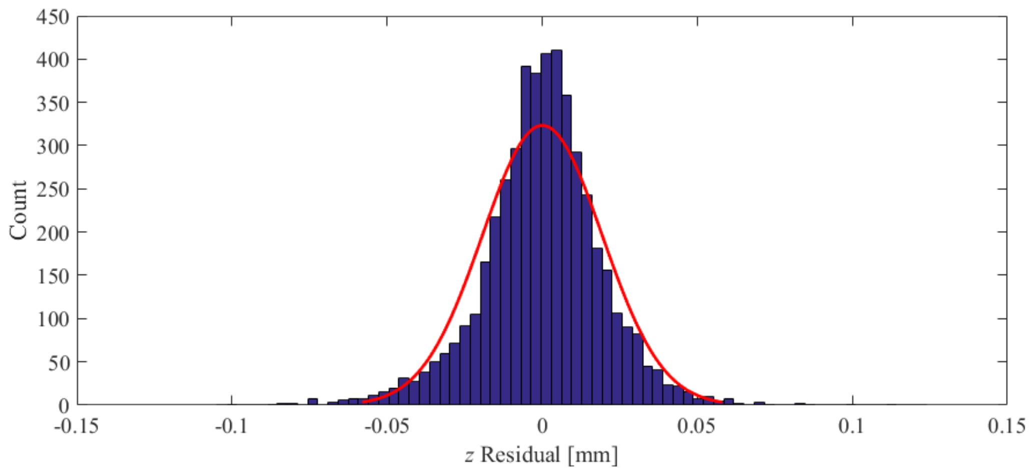
Figure 45 – Detected calibration points for the micro-scale calibration in red overlaid on blue grid of predicted points. In (a-b), the calibration target is aligned with the camera while in (c-d) the target is angled relative to the camera. (a) and (c) are in-plane views of the points, and (b) and (d) are positioned to show the angle of the target.



(a)



(b)



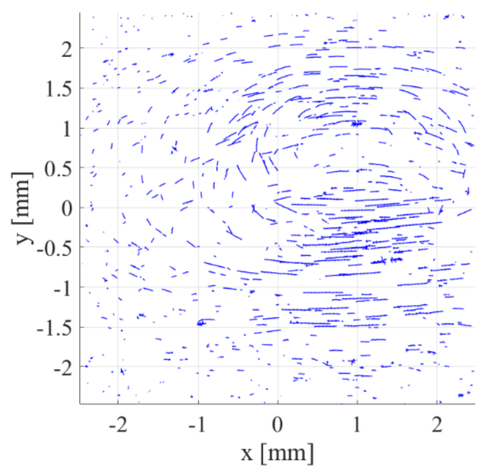
(c)

Figure 46 - Histograms of microscale calibration residuals in (a) *x*, (b) *y*, and (c) *z* directions, overlaid with normal distributions.

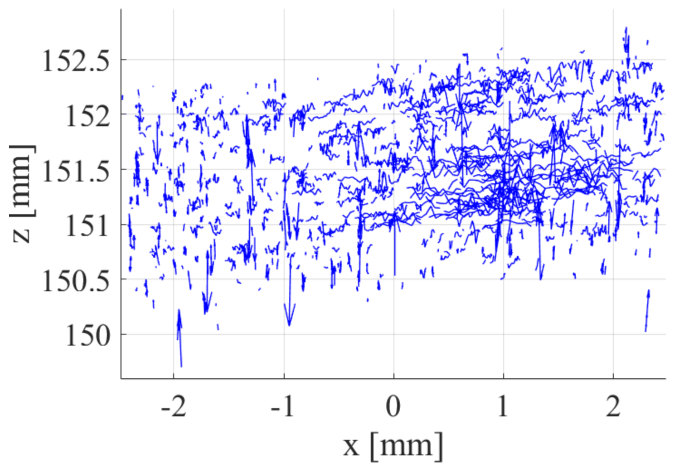
6.3 Data Processing

The ETC approach from Sections 3.3 and 3.5 was applied to the collected data. The resultant particle tracks are shown in Figure 47 and Figure 48. In this series of figures the oil droplet has the largest vectors, while the glycerol flowing around the oil can be seen to recirculate around the droplet as it rises, moving from right to left (gravity in +*x* direction). Walls are present just outside of the field of view, and the droplet is closer to the –*y* side of the channel leading to lower velocities in that region. In the *z* direction, only about the back 1/3 of the droplet is visible as the depth-of-field does not cover the whole 6mm depth of the channel. The recirculation regions behind and to either side of the droplet are most clearly visible in the smoothed-vector images. The vectors in these images can be difficult to make sense of in stationary images. An alternate method of display which cycles through groups of frames to show the motion of all particles in three dimensions is preferred.

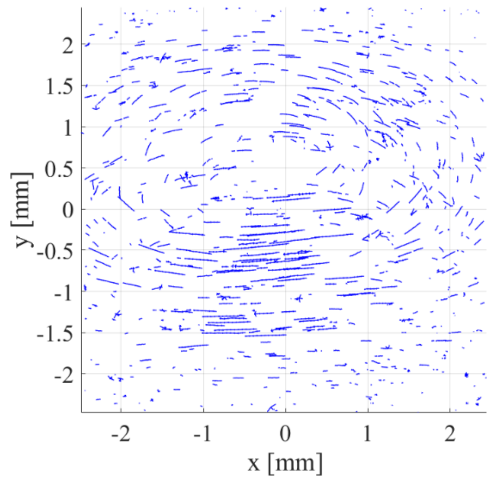
Observing Figure 47, it is clear that a significant level of noise is present in the measured out-of-plane motion, producing spurious vectors and making longer tracks vary randomly in the *z*-direction. This is consistent with the results from Chapter 5. Again, these random variations are a direct result of the level of uncertainty in the determination of the *z*-locations of the particles. This is likely to stem from the 2D particle locations in the micro-lenses. The smoothing method, as applied in Figure 48, greatly reduces the noise but the tracks are still affected. Both figures make it apparent that the in-plane performance of the technique is generally strong and able to reconstruct the expected motion of the flow field to produce coherent particle tracks. The recirculation regions to either side of the droplet are reasonably reconstructed in-plane, although the out-of-plane motion is only partially captured.



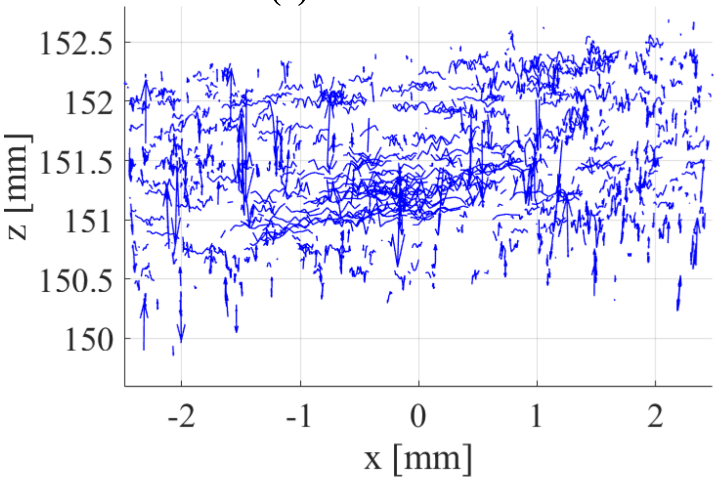
(a): frame 70



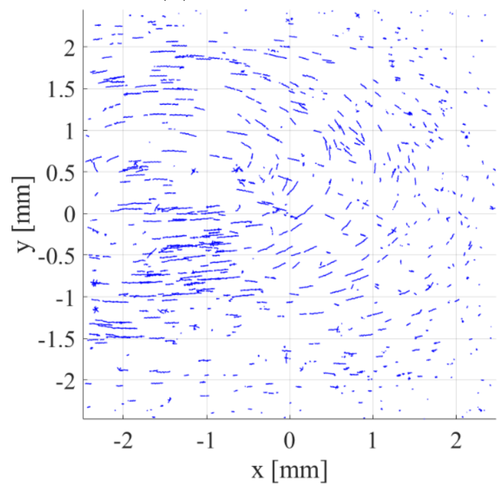
(d) : frame 70



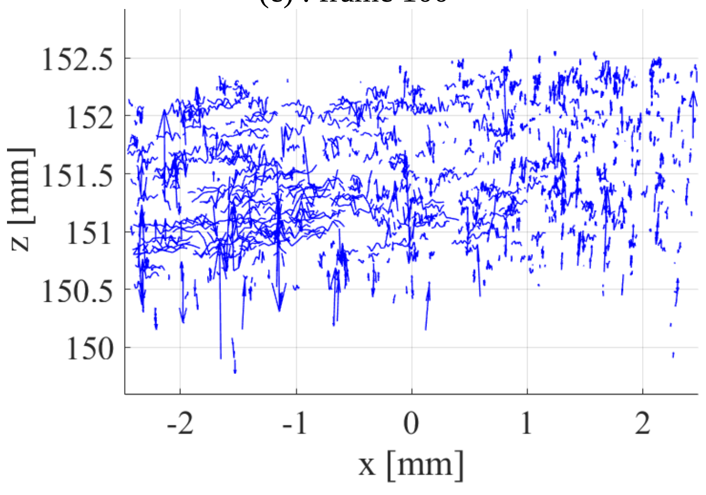
(b) : frame 100



(e) : frame 100

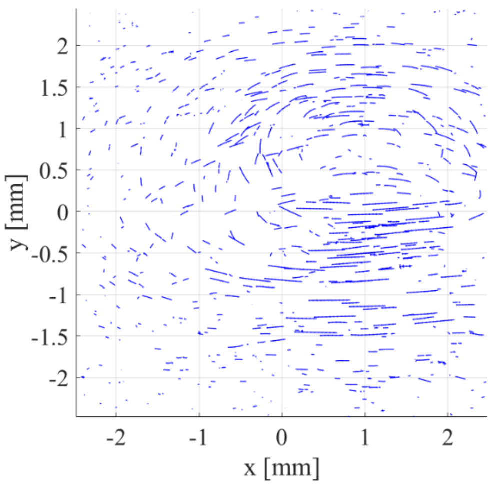


(c) : frame 130

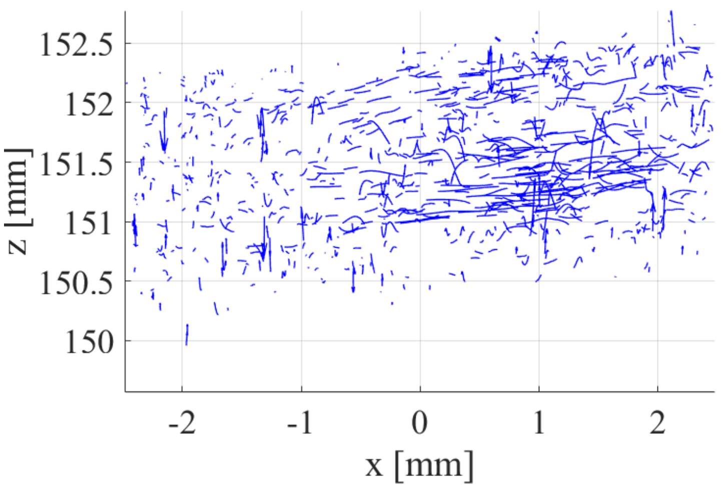


(f) : frame 130

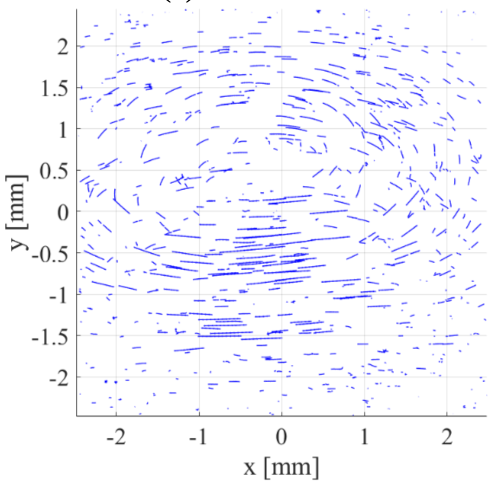
Figure 47 – Raw velocity vectors resultant from the rising droplet test. (a-c) show the vectors viewed in the x-y plane, and (d-f) show the vectors viewed from the x-z plane.



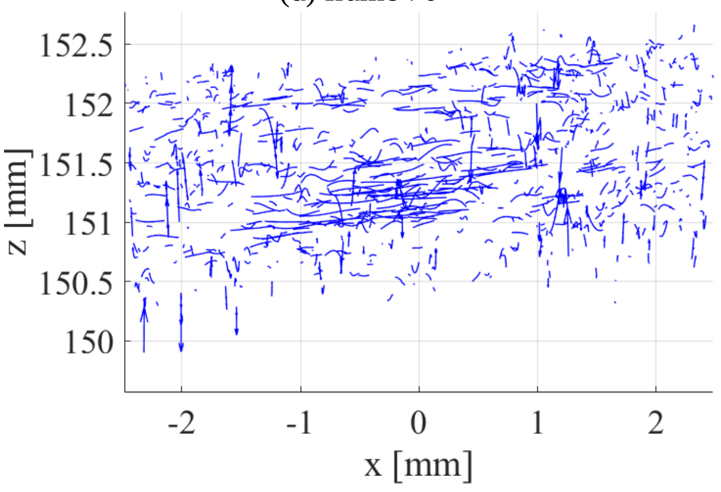
(a): frame 70



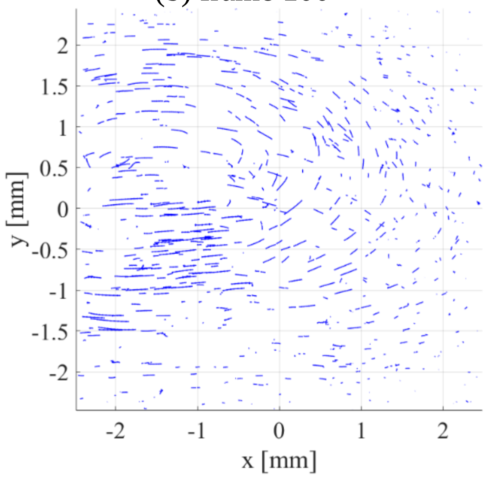
(d) frame 70



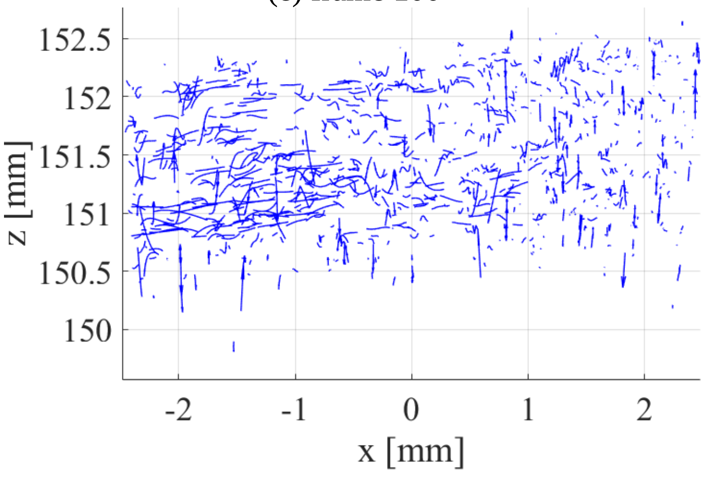
(b) frame 100



(e) frame 100



(c) frame 130



(f) frame 130

Figure 48 – Smoothed velocity vectors resultant from the rising droplet test. (a-c) show the vectors viewed in the *x-y* plane, and (d-f) show the vectors viewed from the *x-z* plane.

6.4 Conclusion

Based on the results of the oil droplet experiment, the ETC method is capable of performing in-plane tracking reasonably well at the microscale. The high level of uncertainty in the depth-location algorithm has affected performance in the out-of-plane direction, producing spurious tracks that are mostly, but not entirely, removed by the smoothing method. The uncertainty in the depth locations found by this method is likely produced through amplification of the uncertainty in the 2D location method by the triangulation approach. This indicates that the error filtering discussed in section 3.3.6 was insufficient in removing incorrect 2D centroids, or that the 2D centroid method has too much associated uncertainty to allow for the triangulation method to be effective. The present hypothesis is that the edges of the micro-lenses partially cut off the particle-images, shifting the 2D centroids in a consistent way that the error filtering method ignores. Improving the 2D particle location method may be necessary to improve the ETC approach.

CHAPTER 7 COMPARISION OF THE REFOCUS AND ETC METHODS: THE VORTEX EXPERIMENT

To investigate and compare the performance of both the ETC technique and the refocusing method, a reasonably well-known, simple three-dimensional fluid flow was desired. Ring vortices, also known as toroidal vortices, were selected for their simplicity and three-dimensional nature with reasonable axial symmetry. A full discussion of the formation and behavior of toroidal vortices can be found in [54]. The expected flow in a ring vortex is a central core surrounded by a vortex that is axially symmetric about the core.

7.1 Experimental setup and data acquisition

The experimental setup used for the vortex experiment is shown in Figure 49. The *f/#* 2.4 (R5, Raytrix GmbH) camera was used with an 85 mm main lens to perform a fluid flow experiment involving the imaging of a simple toroidal vortex. The vortex was generated by forcing fluid from a 25mm syringe, cut off to act as a simple piston-cylinder periodically driving fluid into a stationary reservoir. The motion of the syringe was controlled using a slider-crank assembly connected to a computer controlled stepper motor (CPM-MCVC-34415-RN, Teknic Inc.). The fluid was seeded with 0.5-0.6 mm polystyrene spheres. Pure sodium chloride was added to make the particles neutrally buoyant. Illumination was achieved by using a laser (LRS-0532-PF, Laserglow Tech.) which was diffused into a volume-illuminating beam using a series of spherical lenses and mirrors. The camera lens was manipulated to achieve a roughly cubic 60×60×60 mm imaging volume. A sample raw image collected using this setup is shown in Figure 50. A key point to note here is that, compared to the previous study in Chapter 6, the particle-images here are significantly larger (~twice the radius).

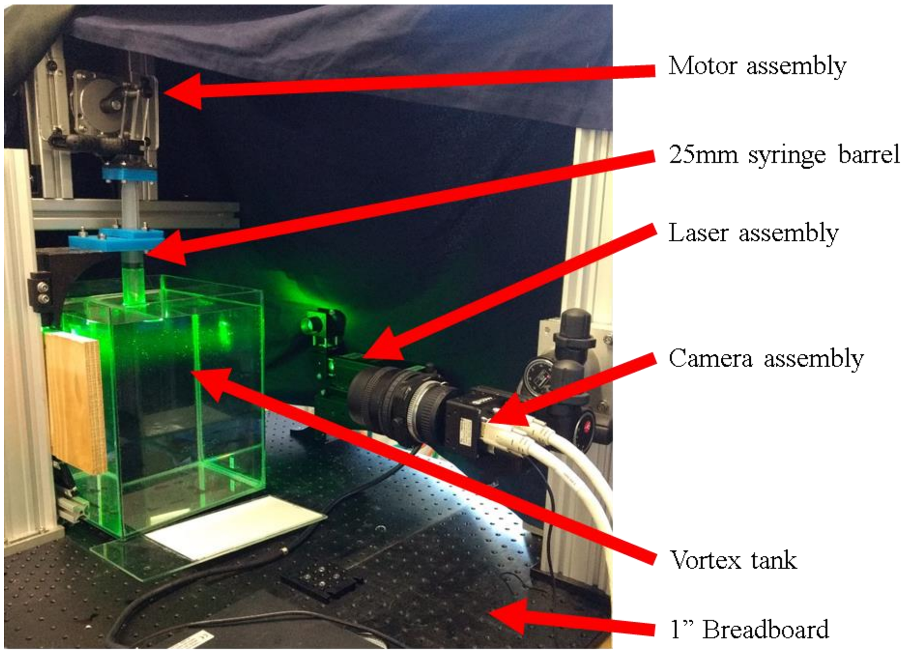


Figure 49 – Experimental setup of vortex generator.

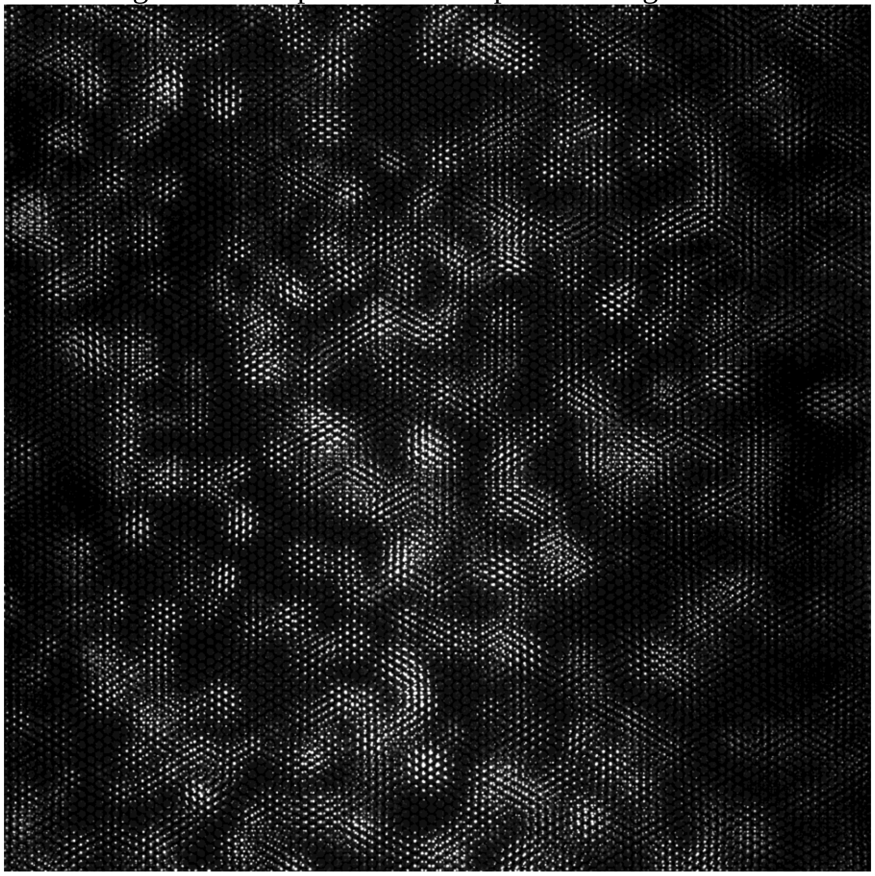


Figure 50 – Partial raw sample image from the vortex tank experiment

7.2 Calibration

The micro-lens location approach discussed in Section 3.3.1 was applied first. Taking a histogram of the residuals resultant from the least-squares determination of the micro-lens location vectors yields Figure 51. Percentile-based statistical analysis indicates that 95% of the data exhibit a residual of less than 0.145 pixels. The resultant vectors that define the positions of neighboring same-type lenses in this case are (40.28, -0.003) and (20.14, 34.88). Note that the vectors are identical to the test from Chapter 5, which is as expected because this is the same camera as in that test.

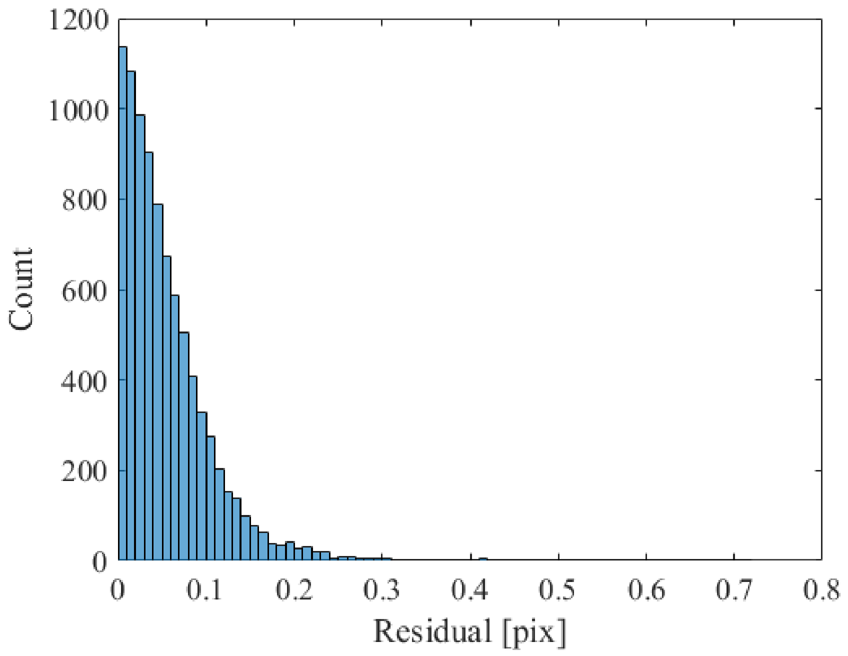
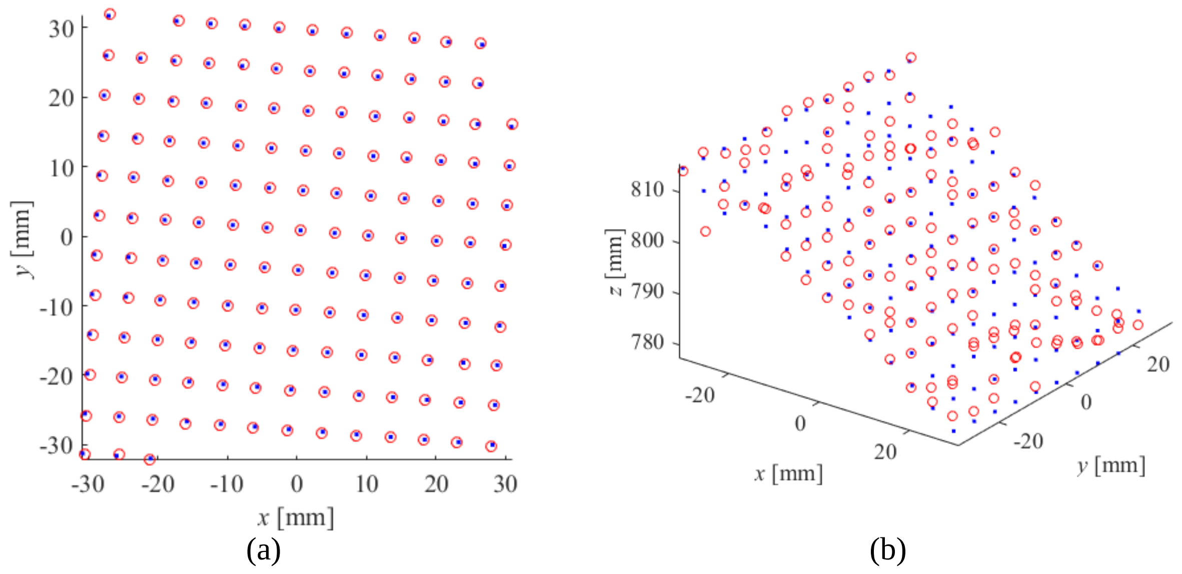


Figure 51 - Histogram of the residuals of the 2D vector norm of the difference between the measured micro-lens locations and the gridded locations for the vortex tank test.

The calibration from section 3.4 in this case was performed within the water tank prior to seeding the water with particles. This was to preserve the refractive behavior of the glass-water interface such that it would be captured by the calibration. The resultant calibration variables for the ETC approach are given in Table 10. Figure 52 shows the detected points in red overlaid with the predictive calibration points in blue. Here, Figure 52(a-b) are the resultant images for one position of the calibration target, viewed in the *x*-*y* plane and from a viewpoint which shows the angle at which the target has been placed. (c-d) are similar, but for an alternate target position. Excellent agreement between the *x* and *y* positions is observed with 95% of the residuals in (-0.27, 0.26) and (-0.26, 0.25) mm, respectively. These are marginally worse than in the first test, likely due to the air-glass and glass-water interface of the tank not being fully accounted for by the calibration method. The *z* residuals are again much higher, with 95% of the residuals in (-1.64, 1.48) mm which is indicative of the higher uncertainty associated with finding the out-of-plane locations relative to the in-plane locations. Corresponding histograms are given in Figure 53.

Table 10 - Calibration variable results for vortex tank tests

T_L	983.30 mm	k_1	1.16e-5
B_1	0.388 mm	k_2	-8.20e-8
B_2	0.397 mm	d_1	1.19e-5
B_3	0.407 mm	d_2	7.78e-8
x_0	-9.22 mm	d_D	0.423
y_0	-1.07 mm		



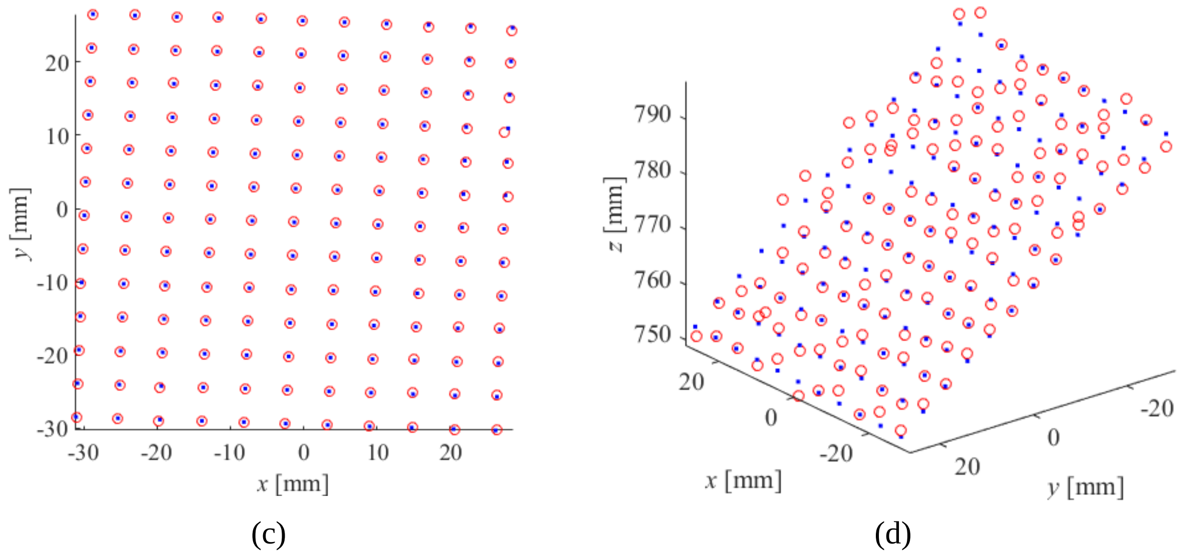
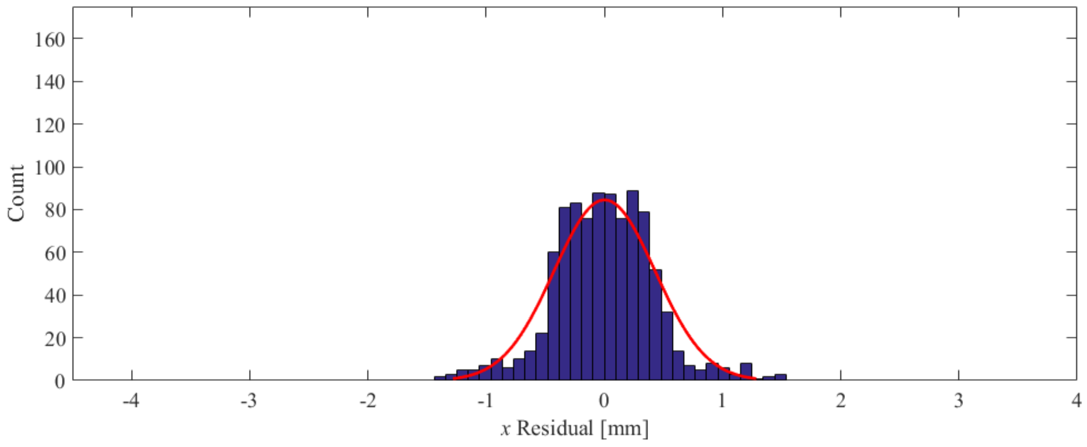
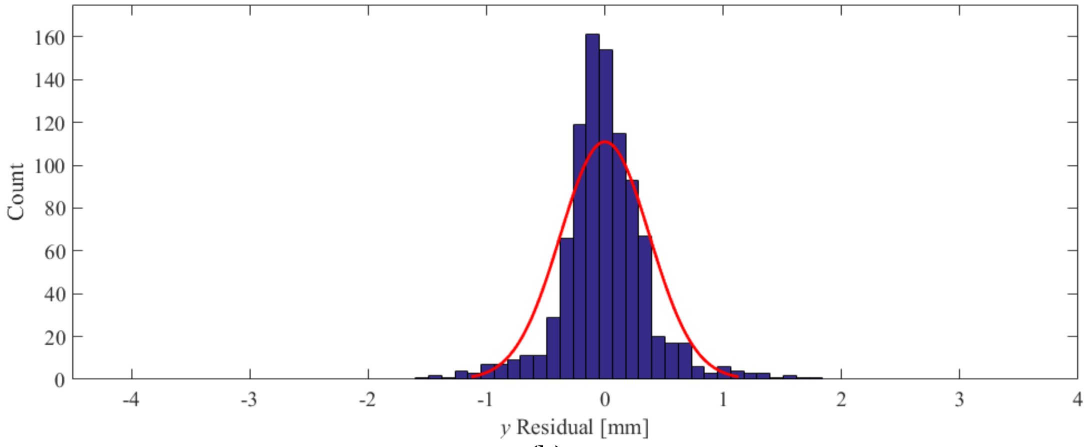


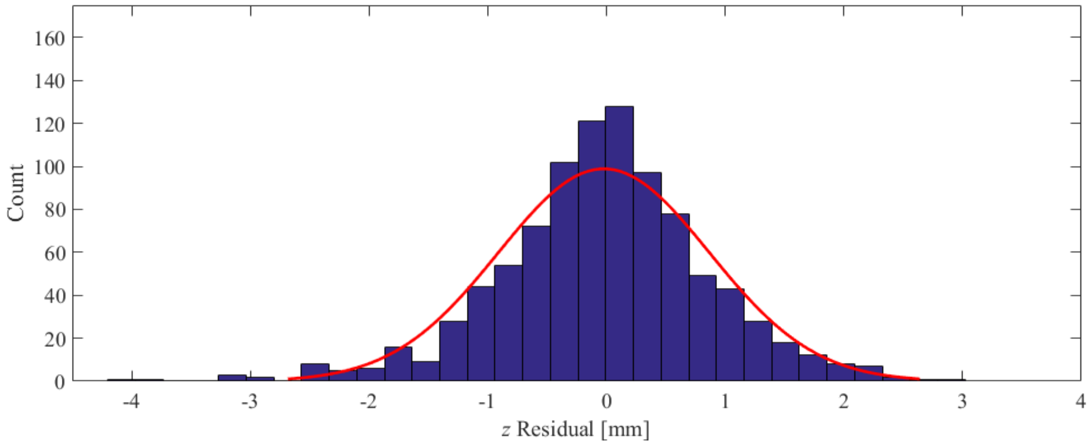
Figure 52 – Detected calibration points in red overlaid on blue grid of predicted points in the vortex tank test. (a-b) and (c-d) show two sample planes.



(a)



(b)



(c)

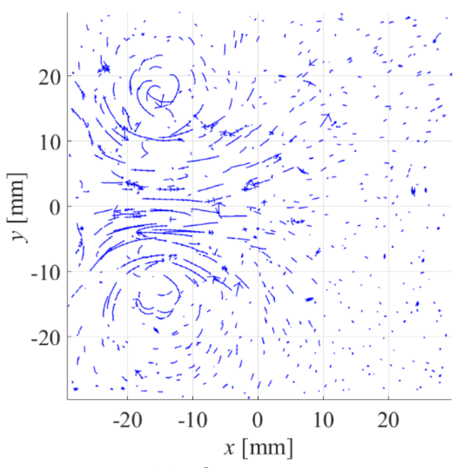
Figure 53 - Histograms of vortex tank micro-lens calibration residuals in (a) *x*, (b) *y*, and (c) *z* directions, overlaid with normal distributions.

7.3 Data Processing

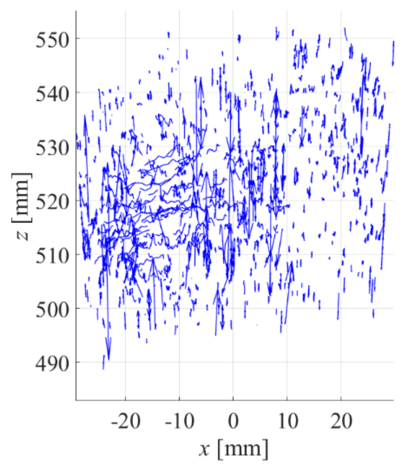
Both the refocusing based approach from Section 3.2 and the ETC approach from Sections 3.3 and 3.5 have been applied to a single sample data set collected in this experiment, using the same calibration approach. Sample results for the refocusing approach are shown in Figure 54 (raw) and Figure 55 (smoothed), and sample results for the ETC approach are shown in Figure 56 (raw) and Figure 57 (smoothed). The progression of the tracks through 10 frames as shown in each image gives a sense of the acceleration of each particle.

The vortex can be seen to progress from left to right in these series of images in Figure 54 to Figure 57. Some noisy elements where tracks have been made incorrectly are also present, particularily in the out-of-plane track sets. The smoothing approach is at least partially successful in eliminating these and does manage to start reconstructing the upper (+*z*) and lower (-*z*) portions of the vortex in Figure 55. The in-plane flow is especially well-reconstructed by the smoothing approach and even somewhat by the basic tracking approach, with the two counter-rotating vorticies clearly visible in all in-plane images. Comparing the sets of figures, the refocusing appraoch has been better able to reconstruct the vortex than the ETC approach. In Figure 55 the refocusing approach has, after smoothing, been able to partially reconstruct the out-of-plane vortices above and below the core of the toroid. The same cannot be said of the smoothed ETC approach in Figure 57, as too much out-of-plane noise is still present to be able to observe the vortices. The smoothing method has also been forced to eliminate more vectors in the ETC, leading to a reduced number of tracks that is apparent even when comparing the in-plane figures of each data set.

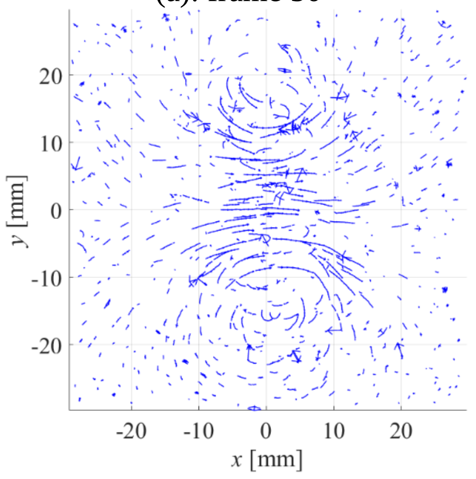
The limitations of the ETC approach’s 2D particle location algorithm is likely the main problem. Here, the algorithm may be lacking when trying to accurately find the 2D positions of particles near the edges of the micro-lens-images. The inability of the ETC or the filtering approach to ignore these particles due to the consistency with which the edge-shifting occurs may also be an issue. Together, the subsequent amplification of these errors by the triangulation approach would lead to poor out of-plane location of the particles. The cross-correlation approach used to make the depth maps and refocused images appears to be more effective at producing good depth estimates, at least for the relatively large particle size used in this case.



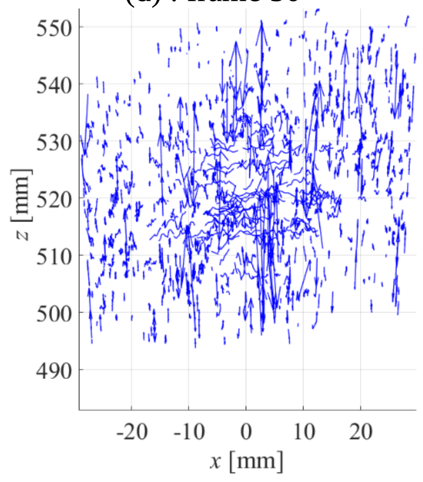
(a): frame 50



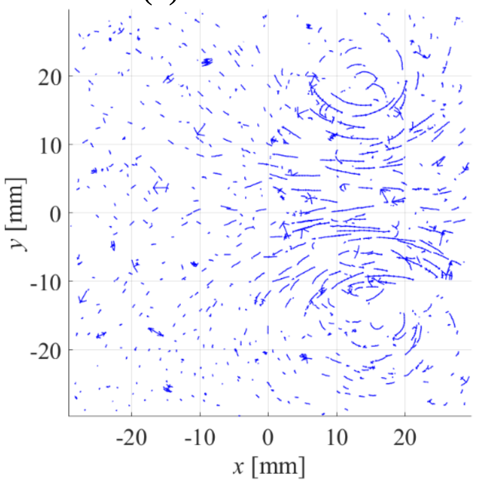
(d) : frame 50



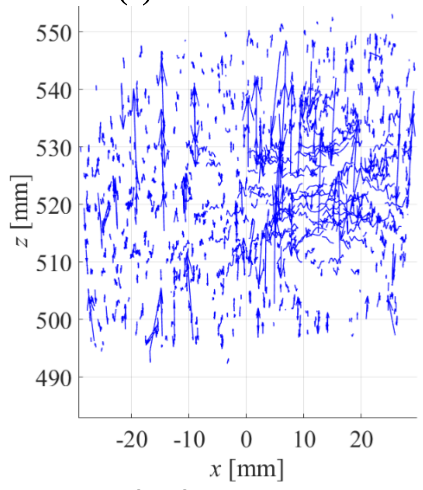
(b) : frame 100



(e) : frame 100

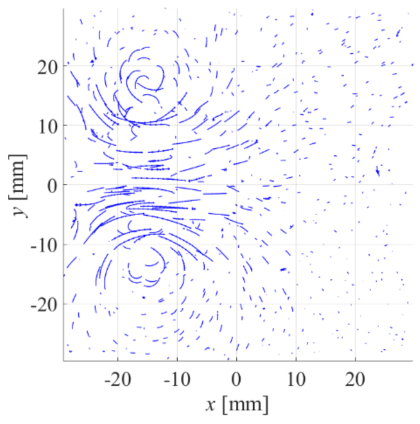


(c) : frame 150

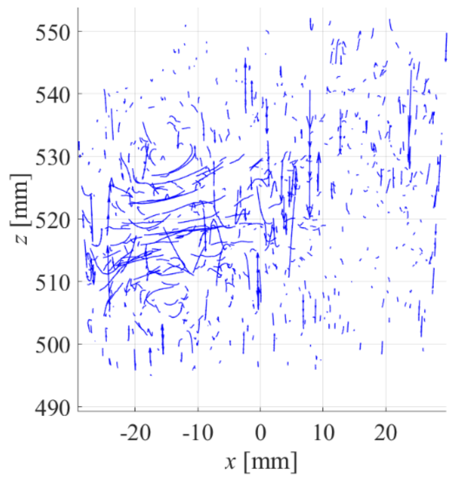


(f) : frame 150

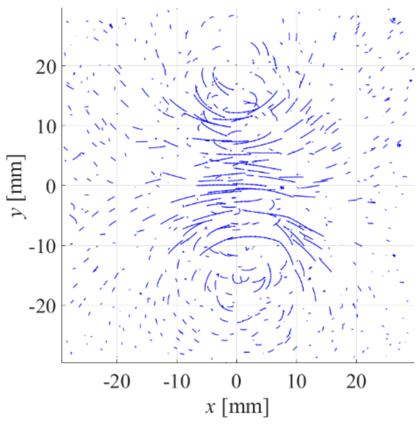
Figure 54 – Raw velocity vectors of the vortex found using the refocusing method. (a-c) show the vectors viewed in the *x*-*y* plane, and (d-f) show the vectors viewed from the *x*-*z* plane.



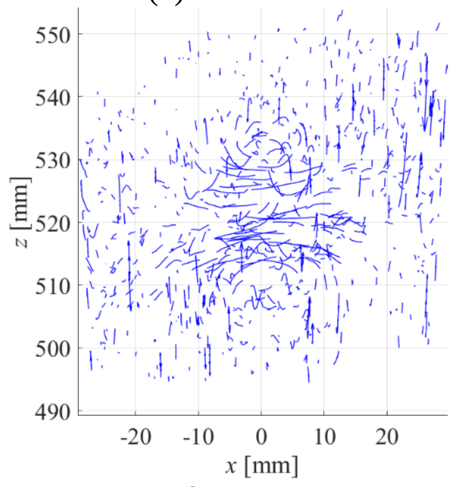
(a): frame 50



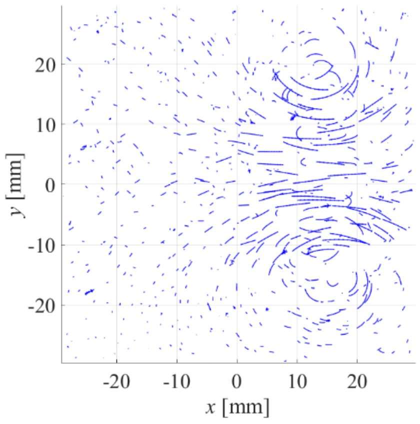
(d) frame 50



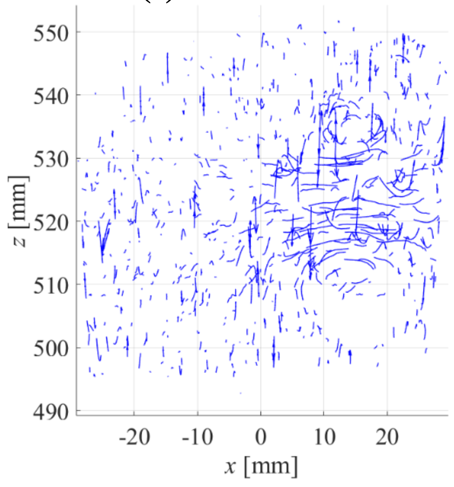
(b) frame 100



(e) frame 100

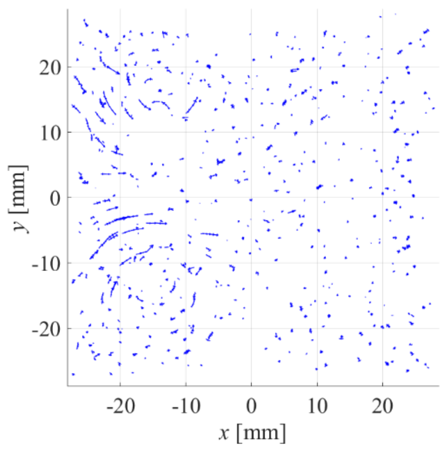


(c) frame 150

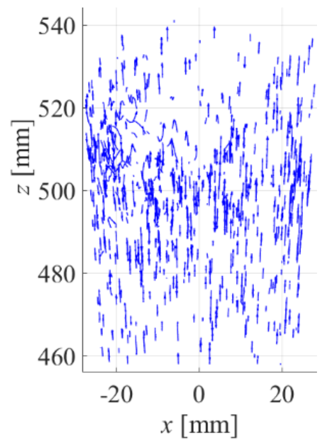


(f) frame 150

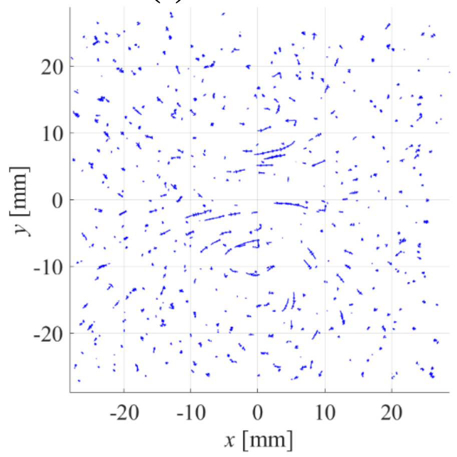
Figure 55 – Smoothed velocity vectors of the vortex found using the refocusing method. (a-c) show the vectors viewed in the *x*-*y* plane, and (d-f) show the vectors viewed from the *x*-*z* plane.



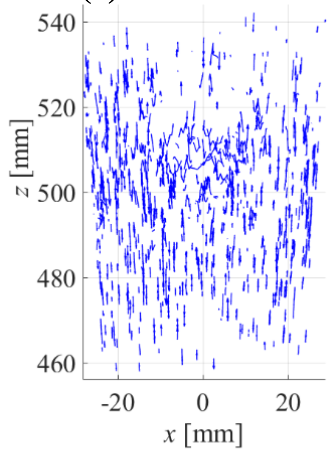
(a): frame 50



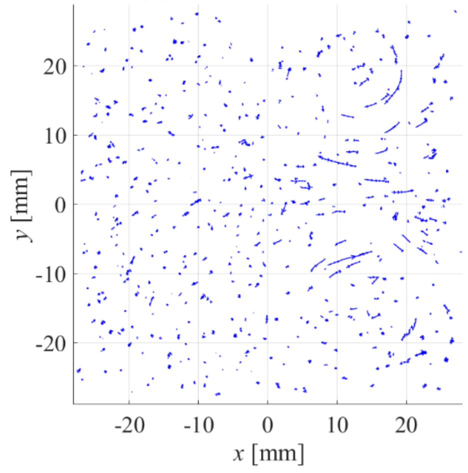
(d) : frame 50



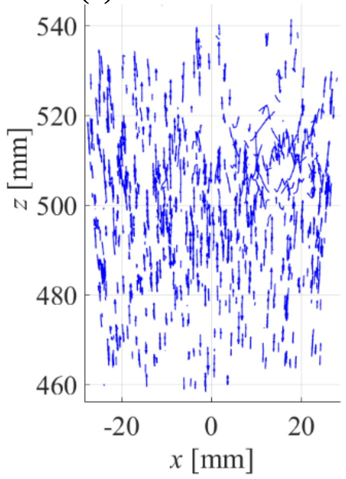
(b) : frame 100



(e) : frame 100

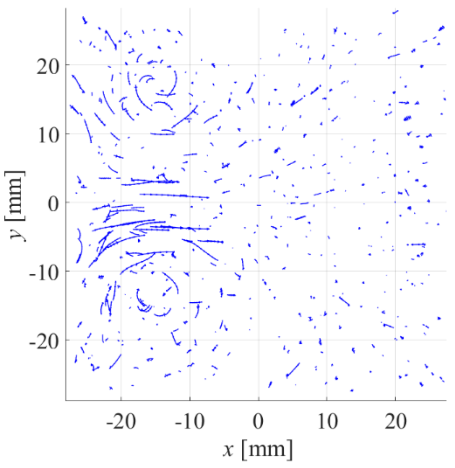


(c) : frame 150

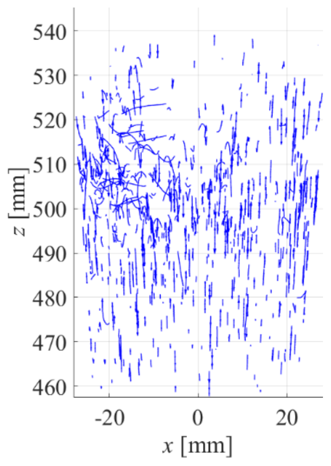


(f) : frame 150

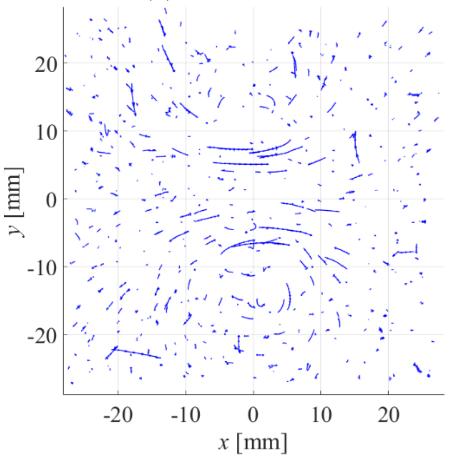
Figure 56 – Raw velocity vectors of the vortex found using the ETC method. (a-c) show the vectors viewed in the *x*-*y* plane, and (d-f) show the vectors viewed from the *x*-*z* plane.



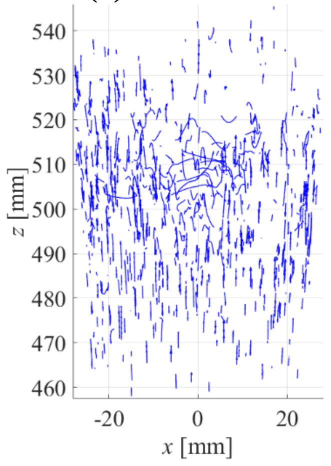
(a): frame 50



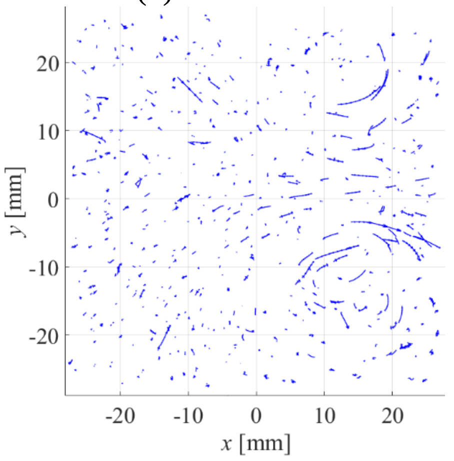
(d) frame 50



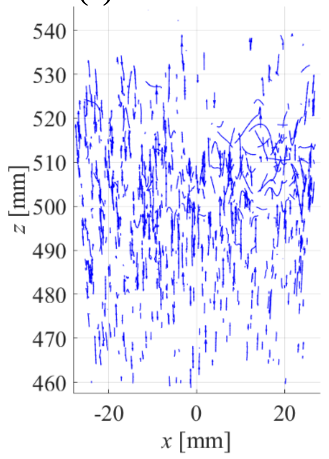
(b) frame 100



(e) frame 100



(c) frame 150



(f) frame 150

Figure 57 – Smoothed velocity vectors of the vortex found using the ETC method. (a-c) show the vectors viewed in the *x*-*y* plane, and (d-f) show the vectors viewed from the *x*-*z* plane.

7.4 Conclusion

The main conclusion that can be drawn from this study is that based on qualitative observations, the refocusing method was better able to reconstruct the toroidal vortex than the ETC method. The ETC method likely had difficulties associated with its ability to correctly determine the virtual depths of individual particles, particularly because the particles were relatively large in this case which could exacerbate occlusion effects near the edges of the micro-lenses. This was likely further influenced by the relatively large particle-image size. As discussed in Chapters 5 and 6, the issues with the out-of-plane particle-location ability of the ETC method are likely to stem from the 2D particle-location method and the failure of the filtering approach from Section 3.3.6.

CHAPTER 8 CONCLUSIONS

To review, the first goal of this study was to develop a method capable of adapting existing commercially available software used for processing multi-focus plenoptic images to locate particles. This has been achieved using a developed Matlab code capable of processing the outputs of the commercial software and extracting the particle locations in pixel and virtual space. The second goal was to implement a calibration algorithm capable of converting the pixel and virtual locations to physical space. An existing multi-focus plenoptic camera calibration algorithm was adapted and implemented for the first time in Matlab. The third goal was to perform time-resolved three-dimensional particle tracking on the found particles. A customized approach specifically designed to address the shortfalls of the particle location algorithm was adapted from existing particle tracking algorithms and implemented in Matlab. The final goal was to develop a custom particle-location approach capable of extracting 3D particle locations from raw plenoptic images. The bulk of this goal has been completed. The developed ETC method is capable of extracting 3D particle locations, but results have indicated that its performance has been hindered by the 2D raw-image particle location algorithm used.

The results from Chapter 5 indicated that use of the R5 *f/#* 2.4 with the ETC method produced a significant increase in track variance after a virtual depth of ~9.35. These results also indicated that the calibration algorithm implemented can under-predict the lens focus distance T_L, producing an incorrect scaling in the depth values and reduced in-plane observed motion. The major success of Chapter 5 was the performance of the smoothing algorithm, which reduced the variance of measured displacements by more than an order of magnitude in all directions and cases. The main area of work remaining in this section would be to repeat the experiment in such a way that the refocusing method could be tested and the results could be compared to confirm which method is more effective.

The results from Chapter 6 showed that the plenoptic camera and ETC particle-location approach combined with the particle tracking and smoothing methods are capable of generating 3D particle tracks at the microscale. The tracks had good in-plane agreement with the expected flow field. The tracking in the out-of-plane direction was reasonable to the extent that some flow structures could be resolved, but was greatly affected by the level of uncertainty in the depth location approach.

The results from Chapter 7 indicated that when tracking macro-scale vortices, the refocusing method performed better than the ETC method. The refocusing method was able to correctly generate some of the out-of-plane structures expected to be present in the toroidal vortex studied. In both methods, but more notable in the ETC method, out-of-plane noise generated from incorrect particle tracks was present. The smoothing method was able to eliminate only some of this noise.

The results of this investigation indicate that the uncertainty associated with the out-of-plane components found by the ETC 3D particle location method is too high when compared the refocusing method, which results in noisy particle tracks. This may mean that the uncertainties associated with the 2D particle centroid location method used are too high and are being amplified by the triangulation method used to find the 3D particle locations. This could be of particular significance near the edges of the micro-lens images, where the 2D centroids are affected by the cutoff of the particle-images.

The major next step in improving upon this work is derived from the new understanding of the issues and problems associated with the developed ETC approach that have become apparent through the conducted studies. This step involves modifying the ETC algorithm to compare the locations of particles based on cross-correlation of the particle-images rather than directly comparing discrete 2D particle centroids, enabling comparison of particles in adjacent micro-lens-images. This could make the ETC method better able to locate particles that are partially cut off by the edges of the micro-lens images in the 2D raw plenoptic images, and therefore better able to reconstruct their positions in 3D.

CHAPTER 9 REFERENCES

[1] D. Schanz, S. Gesemann, and A. Schröder, “Shake-The-Box: Lagrangian particle tracking at high particle image densities,” *Exp. Fluids*, vol. 57, no. 5, p. 70, 2016.

[2] C. E. Willert and M. Gharib, “Digital particle image velocimetry,” *Exp. Fluids*, vol. 10, no. 4, pp. 181–193, Jan. 1991.

[3] H. G. Maas, A. Gruen, and D. Papantoniou, “Particle tracking velocimetry in 3-dimensional flows. 1. Photogrammetric determination of partacle coordinates,” *Exp. Fluids*, vol. 15, no. 2, pp. 133–146, 1993.

[4] S. M. Soloff, R. J. Adrian, and Z.-C. Liu, “Distortion compensation for generalized stereoscopic particle image velocimetry,” *Meas. Sci. Technol.*, vol. 8, no. 12, pp. 1441–1454, 1997.

[5] R. Y. Tsai, “A Versatile Camera Calibration Technique for High-Accuracy 3D Machine Vision Metrology Using Off-the-shelf TV Cameras and Lenses,” *IEEE J. Robot. Autom.*, vol. RA-3, no. 4, 1987.

[6] Z. Zhang, “A Flexible New Technique for Camera Calibration (Technical Report),” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 22, no. 11, pp. 1330–1334, 2000.

[7] G. E. Elsinga, F. Scarano, B. Wieneke, and B. W. van Oudheusden, “Tomographic particle image velocimetry,” *Exp. Fluids*, vol. 41, no. 6, pp. 933–947, 2006.

[8] F. Scarano, “Tomographic PIV: principles and practice,” *Meas. Sci. Technol.*, vol. 24, no. 1, p. 12001, 2013.

[9] D. Homeniuk, “Development and Testing of Macro and Micro Three-Dimensional Particle Tracking Systems,” University of Alberta, 2009.

[10] H. Kinoshita, S. Kaneda, T. Fujii, and M. Oshima, “Three-dimensional measurement and visualization of internal flow of a moving droplet using confocal micro-PIV,” *Lab Chip*, vol. 7, no. 3, pp. 338–346, 2007.

[11] R. Lima, S. Wada, K. Tsubota, and T. Yamaguchi, “Confocal micro-PIV measurements of three-dimensional profiles of cell suspension flow in a square microchannel,” *Meas. Sci. Technol.*, vol. 17, no. 4, pp. 797–808, 2006.

[12] B. Ovryn, “Three-dimensional forward scattering particle image velocimetry applied to a microscopic field-of-view,” *Exp. Fluids*, vol. 29, no. SUPPL. 1, 2000.

[13] M. R. Bown, J. M. MacInnes, R. W. K. Allen, and W. B. J. Zimmerman, “Three-dimensional, three-component velocity measurements using stereoscopic micro-PIV and PTV,” *Meas. Sci. Technol.*, vol. 17, no. 8, pp. 2175–2185, 2006.

[14] E. H. Adelson and J. Y. A. Wang, “Single lens stereo with a plenoptic camera,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 14, no. 2, pp. 99–106, 1992.

[15] M. Levoy and P. Hanrahan, “Light field rendering,” *Proc. 23rd Annu. Conf. Comput. Graph. Interact. Tech. - SIGGRAPH ’96*, pp. 31–42, 1996.

[16] B. Wilburn, N. Joshi, V. Vaish, E.-V. Talvala, E. Antunez, A. Barth, A. Adams, M. Horowitz, and M. Levoy, "High performance imaging using large camera arrays," *ACM Trans. Graph.*, vol. 24, no. 3, p. 765, 2005.

[17] F. Ives, "US patent 725,567," 1903.

[18] G. Lippmann, "Epreuves reversibles, photographies integrales," *Acad. des Sci.*, no. 446451, 1908.

[19] H. E. Ives, "A camera for making parallax panoramagrams," *J. Opt. Soc. Am.*, no. 17, pp. 435–39, 1928.

[20] A. Isaksen, L. McMillan, and S. J. Gortler, "Dynamically reparameterized light fields," *SIGGRAPH 2000*, no. 297306, 2000.

[21] R. Ng, M. Levoy, M. Bredif, G. Duval, M. Horowitz, and P. Hanrahan, "Light Field Photography with a Hand-held Plenoptic Camera," *Stanford Tech Rep.*, vol. 2, 2005.

[22] M. Levoy, R. Ng, A. Adams, M. Footer, and M. Horowitz, "Light field microscopy," *ACM Trans. Graph.*, vol. 3, no. 25, p. 924937, 2006.

[23] A. Lumsdaine and T. Georgiev, "The focused plenoptic camera," in *2009 IEEE International Conference on Computational Photography (ICCP)*, 2009.

[24] T. E. Bishop, S. Zanetti, and P. Favaro, "Light field superresolution," in *2009 IEEE International Conference on Computational Photography (ICCP)*, 2009, pp. 1–9.

[25] T. Georgiev, "New results on the Plenoptic 2.0 camera," in *2009 Conference Record of the Forty-Third Asilomar Conference on Signals, Systems and Computers*, 2009, pp. 1243–1247.

[26] T. Georgiev, G. Chunev, and A. Lumsdaine, "Superresolution with the focused plenoptic camera," in *Proceedings of SPIE - The International Society for Optical Engineering*, 2011, vol. 7873, p. 78730X–78730X–13.

[27] C. Perwaß, L. Wietzke, and R. Gmbh, "Single Lens 3D-Camera with Extended Depth-of-Field," *Proc. SPIE 8291, Hum. Vis. Electron. Imaging XVII*, vol. 49, no. 431, 2012.

[28] Raytrix GmbH, "Raytrix 3D Light Field Camera Technology," 2017. .

[29] T. W. Fahringer, K. P. Lynch, and B. S. Thurow, "Volumetric particle image velocimetry with a single plenoptic camera," *Meas. Sci. Technol.*, vol. 26, no. 11, p. 115201, Nov. 2015.

[30] E. A. Deem, D. Agentis, F. Nicolas, L. N. Cattafesta, T. Fahringer, and B. Thurow, "A Canonical Experiment Comparing Tomographic and Plenoptic PIV," *10th Pacific Symp. Flow Visulaization Image Process.*, no. April 2016, pp. 15–18, 2015.

[31] E. A. Deem, Y. Zhang, L. N. Cattafesta, T. W. Fahringer, and B. S. Thurow, "On the resolution of plenoptic PIV," *Meas. Sci. Technol.*, vol. 27, no. 8, p. 84003, 2016.

[32] E. M. Hall, D. R. GUildenbecher, and B. S. Thurow, "Uncertainty characterization of particle location from refocused plenoptic images," *Opt. Express*, vol. 25, no. 18, pp. 21801–21814, 2017.

[33] E. M. Hall, T. W. Fahringer, B. S. Thurow, and D. R. Guildenbecher, "Volumetric calibration of a plenoptic camera," in *55th AIAA Aerospace Sciences Meeting*, American Institute of Aeronautics

and Astronautics, 2017.

[34] H. Chen and V. Sick, “Three-Dimensional Three-Component Air Flow Visualization in a Steady-State Engine Flow Bench Using a Plenoptic Camera,” *SAE Int. J. Engines*, 2017.

[35] C. Heinze, “Design and test of a calibration method for the calculation of metrical range values for 3D light field cameras,” Fachhochschule Westküste - University of Applied Sciences, 2014.

[36] O. Johannsen, C. Heinze, B. Goldluecke, and C. Perwaß, “On the Calibration of Focused Plenoptic Cameras,” in *Time-of-Flight and Depth Imaging. Sensors, Algorithms, and Applications: Dagstuhl 2012 Seminar on Time-of-Flight Imaging and GCPR 2013 Workshop on Imaging New Modalities*, 2013, pp. 302–317.

[37] E. Halley, “An Instance of the Excellence of the Modern Algebra, in the Resolution of the Problem of Finding the Foci of Optick Glasses Uniuersally. By E. Halley, S. R. S.,” *Philos. Trans.*, vol. 17, pp. 960–969, 1693.

[38] M. Moroni, J. H. Cushman, and A. Cenedese, “A 3D-PTV two-projection study of pre-asymptotic dispersion in porous media which are heterogeneous on the bench scale,” *Int. J. Eng. Sci.*, vol. 41, no. 3–5, pp. 337–370, 2003.

[39] Y. A. Hassan and R. E. Canaan, “Full-field bubbly flow velocity measurements using a multiframe particle tracking technique,” *Exp. Fluids*, vol. 12, no. 1–2, pp. 49–60, 1991.

[40] R. G. Racca and J. M. Dewey, “A method for automatic particle tracking in a three-dimensional flow field,” *Exp. Fluids*, vol. 6, no. 1, pp. 25–32, 1988.

[41] H. Nobach and M. Honkanen, “Two-dimensional Gaussian regression for sub-pixel displacement estimation in particle image velocimetry or particle position estimation in particle tracking velocimetry,” *Exp. Fluids*, vol. 38, no. 4, pp. 511–515, 2005.

[42] S. S. Rogers, T. A. Waigh, X. Zhao, and J. R. Lu, “Precise particle tracking against a complicated background: Polynomial fitting with Gaussian weight,” *Phys. Biol.*, vol. 4, no. 3, pp. 220–227, 2007.

[43] B. Spinewine, H. Capart, M. Larcher, and Y. Zech, “Three-dimensional Voronoi Imaging Methods For The Measurement Of Near-wall Particulate Flows,” *Exp. Fluids*, vol. 34, no. 2, pp. 227–241, 2003.

[44] J. Crocker and D. Grier, “Methods of Digital Video Microscopy for Colloidal Studies,” *J. Colloid Interface Sci.*, vol. 179, no. 1, pp. 298–310, 1996.

[45] D. Blair and E. Dufresne, “The Matlab Particle Tracking Code Repository,” 2007. [Online]. Available: http://site.physics.georgetown.edu/matlab/.

[46] K. Levenberg, “A Method for the Solution of Certain Non-Linear Problems in Least Squares,” *Q. Appl. Math.*, no. 2, pp. 164–168, 1944.

[47] D. Marquardt, “An Algorithm for Least-Squares Estimation of Nonlinear Parameters,” *SIAM J. Appl. Math.*, vol. 11, no. 2, pp. 431–441, 1963.

[48] D. C. Brown, “Decentering distortion of lenses,” *Photom. Eng.*, vol. 32, no. 3, pp. 444–462, 1966.

[49] M. Reidl, “Optical design fundementals for infrared systems,” *SPIE Opt. Eng. Press*, 2001.

[50] B. Lüthi, A. Tsinober, and W. Kinzelbach, “Lagrangian measurement of vorticity dynamics in turbulent flow,” *J. Fluid Mech.*, vol. 528, no. 2005, pp. 87–118, 2005.

[51] B. Rosner, “Percentage Points for a Generalized ESD Many-Outlier Procedure,” *Technometrics*, vol. 25, no. 2, pp. 165–172, May 1983.

[52] J. Hadfield and D. S. Nobes, “Calibration of Plenoptic 2.0 Cameras and use in 3DPTV Investigations,” in *Proceedings of the 18th Annual Symposium on Application of Laser and Imaging Techniques to Fluid Mechanics*, 2016, pp. 3241–3252.

[53] J. Hadfield and D. S. Nobes, “Implementation of Focused Plenoptic Time-Resolved 3D-PTV in the Analysis of a Vortex Flow,” in *Second Thermal and Fluids Engineering Conference*, 2017.

[54] A. Glezer, “The formation of vortex rings,” *Phys. Fluids*, vol. 31, no. 12, p. 3532, 1988.

Appendix A PLENOPTIC SYSTEM DESIGN CODE

The code included here was used to generate the plots of imaging range as a function of image distance in Figure 7 through Figure 9. It is easily adapted for any lens focal length (f) or sensor size (S) by changing the function-evaluation input parameters within.

```
% method for solving a plenoptic system given inputs:
% MLB - micro-lens-sensor distance (maximum)
% selV - maximum desired virtual depth
% outputs plots for 35, 50, 85 mm lenses
% for more information, see Matlab solve, subs, and eval methods
clear
close all

MLB=1.452;
selV=7.5;

syms f Bl B v Bv Tv Tl Fmax D Fmin S positive

e1= 1./(Bl)+1./(Tl-Bl) == 1./f;
e6=S./(2.*(Bl-B))==Fmax./(2.*(Tl-Bl));
e7=Bl+(v-2).*B==Bv;
e8=1./Bv+1./(Tv-Bv)==1./f;
e9=(Tl-Bl)-(Tv-Bv)==D;
e10=S./2./(Bl-B)==Fmin./2./(Tv-Bv);
soln=solve([e1,e6,e7,e8,e9,e10],[Tl,Tv,Bv,Fmax,Fmin,D]);

solsub50.Bl=[55:.25:180];
solsub50.Fmax=eval(subs(soln.Fmax,{f S Bl B},{50 2048*5.5e-3 solsub50.Bl MLB}));
solsub50.Fmin=eval(subs(soln.Fmin,{f S Bl B v},{50 2048*5.5e-3 solsub50.Bl MLB selV}));
solsub50.D=eval(subs(soln.D,{f S Bl B v},{50 2048*5.5e-3 solsub50.Bl MLB selV}));
solsub50.Tl=eval(subs(soln.Tl,{f S Bl B v},{50 2048*5.5e-3 solsub50.Bl MLB selV}));

solsub35.Bl=[38:.25:180];
solsub35.Fmax=eval(subs(soln.Fmax,{f S Bl B},{35 2048*5.5e-3 solsub35.Bl MLB}));
solsub35.Fmin=eval(subs(soln.Fmin,{f S Bl B v},{35 2048*5.5e-3 solsub35.Bl MLB selV}));
solsub35.D=eval(subs(soln.D,{f S Bl B v},{35 2048*5.5e-3 solsub35.Bl MLB selV}));
solsub35.Tl=eval(subs(soln.Tl,{f S Bl B v},{35 2048*5.5e-3 solsub35.Bl MLB selV}));

solsub85.Bl=[94:.25:180];
solsub85.Fmax=eval(subs(soln.Fmax,{f S Bl B},{85 2048*5.5e-3 solsub85.Bl MLB}));
solsub85.Fmin=eval(subs(soln.Fmin,{f S Bl B v},{85 2048*5.5e-3 solsub85.Bl MLB selV}));
solsub85.D=eval(subs(soln.D,{f S Bl B v},{85 2048*5.5e-3 solsub85.Bl MLB selV}));
solsub85.Tl=eval(subs(soln.Tl,{f S Bl B v},{85 2048*5.5e-3 solsub85.Bl MLB selV}));

figure
plot(solsub50.Bl,solsub50.Fmax,'--r',solsub50.Bl,solsub50.Fmin,'--b',solsub50.Bl,solsub50.D,'--
k')
legend({'{\itF}_{MAX}','{\itF}_{MIN}','\itD'},'FontName','Times New Roman','FontSize',12)
```

```
xlabel('Lens-MLA distance [mm]','FontName','Times New Roman','FontSize',12)
ylabel('Distance [mm]','FontName','Times New Roman','FontSize',12)
xlim([70 90])
set(gca,'FontName','Times New Roman')
set(gca,'fontsize',12)

figure
plot(solsub35.Bl,solsub35.Fmax,'-r',solsub35.Bl,solsub35.Fmin,'-b',solsub35.Bl,solsub35.D,'-k')
legend({'{\itF}_{MAX}','{\itF}_{MIN}','\itD'},'FontName','Times New Roman','FontSize',12)
xlabel('Lens-MLA distance [mm]','FontName','Times New Roman','FontSize',12)
ylabel('Distance [mm]','FontName','Times New Roman','FontSize',12)
xlim([45 65])
set(gca,'FontName','Times New Roman')
set(gca,'fontsize',12)

figure
plot(solsub85.Bl,solsub85.Fmax,'-.r',solsub85.Bl,solsub85.Fmin,'-.b',solsub85.Bl,solsub85.D,'-
.k')
legend({'{\itF}_{MAX}','{\itF}_{MIN}','\itD'},'FontName','Times New Roman','FontSize',12)
xlabel('Lens-MLA distance [mm]','FontName','Times New Roman','FontSize',12)
ylabel('Distance [mm]','FontName','Times New Roman','FontSize',12)
xlim([130 150])
set(gca,'FontName','Times New Roman')
set(gca,'fontsize',12)

figure
plot(solsub35.Bl,solsub35.Fmax,'-r',solsub35.Bl,solsub35.Fmin,'-b',solsub35.Bl,solsub35.D,'-
k',solsub50.Bl,solsub50.Fmax,'--r',solsub50.Bl,solsub50.Fmin,'--b',solsub50.Bl,solsub50.D,'--
k',solsub85.Bl,solsub85.Fmax,'-.r',solsub85.Bl,solsub85.Fmin,'-.b',solsub85.Bl,solsub85.D,'-.k')
legend({'{\itf}=35 mm {\itF}_{MAX}','{\itf}=35 mm {\itF}_{MIN}','{\itf}=35 mm \itD','{\itf}=50 mm
{\itF}_{MAX}','{\itf}=50 mm {\itF}_{MIN}','{\itf}=50 mm \itD','{\itf}=85 mm
{\itF}_{MAX}','{\itf}=85 mm {\itF}_{MIN}','{\itf}=85 mm \itD'},'FontName','Times New
Roman','FontSize',12)
xlabel('Lens-MLA distance [mm]','FontName','Times New Roman','FontSize',12)
ylabel('Distance [mm]','FontName','Times New Roman','FontSize',12)
ylim([0 150])
set(gca,'FontName','Times New Roman')
set(gca,'fontsize',12)
xlim([35,180])
```

Published with MATLAB® R2017b

Appendix B REFOCUSING CODE

The Matlab codes for the refocusing method are included here. The functions bpass, pkfnd, and cntrd are available from [45]. All other functions are available in Matlab libraries, or have been included here. The structure of the code is given in the chart below. The red and green blocks are the tracking and tracking auxiliary methods common between this code and the ETC code. These will be included in Appendix E. The required inputs for this code are the refocused images and depth maps from the commercial software package. Also needed is a completed optical system calibration, the code for which is given in Appendix D.

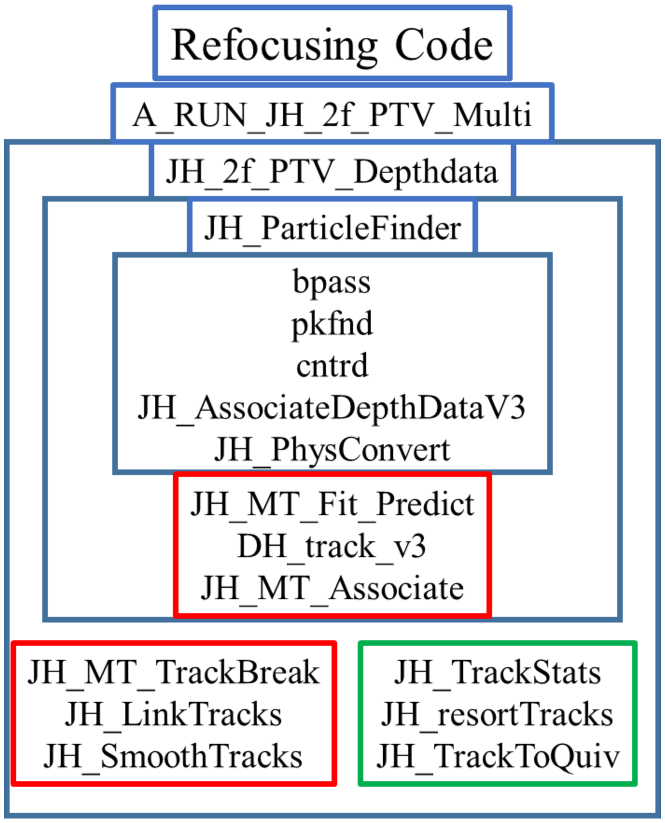


Figure B1 – Flow chart of the refocusing code

B.1 *Wrapper function: A_RUN_JH_2f_PTV_Multi*

The following code serves as a ‘wrapper’, running all other functions in the refocusing code in loops as needed, while ensuring that all inputs and outputs reach the correct locations, and also saving the results. It is set up to allow the inputs to be included either in the first section of the code, or in the function when called.

```
function A_RUN_JH_2f_PTV_Multi(Iloc,imNos,imFmt,path,CalLoc,smoothstrength,settings,display)
if nargin < 7 || length(settings)~=11
    % Default Settings
    % centroid detection (units of pixels)
    inversion = 0;
    pkthresh = 20;
    psize = 9; % should be odd
    noisescale = 3;
    noisethresh = 0;

    % depth detection (for later)
    drt = 8; % required number of depth estimations to keep a point

    % particle tracking (units of mm)
    search = 2;
    neighbor = 20;
    quasi = .3;
    zdt = 0.05;

    % particle prediction
    range = 1;
    fp=5;

    % Track Rejection
    art = 2; % acceleration rejection threshold in mm/frame^2
    zaf = .05; % z-amplification factor, used to reduce z-direction components

    % linkage
    lzdt=0.05;

    % Track Smoothing
    % number of frames to use on either side of particle when smoothing
    % (depends on movement/frame >> more movement, less smoothing).
    % Generally want at least 5 and not more than 20.
    smoothstrength=16;

    settings = [pkthresh psize noisescale noisethresh inversion drt search neighbor quasi range
zdt];

    display = 0;
else
```

```
    pkthresh = settings(1);
    psize = settings(2);
    noisescale = settings(3);
    noisethresh = settings(4);
    inversion = settings(5);
    drt = settings(6);
    search = settings(7);
    neighbor = settings(8);
    quasi = settings(9);
    range = settings(10);
    zdt = settings(11);
end

close all

% initializations
minx=0;
miny=0;
minz=0;
maxx=0;
maxy=0;
maxz=0;
nlostpts=0;

% control to skip initial tracking

% DEFAULT CALIBRATION
if nargin < 5
    CalLoc = 'X:\01_Current_Students\Jake Hadfield\Coding\Matlab\Calibration
Code\Calibration_170529.mat';
end

% LOAD IMAGES
if nargin < 4
    path = 'X:\01_Current_Students\Jake Hadfield\LightField Test Images\FixedField_172905\01X\';
end
if nargin < 3
    Iloc = 'Image';
    imNos = 1024:1088;
    imFmt = '.tiff';
    savedate= '1706001';
end
newtracks=1; % Need to run particle-location code
if newtracks==1

% RUN ALGORITHMS
fprintf(['Running first image pair. Completion time will be estimated soon. Current time is: '
datestr(now) '\n']);
tic;
for i=1:length(imNos)-1
    if i==2
        tic;
    end
    Iloc1 = [Iloc '_' num2str(imNos(i),'%04i')];
    Iloc2 = [Iloc '_' num2str(imNos(i+1),'%04i')];
    if i == 1
        [alltracks,polyhold,p1,mp,ps] =
JH_2f_PTV_Depthdata(Iloc1,Iloc2,imFmt,path,CalLoc,settings,display);
        plist=[ps ones(length(ps),1)];
    else
        [alltracks,polyhold,p1,mp] =
JH_2f_PTV_Depthdata(Iloc1,Iloc2,imFmt,path,CalLoc,settings,display,alltracks,i,polyhold,p1,mp);
    end
    tt=toc;
    if i>1
        fprintf(['Estimated completion is around ' datestr(now+tt/(i-1)*(length(imNos)-1-
i)/84600) '\n']);
    else
        fprintf(['Estimated completion is around ' datestr(now+tt/1.8*(length(imNos)-2)/84600)
'\n']);
    end
    [lp1,~]=size(p1);
    plist=[plist; [p1 ones(lp1,1)*(i+1)]];
end

fprintf(['Actual completion time was: ' datestr(now) '\n']);

% eliminate remaining predicted points
alltracks(alltracks(:,6)==0,:)=[];
save([path 'PList_' savedate '_' Iloc '.mat'],'plist')
save([path 'AllTracks_' savedate '_' Iloc '.mat'],'alltracks')
else
    load([path 'AllTracks_170324_' Iloc '.mat'])
    mp=[];
end
% get track stats
JH_TrackStats(alltracks)

% denoise tracks
lostpts=double.empty(0,5);
while ~isempty(nlostpts)
    [filttracks,nlostpts,~]=JH_MT_TrackBreak(alltracks,art,zaf);
    alltracks=filttracks;
    lostpts=[lostpts;nlostpts];
end
nmp=lostpts(:,[1:3,5]);
save([path 'BreakTracks_' savedate '_' Iloc '.mat'],'filttracks')
% resort tracks
[filttracks,misspts]=JH_resortTracks(filttracks);

% get track stats
JH_TrackStats(filttracks)

mp=sortrows([mp;nmp;misspts],4);
```

```
save([path 'FilTracks_' savedate '_' Iloc '.mat'],'filttracks')
save([path 'MissPts_' savedate '_' Iloc '.mat'],'mp')

linkcount=1;
while linkcount~=0
% link tracks
[linktracks,linkcount]=JH_LinkTracks(filttracks,range,lzdt,fp);
filttracks=linktracks;
% resort tracks
[linktracks,~]=JH_resortTracks(linktracks);
end
% get track stats
JH_TrackStats(linktracks)

% find longest linked track
long=1;
lc=1;
for i=1:max(linktracks(:,4))
    nlc=sum(linktracks(:,4)==i);
    if nlc>lc
        long=i;
        lc=nlc;
    end
end

link=linktracks(linktracks(:,4)==long,:);

% smoothing algorithm
smoothtracks = JH_SmoothTracks(linktracks,smoothstrength);

smooth=smoothtracks(smoothtracks(:,4)==long,:);

% get track stats
JH_TrackStats(smoothtracks)

% % find domain limits
% minx=min(alltracks(:,1));
% miny=min(alltracks(:,2));
% minz=min(alltracks(:,3));
% maxx=max(alltracks(:,1));
% maxy=max(alltracks(:,2));
% maxz=max(alltracks(:,3));
% % save data
% save([path 'Limits_170111_' Iloc '.mat'],'minx','miny','minz','maxx','maxy','maxz')
save([path 'LinkTracks_' savedate '_' Iloc '.mat'],'linktracks')
save([path 'SmoothTracks_' savedate '_' Iloc '.mat'],'smoothtracks')
save([path 'SLink_' savedate '_' Iloc '.mat'],'link')
save([path 'SSmooth_' savedate '_' Iloc '.mat'],'smooth')
JH_TrackToQuiv(path,['AllTracks_' savedate '_' Iloc '.mat']);
JH_TrackToQuiv(path,['SmoothTracks_' savedate '_' Iloc '.mat']);
```

Published with MATLAB® R2015b

B.2 *Location and Tracking: JH_2f_PTV_Depthdata*

The following code is run for each image in the series. It is responsible for 3D location of the particles, and then the initial tracking after the particles in each frame have been located.

```
function [alltracks,polyhold,p2,mp,ps] =
JH_2f_PTV_Depthdata(Iloc1,Iloc2,imFmt,path,CalLoc,settings,display,alltracks,i,polyhold,p1,mp)
```

```
if nargin < 8
    alltracks = [];
end
if nargin < 9
    i = 1;
end
if nargin < 10
    polyhold.pno=[];
    polyhold.polyx=[];
    polyhold.polyy=[];
    polyhold.polyz=[];
    polyhold.mux=[];
    polyhold.muy=[];
    polyhold.muz=[];
end
if nargin < 12
    mp=[];
end
if nargin < 6 || length(settings)~=11
    % Default Settings
    % centroid detection (units of pixels)
    inversion = 0;
    pkthresh = 50;
    psize = 9; % should be odd
    noisescale = 0;
    noisethresh = 20;

    % depth detection (for later)
    drt = 4; % required number of depth estimations to keep a point

    % particle tracking (units of mm)
    search = .5;
    neighbor = 3;
    quasi = 0.2;

    % particle prediction
    range = 0.2;
else
    pkthresh = settings(1);
    psize = settings(2);
    noisescale = settings(3);
    noisethresh = settings(4);
    inversion = settings(5);
    drt = settings(6);
    search = settings(7);
    neighbor = settings(8);
    quasi = settings(9);
    range = settings(10);
    zdt = settings(11);
end

close all

% DEFAULT CALIBRATION
if nargin < 5
    CalLoc = '\\nobes-nas01\NobesGroup\01_Current_Students\jhadfiel\Coding\Matlab\Calibration
Code\Calibration_T1';
end

% LOAD IMAGES
if nargin < 4
    path = 'C:\Users\jhadfiel\Desktop\TestImages\ParticleArc\Recirculation\';
end
if nargin < 3
    Iloc1 = 'SmallTube_80fps_3_0100';
    Iloc2 = 'SmallTube_80fps_3_0101';
    imFmt = '.tiff';
end

% FIND PARTICLES
p2=JH_ParticleFinder(Iloc2,imFmt,path,CalLoc,[pkthresh,psize,noisescale,noisethresh,inversion,drt
],display);
if nargin < 11

p1=JH_ParticleFinder(Iloc1,imFmt,path,CalLoc,[pkthresh,psize,noisescale,noisethresh,inversion,drt
],display);
end

ps=p1;

if display == 2
    figure
    scatter3(p1(:,1),p1(:,2),p1(:,3),.5,'b');
    hold on
    scatter3(p2(:,1),p2(:,2),p2(:,3),.5,'r');
    uiwait(gcf)
end

% TRACK
if i>=3
    fprintf('Attempting forward-predictive tracking on %i frame %i and %i frame %i
particles...\n',sum(~isnan(p2(:,3))),i,sum(~isnan(p2(:,3))),i+1);
```

```
    [p1,p2,alltracks,polyhold]=JH_MT_Fit_Predict(p1,p2,alltracks,i,range,zdt,polyhold,search);
end

%fprintf('Comparing %i particles in frame 1 to %i particles in frame
2\n',sum(~isnan(p1(:,3))),sum(~isnan(p2(:,3))));

fprintf('Attempting two-frame tracking on remaining %i frame %i and %i frame %i
particles...\n',sum(~isnan(p1(:,3))),i,sum(~isnan(p2(:,3))),i+1);

tracks = DH_track_v3(p1,p2,search,neighbor,quasi,zdt);

if ~isempty(tracks)
    nmp=p2(~ismember(p2,tracks(:,1:3),'rows'),:);
    [lnmp,~]=size(nmp);
    mp=[mp;nmp ones(lnmp,1)*i];
    alltracks=JH_MT_Associate(tracks,alltracks,i);
end
```

Published with MATLAB® R2015b

B.3 *Particle location function: JH_ParticleFinder*

The following code locates each particle in 3D based on input refocused images and depth maps.

```
function particles=JH_ParticleFinder(imID,imFmt,path,CalLoc,settings,display)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Finds particles in 3D from light field focused image + depth data.
% Expects one focused image and 3 depth images.
% Inputs:
% imID - Image file name (eg, Image_0001)
% imFmt - Image file extension (eg, .png)
% path - Image file location
% CalLoc - Calibration file location
% Display - how much display to do (integer, 0 to 2)
% Outputs:
% particles - 3D particle field, in the same units as the calibration file

% Settings
if length(settings)~=6
    % centroid detection properties, in pixels
    pkthresh = 30; % minimum acceptable peak threshold
    psize = 31; % should be odd, slightly larger than particles' pixel size
    noisescale = 5; % noise scale filter
    noisethresh = 10; % low-pass filter
    inversion = 1; % wether or not to invert image

    % depth detection values, in pixels
    drt = 7; % required number of depth estimations to keep a point
else
```

```
    pkthresh = settings(1);
    psize = settings(2);
    noisescale = settings(3);
    noisethresh = settings(4);
    inversion = settings(5);
    drt = settings(6);
end

% load calibration
load(CalLoc,'allvars')
caldata = allvars(1:12);

% Get image in memory
focIm = imread([path imID '_Focus_Depth'  imFmt]);
% invert if required
if inversion
    focIm=double(255-focIm);
end

%fprintf('Performing 2d centroid location...\n')
% 2D Centroid Location
% FILTER IMAGE
% IMAGE : Threshold : size of particle (longer)
binIm = bpass(focIm,noisescale,psize*2,noisethresh);

% FIND PARTICLES
% IMAGE : Image Threshold : size of particle (longer)
pk = pkfnd(binIm,pkthresh,psize);

% Do again to sub-pixel
cnt = cntrd(double(focIm),pk,psize);

if display == 2
    figure(1)
    imshow(focIm)
    hold on
    scatter(cnt(:,1),cnt(:,2),'+r');
    hold off
    drawnow
end

%fprintf('Adding 3d location data...\n');
% 3D Centroid Location
% Add in the depth data
for j=1:3
    switch j
        case 1
            depthIm = imread([path imID '_Depth_Far' imFmt]);
        case 2
            depthIm = imread([path imID '_Depth_Mid' imFmt]);
        case 3
            depthIm = imread([path imID '_Depth_Near' imFmt]);
    end
    [cntz(:,j),count(:,j),overs(j)]=JH_AssociateDepthDataV3(cnt,depthIm,1.25*psize,drt,display >=
2,j+1);
    %fprintf('Depth calculation %i complete.\n',j);
end
%fprintf('3D centroid location complete, with %i overlaps detected.\n',sum(overs)/3);

cnts = [cnt(:,1:2),cntz(:,:)];

if display >2
    figure
    scatter3(cnts(:,1),cnts(:,2),cnts(:,3),0.5);
    uiwait(gcf)
end

%fprintf('Performing metric conversion...\n')
% Metric Conversion
particles = JH_PhysConvert(cnts,caldata,count);
```

Published with MATLAB® R2015b

B.4 *Depth location function: JH_AssociateDepthDataV3*

The following code extracts the depth information for each 2D particle location found in the refocused images from the depth maps, and associates the extracted depth with each 2D location.

```
function [cntz,count,overs]=JH_AssociateDepthDataV3(cnt,dm,prad,drt,vis,fig)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Associates virtual depths with 2D image centroids for a calibration
% target. Deals with overlapping particles.
% Inputs:
% cnt    - centroids from cntrd in (x,y,I,rg) format
% dm     - image of depth map. Must be same size as full focus image
% psize  - approximate size of a dot
% border - border around the dot that will contain all depth values for
%          that dot
%
% Outputs:
% cnt    - centroids from cntrd in (x,y,z,I,rg,) format

%masking stuff
xx=repmat(1:1:length(dm),length(dm),1);
yy=repmat(1:1:length(dm),length(dm),1)';

% First depth map
% plot for testing
if vis == 1
    figure(fig)
    vdeptharray=reshape(typecast(reshape(uint32(dm),[],1),'single'),1024,1024);
    imshow(vdeptharray)
    colormap jet
    % min(min(vdeptharray(vdeptharray>0))) is a bit bigger than the
    % theoretical min of 2
    caxis([2 8]);
    cm=colormap;
    cm(1,:)=[0 0 0];
    colormap(cm);
    colorbar
    hold on
    drawnow
end
% add a column to cnt to store depths
cnt(:,5)=cnt(:,4);
cnt(:,4)=cnt(:,3);
cnt(:,3)=0;
cntz=zeros(length(cnt),1);
count = zeros(length(cnt),1);
overs = 0;
%pp=parpool('local',4);
for i=1:length(cnt)
%     str=sprintf('Performing depth calculation on particle %i of %i',i,length(cnt));
%     if i>2
%         [char(8)*ones(1,lStr+10),str]
%     elseif i==2
%         [char(8)*ones(1,lStr+9),str]
%     else
%         disp(str)
%     end
%     lStr = length(str);
    % get distance to other centroids

    if vis == 1
        figure(fig)
        d = prad*2;
        px = cnt(i,1)-prad;
        py = cnt(i,2)-prad;
        rectangle('Position',[px py d d],'Curvature',[1,1],'EdgeColor','g');
        drawnow
    end

    % grab subset of depth map values from around centroid
    subset = dm(sqrt((xx-cnt(i,1)).^2+(yy-cnt(i,2)).^2)<=prad);
    % re-orient into single column of nonzero values
    alldepths=subset(subset>0);
    % convert to virtual depth
    vdepths = typecast(uint32(alldepths),'single');
    vdepths = vdepths(vdepths>2);
    % get standard deviation
    vdsd=std(vdepths);
    vdm=median(vdepths);
    % run a loop to eliminate outliers
    outliers=1;
    while outliers
        nvdepths=vdepths(abs(vdepths-vdm)<vdsd*2);
        if length(nvdepths)==length(vdepths)
            vdepths=nvdepths;
            outliers=0;
        else
            vdm=median(nvdepths);
            vdsd=std(nvdepths);
            vdepths=nvdepths;
        end
    end
    count(i)=length(vdepths);
    if count(i)>drt
%         if vdm<=8
            cntz(i)=vdm;
%         else
%             cntz(i)=NaN;
%         end
    else
        cntz(i)=NaN;
    end
end
%delete(pp)
```

Published with MATLAB® R2015b

B.5 *Physical-space conversion function: JH_PhysConvert*

The following code converts particle locations from pixel and virtual coordinates in image space to metric coordinates in physical space based on the calibration.

```
function mpoints = JH_PhysConvert(impoints,caldata,count)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Scales the pixel+virtual space n-by-5 array impoints into a physical space
% n-by-3 array mpoints, using the parameters set by caldata which should be
% in [fL,tL,b1,b2,b3,x0,y0,k1,k2,d1,d2,dD] form
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% constants
spix = 5.5e-3;% pixel size [mm/pix]
Rx = 1024; % image (sensor?) resolution x [pix]
Ry = 1024; % iamge (sensor?) resolution y [pix]

% shift caldata into variables
% main lens variables
fL = 85;
tL = caldata(1);
```

```
% micro-lens variables
b1= caldata(2);
b2= caldata(3);
b3= caldata(4);

% distortion variables
x0 = caldata(5); % distortion offset x from corner [pix] relative to image centre
y0 = caldata(6); % distortion offset y from corner [pix] relative to image centre
k1 = caldata(7); % radial distortion coeff 1
k2 = caldata(8); % radial distortion coeff 2
d1 = caldata(9); % depth distortion coeff 1
d2 = caldata(10); % depth distortion coeff 2
dD = caldata(11); % depth-dependant depth distortion coeff

% % distortion due to cylinder
% yc0 = caldata(13);
% kc1 = caldata(14);
% kc2 = caldata(15);
% dc1 = caldata(16);
% dc2 = caldata(17);
% dcD = caldata(18);

% lens modeling parameters
bL = tL/2*(1-sqrt(1-4*fL/tL));
Gp = tL-bL;

% Physical Conversion
% convert virtual depths and pixel locations to metric
    zdata=bL+bsxfun(@times,impoints(:,3:5)-2,[b1,b2,b3]);
    metric=[bsxfun(@times,-(impoints(:,1)-Rx/2)*2*spix,bsxfun(@rdivide,zdata,bL)),...
        bsxfun(@times,-(impoints(:,2)-Ry/2)*2*spix,bsxfun(@rdivide,zdata,bL)),...
        zdata];
    % compute r for each point
    r(:,1) = sqrt(sum(abs(bsxfun(@minus,metric(:,[1,4]),[x0,y0])).^2,2));
    r(:,2) = sqrt(sum(abs(bsxfun(@minus,metric(:,[2,5]),[x0,y0])).^2,2));
    r(:,3) = sqrt(sum(abs(bsxfun(@minus,metric(:,[3,6]),[x0,y0])).^2,2));
    % undistort metric-depth points
    undist=[metric(:,1).*(1+k1*r(:,1).^2+k2*r(:,1).^4),...
        metric(:,2).*(1+k1*r(:,2).^2+k2*r(:,2).^4),...
        metric(:,3).*(1+k1*r(:,3).^2+k2*r(:,3).^4),...
        metric(:,4).*(1+k1*r(:,1).^2+k2*r(:,1).^4),...
        metric(:,5).*(1+k1*r(:,2).^2+k2*r(:,2).^4),...
        metric(:,6).*(1+k1*r(:,3).^2+k2*r(:,3).^4),...
        metric(:,7:9)+(1+dD*metric(:,7:9)).*(d1*r(:,1:3).^2+d2*r(:,1:3).^4)];
    % project points to physical space
    zdata=undist(:,7:9)*fL./(undist(:,7:9)-fL);
    phys=[undist(:,1:3)./undist(:,7:9).*zdata,...
        undist(:,4:6)./undist(:,7:9).*zdata,...
        zdata];

% Depth Selection
% we now have 3 depths and 3 (x,y) locations for each particle...only want 1
% so, do a weighted average of these based on how good the initial virtual
% depth measurement was

% determine the weights, based on the original number of depth estimations
w1=count(:,1);
w2=count(:,2);
w3=count(:,3);

% determine the nan locations
nnan1=~isnan(phys(:,7));
nnan2=~isnan(phys(:,8));
nnan3=~isnan(phys(:,9));

% set nans to zero
phys(~nnan1,7)=0;
phys(~nnan2,8)=0;
phys(~nnan3,9)=0;

% do the weigthed average, not including nan weight
mpoints =
[(phys(:,1).*w1.*nnan1+phys(:,2).*nnan2.*w2+phys(:,3).*nnan3.*w3)./(w1.*nnan1+w2.*nnan2+w3.*nnan3
)
(phys(:,4).*w1.*nnan1+phys(:,5).*w2.*nnan2+phys(:,6).*w3.*nnan3)./(w1.*nnan1+w2.*nnan2+w3.*nnan3)
(phys(:,7).*nnan1.*w1+phys(:,8).*nnan2.*w2+phys(:,9).*w3.*nnan3)./(w1.*nnan1+w2.*nnan2+w3.*nnan3)
];
mpoints(mpoints==0)=NaN;
```

Published with MATLAB® R2015b

Appendix C ETC METHOD

The Matlab codes for the ETC method are included here. The functions bpass, pkfnd, and cntrd are available from [45]. All other functions are available in Matlab libraries, or have been included here. The structure of the code is given in the chart below. Before running these codes, a micro-lens calibration must first be performed, the code for which is included in this appendix. The red and green blocks are the tracking and tracking auxiliary methods common between this code and the refocusing code. These will be included in Appendix E. The required inputs for this code are raw plenoptic images. Also needed is a completed optical system calibration, the code for which is given in Appendix D. The JH_PhysConvert method has already been presented in Appendix B and will not be included again.

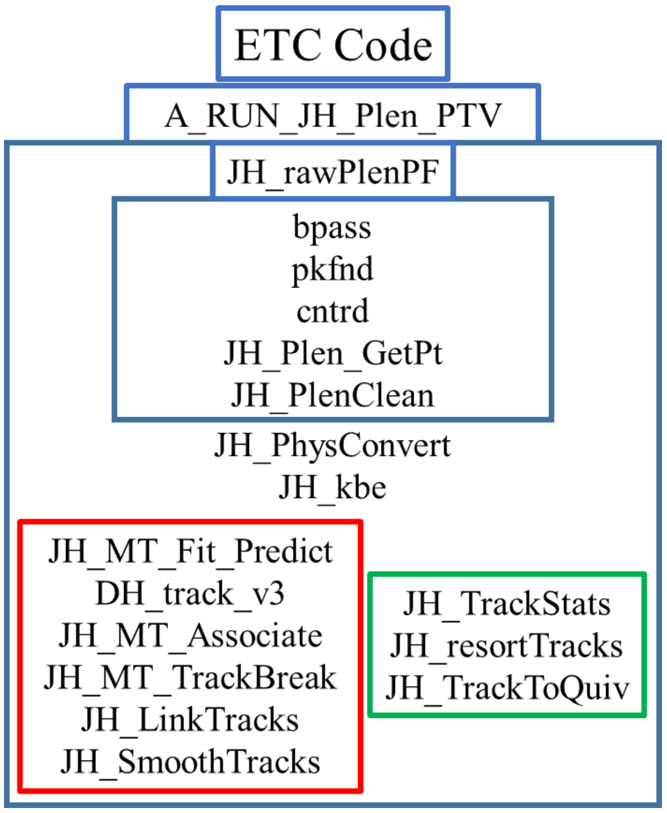


Figure C1 – Flow chart of the ETC code

C.1 *Micro-lens calibration function: micro-lensCal*

The method for the micro-lens calibration is included here. It requires only the input of a white image, or a group of white images if an average white image is desired.

```
function [lensCents,adjlist,typelist,ev,mask]=micro-lensCal(varargin)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Generates a model of the ideal micro-lens centres, given a few
% input parameters, and associates input centroids. first two columns of
% impoints contain [a,b] matrix of valid target points, last two contain
% original centroid locations.
% Inputs:
% cnts       - Detected centroids of grid points, in (x,y,z) form [pix]
% radratio   - Tolerance for expected point location, max 0.5, recommended
%              0.1 or less; depends on target image quality and noise level
% maxmiss    - Number of points
% thresh     - Threshold, the value used as a cutoff when binarizing the
% image to estimate the circle centroids. MUST result in disconnection
% between ALL circles.
% Outputs:
% lensCents  - array of [a,b] positions indicating the pixel locations of
%              each lens centre, sorted from leftmost to rightmost
% adjlist    - an nx6 connectivity list for each of the lenses, storing the
%              indicies of adjacent lenses of the SAME TYPE
% lgroups    - 3 cells containing the sets of all connected lenses
% ev         - epipolar vectors for system
% mask       - A mask that can be used to isolate micro-lens data
%
% Warnings: Code will produce warnings identifying points that deviated
% significantly from the expected location, based on the axis generated.
% These will occur more frequently with highly distorted views. Should
% check these points manually to confirm centroid location was good. If a
% lot of these show up, there may be issues near the centre of the image.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
warn = 0;
% robust input handling
if nargin == 0
    varargin={1 '' '' ''};
end
if ~isa(varargin{1},'string')
    imLoc='X:\01_Current_Students\Jake Hadfield\LightField Test
Images\2017_03_01\17_03_01_09L50HD\WhitIm.tiff';
else
    imLoc=varargin{1};
end
if ~isa(varargin{2},'double')
    radratio = 0.1; % default
else
    radratio = varargin{2};
end
```

```
if ~isa(varargin{3},'double')
    maxmiss = 2; % default
else
    maxmiss = round(varargin{3});
    if maxmiss <= 0
        error('Improper inputs, maxmiss must be a positive integer.');
        return;
    end
end
if ~isa(varargin{4},'double')
    thresh = 160; % default
    load('X:\01_Current_Students\Jake Hadfield\LightField Test Images\WhitIms\WhitAve_11-Sep-
2017.mat');
else
    thresh = varargin{4};
    if thresh <= 0
        error('Improper inputs, thresh must be a positive integer.');
        return;
    end
    imlist=dir([imLoc '\*.png']);
    whitAve=imread([imLoc imlist(1).name]);
    while 1
        imshow(im2bw(whitAve,thresh/255));
        button=questdlg('Are all circles disconnected?');
        if strcmp(button,'Yes')
            break;
        elseif strcmp(button,'Cancel')
            lensCents=[];
            return;
        else
            thresh=str2double(inputdlg('New threshold:'));
        end
    end
    imlist=dir([imLoc '\*.png']);
    for i=1:length(imlist)
        im(:,:,i)=imread([imLoc imlist(i).name]);
    end
    whitAve=mean(im,3)/255;
end
oops = 0;

global ipoints


% roughly find micro-lens centres via thresholding and region detection
bwim=im2bw(whitAve,thresh/255);
imshow(bwim)
props=regionprops(bwim,'centroid');
cnts=cat(1, props.Centroid);

% cnts=pkfnd(image,0,20);
```

```
%remove cnts of lenses on border of image

cnts(~((cnts(:,1)>10) .* (cnts(:,1)<2038) .* (cnts(:,2)>10) .* (cnts(:,2)<2038)),:)=[];

% try to do a better job finding centroids using cntrd
cnts=cntrd(double(whitAve),round(cnts),25);

imshow(whitAve)
hold on
scatter(cnts(:,1),cnts(:,2),'or')
hold off

% Find target axes
% find [x,y] epicentre of centroids
epi= sum(cnts(:,1:2))/length(cnts);

% put [x,y] centroids into a k-d tree
tree = KDTreeSearcher(cnts(:,1:2));

% figure out which centroid is closest to the epicentre
ind = knnsearch(tree,epi);

% set that point to be the origin
ipoints = [0 0 cnts(ind,1:2)];
% set lens type for first point
typelist=1;

% find next 6 nearest points to origin
ind = knnsearch(tree, cnts(ind,1:2), 'K', 7); % single row

% find opposing points
vecs = 2*bsxfun(@minus,cnts(ind,1:2),cnts(ind(1),1:2));
% toss out zeros (origin)
vecs(vecs(:,1)==vecs(:,2),:)=[];
indop = knnsearch(tree,bsxfun(@minus,cnts(ind(2:7),1:2),vecs)); % single column
% get opposites list, remove duplicates and origin
ops = [ind(2:7)' indop];
ops = unique(sort(ops,2),'rows');
% ops should be a 3x2 array with the indicies of points that oppose each other across the origin
check = size(ops);
if check(1)~=3 || check(2) ~=2
    oops = 2;
    ipoints = 0;
    return
end

% determine vector lengths between points in ops
vecs = cnts(ops(:,2),1:2)-cnts(ops(:,1),1:2);

% figure out which axes are good
p=zeros(1,7);
tries=[];
```

```
while all(sort(ind)~=sort(p))
    % most upward pointing vector becomes major axis direction
    majorAxis=vecs(abs(vecs(:,2))==max(abs(vecs(:,2))),:)/2;
    if majorAxis(:,2)<0
        majorAxis=-majorAxis;
    end
    % vector pointed most up and right becomes minor axis direction
    minorAxis=vecs((vecs(:,1)>0) & (vecs(:,2)<0) & ~(abs(vecs(:,2))==max(abs(vecs(:,2)))),:)/2;
    if isempty(minorAxis)
        minorAxis=-vecs(vecs(:,1)<0 .* vecs(:,2)<0 .*
~(abs(vecs(:,2))==max(abs(vecs(:,2)))),:)/2;
    end
    % make sure all 6 closest points to origin can be identified by single
    % combinations of the minor and major axis
    % origin always good
    p(1) = ind(1);
    % on minor
    p(2) = knnsearch(tree, cnts(ind(1),1:2)+minorAxis);
    p(3) = knnsearch(tree, cnts(ind(1),1:2)-minorAxis);
    % on major
    p(4) = knnsearch(tree, cnts(ind(1),1:2)+majorAxis);
    p(5) = knnsearch(tree, cnts(ind(1),1:2)-majorAxis);
    % last
    p(6) = knnsearch(tree, cnts(ind(1),1:2)+majorAxis+minorAxis);
    p(7) = knnsearch(tree, cnts(ind(1),1:2)-majorAxis-minorAxis);
    % try other axes if it didn't work
    if tries==2
        % didn't get 6 good points around origin
        oops = 1;
        ipoints = 0;
        return
    end
    tries=2;
end

% we now should have good major and minor axes
missflags = [];
% Determine coordinates of points in image along minor axis
p=1;
tlist=[1 2 3];
for runs=1:2
    % set up point finding inital loop variables
    findingpoints = 1;
    % which direction we're going
    switch runs
        case 1
            vector = minorAxis;
            c=1;
        case 2
            vector = -minorAxis;
            c=-1;
    end
    loc = cnts(ind(1),1:2);
    misses = 0;
    i=1;
    while findingpoints == 1;
        idx = rangesearch(tree,loc+vector,radratio*norm(vector));
        % did we find a point?
        if isempty(idx{1}) || length(idx)>1
            % if we didn't find one, we should keep going in case only a
            % few points are missing/blocked, but we need to flag the point
            % so we don't put it in the model target
            switch runs
                case 1
                    missflags = [missflags;i,0];
                case 2
                    missflags = [missflags;-i,0];
            end
            loc = loc+vector;
            misses = misses + 1;
            switch runs
                case 1
                    ipoints = [ipoints;[i,0,loc]];
                    if typelist(end)==3
                        typelist(end+1)=1;
                    else
                        typelist(end+1)=typelist(end)+1;
                    end
                case 2
                    ipoints = [ipoints;[-i,0,loc]];
                    if p
                        stag=length(typelist);
                        typelist(end+1)=3;
                    elseif typelist(end)==1
                        typelist(end+1)=3;
                    else
                        typelist(end+1)=typelist(end)-1;
                    end
                    p=0;
            end
        else
            vector = bsxfun(@minus,cnts(idx{1},1:2),loc);
            % compare with minor axis
            if norm(vector-c*minorAxis)>(radratio*norm(minorAxis)) && warn;
                switch runs
                    case 1
                        warning('Point (%i,%i) produced a vector
%1.2d*minorAxis.',i,0,norm(vector-c*minorAxis)/norm(minorAxis))
                    case 2
                        warning('Point (%i,%i) produced a vector %1.2d*minorAxis.',-
i,0,norm(vector-c*minorAxis)/norm(minorAxis))
                end
            end
            loc = cnts(idx{1},1:2);
            misses = 0;
            switch runs
                case 1
                    ipoints = [ipoints;[i,0,loc]];
                    if typelist(end)==3
                        typelist(end+1)=1;
                    else
                        typelist(end+1)=typelist(end)+1;
                    end
                case 2
                    ipoints = [ipoints;[-i,0,loc]];
                    if p
                        stag=length(typelist);
                        typelist(end+1)=3;
                    elseif typelist(end)==1
                        typelist(end+1)=3;
                    else
                        typelist(end+1)=typelist(end)-1;
                    end
                    p=0;
            end
        end
        i=i+1;
        % if we missed too many points, we probably hit the edge
        if misses == maxmiss
            findingpoints = 0;
        end
    end
end

% Determine locations of all points
% should have a line of points along the minor axis. Now, move in the major
% axis direction
[hits,~]=size(ipoints);
for j=1:hits
    for runs = 1:2
        % set up point finding inital loop variables
        findingpoints = 1;
        % which direction we're going
        switch runs
            case 1
                vector = majorAxis;
                c=1;
            case 2
                vector = -majorAxis;
                c=-1;
        end
        loc = ipoints(j,3:4);
        if runs==1;
            ntype=typelist(j)+1;
            if ntype==4; ntype=1; end
        else
```

```
            ntype=typelist(j)-1;
            if ntype==0; ntype=3; end
        end
        p=1;
        misses = 0;
        i=1;
        while findingpoints == 1;
            idx = rangesearch(tree, loc+vector,radratio*norm(vector));
            % did we find a point?
            if isempty(idx{1}) || length(idx)>1
                % if we didn't find one, we should keep going in case only a
                % few points are missing/blocked, but we need to flag the point
                % so we don't put it in the model target
                switch runs
                    case 1
                        missflags = [missflags;ipoints(j,1),i];
                    case 2
                        missflags = [missflags;ipoints(j,1),-i];
                end
                loc = loc+vector;
                misses = misses + 1;
                switch runs
                    case 1
                        ipoints = [ipoints;ipoints(j,1),i,loc];
                        if p
                            typelist(end+1)=ntype;
                            p=0;
                            q=1;
                        elseif q
                            typelist(end+1)=tlist(~ismember(tlist,[ntype typelist(j)]));
                            q=0;
                        elseif typelist(end)==3
                            typelist(end+1)=1;
                        else
                            typelist(end+1)=typelist(end)+1;
                        end
                    case 2
                        ipoints = [ipoints;ipoints(j,1),-i,loc];
                        if p
                            typelist(end+1)=ntype;
                            p=0;
                            q=1;
                        elseif q
                            typelist(end+1)=tlist(~ismember(tlist,[ntype typelist(j)]));
                            q=0;
                        elseif typelist(end)==1
                            typelist(end+1)=3;
                        else
                            typelist(end+1)=typelist(end)-1;
                        end
                end
            else
```

```
                  vector = bsxfun(@minus,cnts(idx{1},1:2),loc);
                  % compare with major axis. Throws more warnings than
                  % necessary, but good info.
                  if norm(vector-c*majorAxis)>(radratio*norm(majorAxis)) && warn;
                      switch runs
                          case 1
                              warning('Point (%i,%i) produced a vector
%1.2d*majorAxis.',ipoints(j,1),i,norm(vector-c*majorAxis)/norm(majorAxis))
                          case 2
                              warning('Point (%i,%i) produced a vector
%1.2d*majorAxis.',ipoints(j,1),-i,norm(vector-c*majorAxis)/norm(majorAxis))
                      end
                  end
                  loc = cnts(idx{1},1:2);
                  misses = 0;
                  switch runs
                      case 1
                          ipoints = [ipoints;ipoints(j,1),i,loc];
                          if p
                              typelist(end+1)=ntype;
                              p=0;
                              q=1;
                          elseif q
                              typelist(end+1)=tlist(~ismember(tlist,[ntype typelist(j)]));
                              q=0;
                          elseif typelist(end)==3
                              typelist(end+1)=1;
                          else
                              typelist(end+1)=typelist(end)+1;
                          end
                      case 2
                          ipoints = [ipoints;ipoints(j,1),-i,loc];
                          if p
                              typelist(end+1)=ntype;
                              p=0;
                              q=1;
                          elseif q
                              typelist(end+1)=tlist(~ismember(tlist,[ntype typelist(j)]));
                              q=0;
                          elseif typelist(end)==1
                              typelist(end+1)=3;
                          else
                              typelist(end+1)=typelist(end)-1;
                          end
                  end
              end

              i=i+1;
              % if we missed too many points, we probably hit the edge
              if misses == maxmiss
                  findingpoints = 0;
              end
```

```
        end
    end
end

% should have a full grid of points in ipoints, including misses. Now need
% to eliminate misses.
[~,crossed,~]=intersect(ipoints(:,1:2),missflags,'rows');
ipoints(crossed,:)=[];
typelist(crossed)=[];

% Model generation has been completed

% find ideal array centre and vectors

% initialize from origin, major and minor axes
x0=ipoints(1,3);
y0=ipoints(1,4);
v1x=minorAxis(1);
v1y=minorAxis(2);
v2x=majorAxis(1);
v2y=majorAxis(2);

initops=optimoptions('lsqnonlin','Algorithm','levenberg-marquardt','InitDamping',1000,'TolX',1e-
9,'MaxFunEvals',100000,'MaxIter',10000,'Display','none');
[result,resnorm,resids]=lsqnonlin(@optimfun,[x0,y0,v1x,v1y,v2x,v2y],[],[],initops);

x0=result(1);
y0=result(2);
v1x=result(3);
v1y=result(4);
v2x=result(5);
v2y=result(6);
lensCents=bsxfun(@plus,ipoints(:,1)*[v1x,v1y]+ipoints(:,2)*[v2x,v2y],[x0,y0]);

imshow(whitAve(1:500,1:500))
hold on
scatter(lensCents(typelist==1,1),lensCents(typelist==1,2),'or')
scatter(lensCents(typelist==2,1),lensCents(typelist==2,2),'og')
scatter(lensCents(typelist==3,1),lensCents(typelist==3,2),'ob')
scatter(lensCents(typelist==1,1),lensCents(typelist==1,2),'.r')
scatter(lensCents(typelist==2,1),lensCents(typelist==2,2),'.g')
scatter(lensCents(typelist==3,1),lensCents(typelist==3,2),'.b')
hold off

mrs=mean(resids)
strs=std(resids)
prc95=prctile(resids,95)
figure
histogram(resids);
xlabel('Deviation [pix]','FontSize',14,'FontName','Times New Roman')
ylabel('Count','FontSize',14,'FontName','Times New Roman')
set(gca,'FontSize',14,'FontName','Times New Roman')
```

```
% sort micro-lenses from left to right
isort=sortrows([lensCents typelist'],1);
lensCents=isort(:,1:2);
typelist=isort(:,3);

% create epipolar vectors for lens system
ev1=[v1x v1y]-[v2x v2y];
ev2=-[v1x v1y]+[v2x v2y];
ev3=-2*[v1x v1y]-[v2x v2y];
ev4=2*[v1x v1y]+[v2x v2y];
ev5=-[v1x v1y]-2*[v2x v2y];
ev6=[v1x v1y]+2*[v2x v2y];
ev=[ev1;ev2;ev3;ev4;ev5;ev6];

% create adjacency index list for micro-lenses; opposed pairs 1-2, 3-4, 5-6
adjlist=zeros(length(lensCents),6);
for i=1:length(lensCents)
    c1=find(ismembertol(lensCents,lensCents(i,:)+ev(1,:),0.001,'ByRows',1));
    c2=find(ismembertol(lensCents,lensCents(i,:)+ev(2,:),0.001,'ByRows',1));
    c3=find(ismembertol(lensCents,lensCents(i,:)+ev(3,:),0.001,'ByRows',1));
    c4=find(ismembertol(lensCents,lensCents(i,:)+ev(4,:),0.001,'ByRows',1));
    c5=find(ismembertol(lensCents,lensCents(i,:)+ev(5,:),0.001,'ByRows',1));
    c6=find(ismembertol(lensCents,lensCents(i,:)+ev(6,:),0.001,'ByRows',1));
    for j=1:6
        if isempty(eval(['c' num2str(j)]))
            evalc(['c' num2str(j) '=' num2str(NaN)]);
        end
    end
    adjlist(i,:)=[c1 c2 c3 c4 c5 c6];
end
% save everything
sloc=strfind(imLoc,'\');
save([imLoc(1:sloc(end)),'MLCal_171116'],'lensCents','adjlist','typelist','ev')

function resid=optimfun(vars)
global ipoints
x0=vars(1);
y0=vars(2);
v1x=vars(3);
v1y=vars(4);
v2x=vars(5);
v2y=vars(6);
lensCents=bsxfun(@plus,(ipoints(:,1)*[v1x,v1y]+ipoints(:,2)*[v2x,v2y]),[x0,y0]);
resid=sum((lensCents-ipoints(:,3:4)).^2,2);
```

Published with MATLAB® R2015b

C.2 *Wrapper function: A_RUN_JH_Plen_PTV*

The following code serves as a ‘wrapper’, running all other functions in the ETC code in loops as needed, while ensuring that all inputs and outputs reach the correct locations, and also saving the results. It is set up to allow the inputs to be included either in the first section of the code, or in the function when called.

```
function A_RUN_JH_Plen_PTV(Iroot,imNos,imFmt,path,mlCalLoc,whitIm,CalLoc,settings,display)
if nargin<9;display=0;end
if nargin < 8 || length(settings)~=11
    % Default Settings

    % Plenoptic Location
    invert=0; % wether or not to invert by subtracting the white image
    cleanup=60; % Divide by the given white image, with this minimum value
    lnoise=0; % Always should be zero, from what I've seen
    psize=5; % Particle size in pixels in raw images
    thresh=0; % cutoff threshold
    peak=.02*255; % peak threshold
    epitol=.5; % pixel distance for max distance from particle to epipolar line
    plotar=50; % aspect ratio of virtual depth for 3D plots
    validrange=[2,12]; % valid range of virtual depths
    bestest = 1; % controls if you want to pull out only the most-focused micro-lens type for
each particle
    xyrt=.1; % tolerance for determining if particles in xy are close enough to be the same
    zrt=2; % tolerance for determining if particles in z are close enough to be the same

    % particle tracking (units of mm)
    search = 1.2;
    neighbor = 3;
    quasi = .3;
    zdt = 0.4;

    % particle prediction
    range = 1;
    fp=5;

    % Track Rejection
    art = 2; % acceleration rejection threshold in mm/frame^2
    zaf = .4; % z-amplification factor, used to reduce z-direction components

    % linkage
    lzdt=0.4;

    % Track Smoothing
    % number of frames to use on either side of particle when smoothing
    % (depends on movement/frame >> more movement, less smoothing).
    % Generally want at least 5 and not more than 20.
    smoothstrength=10;

    plocsettings = [invert cleanup lnoise psize thresh peak epitol plotar validrange display];
else
    invert=settings(1); % wether or not to invert by subtracting the white image
    cleanup=settings(2); % Divide by the given white image, with this minimum value
    lnoise=settings(3); % Always should be zero, from what I've seen
    psize=settings(4); % Particle size in pixels in raw images
    thresh=settings(5); % cutoff threshold
    peak=settings(6); % peak threshold
    epitol=settings(7); % pixel distance for max distance from particle to epipolar line
    plotar=settings(8); % aspect ratio of virtual depth for 3D plots
    validrange=settings(9:10); % valid range of virtual depths
    bestest = settings(11); % controls if you want to pull out only the most-focused micro-lens
type for each particle
    xyrt=settings(12); % tolerance for determining if particles in xy are close enough to be the
same
    zrt=settings(13); % tolerance for determining if particles in z are close enough to be the
same
    search = settings(14);
    neighbor = settings(15);
    quasi = settings(16);
    range = settings(17);
    fp=settings(18);
    art = settings(19);
    zaf = settings(20);
    lzdt=settings(21);
    smoothstrength=settings(22);

    plocsettings = [invert cleanup lnoise psize thresh peak epitol maxz plotar validrange
display];
end

% DEFAULT CALIBRATION
% if nargin < 7
%     CalLoc = 'X:\01_Current_Students\Jake Hadfield\LightField Test
Images\CoreData_170810\WaterCal_170809\Calibration_11-Aug-2017.mat';
% end
% if nargin < 6
%     whitIm = 'X:\01_Current_Students\Jake Hadfield\LightField Test Images\WhitIms\WhitAve_11-
Sep-2017.mat';
% end
% if nargin < 5
%     mlCalLoc = 'X:\01_Current_Students\Jake Hadfield\LightField Test
Images\WhitIms\MLCal_170911.mat';
% end
% %% LOAD IMAGES
% if nargin < 4
%     path = 'X:\01_Current_Students\Jake Hadfield\LightField Test
Images\CoreData_170810\CoreView_6\';
% end
% if nargin < 3
%     Iroot = 'CoreView_6_Raytrix';
%     imNos = 150:250;
%     imFmt = '.png';
% end
if nargin < 7
    CalLoc = 'X:\01_Current_Students\Jake Hadfield\Coding\Matlab\Calibration
Code\Calibration_test.mat';
end
if nargin < 6
    whitIm = 'X:\01_Current_Students\Jake Hadfield\LightField Test
Images\2017_03_01\17_03_01_09L50HD\WhitAve_16-Nov-2017.mat';
end
if nargin < 5
    mlCalLoc = 'X:\01_Current_Students\Jake Hadfield\LightField Test
Images\2017_03_01\17_03_01_09L50HD\MLCal_171116.mat';
end
% LOAD IMAGES
if nargin < 4
    path = 'X:\01_Current_Students\Jake Hadfield\LightField Test
Images\2017_03_01\17_03_01_09L50HD\Raw\';
end
if nargin < 3
    Iroot = 'Image';
    imNos = 1:172;
    imFmt = '.tiff';
end
savedate=datestr(datetime('today'));

newdata=1; % Need to run particle-location code
if newdata

% RUN ALGORITHMS

% load calibration
load(CalLoc,'allvars')
caldata=[allvars(1:11) allvars(end)];

% Particle Location
ppoints=cell(1,length(imNos));
points=cell(1,length(imNos));
ltypes=cell(1,length(imNos));
resnorms=cell(1,length(imNos));
residvecs=cell(1,length(imNos));
pgroups=cell(1,length(imNos));
pstats=cell(1,length(imNos));
irejects=cell(1,length(imNos));
rrejects=cell(1,length(imNos));
% mmpoints=cell(length(imNos),3);
plist=[];
fprintf(['Finding particles. Completion time will be estimated soon. Current time is: '
datestr(now) '\n']);
ctime=now;
```

```
tic;
for i=1:length(imNos)
    imName = [Iroot '_' num2str(imNos(i),'%04i') imFmt];
    % find particles

[ppoints{i},ltypes{i},resnorms{i},residvecs{i},pstats{i},~,~,pgroups{i},~,~,~]=JH_rawPlenPF(path,
imName,[Iroot '_PixPoints'],mlCalLoc,whitIm,CalLoc,plocsettings);

    % convert to physical space
    for j=1:3
        mmpoints{i,j} = JH_PhysConvert(ppoints{i}(ltypes{i}==j,:),[caldata(1) caldata(j+1)
caldata(5:end)]);
    end
    points{i}=[mmpoints{i,1};mmpoints{i,2};mmpoints{i,3}];


%points{i}=[ppoints{i}(ltypes{i}==1,:);ppoints{i}(ltypes{i}==2,:);ppoints{i}(ltypes{i}==3,:)];

    % keep best estimates only
    if bestest

[irejects{i},rrejects{i}]=JH_kbe(points{i},[resnorms{i}(ltypes{i}==1,:);resnorms{i}(ltypes{i}==2,
:);resnorms{i}(ltypes{i}==3,:)],pstats{i},[pgroups{i}(ltypes{i}==1,:);pgroups{i}(ltypes{i}==2,:);
pgroups{i}(ltypes{i}==3,:)],xyrt,zrt);
    end
    rrcrit=1;
    ircrit=0;
    if rrcrit==1
        points{i}(rrejects{i},:)=[];
    elseif ircrit==1
        points{i}(rrejects{i},:)=[];
    end
    plist=[plist;[points{i} ones(length(points{i}),1)*i]];
    tt=toc;
    fprintf(['Particle location should be done around ' datestr(ctime+tt/i*(length(imNos))/84600)
'\n']);
end
save([path 'PList_' savedate '_' Iroot '.mat'],'plist')
save([path 'Points_' savedate '_' Iroot '.mat'],'points')

else
load([path 'PList_' savedate '_' Iroot '.mat'],'plist')
load([path 'Points_' savedate '_' Iroot '.mat'],'points')
end

newtrack=1;
if newtrack
alltracks=[];
polyhold.pno=[];
polyhold.polyx=[];
polyhold.polyy=[];
polyhold.polyz=[];
polyhold.mux=[];
polyhold.muy=[];
polyhold.muz=[];
mp=[];
ctime=now;
tic;
for i=1:length(imNos)-1
    if i>=3
        fprintf('Attempting forward-predictive tracking on %i frame %i and %i frame %i
particles...\n',length(points{i}),i,length(points{i+1}),i+1);

[rp1,rp2,alltracks,polyhold]=JH_MT_Fit_Predict(points{i},points{i+1},alltracks,i,range,zdt,polyho
ld,search);
        fprintf('Attempting two-frame tracking on remaining %i frame %i and %i frame %i
particles...\n',length(rp1),i,length(rp2),i+1);
        tracks = DH_track_v3(rp1,rp2,search,neighbor,quasi,zdt);
        if ~isempty(tracks)
            nmp=rp2(~ismember(rp2,tracks(:,1:3),'rows'),:);
            [lnmp,~]=size(nmp);
            mp=[mp;nmp ones(lnmp,1)*i];
            alltracks=JH_MT_Associate(tracks,alltracks,i);
        end
    else
        fprintf('Attempting two-frame tracking on %i frame %i and %i frame %i
particles...\n',length(points{i}),i,length(points{i+1}),i+1);
        tracks = DH_track_v3(points{i},points{i+1},search,neighbor,quasi,zdt);
        if ~isempty(tracks)
            nmp=points{i+1}(~ismember(points{i+1},tracks(:,1:3),'rows'),:);
            [lnmp,~]=size(nmp);
            mp=[mp;nmp ones(lnmp,1)*i];
            alltracks=JH_MT_Associate(tracks,alltracks,i);
        end
    end
    tt=toc;
    fprintf(['Particle tracking should be done around ' datestr(ctime+tt/i*(length(imNos))/84600)
'\n']);
end
% eliminate remaining predicted points
alltracks(alltracks(:,6)==0,:)=[];
save([path 'AllTracks_' savedate '_' Iroot '.mat'],'alltracks')
end


% get track stats
JH_TrackStats(alltracks)

% denoise tracks
lostpts=double.empty(0,5);
nlostpts=0;
while ~isempty(nlostpts)
    [filttracks,nlostpts,~]=JH_MT_TrackBreak(alltracks,art,zaf);
    alltracks=filttracks;
    lostpts=[lostpts;nlostpts];
end
nmp=lostpts(:,[1:3,5]);
save([path 'BreakTracks_' savedate '_' Iroot '.mat'],'filttracks')
% resort tracks
[filttracks,misspts]=JH_resortTracks(filttracks);

% get track stats
JH_TrackStats(filttracks)

mp=sortrows([mp;nmp;misspts],4);

save([path 'FilTracks_' savedate '_' Iroot '.mat'],'filttracks')
save([path 'MissPts_' savedate '_' Iroot '.mat'],'mp')

linkcount=1;
while linkcount~=0
% link tracks
[linktracks,linkcount]=JH_LinkTracks(filttracks,range,lzdt,fp);
filttracks=linktracks;
% resort tracks
[linktracks,~]=JH_resortTracks(linktracks);
end
% get track stats
JH_TrackStats(linktracks)

% find longest linked track
long=1;
lc=1;
tl=0;
for i=1:max(linktracks(:,4))
    nlc=sum(linktracks(:,4)==i);
    if nlc>lc
        long=i;
        lc=nlc;
    elseif nlc==lc
        trak=linktracks(linktracks(:,4)==1,:);
        ntl=sum(sqrt(sum(diff(trak(:,1:2)).^2,2)));
        if ntl>tl
            long=i;
            tl=ntl;
        end
    end
end

link=linktracks(linktracks(:,4)==long,:);

% smoothing algorithm
smoothtracks = JH_SmoothTracks(linktracks,smoothstrength);

smooth=smoothtracks(smoothtracks(:,4)==long,:);
```

```
% get track stats
JH_TrackStats(smoothtracks)

% % find domain limits
% minx=min(alltracks(:,1));
% miny=min(alltracks(:,2));
% minz=min(alltracks(:,3));
% maxx=max(alltracks(:,1));
% maxy=max(alltracks(:,2));
% maxz=max(alltracks(:,3));
% % save data
% save([path 'Limits_170111_' Iloc '.mat'],'minx','miny','minz','maxx','maxy','maxz')
save([path 'LinkTracks_' savedate '_' Iroot '.mat'],'linktracks')
save([path 'SmoothTracks_' savedate '_' Iroot '.mat'],'smoothtracks')
save([path 'SLink_' savedate '_' Iroot '.mat'],'link')
save([path 'SSmooth_' savedate '_' Iroot '.mat'],'smooth')
JH_TrackToQuiv(path,['AllTracks_' savedate '_' Iroot '.mat']);
JH_TrackToQuiv(path,['SmoothTracks_' savedate '_' Iroot '.mat']);
```

Published with MATLAB® R2015b

C.3 *Particle location function: JH_rawPlenPF*

The following function locates the particles in the raw plenoptic images and applies the ETC criterion, then applies triangulation to get the 3D locations.

```
function
[points,ltypes,resnorms,residvecs,pstats,pcd,plocs,pgroups,lensCents,pli,typelist]=JH_rawPlenPF(p
ath,imName,saveName,calLoc,whitIm,mlCalLoc,settings)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Finds particles in a raw plenoptic image, organizing them by grouping
% together the multiple micro-image particle locations
% Inputs: imLoc-     The location of the image to be processed
%         lensCents-  The location of the lens centroids from micro-lensCal
%         settings-   Processing settings for the particle-location
%                     algorithms
% Outputs: ploc3D-    A nx3 array containing the [x,y] location of each
%                     particle; the third column contains the particle number
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
close all
% defaults if missing inputs
if nargin < 7
    invert=0; % wether or not to invert by subtracting the white image
    cleanup=0; % Divide by the given white image, with this minimum value
    lnoise=0; % Always should be zero, from what I've seen
    psize=3; % Particle size in pixels in raw images
    thresh=0; % cutoff threshold
    peak=20; % peak threshold
    epitol=.5; % pixel distance for max distance from particle to epipolar line
    plotar=50; % aspect ratio of virtual depth for 3D plots
    validrange=[2,16]; % valid range of virtual depths
    display=1;
else
    invert=settings(1); % wether or not to invert by subtracting the white image
    cleanup=settings(2); % Divide by the given white image, with this minimum value
    lnoise=settings(3); % Always should be zero, from what I've seen
    psize=settings(4); % Particle size in pixels in raw images
    thresh=settings(5); % cutoff threshold
    peak=settings(6); % peak threshold
    epitol=settings(7); % pixel distance for max distance from particle to epipolar line
    plotar=settings(8); % aspect ratio of virtual depth for 3D plots
    validrange=settings(9:10); % valid range of virtual depths
    display=settings(11); % input 1 to display stuff
end
if nargin <6
    mlCalLoc='';
end
if nargin <5
    whitIm='X:\01_Current_Students\Jake Hadfield\LightField Test Images\WhitAve_28-Jul-2017.mat';
end
if nargin < 4
    load('X:\01_Current_Students\Jake Hadfield\LightField Test Images\17_06_21\MLCal.mat')
else
    load(calLoc)
end
if nargin < 3
    path='X:\01_Current_Students\Jake Hadfield\LightField Test Images\FF_170712\Helix40\';
    imName='Image_Raw_0065.png';
    saveName='Image_Points';
end

% Load in image, run bpass, pkfnd and cntrd to get particle locations
if invert
    load(whitIm)
    whitAveG=whitAve;
    im=im2double(imread([path,imName]));
    image=whitAve-im/max(max(im))*1;
    image(image<0)=0;
elseif cleanup
    load(whitIm)
    whitAve=im2double(whitAve);
    image=imread([path,imName]);
    if size(image,3)==3
        image=rgb2gray(image);
        image=image(1:2048,1:2048);
    end
    image=im2double(image)./(((whitAve-min(min(whitAve)))./(max(max(whitAve))-
min(min(whitAve))).*(1-cleanup/255)+cleanup/255));
    image=image./max(max(image));
else
    image=im2double(imread([path,imName]));
```

```
end
bpim=bpass(image,lnoise,psize,thresh);
plocs=cntrd(bpim,pkfnd(bpim,peak/255,psize+2),psize);
if display==1
    imshow(1.5*image)
    hold on
    plot(plocs(:,1),plocs(:,2),'+r','markers',15,'linewidth',2)
    drawnow
%     pl1=[lensCents(typelist==1,1)-12,lensCents(typelist==1,2)-
12,23*ones(sum(typelist==1),1),23*ones(sum(typelist==1),1)];
%     pl2=[lensCents(typelist==2,1)-12,lensCents(typelist==2,2)-
12,23*ones(sum(typelist==2),1),23*ones(sum(typelist==2),1)];
%     pl3=[lensCents(typelist==3,1)-12,lensCents(typelist==3,2)-
12,23*ones(sum(typelist==3),1),23*ones(sum(typelist==3),1)];
%     for i=1:length(pl1)
%         rectangle('position',pl1(i,:),'curvature',[1 1],'EdgeColor','r','linewidth',2)
%     end
%     for i=1:length(pl2)
%         rectangle('position',pl2(i,:),'curvature',[1 1],'EdgeColor','g','linewidth',2)
%     end
%     for i=1:length(pl3)
%         rectangle('position',pl3(i,:),'curvature',[1 1],'EdgeColor','c','linewidth',2)
%     end
    xlabel('{\itx} [pix]','FontName','Times New Roman','FontSize',20)
    ylabel('{\ity} [pix]','FontName','Times New Roman','FontSize',20)
    set(gca,'outerposition',[0 0 1 1],'fontname','Times New Roman','fontsize',20)
%     ylim([1725.8 1749+23*3])
%     xlim([1704 1810])
end

pstats=plocs(:,3:4);
plocs=plocs(:,1:2);

% determine which micro-lens each particle is in (closest lens centre)
[pll,~]=size(plocs);
pcd=zeros(pll,1);
pli=zeros(pll,1);
% should map pixels to lenses ahead of time!!!! faster
for i=1:pll
    [pcd(i),pli(i,:)]=min(sqrt(sum(bsxfun(@minus,lensCents,plocs(i,:)).^2,2)));
end

% if not calibrating, modify particle location based on calibration
% if ~strcmp(mlCalLoc,'')
%     load(mlCalLoc);
%     mlVars=allvars(12:18);
%     plocs=JH_Plen_mlCorrect(plocs,lensCents,pli,typelist,pcd,mlVars);
% end

% scan through particle list, number similar particles
pcount=length(plocs);
plist=1:pcount;
```

```
pgroups=cell(1,1);
ped=cell(pcount,1);
pgepd=cell(1,1);
pparray=cell(pcount,1);
% for each particle, identify epipolar connections
for cpart=plist
    clens=pli(cpart);
    % look at adjacent lenses
    adjlenses=adjlist(clens,:);
    adjset=[];
    for lens=adjlenses
        % what particles are in each adjacent micro-lens?
        pnew=find(pli==lens);
        if ~isempty(pnew)
            adjset=[adjset;lens*ones(length(pnew),1),pnew];
        end
    end
    % can skip the rest if there are no particles in adjacent lenses
    if isempty(adjset)
        continue
    end

    % which particles are along an epipolar line from the parent
    % particle?
    for p=adjset(:,2)'
        lens = adjset(adjset(:,2)==p,1);
        lcheck=(lens==adjlenses)';
        if any(lcheck)
            epid=ptl(plocs(p,:),plocs(cpart,:),ev(lcheck,:));
            if epid<epitol
                % add particle to potential array
                pparray{cpart,:}=[pparray{cpart,:},p];
                ped{cpart,:}=[ped{cpart,:},epid];
            end
        end
    end
end
% want to find triangular epipolar connections between particles
triangles=[];
tepd=[];
for p=1:size(pparray,1)
    lens=pli(p);
    for j=pparray{p}
        o2=[];
        cl=ismember(pparray{p},pparray{j});
        t3=pparray{p}(cl);
        el=~ismember(pli(t3),lens);
        t3=t3(el);
        if ~isempty(t3)
            ntri=[ones(length(t3),1)*[p j],t3'];
            triangles=[triangles;ntri];
            for k=ntri'
```

```
                o2=[o2;ped{j}(k(3)==pparray{j}),ped{p}(k(3)==pparray{p}),ped{p}(j==pparray{p})];
            end
            tepd=[tepd;o2];
        end
    end
end
[triangles,ind]=sort(triangles,2);
ltp=length(tepd);
temp=tepd';
tepd=temp(ind+[[0:3:3*ltp-3]',[0:3:3*ltp-3]',[0:3:3*ltp-3]']);
[triangles,ia]=unique(triangles,'rows');
tepd=tepd(ia,:);

% plot triangles
if display==1
    for i=1:size(triangles,1)
        line([plocs(triangles(i,1),1)
plocs(triangles(i,2),1)],[plocs(triangles(i,1),2),plocs(triangles(i,2),2)],'Color','m');
        line([plocs(triangles(i,2),1)
plocs(triangles(i,3),1)],[plocs(triangles(i,2),2),plocs(triangles(i,3),2)],'Color','m');
        line([plocs(triangles(i,1),1)
plocs(triangles(i,3),1)],[plocs(triangles(i,1),2),plocs(triangles(i,3),2)],'Color','m');
    end
    drawnow
    hold off
end

% with triangular connections established, separate sets using connectivity
group=1;
vparts=triangles(1,:);
ovparts=[];
epdstack=tepd(1,:);
while ~isempty(triangles)
    acond=any(ismember(triangles,vparts),2);
    nvparts=triangles(acond,:);
    nepd=tepd(acond,:);
    vparts=[vparts;nvparts];
    epdstack=[epdstack;nepd];
    [vparts,ulocs]=unique(vparts,'rows');
    epdstack=epdstack(ulocs,:);
    if size(vparts,1)==size(ovparts,1)
        pgroups{group,1}=vparts;
        pgepd{group,1}=epdstack;
        if size(pgroups{group,1},2)>1
            pgroups{group,1}=pgroups{group,1}';
            pgepd{group,1}=pgepd{group,1}';
        end
        group=group+1;
        bcond=~ismember(triangles,vparts,'rows');
        triangles=triangles(bcond,:);
        tepd=tepd(bcond,:);
        if ~isempty(triangles)
```

```
            vparts=triangles(1,:);
            epdstack=tepd(1,:);
            ovparts=[];
        end
    else
        ovparts=vparts;
    end
end

% adjust particle locations in connected sets to improve epipolar
% connectivity

for i=1:group-1
%     % look for particle images that have only been captured once
%     [~,ia,~]=unique(pgroups{i});
%     pgsolo=pgroups{i}(~ismember(pgroups{i},pgroups{i}(~ismember(1:numel(pgroups{i}),ia))));
%
%     % these locations are 'safe' to manipulate to improve the epipolar
%     % connections, because they're on the edge of the groups. Only want
%     % to manipulate if their opposed epipolar line
%     % is better than both the connected epipolar lines.
%     ignorep=[];
%     for p=pgsolo'
%         % eliminate locations from the group if they're a major source of
%         % error
%         tritag=pgepd{i}(p==pgroups{i})==min(pgepd{i});
%         if any(tritag)
%             if pgepd{i}(p==pgroups{i})/sum((pgepd{i}(:,sum(p==pgroups{i})==1)))<0.1
%                 % point is a big source of error, tag to be ignored.
%                 ignorep=[ignorep;p];
%             end
%         end
%     end
%     if ~isempty(ignorep)
%         pgroups{i}(ismember(pgroups{i},ignorep))=[];
%     end
    pgroups{i}=unique(pgroups{i});
end

% having separated the sets, now ready to perform 3D particle location
[points,ltypes,resnorms,residvecs]=JH_Plen_GetPt(plocs,pgroups,pgepd,lensCents,pli,pcd,typelist,d
isplay);

% remove all data for points with a depth outside the valid range
[points,ltypes,resnorms,residvecs]=JH_PlenClean(points,ltypes,resnorms,residvecs,validrange);

if display==1
    set1=ltypes==1;
    set2=ltypes==2;
    set3=ltypes==3;
    scatter3(points(set1,1),points(set1,2),points(set1,3),200,'or')
    scatter3(points(set2,1),points(set2,2),points(set2,3),200,'og')
    scatter3(points(set3,1),points(set3,2),points(set3,3),200,'ob')
    xlim([-10,2060])
    ylim([-10,2060])
    zlim([0,validrange(2)])
    axis ij
    daspect([1,1,1/plotar])
    hold off
%     ylim([1725.8 1749+23*3])
%     xlim([1704 1810])
%     zlim([0 4.5])
    xlabel('{\itx} [pix]','FontName','Times New Roman','FontSize',20)
    ylabel('{\ity} [pix]','FontName','Times New Roman','FontSize',20)
    zlabel('{\itz} [pix]','FontName','Times New Roman','FontSize',20)
    set(gca,'outerposition',[0 0 1 1],'fontname','Times New Roman','fontsize',20)
    grid on
end

fprintf('%i points found, with an average residual of %d, SD %d, max
%d\n',size(points,1),mean(resnorms),std(resnorms),max(resnorms))
if ~isempty(saveName)
    save([path,saveName],'points','ltypes','resnorms','residvecs','pgroups','pstats')
end
end

function d=ptl(p, pc, v)
d=abs(det([p-pc;v]))/norm(v);
end
```

Published with MATLAB® R2015b

C.4 *Triangulation function: JH_Plen_GetPt*

The following code performs the triangulation method conducted based on the ETC groups.

```
function [points,ltypes,resnorms,
residvecs]=JH_Plen_GetPt(plocs,pgroups,pgepd,lensCents,pli,pcd,typelist,display)
global pcdsend
global setvecs
global disvecs
% global pgepdsend
% global iccount
gcount=size(pgroups,1);
points=zeros(gcount,3);
resnorms=zeros(gcount,1);
residvecs=cell(gcount,1);
ltypes=zeros(gcount,1);
if display==1
    figure(2)
    hold on
```

```
end
for pset=1:gcount
    % create a vector from the centre of each particle's micro-lens to the
    % particle, in 3D. Value of the z-component is -1, so results will be in
    % virtual depth units. Vectors are in [x1,x2,y1,y2,z1,z2] format
%     [pgroups{pset,1},ia,ic]=unique(pgroups{pset,1});

setvecs=[plocs(pgroups{pset,1},1),plocs(pgroups{pset,1},2),ones(length(pgroups{pset,1}),1),...

lensCents(pli(pgroups{pset,1}),1),lensCents(pli(pgroups{pset,1}),2),zeros(length(pgroups{pset,1})
,1)];
    pcdsend=pcd(pgroups{pset,1});
    % apply lens types to points
    ltypes(pset)=typelist(pli(pgroups{pset,1}(1)));
    % create an initial point
    pinit=[mean(setvecs(:,1:2),1),4];
    % use least-squares optimization to minimize the distance from the
    % point to all the lines by moving the point
    initops=optimoptions('lsqnonlin','Algorithm','levenberg-
marquardt','InitDamping',1000,'TolFun',1e-9,'TolX',1e-
9,'MaxFunEvals',100000,'MaxIter',10000,'Display','none');
    [points(pset,:),~]=lsqnonlin(@optimfun,pinit,[],[],initops);
    % plot stuff
    if display==1
        for i =1:size(setvecs,1)
            switch ltypes(pset)
                case 1
                    col='r';
                case 2
                    col='g';
                case 3
                    col='b';
            end
            line([setvecs(i,4) setvecs(i,4)+(1+points(pset,3))*(setvecs(i,1)-
setvecs(i,4))],[setvecs(i,5) setvecs(i,5)+(1+points(pset,3))*(setvecs(i,2)-
setvecs(i,5))],[setvecs(i,6) setvecs(i,6)+(1+points(pset,3))*(setvecs(i,3)-
setvecs(i,6))],'Color',col);
        end
    end
    residvecs{pset,1}=disvecs;
    resnorms(pset)=sum(sqrt(sum(residvecs{pset,1}.^2,2)));
end
end

function resid=optimfun(point)
% calculate vector distances from point to all lines
global setvecs
global disvecs
% global pcdsend
% global pgepdsend
% global iccount
svs=size(setvecs,1);
```

```
disvecs=ones(svs,3);
ndv=ones(svs,3);
for i =1:size(setvecs,1)
    disvecs(i,:)=setvecs(i,4:6)+dot(-setvecs(i,4:6)+point,(-setvecs(i,4:6)+setvecs(i,1:3))/norm(-
setvecs(i,4:6)+setvecs(i,1:3)))*(-setvecs(i,4:6)+setvecs(i,1:3))/norm(-
setvecs(i,4:6)+setvecs(i,1:3))-point;
    ndv(i,:)=disvecs(i,:);%./(pgepdsend(i)/iccount(i)/sum(pgepdsend)); % normalizes based on how
good the epipolar triangles are
end
resid=sqrt(ndv(:,1:3).^2)/size(ndv,1);
end

function d=ptl(p, pc, v)
d=abs(det([p-pc;v]))/norm(v);
end
```

Published with MATLAB® R2015b

C.5 *Point rejection function: JH_PlenClean*

This code performs some basic filtering to remove particles that are out of the virtual depth range or are too close to the edge of the image to have been reliably located

```
function
[points,ltypes,resnorms,residvecs]=JH_PlenClean(points,ltypes,resnorms,residvecs,validrange)
if ~isempty(validrange)
    ltypes(points(:,3)<validrange(1)|points(:,3)>validrange(2),:)=[];
    resnorms(points(:,3)<validrange(1)|points(:,3)>validrange(2),:)=[];
    residvecs(points(:,3)<validrange(1)|points(:,3)>validrange(2),:)=[];
    points(points(:,3)<validrange(1)|points(:,3)>validrange(2),:)=[];
end

% remove all points that are too close (within 1 micro-lens distance) to the
% edge

ltypes(points(:,1)<25|points(:,1)>2024|points(:,2)<25|points(:,2)>2024,:)=[];
resnorms(points(:,1)<25|points(:,1)>2024|points(:,2)<25|points(:,2)>2024,:)=[];
residvecs(points(:,1)<25|points(:,1)>2024|points(:,2)<25|points(:,2)>2024,:)=[];
points(points(:,1)<25|points(:,1)>2024|points(:,2)<25|points(:,2)>2024,:)=[];
```

Published with MATLAB® R2015b

C.6 *Point filtering function: JH_kbe*

The following code aims to determine which particle locations produced by different micro-lenses correspond to the same particle, and passes back the rejected locations based on intensity and triangulation residuals.

```
function [irejects,rrejects]=JH_kbe(points,resnorms,pstats,pgroups,xyrt,zrt)
spcell={};
irejects=[];
rrejects=[];
% first look for particles with similar (x,y,z) locations
for i=1:size(points,1)

spts=ismembertol(points(:,1:2),points(i,1:2),xyrt,'DataScale',1,'ByRows',1)&ismembertol(points(:,
3),points(i,3),zrt,'DataScale',1,'ByRows',1);
    spts(i,:)=0;
    added=0;
    if any(spts)
        npts=[i;find(spts)];
        if isempty(spcell)
            spcell{length(spcell)+1,1}=npts;
        else
            for j=1:size(spcell,1)
                if ismember(spcell{j},npts)
                    spcell{j}=unique([spcell{j};npts]);
                    added=1;
                    break;
                end
            end
            if ~added
                spcell{length(spcell)+1,1}=npts;
            end
        end
    end
end

% grab the initial particle numbers for each member of each spcell,
% return to the particle stats to get intensity data. Add worst
% intensities to the intensity reject array. Also, look at residuals
% for each particle
for j=1:length(spcell)
    bpimean=zeros(size(spcell{j}));
    bprmean=zeros(size(spcell{j}));
    for k=1:size(spcell{j})
        try
        bpimean(k)=mean(pstats(pgroups{spcell{j}(k)}));
        catch
            pause
        end
        bprmean(k)=mean(resnorms(spcell{j}(k)));
    end
    [~,bpii]=max(bpimean);
    [~,bpri]=min(bprmean);
    irejects=[irejects;spcell{j}(1:end~=bpii)];
    rrejects=[rrejects;spcell{j}(1:end~=bpri)];
end
irejects=sort(unique(irejects));
rrejects=sort(unique(rrejects));
```

Published with MATLAB® R2015b

Appendix D CALIBRATION CODES

The calibration code relies on using either of the two particle location methods to find the pixel and virtual coordinates of the target points for each lens type, and a separate version with similar structure is included here for each location method.

D.1 *Refocusing calibration: JH_RayCam_Calibration*

The following code is the calibration method applied with the refocusing method:

```
function JH_RayCam_Calibration
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calibrates a camera using Matlab's calibration software and a few other
% things
% Inputs:
% calimages - Raytrix-generated total focus images to be used for
% calibration
% depthmaps - Raytrix-generated depth maps, one per cal image
% drt       - depth reject threshold, depth map needs more calculated
%             points than this per grid point, or grid point is rejected
% spacing   - centre-to-centre distance between adjacent cal grid points
% Based on Christian Heinze's 2014 masters thesis
newrun = 1;
runs = 1;
clear global
tic
global impoints;
global modelpoints;
global iters
iters=0;
global noIms
global pass

% IMAGE LOCATIONS
% file path to images
path = 'X:\01_Current_Students\Jake Hadfield\LightField Test Images\2017_02_24_Cal\';
% Shared tag, before _Focus_Depth or _Depth
imID = 'Image';
% List of image numbers in a 1 by n array
imNos = [1:8];
noIms = length(imNos);
% file format
fmt = '.tiff';

% convert image numbers to n strings
imStr = repmat('0000',noIms,1);
for i=1:noIms
    imStr(i,:)=num2str(imNos(i),'%04i');
```

```
end

% Default Settings
% centroid detection properties, in pixels
pkthresh = 30;
dotpix = 49; % should be odd, slightly larger than dot on target
noisescale = 3;
noisethresh = 10;

% depth detection values, in pixels
drt = 10; % required number of depth estimations to keep a point
dfrt = 0.3; % depth fit rejection threshold, virtual depth units

% model generation stuff
radratio = 0.1;

% target model values, in mm
dotsize = 4.953/3;
dotspacing = 4.953;

% Iterative solver initial parameter values

% Camera Constants
global Rx
global Ry
global spix

spix = 5.5e-3;% pixel size [mm/pix]
Rx = 1024; % image (sensor?) resolution x [pix]
Ry = 1024; % image (sensor?) resolution y [pix]

% Calibration Variables
try
    load('nope','allvars')
    % main lens variables
    fL = allvars(end); % focal length of main lens [mm]
    tL = allvars(1); % focus distance [mm]

    % micro-lens variables
    b1= allvars(2); % MLA-CCD distance [mm], far field lenses
    b2= allvars(3); % MLA-CCD distance [mm], mid field lenses
    b3= allvars(4); % MLA-CCD distance [mm], near field lenses

    % distortion variables
    x0 = allvars(5); % distortion offset x from corner [pix] relative to image centre
    y0 = allvars(6); % distortion offset y from corner [pix] relative to image centre
    k1 = allvars(7); % radial distortion coeff 1
    k2 = allvars(8); % radial distortion coeff 2
    d1 = allvars(9); % depth distortion coeff 1
    d2 = allvars(10); % depth distortion coeff 2
    dD = allvars(11); % depth-dependant depth distortion coeff
catch
```

```
    % main lens variables
    fL = 85; % focal length of main lens [mm]
    tL = 750; % focus distance [mm]

    % micro-lens variables
    b1= 0.3; % MLA-CCD distance [mm], far field lenses
    b2= 0.3; % MLA-CCD distance [mm], mid field lenses
    b3= 0.3; % MLA-CCD distance [mm], near field lenses

    % distortion variables
    x0 = 0; % distortion offset x from corner [pix] relative to image centre
    y0 = 0; % distortion offset y from corner [pix] relative to image centre
    k1 = 0; % radial distortion coeff 1
    k2 = 0; % radial distortion coeff 2
    d1 = 0; % depth distortion coeff 1
    d2 = 0; % depth distortion coeff 2
    dD = 0; % depth-dependant depth distortion coeff
end

% Extrinsics
% Translations
Tx = 0;
Ty = 0;
Tz = tL;

% Quaternion Rotation
Q1 = 0;
Q2 = 0;
Q3 = 0;
Q4 = 1;

global defaults
defaults=[fL,tL,b1,b2,b3,x0,y0,k1,k2,d1,d2,dD,Tx,Ty,Tz,Q1,Q2,Q3,Q4];

% POINT DETECTION LOOP
% set up figures
cnts=cell(noIms,1);
for i=1:noIms
    cntz=[];
    % load calibration image, invert
    calIm = 255-imread([path imID '_' imStr(i,:) '_Focus_Depth'  fmt]);

    % FILTER IMAGE
    % IMAGE : Threshold : size of particle (longer)
    calIm = bpass(calIm,noisescale,dotpix,noisethresh);

    % FIND PARTICLES
    % IMAGE : Image Threshold : size of particle (longer)
    pk = pkfnd(calIm,pkthresh,dotpix);

    % Do again to sub-pixel
    cnt = cntrd(calIm,pk,dotpix);
    %Plot detected centroids
%         figure
%         imagesc(calIm)
%         axis equal
%         colormap gray
%         hold on
%         plot(cnt(:,1),cnt(:,2),'r+','MarkerSize',20)
%         hold off
%         uiwait(gcf)
    %
    % Should probably filter based on dot size, but cntrd can't right now.
    % Could also filter by intensity.

    % Add in the depth data
    for j=1:3
        switch j
            case 1
                depthIm = imread([path imID '_' imStr(i,:) '_Depth_Far' fmt]);
            case 2
                depthIm = imread([path imID '_' imStr(i,:) '_Depth_Mid' fmt]);
            case 3
                depthIm = imread([path imID '_' imStr(i,:) '_Depth_Near' fmt]);
        end
        cntz(:,j)=JH_AssociateDepthData_V3(cnt,depthIm,dotpix*dotspacing/dotsize/3,drt,0);
    end
    farnans=sum(isnan(cntz(:,1)));
    midnans=sum(isnan(cntz(:,2)));
    nearnans=sum(isnan(cntz(:,3)));
    fprintf('%i far, %i mid and %i near depths had too few estimations in image
%i\n',farnans,midnans,nearnans,i)
    % filter out bad depths
    cntfar=[cnt(:,1:2) cntz(:,1)];
    cntmid=[cnt(:,1:2) cntz(:,2)];
    cntnear=[cnt(:,1:2) cntz(:,3)];
    cntfar(isnan(cntfar(:,3)),:)=[];
    cntmid(isnan(cntmid(:,3)),:)=[];
    cntnear(isnan(cntnear(:,3)),:)=[];
    badfar=depthfilter(cntfar,dfrt);
    badmid=depthfilter(cntmid,dfrt);
    badnear=depthfilter(cntnear,dfrt);
    cnts{i}=[cnt(:,1:2) cntz];
    cnts{i}(badfar,3)=NaN;
    cnts{i}(badmid,4)=NaN;
    cnts{i}(badnear,5)=NaN;
    fprintf('%i far, %i mid and %i near depths rejected by fit criteria in image
%i\n',length(badfar),length(badmid),length(badnear),i)
%     figure(1)
%     scatter3(cntfar(:,1),cntfar(:,2),cntfar(:,3),'ro')
%     hold on
%     scatter3(cntfar(badfar,1),cntfar(badfar,2),cntfar(badfar,3),'k+')
%     hold off
%     figure(2)
%     scatter3(cntmid(:,1),cntmid(:,2),cntmid(:,3),'go')
%     hold on
%     scatter3(cntmid(badmid,1),cntmid(badmid,2),cntmid(badmid,3),'k+')
%     hold off
%     figure(3)
%     scatter3(cntnear(:,1),cntnear(:,2),cntnear(:,3),'bo')
%     hold on
%     scatter3(cntnear(badnear,1),cntnear(badnear,2),cntnear(badnear,3),'k+')
%     hold off
%     drawnow
%     uiwait(gcf)
%     close all
end
% now we have all the centroids in pixel and virtual depth space

% GENERATE TARGET MODEL
impoints = cell(noIms,1);
modelpoints = cell(noIms,1);
for i=1:noIms
    [impoints{i},error] = JH_Generate_Model(cnts{i}(:,1:5), radratio, 2);
    switch error
        case 1
            fprintf('Image no. %i failed axis check. Grid may be on too much of an angle, or
centroid detection properties may be wrong.\n',i)
            return
        case 2
            fprintf('Image no. %i had issues at the centre of the grid. Check centroid list &
detection properites.\n',i)
            return
    end
    [height,~]=size(impoints{i});
    modelpoints{i} = [impoints{i}(:,1:2) impoints{i}(:,1:2)*dotspacing zeros(height,1)]; %
locations of model points, in mm
    % make an array of extrinsic parameters (for later)
end

% figure
% imagesc(calIm)
% colormap gray
% axis equal
% hold on
% plot(impoints{i}(:,3),impoints{i}(:,4),'r+','MarkerSize',20)
% hold off

% ITERATIVE SOLVER
global phys
global trans
for run = 1:runs
    % solve for the depth-specific parameters
    if run == 1
        extrinsics = [];
        for i=1:noIms
            extrinsics=[extrinsics,[Tx,Ty,Tz,Q1,Q2,Q3,Q4]];
        end
    else
        vbles = num2cell(defaults);
        [tL,b1,b2,b3,x0,y0,k1,k2,d1,d2,dD]=vbles{2:12};
        b1=rand
        b2=b1;
        b3=b1;
        extrinsics = allvars(12:end-1);
        fL = allvars(end);
        defaults=[fL,tL,b1,b2,b3,x0,y0,k1,k2,d1,d2,dD,Tx,Ty,Tz,Q1,Q2,Q3,Q4];
    end
    initops=optimoptions('lsqnonlin','Algorithm','levenberg-
marquardt','InitDamping',1000,'TolFun',1e-
6,'MaxFunEvals',1000*length(extrinsics+4),'MaxIter',1000,'Display','none');
    result=lsqnonlin(@initialize,[tL,extrinsics],[],[],initops);
    tL=result(1);
    extrinsic=result(2:end);
    pass = tL;

    % solve for the b's
    result=lsqnonlin(@initialize2,[extrinsic,b1,b2,b3],[],[],initops);
    % variable assignment for in-plane distortion solver
    b1=result(end-2);
    b2=result(end-1);
    b3=result(end);
    % variable re-assignments
    extrinsic=result(1:end-3);
    intrins = [tL,b1,b2,b3,x0,y0,k1,k2,d1,d2,dD];
    allvars = [intrins extrinsic fL];
    pass = [tL,b1,b2,b3];

    % solve for the in-plane distortion variables
    intops=optimoptions('lsqnonlin','Algorithm','levenberg-
marquardt','InitDamping',1000,'TolFun',1e-
6,'MaxFunEvals',1000*length(allvars),'MaxIter',1000,'Display','none');
    [allvars,totresid,resids,~,~]=lsqnonlin(@intrinfun,allvars,[],[],intops);
    allvars(1:11)
    totresid
    normres=sqrt(sum(resids.^2,2));
    prc95=prctile(normres,95)
    prc2_1=prctile(resids(:,1),2.5)
    prc98_1=prctile(resids(:,1),97.5)
    prc2_2=prctile(resids(:,2),2.5)
    prc98_2=prctile(resids(:,2),97.5)
    prc2_3=prctile(resids(:,3),2.5)
    prc98_3=prctile(resids(:,3),97.5)

    % save result if it's better than the old one
    if newrun == 0
        try
```

```
            old=load(['Calibration_170529']);
        catch
            newrun = 1;
            old.totresid = 1;
        end
    else
        old.totresid = 1;
    end
    if (old.totresid>totresid || newrun) && imag(allvars(1))==0
        save(['Calibration_test'],'allvars','totresid');
        newrun=0;
        % kill the loop if nothing is changing
%         if abs(old.totresid-totresid)<10^-6
%             fprintf('Precision limit reached!\n');
%             break;
%         end
    else
        %allvars=old.allvars;
        %fprintf('Got worse...\n');
        %break;
    end
    % toc
end

% result plots
for i=1:noIms
    k=0;
    for j=0:2
        k=k+1;
        figure(k)
        hold on
        scatter3(phys{i}(:,3+3*j),phys{i}(:,4+3*j),phys{i}(:,5+3*j),'or');
        scatter3(trans{i}(:,3),trans{i}(:,4),trans{i}(:,5),'.b');
        axis tight
        axis equal
        hold off
    end
end
figure
histogram(normres);
xlabel('Residual [mm]','FontSize',14,'FontName','Times New Roman')
ylabel('Count','FontSize',14,'FontName','Times New Roman')
set(gca,'FontSize',14,'FontName','Times New Roman')
figure
histfit(resids(:,1));
xlabel('{\itx} Residual [mm]','FontSize',14,'FontName','Times New Roman')
ylabel('Count','FontSize',14,'FontName','Times New Roman')
set(gca,'FontSize',14,'FontName','Times New Roman')
figure
histfit(resids(:,2));
xlabel('{\ity} Residual [mm]','FontSize',14,'FontName','Times New Roman')
ylabel('Count','FontSize',14,'FontName','Times New Roman')
```

```
set(gca,'FontSize',14,'FontName','Times New Roman')
figure
histfit(resids(:,3));
xlabel('{\itz} Residual [mm]','FontSize',14,'FontName','Times New Roman')
ylabel('Count','FontSize',14,'FontName','Times New Roman')
set(gca,'FontSize',14,'FontName','Times New Roman')
drawnow

end

function initres = initialize(vars)
% Try to get some of the initial parameters right by ignoring distortion
global defaults
global impoints
global noIms
global spix
global Rx
global Ry
global phys
global i

% main lens variables
fL = defaults(1);
tL = vars(1);

% micro-lens variables
b1= defaults(3);
b2= defaults(4);
b3= defaults(5);

% distortion variables
x0 = defaults(6); % distortion offset x from corner [pix] relative to image centre
y0 = defaults(7); % distortion offset y from corner [pix] relative to image centre
k1 = defaults(8); % radial distortion coeff 1
k2 = defaults(9); % radial distortion coeff 2
d1 = defaults(10); % depth distortion coeff 1
d2 = defaults(11); % depth distortion coeff 2
dD = defaults(12); % depth-dependant depth distortion coeff

% lens modeling parameters
bL = tL/2*(1-sqrt(1-4*fL/tL));
Gp = tL-bL;

initres=[0,0,0,0,0,0,0,0,0];
r=cell(noIms,1);
metric=cell(noIms,1);
undist=cell(noIms,1);
phys = cell(noIms,1);

for i=1:noIms
    % Extrinsics
    % Translations
    Tx = vars(2+(i-1)*7);
    Ty = vars(3+(i-1)*7);
    Tz = vars(4+(i-1)*7);

    % Quaternion Rotation
    Q1 = vars(5+(i-1)*7);
    Q2 = vars(6+(i-1)*7);
    Q3 = vars(7+(i-1)*7);
    Q4 = vars(8+(i-1)*7);

    % send to depth conversion algorithm

[metric{i},r{i},undist{i},phys{i}]=depthconv(impoints{i},b1,b2,b3,bL,Rx,Ry,spix,x0,y0,k1,k2,d1,d2
,dD,fL);
    % perform extrinsic shifts, find residuals
    exres = extrinfun([Tx,Ty,Tz,Q1,Q2,Q3,Q4]);
    % Add to total residuals
    initres = [initres;exres];
end % for
end

function intres = initialize2(extrins)
% CCD constants
global spix
global Rx
global Ry
global defaults
% globals
global impoints;
global phys;
global i;
global noIms
global pass

% shift intrins into variables
% main lens variables
fL = defaults(1);
tL = pass;

% micro-lens variables
b1= abs(extrins(end-2));
b2= abs(extrins(end-1));
b3= abs(extrins(end));

% distortion variables
x0 = defaults(6); % distortion offset x from corner [pix] relative to image centre
y0 = defaults(7); % distortion offset y from corner [pix] relative to image centre
k1 = defaults(8); % radial distortion coeff 1
k2 = defaults(9); % radial distortion coeff 2
d1 = defaults(10); % depth distortion coeff 1
d2 = defaults(11); % depth distortion coeff 2
dD = defaults(12); % depth-dependant depth distortion coeff
```

```
% lens modeling parameters
bL = tL/2*(1-sqrt(1-4*fL/tL));
Gp = tL-bL;

intres=[];
r=cell(noIms,1);
metric=cell(noIms,1);
undist=cell(noIms,1);
phys = cell(noIms,1);

for i=1:noIms
    % get extrinsic parameters
    extr=extrins(1+(i-1)*7:7+(i-1)*7);
    % send to depth conversion algorithm

[metric{i},r{i},undist{i},phys{i}]=depthconv(impoints{i},b1,b2,b3,bL,Rx,Ry,spix,x0,y0,k1,k2,d1,d2
,dD,fL);
    % move modelpoints array into position, calculate residual
    exres=extrinfun(extr);
    % Add to total residuals
    intres = [intres;exres];
end % for
end % function

function intres = intrinfun(intrins)
% CCD constants
global spix
global Rx
global Ry
global defaults
% globals
global impoints;
global phys;
global i;
global noIms
global pass

% shift intrins into variables
fL=defaults(1);
intr=num2cell(intrins);
[tL,b1,b2,b3,x0,y0,k1,k2,d1,d2,dD]=intr{1:11};
% ps = num2cell(pass);
% [tL,b1,b2,b3]=ps{:};

b1=abs(b1);
b2=abs(b2);
b3=abs(b3);

% lens modeling parameters
bL = tL/2*(1-sqrt(1-4*fL/tL));
Gp = tL-bL;
```

```
intres=[0,0,0,0,0,0,0,0,0,0];
r=cell(noIms,1);
metric=cell(noIms,1);
undist=cell(noIms,1);
phys = cell(noIms,1);

for i=1:noIms
    % get extrinsic parameters
    extrins=intrins(12+(i-1)*7:18+(i-1)*7);
    % send to depth conversion algorithm

[metric{i},r{i},undist{i},phys{i}]=depthconv(impoints{i},b1,b2,b3,bL,Rx,Ry,spix,x0,y0,k1,k2,d1,d2
,dD,fL);
    % move modelpoints array into position, calculate residual
    exres=extrinfun(extrins);
    % Add to total residuals
    intres = [intres;exres];
end % for
end % function

function exres = extrinfun(extrins)
global phys;
global modelpoints;
global i;
global trans;

extr = num2cell(extrins);
[Tx,Ty,Tz,Q1,Q2,Q3,Q4]=extr{:};

% set up translation variable
trans{i}(:,1:2)=modelpoints{i}(:,1:2);

% Convert quaternions to rotation matrix
mat = zeros(3);
n = Q4 * Q4 + Q1 * Q1 + Q2 * Q2 + Q3 * Q3;
if n == 0
    s=0;
else
    s=2 / n;
end
xx      = Q1 * Q1 * s;
xy      = Q1 * Q2 * s;
xz      = Q1 * Q3 * s;
xw      = Q1 * Q4 * s;
yy      = Q2 * Q2 * s;
yz      = Q2 * Q3 * s;
yw      = Q2 * Q4 * s;
zz      = Q3 * Q3 * s;
zw      = Q3 * Q4 * s;
mat(1,1)  = 1 - ( yy + zz );
mat(2,1)  =     ( xy - zw );
mat(3,1)  =     ( xz + yw );
mat(1,2)  =     ( xy + zw );
mat(2,2)  = 1 - ( xx + zz );
mat(3,2)  =     ( yz - xw );
mat(1,3)  =     ( xz - yw );
mat(2,3)  =     ( yz + xw );
mat(3,3) = 1 -  ( xx + yy );

% apply quaternion rotation to each model point
trans{i}(:,3:5) = (mat'*modelpoints{i}(:,3:5)')';

% apply translation to each model point
trans{i}(:,3:5)=bsxfun(@plus,trans{i}(:,3:5),[Tx,Ty,Tz]);

% calculate residuals for each impoint, excluding NaN depths
resids=[phys{i}(:,3:5)-trans{i}(:,3:5),phys{i}(:,6:8)-trans{i}(:,3:5),phys{i}(:,9:11)-
trans{i}(:,3:5)];
% eliminate NaN results
resids(isnan(resids(:,3)),1:3)=0;
resids(isnan(resids(:,6)),4:6)=0;
resids(isnan(resids(:,9)),7:9)=0;
% % combine all residuals into a single vector
% exres = reshape(resids,1,[]);
% if any(imag(exres)~=0)
%     warning('Regression method tested an imaginary value.')
% end
exres=[resids(:,1:3),resids(:,4:6),resids(:,7:9)];
end

function
[metric,r,undist,phys]=depthconv(impoints,b1,b2,b3,bL,Rx,Ry,spix,x0,y0,k1,k2,d1,d2,dD,fL)
% convert virtual depths and pixel locations to metric
    zdata=bL+bsxfun(@times,impoints(:,5:7)-2,[b1,b2,b3]);
    metric=[impoints(:,1:2),...
        bsxfun(@times,-(impoints(:,3)-Rx/2)*2*spix,bsxfun(@rdivide,zdata,bL)),...
        bsxfun(@times,-(impoints(:,4)-Ry/2)*2*spix,bsxfun(@rdivide,zdata,bL)),...
        zdata];
    % compute r for each point
    r(:,1) = sqrt(sum(abs(bsxfun(@minus,metric(:,[3,6]),[x0,y0])).^2,2));
    r(:,2) = sqrt(sum(abs(bsxfun(@minus,metric(:,[4,7]),[x0,y0])).^2,2));
    r(:,3) = sqrt(sum(abs(bsxfun(@minus,metric(:,[5,8]),[x0,y0])).^2,2));
    % undistort metric-depth points
    undist=[metric(:,1:2),...
        metric(:,3).*(1+k1*r(:,1).^2+k2*r(:,1).^4),...
        metric(:,4).*(1+k1*r(:,2).^2+k2*r(:,2).^4),...
        metric(:,5).*(1+k1*r(:,3).^2+k2*r(:,3).^4),...
        metric(:,6).*(1+k1*r(:,1).^2+k2*r(:,1).^4),...
        metric(:,7).*(1+k1*r(:,2).^2+k2*r(:,2).^4),...
        metric(:,8).*(1+k1*r(:,3).^2+k2*r(:,3).^4),...
        metric(:,9:11)+(1+dD*metric(:,9:11)).*(d1*r(:,1:3).^2+d2*r(:,1:3).^4)];
    % project points to physical space
    zdata=undist(:,9:11)*fL./(undist(:,9:11)-fL);
    phys=[undist(:,1:2),...
        undist(:,3:5)./undist(:,9:11).*zdata,...
        undist(:,6:8)./undist(:,9:11).*zdata,...
        zdata];
    % reposition some stuff
    phys=[phys(:,1:3),phys(:,6),phys(:,9),...
        phys(:,4),phys(:,7),phys(:,10),...
        phys(:,5),phys(:,8),phys(:,11)];
    % we now have phys in (a,b,x1,y1,z1,x2,y2,z2,x3,y3,z3) form. This
    % should correspond directly to our modelpoints array
end
```

Published with MATLAB® R2015b

D.2 ***ETC calibration function: metricCal***

The following code is the calibration method applied with the ETC method:

```
function metricCal
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calibrates a Raytrix camera
% Based on Christian Heinze's 2014 masters thesis
newrun = 1;
runs = 1;
clear global
tic
global impoints;
global modelpoints;
global iters
iters=0;
global noIms
global pass
global zred
global mlHold
global from

% IMAGE LOCATIONS
% file path to white image
whitIm='X:\01_Current_Students\Jake Hadfield\LightField Test Images\whitIms\whitAve_11-Sep-
2017.mat';
% file path to micro-lens locations
lensCalLoc='X:\01_Current_Students\Jake Hadfield\LightField Test
Images\whitIms\MLCal_170911.mat';
% file path to images
path = 'X:\01_Current_Students\Jake Hadfield\LightField Test Images\waterCal_170809\';
% Shared tag
imID = 'Image';
% List of image numbers in a 1 by n array
%[4,6:9,11:27,29:31,34:43
```

```
%[389:390,405:406,410,411,413,414,417]
imNos = [15 17 21 25 26 27 28];
noIms = length(imNos);
% file format
fmt = '.png';

% Dot Detection Settings
invert=1; % wether or not to invert by subtracting the white image
cleanup=225; % wether or not to normalize the image by dividing by the white image
lnoise=0; % Always should be zero, from what I've seen. see bpass for more.
psize=7; % Particle size in pixels in raw images
thresh=0; % cutoff threshold
peak=50; % peak threshold
epitol=.5; % pixel distance for max distance from particle to epipolar line
plotar=50; % aspect ratio of virtual depth for 3D plots
validrange=[2,12]; % valid range of virtual depths
display=0;
settings=[invert cleanup lnoise psize thresh peak epitol plotar validrange display];

% Target Model Generation Settings
% model generation stuff
radratio = 0.1;

% target model values, in mm
dotspacing = 5.72666666666666666666667;

% error modifier
zred=3;

% Iterative solver initial parameter values

% Camera Constants
global Rx
global Ry
global spix

spix = 5.5e-3;% pixel size [mm/pix]
Rx = 2048; % image (sensor?) resolution x [pix]
Ry = 2048; % image (sensor?) resolution y [pix]

% Calibration Variables
try
    load('nope','allvars')
    % main lens variables
    fL = allvars(end); % focal length of main lens [mm]
    tL = allvars(1)-fL*4; % focus distance [mm]

    % micro-lens variables
    b1= allvars(2); % MLA-CCD distance [mm], far field lenses
    b2= allvars(3); % MLA-CCD distance [mm], mid field lenses
    b3= allvars(4); % MLA-CCD distance [mm], near field lenses
    % distortion variables
    x0 = allvars(5); % distortion offset x from corner [pix] relative to image centre
    y0 = allvars(6); % distortion offset y from corner [pix] relative to image centre
    k1 = allvars(7); % radial distortion coeff 1
    k2 = allvars(8); % radial distortion coeff 2
    d1 = allvars(9); % depth distortion coeff 1
    d2 = allvars(10); % depth distortion coeff 2
    dD = allvars(11); % depth-dependant depth distortion coeff

    % ML distortion by type
    mlvars=allvars(12:17);
catch
    % main lens variables
    fL = 85; % focal length of main lens [mm]
    tL = 800; % focus distance [mm]

    % micro-lens variables
    b1= .35; % MLA-CCD distance [mm], far field lenses
    b2= .35; % MLA-CCD distance [mm], mid field lenses
    b3= .35; % MLA-CCD distance [mm], near field lenses

    % distortion variables
    x0 = 0; % distortion offset x from corner [pix] relative to image centre
    y0 = 0; % distortion offset y from corner [pix] relative to image centre
    k1 = 0; % radial distortion coeff 1
    k2 = 0; % radial distortion coeff 2
    d1 = 0; % depth distortion coeff 1
    d2 = 0; % depth distortion coeff 2
    dD = 0; % depth-dependant depth distortion coeff

    % ML distortion by type
    mlvars= [0 0 0 0 0 0];
end

% Extrinsics
% Translations
Tx = 0;
Ty = 0;
Tz = abs(tL)+4*fL;

% Quaternion Rotation
Q1 = 0;
Q2 = 0;
Q3 = 0;
Q4 = 1;

global defaults
defaults=[fL,tL,b1,b2,b3,x0,y0,k1,k2,d1,d2,dD,Tx,Ty,Tz,Q1,Q2,Q3,Q4];

% POINT DETECTION LOOP
% set up figures
points=cell(noIms,1);
```

```
ltypes=cell(noIms,1);
pgroups=cell(noIms,1);
for i=1:noIms
    imName=[imID '_Raw_' num2str(imNos(i),'%04i') fmt];

[points{i},ltypes{i},~,~,~,pcd{i},plocs{i},pgroups{i},lensCents,pli{i},typelist]=JH_rawPlenPF(pat
h,imName,[],lensCalLoc,whitIm,'',settings);
end
% now we have all the centroids in pixel and virtual depth space

% GENERATE TARGET MODEL
impoints = cell(noIms*3,1);
from=cell(noIms*3,1);
modelpoints = cell(noIms*3,1);
skips=[];
for i=1:noIms
    for j=1:3
        [impoints{(i-1)*3+j},error,from{(i-1)*3+j}] =
JH_Generate_Model(points{i}(ltypes{i}==j,:), radratio, 2);
        switch error
            case 1
                fprintf('Image no. %i - %i failed axis check. Grid may be on too much of an
angle, or centroid detection properties may be wrong.\n',i,j)
                skips=[skips;(i-1)*3+j];
            case 2
                fprintf('Image no. %i - %i had issues at the centre of the grid. Check centroid
list & detection properites.\n',i,j)
                skips=[skips;(i-1)*3+j];
            case 3
                fprintf('Image no. %i - %i had too few points. Check centroid list & detection
properites.\n',i,j)
                skips=[skips;(i-1)*3+j];
        end
    end
    % readjust to align grid centres, if possible.
    [~,ind]=max([length(impoints{(i-1)*3+1}),length(impoints{(i-1)*3+2}),length(impoints{(i-
1)*3+3})]);
    try
        loc=impoints{(i-1)*3+ind}(1,3:4);
    catch
        continue
    end
    adjind=1:3;
    adjind(adjind==ind)=[];
    var1=1;
    pos=zeros(3,1);
    tag=0;
    while tag ~=2
        tag=0;
        pos(ind)=var1;
        for j=adjind
            if ~isempty(impoints{(i-1)*3+j})
```

```
                [dist,pos(j)]=min(sum(bsxfun(@minus,impoints{(i-1)*3+j}(:,3:4),loc).^2,2).^0.5);
                if dist<1
                    tag=tag+1;
                end
            else
                tag=tag+1;
            end
        end
        if tag==2
            for j=1:3
                if ~isempty(impoints{(i-1)*3+j})
                    % adjust grid origin and generate modelpoints
                    impoints{(i-1)*3+j}(:,1:2)=bsxfun(@minus,impoints{(i-
1)*3+j}(:,1:2),impoints{(i-1)*3+j}(pos(j),1:2));
                    modelpoints{(i-1)*3+j} = [impoints{(i-1)*3+j}(:,1:2) impoints{(i-
1)*3+j}(:,1:2)*dotspacing zeros(size(impoints{(i-1)*3+j},1),1)]; % locations of model points, in
mm
                end
            end
        else
            % didn't share centrepoint, try the next one
            varl=varl+1;
            try
                loc=impoints{(i-1)*3+ind}(varl,3:4);
            catch
                % couldn't align grid, empty smallest member of impoints
                % and retry with just 2 (or use only 1 if that's all we
                % have)
                [~,small]=min([length(impoints{(i-1)*3+1}),length(impoints{(i-
1)*3+2}),length(impoints{(i-1)*3+3})]);
                if ~isempty(impoints{(i-1)*3+small})
                    impoints{(i-1)*3+small}=[];
                else
                    ls=[length(impoints{(i-1)*3+1}),length(impoints{(i-
1)*3+2}),length(impoints{(i-1)*3+3})];
                    [~,small]=find(ls==min(ls(ls>0)));
                    impoints{(i-1)*3+small}=[];
                end
                varl=1;
                try
                    loc=impoints{(i-1)*3+ind}(varl,3:4);
                catch
                    continue
                end
            end
        end
    end
end
impoints(skips)=cell(1,1);
modelpoints(skips)=cell(1,1);
% figure
% imagesc(calIm)
% colormap gray
% axis equal
% hold on
% plot(impoints{i}(:,3),impoints{i}(:,4),'r+','MarkerSize',20)
% hold off

% ITERATIVE SOLVER
global phys
global trans
for run = 1:runs
    % solve for the depth-specific parameters
    if run == 1
        extrinsics = [];
        for i=1:noIms
            extrinsics=[extrinsics,[Tx,Ty,Tz,Q1,Q2,Q3,Q4]];
        end
    else
        vbles = num2cell(defaults);
        [tL,b1,b2,b3,x0,y0,k1,k2,d1,d2,dD]=vbles{2:12};
        b1=rand
        b2=b1;
        b3=b1;
        extrinsics = allvars(12:end-1);
        fL = allvars(end);
        defaults=[fL,tL,b1,b2,b3,x0,y0,k1,k2,d1,d2,dD,Tx,Ty,Tz,Q1,Q2,Q3,Q4];
    end
    initops=optimoptions('lsqnonlin','Algorithm','levenberg-
marquardt','InitDamping',10,'TolFun',1e-
6,'MaxFunEvals',1000*length(extrinsics+4),'MaxIter',1000,'Display','none');
    result=lsqnonlin(@initialize,[tL,extrinsics],[],[],initops);
    tL=result(1);
    extrinsic=result(2:end);

    intrins = [tL,b1,b2,b3,x0,y0,k1,k2,d1,d2,dD];
    allvars = [intrins extrinsic fL];

    % solve for the in-plane distortion variables
    intops=optimoptions('lsqnonlin','Algorithm','levenberg-
marquardt','InitDamping',10,'TolFun',1e-
6,'MaxFunEvals',1000*length(allvars),'MaxIter',1000,'Display','none');
    [allvars,totresid,resids,~,~]=lsqnonlin(@intrinfun,allvars,[],[],intops);

    % solve for micro-lens distortions
%     pass = allvars;
%     mlHold = {plocs,pgroups,lensCents,pli,typelist,pcd};
%     [mlvars,totresid,resids,~,~]=lsqnonlin(@mlfun,mlvars,[],[],initops);
    allvars=[allvars(1:11) mlvars allvars(11:end)];
    allvars(1)=allvars(end)*4+abs(allvars(1));
    allvars(1:17)

    normres=sqrt(sum(resids.^2,2));
    prc95=prctile(normres,95)
    prc2_1=prctile(resids(:,1),2.5)
    prc98_1=prctile(resids(:,1),97.5)
    prc2_2=prctile(resids(:,2),2.5)
    prc98_2=prctile(resids(:,2),97.5)
    prc2_3=prctile(resids(:,3),2.5)
    prc98_3=prctile(resids(:,3),97.5)

    figure
    histogram(normres);
    xlabel('Residual [mm]','FontSize',14,'FontName','Times New Roman')
    ylabel('Count','FontSize',14,'FontName','Times New Roman')
    set(gca,'FontSize',14,'FontName','Times New Roman')
    figure
    histfit(resids(:,1));
    xlabel('{\itx} Residual [mm]','FontSize',14,'FontName','Times New Roman')
    ylabel('Count','FontSize',14,'FontName','Times New Roman')
    set(gca,'FontSize',14,'FontName','Times New Roman')
    figure
    histfit(resids(:,2));
    xlabel('{\ity} Residual [mm]','FontSize',14,'FontName','Times New Roman')
    ylabel('Count','FontSize',14,'FontName','Times New Roman')
    set(gca,'FontSize',14,'FontName','Times New Roman')
    figure
    histfit(resids(:,3));
    xlabel('{\itz} Residual [mm]','FontSize',14,'FontName','Times New Roman')
    ylabel('Count','FontSize',14,'FontName','Times New Roman')
    set(gca,'FontSize',14,'FontName','Times New Roman')

    % output residuals
    tpoints=0;
    for m=1:length(modelpoints)
        tpoints=sum([tpoints length(modelpoints{m})]);
    end
    avresid=sum(sqrt(sum([resids(:,1:2) resids(:,3)*zred].^2,2)))/length(resids)

    % save result if it's better than the old one
    if newrun == 0
        try
            old=load(['Calibration_170529']);
        catch
            newrun = 1;
            old.totresid = 1;
        end
    else
        old.totresid = 1;
    end
    if (old.totresid>totresid || newrun) && imag(allvars(1))==0
        save(['Calibration_' datestr(datetime('today'))],'allvars','totresid');
        newrun=0;
        % kill the loop if nothing is changing
%         if abs(old.totresid-totresid)<10^-6
%             fprintf('Precision limit reached!\n');
%             break;
%         end
    else
        %allvars=old.allvars;
        %fprintf('Got worse...\n');
        %break;
    end
    % toc
end

% result plots
for i=1:length(phys)
    figure
    if ~isempty(phys{i})
        hold on
        scatter3(phys{i}(:,3),phys{i}(:,4),phys{i}(:,5),'or');
        scatter3(trans{i}(:,3),trans{i}(:,4),trans{i}(:,5),'.b');
        axis tight
        axis equal
        hold off
        xlabel('{\itx} [mm]','FontSize',14,'FontName','Times New Roman')
        ylabel('{\ity} [mm]','FontSize',14,'FontName','Times New Roman')
        zlabel('{\itz} [mm]','FontSize',14,'FontName','Times New Roman')
        set(gca,'FontSize',14,'FontName','Times New Roman')
    end
end
figure
hold on
for i=1:length(phys)
    if ~isempty(phys{i})
        scatter3(phys{i}(:,3),phys{i}(:,4),phys{i}(:,5),'or');
        scatter3(trans{i}(:,3),trans{i}(:,4),trans{i}(:,5),'.b');
        axis tight
        axis equal
    end
end
xlabel('{\itx} [mm]','FontSize',14,'FontName','Times New Roman')
ylabel('{\ity} [mm]','FontSize',14,'FontName','Times New Roman')
zlabel('{\itz} [mm]','FontSize',14,'FontName','Times New Roman')
set(gca,'FontSize',14,'FontName','Times New Roman')
drawnow

end

function initres = initialize(vars)
% Try to get some of the initial parameters right by ignoring distortion
global defaults
global impoints
global noIms
global spix
global Rx
global Ry
global phys
global i
global j

% main lens variables
fL = defaults(1);
tL = vars(1);
tL=abs(tL)+4*fL;

% micro-lens variables
b1= defaults(3);
b2= defaults(4);
b3= defaults(5);

% distortion variables
x0 = defaults(6); % distortion offset x from corner [pix] relative to image centre
y0 = defaults(7); % distortion offset y from corner [pix] relative to image centre
k1 = defaults(8); % radial distortion coeff 1
k2 = defaults(9); % radial distortion coeff 2
d1 = defaults(10); % depth distortion coeff 1
d2 = defaults(11); % depth distortion coeff 2
dD = defaults(12); % depth-dependant depth distortion coeff

% lens modeling parameters
bL = tL/2*(1-sqrt(1-4*fL/tL));

initres=[];
r=cell(noIms,1);
metric=cell(noIms,1);
undist=cell(noIms,1);
phys = cell(noIms,1);
for i=1:noIms
    for j=1:3
    % Extrinsics
    % Translations
    Tx = vars(2+(i-1)*7);
    Ty = vars(3+(i-1)*7);
    Tz = vars(4+(i-1)*7);

    % Quaternion Rotation
    Q1 = vars(5+(i-1)*7);
    Q2 = vars(6+(i-1)*7);
    Q3 = vars(7+(i-1)*7);
    Q4 = vars(8+(i-1)*7);
        % send to depth conversion algorithm
    if ~isempty(impoints{(i-1)*3+j})
        [metric{(i-1)*3+j},r{(i-1)*3+j},undist{(i-1)*3+j},phys{(i-1)*3+j}]=depthconv(impoints{(i-
1)*3+j},j,b1,b2,b3,bL,Rx,Ry,spix,x0,y0,k1,k2,d1,d2,dD,fL);
        % perform extrinsic shifts, find residuals
        exres = extrinfun([Tx,Ty,Tz,Q1,Q2,Q3,Q4]);
        % Add to total residuals
        initres = [initres;exres];
    end
    end
end % for
end

function intres = intrinfun(intrins)
% CCD constants
global spix
global Rx
global Ry
global defaults
% globals
global impoints;
global phys;
global i;
global j
global noIms

% shift intrins into variables
fL=intrins(end);
intr=num2cell(intrins);
[tL,b1,b2,b3,x0,y0,k1,k2,d1,d2,dD]=intr{1:11};
tL=abs(tL)+4*fL;
% ps = num2cell(pass);
% [tL,b1,b2,b3]=ps{:};

b1=abs(b1);
b2=abs(b2);
b3=abs(b3);

% lens modeling parameters
bL = tL/2*(1-sqrt(1-4*fL/tL));
intres=[];
r=cell(noIms,1);
metric=cell(noIms,1);
undist=cell(noIms,1);
phys = cell(noIms,1);

for i=1:noIms
    for j=1:3
    % get extrinsic parameters
    extrins=intrins(12+(i-1)*7:18+(i-1)*7);
    % send to depth conversion algorithm
    if ~isempty(impoints{(i-1)*3+j})
        [metric{(i-1)*3+j},r{(i-1)*3+j},undist{(i-1)*3+j},phys{(i-1)*3+j}]=depthconv(impoints{(i-
1)*3+j},j,b1,b2,b3,bL,Rx,Ry,spix,x0,y0,k1,k2,d1,d2,dD,fL);
        % move modelpoints array into position, calculate residual
        exres=extrinfun(extrins);
        % Add to total residuals
        intres = [intres;exres];
    end
    end
```

```
end % for
end % function

function exres = extrinfun(extrins)
global phys;
global modelpoints;
global i;
global j
global trans;
global zred;

extr = num2cell(extrins);
[Tx,Ty,Tz,Q1,Q2,Q3,Q4]=extr{:};

% set up translation variable
trans{(i-1)*3+j}(:,1:2)=modelpoints{(i-1)*3+j}(:,1:2);

% Convert quaternions to rotation matrix
mat = zeros(3);
n = Q4 * Q4 + Q1 * Q1 + Q2 * Q2 + Q3 * Q3;
if n == 0
    s=0;
else
    s=2 / n;
end
xx      = Q1 * Q1 * s;
xy      = Q1 * Q2 * s;
xz      = Q1 * Q3 * s;
xw      = Q1 * Q4 * s;
yy      = Q2 * Q2 * s;
yz      = Q2 * Q3 * s;
yw      = Q2 * Q4 * s;
zz      = Q3 * Q3 * s;
zw      = Q3 * Q4 * s;
mat(1,1)  = 1 - ( yy + zz );
mat(2,1)  =     ( xy - zw );
mat(3,1)  =     ( xz + yw );
mat(1,2)  =     ( xy + zw );
mat(2,2)  = 1 - ( xx + zz );
mat(3,2)  =     ( yz - xw );
mat(1,3)  =     ( xz - yw );
mat(2,3)  =     ( yz + xw );
mat(3,3) = 1 -  ( xx + yy );

% apply quaternion rotation to each model point
trans{(i-1)*3+j}(:,3:5) = (mat'*modelpoints{(i-1)*3+j}(:,3:5)')';

% apply translation to each model point
trans{(i-1)*3+j}(:,3:5)=bsxfun(@plus,trans{(i-1)*3+j}(:,3:5),[Tx,Ty,Tz]);

% calculate residuals for each impoint, excluding NaN depths
resids=phys{(i-1)*3+j}(:,3:5)-trans{(i-1)*3+j}(:,3:5);
```

```
% eliminate NaN results
resids(isnan(resids(:,3)),:)=[];
% combine all residuals into a single vector
exres = [resids(:,1:2) resids(:,3)/zred];
% if any(imag(exres)~=0)
%     warning('Regression method tested an imaginary value.')
% end
end

function
[metric,r,undist,phys]=depthconv(impoints,j,b1,b2,b3,bL,Rx,Ry,spix,x0,y0,k1,k2,d1,d2,dD,fL)
switch j
    case 1
        b=b1;
    case 2
        b=b2;
    case 3
        b=b3;
end
% convert virtual depths and pixel locations to metric
    zdata=bL+bsxfun(@times,impoints(:,5)-2,b);
    metric=[impoints(:,1:2),...
        (impoints(:,3)-Rx/2)*spix,...
        (impoints(:,4)-Ry/2)*spix,...
        zdata];
    % compute r for each point
    r(:,1) = sqrt(sum(abs(bsxfun(@minus,metric(:,3:4),[x0,y0])).^2,2));
    % undistort metric-depth points
    undist=[metric(:,1:2),...
        metric(:,3).*(1+k1*r(:,1).^2+k2*r(:,1).^4),...
        metric(:,4).*(1+k1*r(:,1).^2+k2*r(:,1).^4),...
        metric(:,5)+(1+dD*metric(:,5)).*(d1*r(:,1).^2+d2*r(:,1).^4)];
    % project points to physical space
    zdata=undist(:,5)*fL./(undist(:,5)-fL);
    phys=[undist(:,1:2),...
        -undist(:,3)./undist(:,5).*zdata,...
        -undist(:,4)./undist(:,5).*zdata,...
        zdata];
    % we now have phys in (a,b,x,y,z) form. This
    % should correspond directly to our modelpoints array
end

% function resids=mlfun(mlVars)
% global pass
% global mlHold
% global impoints
% global from
%
% plocs=mlHold{1};
% pgroups=mlHold{2};
% lensCents=mlHold{3};
% pli=mlHold{4};
% typelist=mlHold{5};
% pcd=mlHold{6};
%
%
% % modify raw particle locations
% nplocs=JH_Plen_mlCorrect(plocs,lensCents,pli,typelist,pcd,mlVars);
%
%
% for i = 1:length(nplocs)
%     % solve for new 3D particle locations
%     [points,~,~]=JH_Plen_GetPt(nplocs{i},pgroups{i},lensCents,pli{i},pcd{i},typelist,0);
%
%     % replace values in model array
%     for j=1:size(impoints{i},1)
%         impoints{i}(j,3:5)=points(from{i}(j),:);
%     end
% end
%
% % convert to physical position and calculate residual
% resids = intrinfun(pass);
% end
```

Published with MATLAB® R2015b

D.3 *Model generation function: JH_Generate_Model*

The target model function has been isolated in the ETC code, accomplished using the following code:

```
function [impoints, error, from] = JH_Generate_Model(cnts,varargin)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Generates a model of a calibration target with points corresponding to
% input parameters, and associates input centroids. first two columns of
% impoints contain [a,b] matrix of valid target points, last three contain
% centroid locations.
% Inputs:
% cnts       - Detected centroids of grid points, in (x,y,z) form [pix]
% radratio   - Tolerance for expected point location, max 0.05, recommended
%              0.01 or less; depends on target image quality and noise level
% maxmiss    - Number of points
% Outputs:
% impoints - array of [a,b,x,y,z] locations containing:
%              a - relative x position of the point in the array along minor axis [int]
%              b - relative y position of the point in the array along major axis [int]
%        [x,y,z] - centroid of point [pix]

% error    - variable for identifying errors
%
% Warnings: Code will produce warnings identifying points that deviated
% significantly from the expected location, based on the axis generated.
% These will occur more frequently with highly distorted views. Should
% check these points manually to confirm centroid location was good. If a
% lot of these show up, there may be issues near the centre of the image.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
warn = 0;
from=[];
% robust input handling
if ~isa(varargin{1},'double')
    radratio = 0.1; % default
else
    radratio = varargin{1};
end
if ~isa(varargin{2},'int')
    maxmiss = 2; % default
else
    maxmiss = varargin{2};
end
if isempty(cnts)
    impoints=[];
    error=3;
    from=[];
    return;
end
error = 0;
% figure
% scatter3(cnts(:,1),cnts(:,2),cnts(:,3))
% Find target axes
% find [x,y] epicentre of centroids
epi= sum(cnts(:,1:2),1)/size(cnts,1);

% put [x,y] centroids into a k-d tree
tree = KDTreeSearcher(cnts(:,1:2));
okay=0;
flip=0;
while ~okay
    % figure out which centroid(s) is closest to the epicentre
    ind = knnsearch(tree,epi);

    % set that point to be the origin
    impoints = [0 0 cnts(ind,1:3)];

    % find next 8 nearest points to origin
    ind = knnsearch(tree, cnts(ind,1:2), 'K', 9); % single row

    % find opposing points
    try
    vecs = 2*bsxfun(@minus,cnts(ind,1:2),cnts(ind(1),1:2));
    catch
        pause
    end
    % toss out zeros (origin)
    vecs(vecs(:,1)==vecs(:,2),:)=[];
    try
        indop = knnsearch(tree,bsxfun(@minus,cnts(ind(2:9),1:2),vecs)); % single column
    catch
        error=3;
        return;
    end
    % get opposites list, remove duplicates and origin
    ops = [ind(2:9)' indop];
    ops = unique(sort(ops,2),'rows');
    % ops should be a 4x2 array with the indicies of points that oppose each other accross the
origin
    check = size(ops);
    if check(1)~=4 || check(2) ~=2
        if epi<2048
            epio=epi;
            epi=epi+[100,0];
        else
            epi=[10,epi(2)];
            flip=1;
        end
        if flip && epi(1)>epio(1)
            error = 2;
            impoints = [];
            return
        end
    else
        okay=1;
    end
end

% determine vector lengths between points in ops
vecs = cnts(ops(:,2),1:2)-cnts(ops(:,1),1:2);
vlengths = sqrt(sum(abs(vecs).^2,2));
[~, vind] = sort(vlengths);

% figure out which axes are good
p=zeros(1,9);
try1=1;
try2=2;
tries = 1;
while all(sort(ind)~=sort(p))
    % shortest vector becomes minor axis direction
    minorAxis=vecs(vind(try1),:)/2;
    % next shortest vector becomes major axis direction
    majorAxis=vecs(vind(try2),:)/2;
    % make sure all 8 closest points to origin can be identified by single
    % combinations of the minor and major axis
    % origin always good
    p(1) = ind(1);
    % on minor
    p(2) = knnsearch(tree, cnts(ind(1),1:2)+minorAxis);
```

```
        p(3) = knnsearch(tree, cnts(ind(1),1:2)-minorAxis);
        % on major
        p(4) = knnsearch(tree, cnts(ind(1),1:2)+majorAxis);
        p(5) = knnsearch(tree, cnts(ind(1),1:2)-majorAxis);
        % diagonals
        p(6) = knnsearch(tree, cnts(ind(1),1:2)+majorAxis+minorAxis);
        p(7) = knnsearch(tree, cnts(ind(1),1:2)-majorAxis+minorAxis);
        p(8) = knnsearch(tree, cnts(ind(1),1:2)+majorAxis-minorAxis);
        p(9) = knnsearch(tree, cnts(ind(1),1:2)-majorAxis-minorAxis);
        % try other axes if it didn't work
        if tries == 1
            try1=2;
            try2=3;
        elseif tries == 2
            try1=1;
            try2=3;
        else
            % didn't get 8 good points around origin
            error = 1;
            impoints = 0;
            return
        end
    end

    % we now should have good major and minor axes
    missflags = [];
    % Determine coordinates of points in image along minor axis
    for runs=1:2
        % set up point finding inital loop variables
        findingpoints = 1;
        % which direction we're going
        switch runs
            case 1
                vector = minorAxis;
                c=1;
            case 2
                vector = -minorAxis;
                c=-1;
        end
        loc = cnts(ind(1),1:2);
        misses = 0;
        i=1;
        while findingpoints == 1;
            idx = rangesearch(tree,loc+vector,radratio*norm(vector));
            % did we find a point?
            if isempty(idx{1}) || length(idx{1})>1
                % if we didn't find one, we should keep going in case only a
                % few points are missing/blocked, but we need to flag the point
                % so we don't put it in the model target
                switch runs
                    case 1
                        missflags = [missflags;i,0];
                    case 2
                        missflags = [missflags;-i,0];
                end
                loc = loc+vector;
                misses = misses + 1;
                switch runs
                    case 1
                        impoints = [impoints;[i,0,loc,NaN]];
                    case 2
                        impoints = [impoints;[-i,0,loc,NaN]];
                end
            else
                vector = bsxfun(@minus,cnts(idx{1},1:2),loc);
                % compare with minor axis
                try
                if norm(vector-c*minorAxis)>(radratio*norm(minorAxis)) && warn
                    switch runs
                        case 1
                            warning('Point (%i,%i) produced a vector
    %1.2d*minorAxis.',i,0,norm(vector-c*minorAxis)/norm(minorAxis))
                        case 2
                            warning('Point (%i,%i) produced a vector %1.2d*minorAxis.',-
    i,0,norm(vector-c*minorAxis)/norm(minorAxis))
                    end
                end
                catch
                    error('Detected multiple points where only 1 should be present. Increase epitol
    and/or decrease peak threshold.');
                end
                loc = cnts(idx{1},1:2);
                misses = 0;
                switch runs
                    case 1
                        impoints = [impoints;[i,0,loc,cnts(idx{1},3)]];
                    case 2
                        impoints = [impoints;[-i,0,loc,cnts(idx{1},3)]];
                end
            end
            i=i+1;
            % if we missed too many points, we probably hit the edge
            if misses == maxmiss
                findingpoints = 0;
            end
        end
    end

    % Determine locations of all points
    % should have a line of points along the minor axis. Now, move in the major
    % axis direction
    [hits,~]=size(impoints);
    for j=1:hits
        for runs = 1:2
```

```
        % set up point finding inital loop variables
        findingpoints = 1;
        % which direction we're going
        switch runs
            case 1
                vector = majorAxis;
                c=1;
            case 2
                vector = -majorAxis;
                c=-1;
        end
        loc = impoints(j,3:4);
        misses = 0;
        i=1;
        while findingpoints == 1;
            idx = rangesearch(tree, loc+vector,radratio*norm(vector));
            % did we find a point?
            if isempty(idx{1}) || length(idx)>1
                % if we didn't find one, we should keep going in case only a
                % few points are missing/blocked, but we need to flag the point
                % so we don't put it in the model target
                switch runs
                    case 1
                        missflags = [missflags;impoints(j,1),i];
                    case 2
                        missflags = [missflags;impoints(j,1),-i];
                end
                loc = loc+vector;
                misses = misses + 1;
                switch runs
                    case 1
                        impoints = [impoints;impoints(j,1),i,loc,NaN,];
                    case 2
                        impoints = [impoints;impoints(j,1),-i,loc,NaN,];
                end
            elseif length(idx{1})>1
                for p=length(idx{1})
                    vector = bsxfun(@minus,cnts(idx{1}(p),1:2),loc);
                    % compare with major axis. Throws more warnings than
                    % necessary, but good info.
                    if norm(vector-c*majorAxis)>(radratio*norm(majorAxis)) && warn;
                        switch runs
                            case 1
                                warning('Point (%i,%i) produced a vector
%1.2d*majorAxis.',impoints(j,1),i,norm(vector-c*majorAxis)/norm(majorAxis))
                            case 2
                                warning('Point (%i,%i) produced a vector
%1.2d*majorAxis.',impoints(j,1),-i,norm(vector-c*majorAxis)/norm(majorAxis))
                        end
                    end
                    loc = cnts(idx{1}(p),1:2);
                    misses = 0;
                    switch runs
                        case 1
                            impoints = [impoints;impoints(j,1),i,loc,cnts(idx{1}(p),3)];
                        case 2
                            impoints = [impoints;impoints(j,1),-i,loc,cnts(idx{1}(p),3)];
                    end
                end
            else
                vector = bsxfun(@minus,cnts(idx{1},1:2),loc);
                % compare with major axis. Throws more warnings than
                % necessary, but good info.
                if norm(vector-c*majorAxis)>(radratio*norm(majorAxis)) && warn;
                    switch runs
                        case 1
                            warning('Point (%i,%i) produced a vector
%1.2d*majorAxis.',impoints(j,1),i,norm(vector-c*majorAxis)/norm(majorAxis))
                        case 2
                            warning('Point (%i,%i) produced a vector
%1.2d*majorAxis.',impoints(j,1),-i,norm(vector-c*majorAxis)/norm(majorAxis))
                    end
                end
                loc = cnts(idx{1},1:2);
                misses = 0;
                switch runs
                    case 1
                        impoints = [impoints;impoints(j,1),i,loc,cnts(idx{1},3)];
                    case 2
                        impoints = [impoints;impoints(j,1),-i,loc,cnts(idx{1},3)];
                end
            end

            i=i+1;
            % if we missed too many points, we probably hit the edge
            if misses == maxmiss
                findingpoints = 0;
            end
        end
    end
end

% should have a full grid of points in impoints, including misses. Now need
% to eliminate misses.
[~,crossed,~]=intersect(impoints(:,1:2),missflags,'rows');
impoints(crossed,:)=[];

% perform iterative planar fitting and remove outliers
fitz=fit([impoints(:,3),impoints(:,4)],impoints(:,5),'poly22');

fdata = feval(fitz,[impoints(:,3),impoints(:,4)]);
I = abs(fdata - impoints(:,5)) > 1*std(impoints(:,5));

display=0;
```

```
if display==1
    figure
    plot(fitz,[impoints(:,3),impoints(:,4)],impoints(:,5))
    fitzelim= fit([impoints(~I,3),impoints(~I,4)],impoints(~I,5),'poly22');
    figure
    plot(fitzelim,[impoints(~I,3),impoints(~I,4)],impoints(~I,5))
    hold on
    scatter3(impoints(I,3),impoints(I,4),impoints(I,5),'*m')
    hold off
end

% remove outliers from impoints
impoints(I,:)=[];
% And we're done!

% now, where did all the impoints come from in cnts?
from=zeros(size(impoints,1),1);
for i=1:length(impoints)
    [~,from(i)]=ismember(impoints(i,3:5),cnts,'rows');
end
```

Published with MATLAB® R2015b

Appendix E PARTICLE TRACKING METHODS

The particle tracking methods are used by both the refocusing and ETC methods. These codes are implemented after the particles in at least two frames have been located and require an outer loop to manage properly as they only deal with the existing tracks and the last two particle sets in the series.

E.1 *Track initialization: DH_track_v3*

The following code is responsible for the two-frame track initialization. This code was written initially by [9], but a slight modification has been made to make it suitable for the particle locations produced from the plenoptic images.

```
function tracks = DH_track_v3(xyz_a,xyz_b,search,neighbor,quasi,zdt)

%A script to track particles through two time frames and three dimensions.
%It follows the general technique of Baek and Lee's 1996 paper, extended
%out to three dimensions.
%
%Inputs: xyz_a, in the form of [x  y  z]
%           where (x,y,z) is the three-dimensional coordinates of particles
%           in time frame 'a'
%
%        xyz_b, in the form of [x  y  z]
%           where (x,y,z) is the three-dimensional coordinates of particles
%           in time frame 'b'
%
%        search, an integer
%           This variable defines the maximum displacement that should be
%           seen by a single particle in any given direction
%
%        neighbor, an integer
%           This variable defines the neighborhood radius, inside of which
%           other particles will be found and may be assumed to displace
%           themselves similarly to the particle to be tracked
%
%        quasi, an integer
%           This variable defines the maximum amount of variability in any
%           group's movement, ie random-type error.  If a group of 10
%           particles uniformly displaces 3 units in the x-direction, except
%           for one which displaces 3.5 units, this variable will account
%           for that, and allow a match.
%
%        zdw, a double
%           This variable defines the reliability of the particle location
%           algorithm in the out-of-plane direction relative to the
%           in-plane direction. When performing nearest-neighbor tracking
%           or using the quasi radius, this weight is applied to reduce the
%           algorithm's reliance on out-of-plane location data
%Output: tracks, in the form of [x y z]
%           The particle number and time frame values have been removed
%           from previous versions to allow a continuous list to be
%           created.  Now, with multiple image sets, the list can be added
%           onto itself, and plotted last.  It is assumed that the first
%           row is a particle in time frame 'a', and the second row is it's
%           matching particle in time frame 'b'.  Row three is once again
%           from time frame 'a', and row four is row three's matching
%           particle from time frame 'b', and so on.
%           Any particles that are not tracked to a second frame are
%           removed from the list.
%
%On choosing parameter values:
%   -Both the search and neighborhood radii should be large enough to
%   contain approximately 5 particles inside of them.  Having less will
%   make the algorithm fall over more often, and having more will greatly
%   slow down the process.  At the same time, they should be small compared
%   to the motion of the flow - having two directions of flow or a large
%   curvature in one radius is a bad idea.
%   -The quasi-rigidity radius should be chosen small compared to the other
%   radii, and the smaller the quasi radius, the more rigid the flow must
%   be in order to track particles.  Having this radius too large will
%   allow too many fluke particles to have an influence on the correct
%   match.
%   -The neighborhood radii's size is strictly defined by the system flow
%   geometry.  If a linear flow exists, a large neighborhood radius can be
%   large, because most particles will flow in a consistent direction.  If
%   a sharp circular geometry exists, the neighborhood radius should be
%   very close to the search radius, if not less than the search radius.
%
%Written by Darren Homeniuk, October 22, 2007
%
%Inspiration for process given by paper written by S.J. Baek and S.J. Lee,
%entitled "A new two-frame particle tracking algorithm using match
%probability", 1996

% Jake Hadfield 2016/17: Added zdt/zdw for plenoptic tracking

% Test Variables
zdw = zdt;              %z-displacement weighting; see input variables

%initial declarations

in_search = [];         %structure for positions of particles in frame 'b'
                        %that are close enough to match the particle
                        %to be tracked

in_neighbor = [];       %structure for positions of particles in frame 'a'
                        %that are close enough to follow the same
                        %motion as the particle to be tracked

a = 1;                  %index to the final "tracks" array

displacement = [];      %multi-use structure

nn_track = 0;           %number of times nearest neighbor was used

pos_search = xyz_b;     %possible search particles.  Neighbors get pulled
                        %from the xyz_a array.  Potential trajectories get
                        %pulled from this array.  After being matched, a
                        %particle will be eliminated from pos_search, so it
                        %doesn't get matched again.

tracks = [];

len_frame_a = size(xyz_a,1);
len_frame_b = size(xyz_b,1);
len_positions_search = len_frame_b;

%look through frame 'a' to determine particle matches
i = 1;
%FOR PARTICLE 'i'
while i <= len_frame_a
    % FIRST TRY PREDICTIVE TRACKING


        %FIND NEIGHBORS OF PARTICLE
        for j = 1:len_frame_a
            dist1 = sqrt( (xyz_a(i,1)-xyz_a(j,1))^2 + (xyz_a(i,2)-xyz_a(j,2))^2 ...
                + (xyz_a(i,3)-xyz_a(j,3))^2 );
            if dist1 < neighbor && i ~= j
                %if a neighbor, put x,y,z coords into array
                in_neighbor = [in_neighbor; xyz_a(j,1) xyz_a(j,2) xyz_a(j,3)];
            end
        end
        [num_neighbor,~] = size(in_neighbor);

        if(num_neighbor > 0)
            %HAVE NEIGHBORS - PERFORM ADVANCED TRACKING

            %FIND FRAME 'b' PARTICLES INSIDE SEARCH RADIUS
            num_search = 0;
            for j = 1 : len_positions_search

                %calculate distance between particle being tracked in frame 'a'
                %and every particle in frame 'b'
                xd = xyz_a(i,1)-pos_search(j,1);
                yd = xyz_a(i,2)-pos_search(j,2);
                zd = xyz_a(i,3)-pos_search(j,3);
                dist1 = sqrt( (xd)^2 + (yd)^2 + (zd*zdw)^2 );
                %if the distance between frame 'a' and 'b' particles is
                %less than the search radius, it is a potential match
%                 if dist1 < search && xyz_b(j,3) > min_h
                if dist1 < search

                    %place particle in array - used later if it is a match
                    in_search = [in_search; ...
                        pos_search(j,1) pos_search(j,2) pos_search(j,3)];

                    %find distance to particle being tracked from this particle
                    displacement = [displacement; xd yd zd];

                    %increment number of search particles
                    num_search = num_search + 1;
                end
            end

            if(num_search > 0)

                %HAVE SEARCH PARTICLES - PARTICLE HAS SOME POTENTIAL MATCH

                %number of times particle found in quasi-rigidity radius
                count = zeros(num_search,1);

                %distance from centre of quasi-rigidity radius to where
                %particle actually is found - may be used later
                distance = zeros(num_search,1);

                for j = 1 : num_search
                    for k = 1 : num_neighbor
                        l = 1; flag = 0;
                        while l <= len_frame_b && flag == 0

                            %-> in_neighbor(k,:) is the k-th neighbor of the
                            %particle to be tracked

                            %-> displacement(j,:) is the displacement measured
                            %between the j-th potential match

                            %-> xyz_b(l,:) the set of frame 'b' particles
                            %that should appear somewhere around the calculated
                            %position, if the track tested is correct

                            %this basically checks the distance between where
                            %you would expect the particle to be (if it is a
                            %correct tracking direction/magnitude) against
                            %where particles actually appear in the second
                            %frame
                            xdist = in_neighbor(k,1)-displacement(j,1)-xyz_b(l,1);
                            ydist = in_neighbor(k,2)-displacement(j,2)-xyz_b(l,2);
                            zdist = in_neighbor(k,3)-displacement(j,3)-xyz_b(l,3);
                            dist1 = sqrt( (xdist)^2 + (ydist)^2 + (zdist*zdw)^2 );

                            if(dist1 < quasi)
                            %if a particle appears inside quasi-rigidity radius,
                            %then this displacement is more likely correct

                                %increment correct count
                                count(j) = count(j) + 1;

                                %add onto distance measure
                                distance(j) = distance(j) + dist1;

                                %match found - stop checking particles,
                                %check next neighbor
                                flag = 1;

                            end

                            l = l + 1;
                        end
                    end
                end

                %find displacement with most correct counts
                [maximum,index] = max(count);

                if maximum ~= 0

                    %IF HAVE PARTICLES INSIDE QUASI-RIGIDITY RADIUS
                    ind = find(count == maximum);
                    [sz_ind,~] = size(ind);

                    if(sz_ind > 1 && maximum ~= 0)

                        %if displacements have equal number of correct matches
                        %compute distance from calculated placement to actual
                        %location of particle; choose the lowest value as correct
                        index = ind(1);
                        for j = 2 : size(ind)
                            if(distance(index) > distance(ind(j)))
                                index = ind(j);
                            end
                        end
                    end

                    %this displacement is the "correct" track
                    tracks(a,:) = xyz_a(i,:);
    %                 xyz_a(i,:) = []; len_frame_a = len_frame_a - 1;
                    i = i + 1;
                    a = a + 1;

                    %note: assumption here is that the x-position will be
```

```
                    %rather unique.  The z-position is very non-unique, while
                    %the x-y locations are.  If however, two exact x-positions
                    %exist, trouble will be created by this!
                    tracks(a,:) = in_search(index,:);
                    ind1 = find(in_search(index,1) == pos_search(:,1),1,'first');
                    pos_search(ind1,:) = [];
                    len_positions_search = len_positions_search - 1;
    %                 ind1 = find(in_search(index,1) == xyz_b(:,1),1,'first');
    %                 xyz_b(ind1,:) = []; len_frame_b = len_frame_b - 1;


                    %sample plotting part 2
%                     quiver3(tracks(a-1,1),tracks(a-1,2),tracks(a-1,3),...
%                         tracks(a,1)-tracks(a-1,1),...
%                         tracks(a,2)-tracks(a-1,2),...
%                         tracks(a,3)-tracks(a-1,3),'r');

                    a = a + 1;

                else

                    %NO PARTICLES INSIDE QUASI-RIGIDITY RADIUS - ELIMINATE
                    %PARTICLE
    %                 xyz_a(i,:) = [];
    %                 len_frame_a = len_frame_a - 1;
                    i = i + 1;
                end

            else

                %NO PARTICLES INSIDE OF SEARCH RADIUS - MATCH TO PARTICLE HAS
                %DISAPPEARED AND/OR DROPPED OFF THE FACE OF THE EARTH -
                %ELIMINATE PARTICLE
    %             xyz_a(i,:) = [];
    %             len_frame_a = len_frame_a - 1;
                i = i + 1;
            end
        else

            %NO NEIGHBORS - NEAREST NEIGHBOR (WITHIN SEARCH RADIUS) KICKS IN
            b = 1;
            for k = 1 : len_positions_search
                displacement(b) = sqrt( (xyz_a(i,1)-pos_search(k,1))^2 + ...
                    (xyz_a(i,2)-pos_search(k,2))^2 + ...
                    ((xyz_a(i,3)-pos_search(k,3))*zdw)^2 );
                b = b + 1;
            end
            [minimum,index] = min(displacement);
            if(minimum < search)

                %HAVE PARTICLE INSIDE OF SEARCH RADIUS - DETERMINE SMALLEST
                %DISTANCE, AND MATCH TO THAT ONE
                nn_track = nn_track + 1;
                tracks(a,:) = xyz_a(i,:);
                i = i + 1;
    %             xyz_a(i,:) = []; len_frame_a = len_frame_a - 1;
                a = a + 1;
                tracks(a,:) = pos_search(index,:);
                pos_search(index,:) = [];
                len_positions_search = len_positions_search - 1;
    %             xyz_b(index,:) = []; len_frame_b = len_frame_b - 1;
                a = a + 1;
            else

                %NO PARTICLES INSIDE OF SEARCH RADIUS - ELIMINATE PARTICLE
                i = i + 1;
    %             xyz_a(i,:) = [];
    %             len_frame_a = len_frame_a - 1;
            end
        end

        %DONE MATCHING FOR THIS PARTICLE
        in_search = []; %clear data structures
        in_neighbor = [];
        displacement = [];


%     else
%         below_min = below_min + 1;
%         i = i + 1;
%     end
end

if ~isempty(tracks)
    [proper_length, dump] = size(tracks);
%     i = 1; num = proper_length;
%
%     %if trajectory is less than 1 um, delete it, as it will only take up
%     %room and space in the plot
%     while i <= num
%         x = (tracks(i,1) - tracks(i+1,1))^2;
%         y = (tracks(i,2) - tracks(i+1,2))^2;
%         z = (tracks(i,3) - tracks(i+1,3))^2;
%         if sqrt(x+y+z) < 1
%             tracks(i,:) = []; tracks(i,:) = [];
%             num = num - 2;
%         else
%             i = i + 2;
%         end
%     end

    %final displays
%     [mess,err] = sprintf...
%         ('%d particles have been tracked successfully...',proper_length/2);
%     disp(mess);
%     [mess,err] = sprintf...
%         ('%d particle(s) have been tracked using nearest neighbor!',nn_track);
%     disp(mess);
%     [mess,err] = sprintf...
%         ('%d particle(s) were on the bottom of the tank',below_min);
%     disp(mess);
%     elim = proper_length/2 - num/2;
%     [mess,err] = sprintf('%d tracks were too short and were hence eliminated',elim);
%     disp(mess);
else
    disp('No tracks reported here!?!');
end
```

Published with MATLAB® R2015b

E.2 *Track association: JH_MT_Associate*

The following code is responsible for associating tracks produced by the two-frame tracking algorithm that have similar endpoints.

```
function alltracks=JH_MT_Associate(tracks,alltracks,currframe)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Keeps track of which particles in a series of particle tracks are the
% same
% tracks - 2 frame tracking results
% alltracks - multi tracking results, with columns 4 and 5 incorporating
% particle number and position number, respectively. Column 6 is binary,
% representing wether particle is true (1) or predicted (0). If this input
% is left blank, initialization protocal will be performed.
% firstframe - number of first frame in tracks. May be left blank on first
% pass.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if nargin < 3 || currframe==1
    % first pass, initialize alltracks variable
    alltracks = tracks;
    for i = 1:length(tracks)/2
        alltracks(1+(i-1)*2,4)=i; % particle number
        alltracks(2+(i-1)*2,4)=i; % particle number
        alltracks(1+(i-1)*2,5)=1; % frame number
        alltracks(2+(i-1)*2,5)=2; % frame number
        alltracks(1+(i-1)*2,6)=1; % true particles only
        alltracks(2+(i-1)*2,6)=1; % true particles only
    end
else
    % subsequent passes, look at first position for matches in last
    % alltracks pass
    trackcount = max(alltracks(:,4));
    % separate out only the previous frame's tracked particles
    trackedLast = alltracks(alltracks(:,5)==currframe,:);
    % loop through new tracked particles
    new = 0;
    [ltracks,~]=size(tracks);
    for i = 1:2:ltracks
        % see if this particle was tracked successfully last time
        prevTrack = ismember(trackedLast(:,1:3),tracks(i,:),'rows');
        if any(prevTrack)
            % was tracked last time, keep old particle number and don't
            % re-add current particle
            alltracks=[alltracks;[tracks(i+1,:),trackedLast(prevTrack,4),currframe+1,1]];
            % remove predicted point, if present

alltracks(ismember(alltracks(:,4:6),[trackedLast(prevTrack,4),currframe+1,0],'rows'),:)=[];
        else
            % particle is new, assign new particle number and add particle for both frames
            new=new+1;

alltracks=[alltracks;[tracks(i,:),trackcount+new,currframe,1];[tracks(i+1,:),trackcount+new,currf
rame+1,1]];
        end
    end
    % sort the final list by particle and by frame
    alltracks = sortrows(alltracks,[4,5]);
end
```

Published with MATLAB® R2015b

E.3 *Forward-predictive tracking: JH_MT_Fit_Predict*

The following code is responsible for the forward-predictive tracking approach.

```
function
[p1,p2,alltracks,polyhold]=JH_MT_Fit_Predict(p1,p2,alltracks,frame,range,zdt,polyhold,search)
% Predicts future points from tracks' current points using polynomial
% least-squares fitting of the last n points of a track.

% set n, number of previous particles to include
na=5;
% set ap, maximum number of predicted particles allowed
ap=3;

%     estimate particle domain
%     minx=min([alltracks(:,1),p1(:,1),p2(:,1)]);
%     miny=min([alltracks(:,2),p1(:,1),p2(:,1)]);
%     minz=min([alltracks(:,3),p1(:,1),p2(:,1)]);
%     maxx=max([alltracks(:,1),p1(:,1),p2(:,1)]);
%     maxy=max([alltracks(:,2),p1(:,1),p2(:,1)]);
%     maxz=max([alltracks(:,3),p1(:,1),p2(:,1)]);

% figure out which particles were tracked last frame;
% pull out only long-enough tracks from alltracks
pNos = [];
for pN=[alltracks(alltracks(:,5)==frame,4)]';
    if sum(alltracks(:,4)==pN)>=3
        pNos = [pNos,pN];
    end
end
ltracks=alltracks(ismember(alltracks(:,4),pNos),:);
% Predict next particle location from each track
trackcount=0;
predcount=0;
for i=pNos
    pp=0;
    % fit polynomials to the particle positions
    ctracks = ltracks(ltracks(:,4)==i,:);
    [sct,~]=size(ctracks);
    pct=sum(ctracks(end-ap+1:end,6));
    if sct<na
        % track is short, but may still be useful
        if sct>=3
            n=sct;
        else
            % track too short, skip this loop iteration
            continue
        end
    else
        n=na;
    end
    if pct==0
        % too many predicted particles with nothing found, remove predicted
        % particles from alltracks and skip iteration
        alltracks(ismember(alltracks,ctracks(end-ap+1:end,:),'rows'),:)=[];
        continue
    end
    ctracks = ctracks(end-n+1:end,:);
    nt=ctracks(end,5)+1;
    if ctracks(end,6)==0
        % working with predicted point, use previous polynomials
        try
        pp=1;
        loc=find(polyhold.pno==i);
        polyx = polyhold.polyx(min(loc),:);
        polyy = polyhold.polyy(min(loc),:);
        polyz = polyhold.polyz(min(loc),:);
        mux = polyhold.mux(:,min(loc));
        muy = polyhold.muy(:,min(loc));
        muz = polyhold.muz(:,min(loc));
        catch
            error('something screwed up')
        end
    else
        % need new polynomial
        [polyx,~,mux]=polyfit(ctracks(:,5),ctracks(:,1),2);
        [polyy,~,muy]=polyfit(ctracks(:,5),ctracks(:,2),2);
        [polyz,~,muz]=polyfit(ctracks(:,5),ctracks(:,3),2);
    end

    % predict next location
    try
        predict=[polyval(polyx,nt,[],mux),polyval(polyy,nt,[],muy),polyval(polyz,nt,[],muz)];
    catch
        error('No mux, muy or muz.')
    end

    % determine if there's a match in p2
    displacement = sqrt(sum(bsxfun(@times,(bsxfun(@minus,p2,predict).^2),[1,1,zdt]),2));
    [minimum,index] = min(displacement);

    % check to make sure match doesn't exceed allowed displacement
    if sqrt(sum(bsxfun(@times,(p2(index,:)-ctracks(end,1:3)).^2,[1,1,zdt])))<(search+range)
        if(minimum < range) %*(ap-pct+1)
            % found a particle, put it in the ptracks array as a real particle
            alltracks=[alltracks;p2(index,:),i,nt,1];

            % remove p1 and p2 from their lists
            p1(ismember(p1,ctracks(end,1:3),'rows'),:)=[];
            p2(index,:)=[];

            trackcount = trackcount+1;
        else
            % put the predicted point in as a predicted particle
            alltracks=[alltracks;predict,i,nt,0];
            predcount = predcount+1;
            % check if this is a new predicted particle
            if pp==0
                % hold on to the polynomials, associate with particle number
                polyhold.pno=[polyhold.pno,i];
                polyhold.polyx=[polyhold.polyx;polyx];
                polyhold.polyy=[polyhold.polyy;polyy];
                polyhold.polyz=[polyhold.polyz;polyz];
                polyhold.mux=[polyhold.mux,mux];
                polyhold.muy=[polyhold.mux,muy];
                polyhold.muz=[polyhold.mux,muz];
            end
        end
    else
        % prediction was probably bad, send to 2f algorithm
        predcount=predcount+1;
    end
end
fprintf('%3i particles tracked using forward prediction!\n', trackcount);
fprintf('%3i predicted particles not found.\n', predcount);
% resort alltracks
alltracks = sortrows(alltracks,[4,5]);
```

Published with MATLAB® R2015b

E.4 *Track acceleration filtering: JH_MT_TrackBreak*

The following code is responsible for the acceleration-based filter that removes bad tracks.

```
function [filttracks,lostpts,acamags]=JH_MT_TrackBreak(intracks,art,zaf)
% Performs denoising operation on particle tracks by removing bad vectors
% from individual tracks

% enable acceptance of location string or full alltracks array
if isa(intracks,'char')
    load(intracks);
else
    alltracks = intracks;
end

% operation
% look at each particle track individually
tcount=max(alltracks(:,4));
ntidx=1;
lpidx=1;
filttracks = [];
lostpts = [];
warn=0;
tsf=0;
acamags=[];
%try
for i = 1:tcount
%     [lps,~]=size(lostpts);
%     [fts,~]=size(unique(filttracks,'rows'));
%     if lps+fts~=tsf
%         missing=tsf-(lps+fts)
%     end
    % pull out one track
    ctracks = alltracks(alltracks(:,4)==i,:);
    [trows,~]=size(ctracks);
    tsf=tsf+trows;
    % calculate track velocity at each point
    cvels = [ctracks(2:end,1:3)-ctracks(1:end-1,1:3) ctracks(1:end-1,4:6)];
    % calculate track acceleration if possible
    if trows>2
        caccs=[cvels(2:end,1:3)-cvels(1:end-1,1:3) ctracks(2:end-1,4:6)];
        % calculate magnitude of acceleration, include zaf
        camag=sqrt(caccs(:,1).^2+caccs(:,2).^2+zaf.*caccs(:,3).^2);
        acamags=[acamags;camag];
        % find pointlist of excessive accelerations
        exacc=find(camag>art);
        if isempty(exacc)
            filttracks = [filttracks; ctracks(:,1:3) ntidx*ones(trows,1) ctracks(:,5:6)];
            ntidx=ntidx+1;
            continue;
        end
        % acceleration warning criteria
%         warn = warn+length(exacc)/trows-0.25;
%         if warn > 25
%             warning('Many accelerations are exceeding the art, suggest higher threshold.')
%             warn=0;
%         end
%         if warn < 0
%             warn = 0;
%         end
        % break up ctracks at that (those) point(s) to remove bad links
        if trows == 3
            % points shouldn't be joined at all, send them to the lostpts array
            lostpts = [lostpts; ctracks(:,1:3) lpidx*ones(trows,1) ctracks(:,5:6)];
            lpidx = lpidx+1;
            continue;
        end
        % longer tracks: need to handle multiple bad accelerations at
        % different points along the track
        splits=diff(exacc)==1;
        groups = length(exacc)-sum(splits);
        if groups > 1
            spllocs1=[1,exacc(find(splits==0))'+2];
            spllocs2=[exacc(find(splits==0)+1)',trows+1];
            ff1=1;
            ff2=2;
        else
            spllocs1=1;
            spllocs2=trows+1;
            ff1=0;
            ff2=0;
        end
        % now have all the track-groups
        for j=1:groups
            if groups==1
                ff1=0;
                ff2=0;
                ff3=1;
            elseif j<groups
                ff1=1;
                ff2=1;
                ff3=1;
            else
                ff1=0;
                ff2=0;
                ff3=0;
            end
            csett = ctracks(spllocs1(j):spllocs2(j)-1+ff1,:);
            csetam = camag(spllocs1(j)-1+ff3:spllocs2(j)-3+ff2);
            % exaccs are all together or there's only one
            waval=max(csetam);
            wacc=find(csetam==waval);
            % make sure we aren't at the ends of the track
            try
                catvar=0;
                csetam(wacc-1);
                catvar=1;
                csetam(wacc+1);
            catch
                if catvar == 0
                    % first point is garbage
                    lostpts = [lostpts; csett(1,1:3) lpidx*ones(1,1) csett(1,5:6)];
                    csett = csett(2:end,:);
                    lpidx=lpidx+1;
                    inc=0;
                else
                    % last point is garbage
                    lostpts = [lostpts; csett(end,1:3) lpidx*ones(1,1) csett(end,5:6)];
                    csett = csett(1:end-1,:);
                    lpidx=lpidx+1;
                    inc=1;
                end
                [csl,~]=size(csett);
                filttracks = [filttracks; csett(:,1:3) ntidx*ones(csl,1) csett(:,5:6)];
                continue;
            end
            % weren't at ends, figure out which track point belongs to
            if csetam(wacc-1)>csetam(wacc+1)
                % acceleration is worse before the point, point is included after
                filttracks = [filttracks; csett(1:wacc,1:3) ntidx*ones(length(1:wacc),1)
csett(1:wacc,5:6)];
                ntidx=ntidx+1;
                filttracks = [filttracks; csett(wacc+1:end,1:3)
ntidx*ones(length(csett(wacc+1:end,1)),1) csett(wacc+1:end,5:6)];
            else
                % acceleration is worse after the point, point is included before
                filttracks = [filttracks; csett(1:wacc+1,1:3) ntidx*ones(length(1:wacc+1),1)
csett(1:wacc+1,5:6)];
                ntidx=ntidx+1;
                filttracks = [filttracks; csett(wacc+2:end,1:3)
ntidx*ones(length(csett(wacc+2:end,1)),1) csett(wacc+2:end,5:6)];
            end
        end
        ntidx=ntidx+1;
    else
        filttracks = [filttracks; ctracks(:,1:3) ntidx*ones(trows,1) ctracks(:,5:6)];
        ntidx=ntidx+1;
    end
    % double-check that we added more than one point
    if sum(filttracks(:,4)==ntidx-1)==1
        lostpts=[lostpts;filttracks(end,:)];
        filttracks=filttracks(1:end-1,:);
    end
end
filttracks = unique(filttracks,'rows');
try
    lostpts = lostpts(~ismember(lostpts(:,[1:3,5]),filttracks(:,[1:3,5]),'rows'),:);
catch
    lostpts = [];
end
%collect similar track numbers
filttracks = sortrows(filttracks,[4,5]);
%catch
%    error()
%end
end
```

Published with MATLAB® R2015b

E.5 *Track segment linkage: JH_LinkTracks*

The following code is responsible for the forward- and backward- fitting method that links together track segments.

```
function [alltracks,linkcount]=JH_LinkTracks(alltracks,range,zdt,fp)
% Backward-looking tracking trying to further link together tracked
% particles once tracking has been completed.

% look at each individual track
endtrack=max(alltracks(:,4));
endframe=max(alltracks(:,5));

selt=[];
linkcount=0;
% for all tracks
for i=endtrack:-1:1
    %fprintf('Working on track %4i of %4i.\n',endtrack-i+1,endtrack)
    if ~isempty(selt)
        fprintf('Track %4i linked to track %4i!\n',i,selt)
    end
    if i==423
        mfasdf=1;
    end
    selt = [];
    % Grab the track, find its size, select the method
    ptrack=alltracks(alltracks(:,4)==i,:);
    [tlength,~]=size(ptrack);
    if tlength==0
        continue;
    end
    if tlength==2
        % only have 2 frames, predict with velocity only in x,y. Maybe
        % just take average z for next frames.
        [polyx,~,mux]=polyfit(ptrack(:,5),ptrack(:,1),1);
        [polyy,~,muy]=polyfit(ptrack(:,5),ptrack(:,2),1);
        [polyz,~,muz]=polyfit(ptrack(:,5),ptrack(:,3),0);
    elseif tlength<5
        % don't have enough frames for z-acceleration, only use
        % average z-velocity for prediction. x and y still try to get
        % accelerations.
        [polyx,~,mux]=polyfit(ptrack(:,5),ptrack(:,1),2);
        [polyy,~,muy]=polyfit(ptrack(:,5),ptrack(:,2),2);
        [polyz,~,muz]=polyfit(ptrack(:,5),ptrack(:,3),0);
    else
        % can use polynomial fits for all 3 directions, but only want last
        % few tracks
        [polyx,~,mux]=polyfit(ptrack(end-4:end,5),ptrack(end-4:end,1),2);
        [polyy,~,muy]=polyfit(ptrack(end-4:end,5),ptrack(end-4:end,2),2);
        [polyz,~,muz]=polyfit(ptrack(end-4:end,5),ptrack(end-4:end,3),1);
    end
    % get frame at end of track
    ftime=ptrack(end,5);
    % find expected locations and velocities
    c=1;
    ftimes=(ftime+1):ftime+fp;
    ftimes=ftimes(ftimes<endframe);
    if isempty(ftimes)
        continue;
    end
    for j=ftimes
        ppoint(c,:)=[polyval(polyx,j,[],mux),polyval(polyy,j,[],muy),polyval(polyz,j,[],muz),j];
        differx=polyder(polyx);
        differy=polyder(polyy);
        differz=polyder(polyz);
        vpoint(c,:)=[polyval(differx,(j-mux(1))/mux(2).^2),polyval(differy,(j-
muy(1))/muy(2).^2),polyval(differz,(j-muz(1))/muz(2).^2),j];
        prange(c)=range*c^(1/8);
        vrange(c)=range;
        c=c+1;
    end
    % scan for candidate tracks
    tlist = [];
    % check position/time conditions of all members of alltracks
    tcond=ismember(alltracks(:,5),ftimes);
    xcond=(alltracks(:,1)<=max([ppoint(1,1)+prange(1),ppoint(1,1)-
prange(1),ppoint(end,1)+prange(end),ppoint(end,1)-
prange(end)])).*(alltracks(:,1)>=min([ppoint(1,1)+prange(1),ppoint(1,1)-
prange(1),ppoint(end,1)+prange(end),ppoint(end,1)-prange(end)]));
```

```
    ycond=(alltracks(:,2)<=max([ppoint(1,2)+prange(1),ppoint(1,2)-
prange(1),ppoint(end,2)+prange(end),ppoint(end,2)-
prange(end)])).*(alltracks(:,2)>=min([ppoint(1,2)+prange(1),ppoint(1,2)-
prange(1),ppoint(end,2)+prange(end),ppoint(end,2)-prange(end)]));
    %zcond=(alltracks(:,3)<=max([ppoint(1,3)+prange(1),ppoint(1,3)-
prange(1),ppoint(end,3)+prange(end),ppoint(end,3)-
prange(end)])).*(alltracks(:,3)>=min([ppoint(1,3)+prange(1),ppoint(1,3)-
prange(1),ppoint(end,3)+prange(end),ppoint(end,3)-prange(end)]));
    allcond=logical(tcond.*xcond.*ycond);
    if isempty(allcond)
        continue;
    end
    for k=(alltracks(allcond,4))'
        ctrack=alltracks(alltracks(:,4)==k,:);
        [ctl,~]=size(ctrack);
        if ctl<2
            continue;
        end
        % get stats
        for j=1:c-1
            sframe=ctrack(1,5);
            if sframe>max(ftimes)
                % all other tracks must start after valid ftimes
                break;
            end
            if all(sframe ~= ftimes)
                continue;
            end
            pdiff=sqrt(sum(bsxfun(@times,(ctrack(1,1:3)-ppoint(j,1:3)).^2,[1,1,zdt])));
            if pdiff > prange(j)
                continue;
            end
            vdiff=sqrt(sum(bsxfun(@times,(ctrack(2,1:3)-ctrack(1,1:3)-
vpoint(j,1:3)).^2,[1,1,zdt])));
            if vdiff < vrange(j)
                % link may be correct, put track number into tlist
                tlist=[tlist;k,pdiff,vdiff];
            end
        end
    end
    if isempty(tlist)
        continue;
    end
    % select the best track out of the candidate tracks
    [stl,~]=size(tlist);
    if stl==1
        % find the track number in alltracks, replace with initial track
        % number
        alltracks(alltracks(:,4)==tlist(1),4)=i;
    else
        % find the best match
        % best match will have the most similar velocity
        [~,mvd]=min(tlist(:,3));
        alltracks(alltracks(:,4)==tlist(mvd,1),4)=i;
    end
    linkcount=linkcount+1;
end
```

Published with MATLAB® R2015b

E.6 *Track smoothing: JH_SmoothTracks*

The following code is responsible for the polynomial temporal smoothing method.

```
function smoothtracks = JH_SmoothTracks(alltracks,fp)
% for each track
tcount=max(alltracks(:,4));
smoothtracks=[];
for i=1:tcount
    % grab current track
    ctrack = alltracks(alltracks(:,4)==i,:);
    [sct,~]=size(ctrack);
    % ignore if too short
    if sct<5
        continue;
    end
    % find start and end frame
    sf = ctrack(1,5);
    ef = ctrack(end,5);
    % for all frames in between
    for j=sf:ef
        % generate a fit curve
        fitrange=(j-fp):(j+fp);
        fitpts=ctrack(ismember(ctrack(:,5),fitrange),[1:3,5]);
        [polyx,~,mux]=polyfit(fitpts(:,4),fitpts(:,1),3);
        [polyy,~,muy]=polyfit(fitpts(:,4),fitpts(:,2),3);
        [polyz,~,muz]=polyfit(fitpts(:,4),fitpts(:,3),3);
        differx=polyder(polyx);
        differy=polyder(polyy);
        differz=polyder(polyz);
        % evaluate the fit curve at the time
        fitpoint=[polyval(polyx,j,[],mux) polyval(polyy,j,[],muy) polyval(polyz,j,[],muz)
ctrack(1,4) j polyval(differx,(j-mux(1))/mux(2).^2)*180 polyval(differy,(j-muy(1))/muy(2).^2)*180
polyval(differz,(j-muz(1))/muz(2).^2)*180];
        smoothtracks=[smoothtracks;fitpoint];
    end
    if ~isempty(ctrack)
        clc
        fprintf('Track %4i of %4i completed, with %3i points generated.\n',i,tcount,ef-sf+1);
    end
```

```
end
end
```

Published with MATLAB® R2015b

E.7 *Track length statistics: JH_TrackStats*

The following auxiliary code generates histograms for the tracking data, based on track length

```
function JH_TrackStats(alltracks)
if nargin==0

load('C:\Users\jhadfiel\Desktop\2016_12_05_Vortex\Processed\180fps\FilTracks_170106_2_Image.mat')
;
    alltracks=filttracks2;
end
% Track Length Distribution
ltracks=zeros(1,max(alltracks(:,4)));
for i=1:max(alltracks(:,4))
       ctracks = alltracks(alltracks(:,4)==i,:);
    [ltracks(i),~]=size(ctracks);
end
figure
histogram(ltracks,1:max(ltracks))
end
```

Published with MATLAB® R2015b

E.8 *Track rearrangement: JH_resortTracks*

The following auxiliary code rearranges the tracks such that all the other algorithms can work with the tracks correctly.

```
function [newtracks,misspts] = JH_resortTracks(alltracks)
frames = max(alltracks(:,5));
newtracks=alltracks;
misspts=[];
ntl=1;
for i = 1:frames
    % find this i as track numbers
    tlist=alltracks(alltracks(:,5)==i,4);
    for j=tlist'
        tlocs=alltracks(:,4)==j;
        ctracks=alltracks(tlocs,:);
        alltracks=alltracks(~tlocs,:);
        [sct,~]=size(ctracks);
        if sct>1
            newtracks(ntl:ntl+sct-1,:)=ctracks;
            ntl=ntl+sct;
        else
            misspts=[misspts;ctracks];
        end
    end
end
```

Published with MATLAB® R2015b

E.9 *Track output: JH_TrackToQuiv*

The following auxiliary code outputs tracks in a format expected by the plotting code.

```
function JH_TrackToQuiv(path,TrackName)
if nargin~=2
    [TrackName,path]=uigetfile('*.mat');
end
tracks=importdata([path,TrackName]);

quivarray=zeros(length(tracks)-max(tracks(:,4)),7);
lasti=0;
for j=1:max(tracks(:,4))
    track=tracks(tracks(:,4)==j,:);
    if isempty(track)
        continue
    end
    [s,~] = size(track);     %grab height of full matrix

    X = zeros(s-1,1); Y = zeros(s-1,1); Z = zeros(s-1,1); F = zeros(s-1,1); %initialize starting
points as null for quiver use
    U = zeros(s-1,1); V = zeros(s-1,1); W = zeros(s-1,1); %initialize vector lengths as null for
quiver use

    for i=1:s-1
        %grab all x,y,z (starting coordinates)
        X(i) = track(i,1);
        Y(i) = track(i,2);
        Z(i) = track(i,3);
        %grab time
        F(i) = track(i,5);
        %get vectors
        U(i) = (track(i+1,1) - X(i));
        V(i) = (track(i+1,2) - Y(i));
        W(i) = (track(i+1,3) - Z(i));
    end
    quivarray(lasti+1:lasti+i,:)=[X,Y,Z,U,V,W,F];
    lasti=lasti+i;
```

```
end
save([path TrackName(1:end-4) '_quiv.mat'],'quivarray')
```

Published with MATLAB® R2015b