Three-dimensional Fluid Flow Measurement Techniques with Applications to Biological Flows

Roderick R. La Foy

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Mechanical Engineering

John J. Socha, Chair
Thomas E. Diller
Mark R. Paul
Mark A. Stremler
Danesh K. Tafti

August 12, 2022
Blacksburg, Virginia

Keywords: PIV, Tomography, Fluid Flow

Three-dimensional Fluid Flow Measurement Techniques with Applications to Biological Flows

Roderick R. La Foy

(ABSTRACT)

The accuracy of plenoptic and tomographic particle image velocimetry (PIV) experimental methods is measured by simulating three dimensional flows and measuring the errors in the estimated versus true velocity fields. Parametric studies investigate the accuracy of these methods by simulating a range of camera numbers, camera angles, calibration errors, and particle densities. The plenoptic simulations combine lightfield imaging techniques with standard tomographic techniques and are shown to produce higher fidelity measurements than either technique alone. The tomographic PIV simulations are centered around testing software developed for processing large quantities of data that were produced during an experimental investigation of the flow field about a 3D printed model of the flying snake *Chrysopelea paradisi*. A description of this tomographic PIV experiment is given along with basic results and recommendations for future investigation.

Three-dimensional Fluid Flow Measurement Techniques with Applications to Biological Flows

Roderick R. La Foy

(GENERAL AUDIENCE ABSTRACT)

Two different experimental measurement techniques that can be used to measure three-dimensional fluid flow fields are discussed. The first measurement technique that is investigated in simulations uses cameras with arrays of lenses to simultaneously capture images of a flow field from multiple different angles. A method of combining the data from multiple cameras is discussed and shown to yield more accurate estimates of the three-dimensional flow fields than from a single camera alone. An additional measurement technique that uses a group of standard cameras to measure three-dimensional flow fields is also discussed with respect to software that was developed for processing a large volume dataset. This software was developed for processing data collected during an experimental investigation of the flow field about a 3D printed model of the flying snake *Chrysopelea paradisi*. A description of this experiment is given along with basic results and recommendations for future investigation.

Acknowledgments

Many thanks are due to Harriet Angel who has spent significant time analyzing data, looking through code, and generally providing encouragement with this work. Her support on this project has been invaluable.

I would also like to thank Susan Carr for being an amazing friend through all of this. You have given me advice, someone to talk to when needed, a distraction when I am stressed, and above all, unwavering support and love. This likely would not have been possible without you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Complex fluid flows are ubiquitous in the world and occur in many purely natural phenomena as well as a very large number of engineered systems. Understanding the flow in these systems is important both for a clearer understanding of the physical world and for creating more efficient and robust engineering designs. Yet despite the prevalence and importance of fluid flows, much of the dynamics of these systems are not well understood. There are many reasons for this lack of understanding including the highly complex nature of many flows, the difficulty in modeling and simulating flows, and the problematic nature of observing and accurately measuring flows. All of these problems have been intensely studied in an attempt to contribute to this full body of knowledge, but this work will primarily focus on those problems related to the measurement of flows. In particular, this work specifically investigates methods for collecting volumetric flow field measurements applied to the natural system of the flying snake.

1.1 Motivation

There are many interesting fluid dynamics phenomena that could be investigated that require an understanding of the fully three-dimensional nature of the flow. For various reasons that will be explained later, it is significantly easier to study two-dimensional (or even one-dimensional) flow fields. Many interesting flows have been investigated from a purely planar

viewpoint. While this can yield many valuable insights and in some cases, can relatively accurately model the full dynamics of the flow field, many flows are inherently three-dimensional. Despite the importance of measuring the fully three-dimensional nature of the flows, many experimental investigations still primarily focus on two-dimensional systems due to the experimental and computational simplicity of the measurements. Experimental measurements studying the fluid flows governing the dynamics of the flying snake *Chrysopelea paradisi* is one such case.

The genus of flying snakes native to Southeast Asia have evolved the ability to leap from trees, spread their ribs to flatten dorsoventrally to form an airfoil, and then glide to the ground or other trees while undulating in an 'S'-shape [58, 59]. This specific mode of flight is unique among animals and is interesting to investigate for this reason alone. In addition to this, previous two-dimensional studies of the fluid dynamics governing the flight of the snake have shown interesting dynamics including a large jump in the lift-to-drag ratio of the snake at a 35 degree angle of attack and a very gentle stall behavior up to large angles of attack [25, 27].

While these insights are inherently valuable, the fluid flow around the body of the snake is inherently three-dimensional due to the 'S'-shaped asymmetry of the body during flight, the undulatory motion of the snake, and the angle of attack of the snake body causing the flow to transition into the turbulent regime for at least some flight cases [25]. Thus it is likely that some of the important dynamics governing the snake's flight could not be experimentally observed in the two-dimensional model case.

Fully understanding the three-dimensional flow dynamics about the flying snake would require measuring the flow around living specimen during flight. However, measuring the fluid velocity field about a living snake during flight is difficult for several reasons, both logistically and technically. From a logistical standpoint, working with animals is difficult due to

getting experimental approval and the animals not always being cooperative with the experimentalist. But from a technical perspective, collecting tomographic measurements around the snakes during gliding would be very difficult. To accurately measure the velocity field at a sufficiently high resolution to resolve important flow features about the snake requires that the field of view of the cameras be approximately 20 cm on a side. However, during gliding experiments, the snakes typically traverse tens of meters, so ensuring the snake travels through this narrow field of view would be very difficult. Additionally, seeding the air surrounding the snake with particles would be difficult since typically fog machines are used to aerosolize fine droplets of oil for air flows - which might change the snake's behavior or potentially be dangerous to the snake.

For these reasons, the experiment designed to study the three-dimensional flow characteristics about the snake body was designed to use a 3D printed model of the snake mounted in a water tunnel. A technique known as tomographic particle image velocimetry (PIV) was selected to measure the three-dimensional flow field about the snake model in the water tunnel experiments.

1.2 Tomographic PIV

In two dimensions, PIV typically involves illuminating a fluid seeded with reflective, low mass particles with a focused light source, commonly a laser, while cameras record the light emitted from the particles. By imaging many particles at once over at least two time steps, a map of the full flow field across a two-dimensional plane is recorded. To extract the velocity field from these images, the motion of the particles must be tracked. A variety of methods can be used to perform this measurement, but typically small windows of the image are extracted and cross-correlated across multiple time steps. For a coherent, spatially and

temporarily resolved flow field, the correlation function will have a peak intensity located at a distance from the origin that directly corresponds to the average displacement over the windows. In this way, each cross-correlation will yield a single velocity vector. By combining multiple cross-correlations over a large spatial and possibly temporal region, the flow field can be measured in a two-dimensional plane.

This technique is relatively straight-forward to implement both experimentally and computationally. However, extending this to the measurement of three-dimensional flow fields requires a significantly more complex experimental setup, additional data to be collected, and much higher computational cost [8, 9, 10]. Since the position of the particles in the flow field must be triangulated, either through direct calculation of their position or through the reconstruction of a three-dimensional image, at least one additional camera is required during the data acquisition process; in practice four or more cameras are typically used [8, 52]. The relative positions of the cameras with respect to one-another and to the illuminated fluid must also be recorded as a calibration function for each camera. This calibration function may be recorded through a variety of methods, but generally a series of images are collected from all cameras of a grid of precisely positioned points that is translated through the volume of interest [60].

Small errors in the calibration functions may result in relatively large errors in the measured velocity flow field [65]. Thus a self-calibration procedure is applied utilizing the particle image data to refine the measured calibration function. This method utilizes the fact that since the particles must have had a single physical location during the recording process, their triangulated positions must align amongst all cameras. Any discrepancies between the triangulated positions from the different subsets of cameras must be due purely to error in the calibration functions. By measuring these discrepancies for a large number of different particles, the error of the calibration function may be estimated over the entire measured

flow field. Then to correct the calibration functions, this error is added back onto the original calibration function estimate. This process may be repeated iteratively to refine the calibration functions to a very low level of error that will result in accurate measurement of the flow field.

The velocity field may then be measured by either directly calculating the displacement of the particles in the field or by reconstructing a three-dimensional image of the particle field and applying the cross-correlation techniques applied to the two-dimensional PIV system. Calculation of this three-dimensional image is known as tomographic reconstruction [8, 9, 10].

The generation of the three-dimensional image from the collected camera data involves two steps. First, the calibration function is used to generate a weighting matrix that relates the voxels in the three-dimensional image to each of the pixels in the cameras' two-dimensional images [1, 2, 3, 8, 10]. This process is relatively computationally expensive, but only must be calculated once for a single set of camera positions. After this, the individual particle image frames can be used along with this weighting matrix to reconstruct the three-dimensional representation of the particle field.

In two dimensions, the particle image that is processed (using cross-correlation or other velocimetric methods) is an accurate representation of the physical particle field. There will be errors in the image data due to a variety of sources including the focusing of the lens and depth-of-field effects, the finite resolution of the imaging system, mis-alignment of the light sheet and the focal plane of the camera, light scattering effects about the particles, along with many other potential error sources. While all of these sources do contribute to increasing noise in the calculated velocity field, the image that the camera records is still an accurate representation of the particle field. Any particles imaged by the camera correspond to physical particles that are illuminated in the flow field. However, this is not the case with the reconstructed three-dimensional images used in tomographic PIV.

The tomographic reconstruction of the three-dimensional particle field image requires solving a massively underdetermined system. Typically the number of unknown variables outranks the known values by a factor of hundreds. Assumptions about the nature of PIV recordings, specifically that small bright regions appear on an otherwise dark background, can be used in the reconstruction calculations to lessen the effect of the indeterminacy of the system [2, 3]. However, errors in the reconstructed image are unavoidable and unlike in the two-dimensional case, the reconstructed images do not represent a physical system. Specifically, particles will be generated in the image that did not exist in physical reality. These non-physical particles are commonly referred to as ghost particles and are the largest source of noise in tomographically measured velocity fields [1, 2, 3, 8, 11].

The effects of ghost particles may be minimized during the experimental phase of the investigation through several means. First, the number of ghost particles scales with the particle density. This suggests that lowering the total number of particles will help to eliminate noise from the ghost particles. However, decreasing the particle density will inevitably result in lower resolution velocity field measurements since at best, only one velocity vector can be measured per particle. Realistically, the actual velocity field resolution is much lower than this, so in practice the particle density cannot be significantly lowered. A second source of ghost particles is due to sensor noise being reconstructed. In general, this problem is easier to solve since the signal-to-noise ratio of the recorded particle images scales with the intensity of source used to illuminate the particles within the fluid. The snake model experiment, however, required cutting out a large portion of the particle illumination due to the nature of the experiment. This resulted in more advanced tomographic PIV algorithms being required to process the data from the experiment.

1.3 Plenoptic PIV

Plenoptic or lightfield cameras have been proposed for performing single camera three-dimensional fluid velocity field PIV measurements due to their unique advantage of recording both the spatial variation and the propagation direction of the incoming light. The combination of both the spatial and angular information of the light yields the lightfield function, which in turn can be used to computationally reconstruct the three-dimensional particle field.

Unfortunately, single camera plenoptic PIV measurements suffer from low angular resolution limitations similar to single camera defocussing PIV [26, 46, 47] and holographic PIV techniques, which are detrimental to PIV measurements. To increase the quality of lightfield volumetric PIV, a multi-camera plenoptic reconstruction algorithm is developed and presented. This algorithm combines the three-dimensional reconstructions from each individual camera by taking a scaled product of the individual reconstructions. Using lightray simulations it is demonstrated that using two plenoptic cameras can produce three-dimensional intensity fields with much higher fidelity to the true field than the reconstructions produced by a single plenoptic camera. Using additional plenoptic cameras is shown to only incrementally improve the reconstruction quality. To directly compare the dual plenoptic camera PIV methodology with traditional volumetric PIV techniques, standard camera images from two to four cameras were also simulated and used to create tomographic reconstructions. The intensity fields produced using two plenoptic cameras were shown to have higher fidelity than those produced using tomographic PIV techniques with up to four standard cameras in the same configuration. This work shows that using multiple plenoptic cameras for PIV measurements has the potential to produce higher fidelity data than using traditional three-dimensional measurement methods.

1.4 Summary of Work

The following chapters begin by summarizing the work that was completed during a tomographic PIV experiment investigating the flying snakes in Chapter 2. This experiment was motivated by the interesting flight dynamics that are already known from previous studies with the flying snakes. Specifically, the experiment intended to use the tools of tomographic PIV to investigate the three-dimensional flow dynamics around a static model of a snake. The data from this experiment would ideally provide valuable insights into the flight dynamics as well as provide a test case for the development of three-dimensional measurement techniques.

The collecting and processing of the tomographic PIV snake data is discussed in detail in this chapter. Descriptions of how the data is processed and validated are given, but these primarily focus on setting the data up to the point that it can be analyzed with standard tomographic PIV tools, which is described more fully in the following chapter.

Chapter 3 describes the tomographic PIV software that was used to process the snake model experimental data. This chapter summarizes the tomographic tools that were developed as well as describes novel processing methods that were incorporated into the software to aid in the processing of the snake model data. Since the software was newly developed for processing the experimental data, simulated test cases are also described. These test cases are used to validate the software and show that it gives outputs that are similar to other tomographic PIV software that has been developed.

In Chapter 4, a novel three-dimensional PIV measurement technique is described. This method uses unique cameras that not only record the radiance information of a scene, but also record the directional information of the light in the scene. Since the primary concept behind tomography is that a three-dimensional image can be built up from two-dimensional

images taken from different directions, this means that these cameras, known as plenoptic or lightfield cameras, can create tomographic reconstructions with a single camera.

The work in this chapter investigates combining the data from multiple plenoptic cameras to generate three-dimensional PIV data. It is shown that using just two plenoptic cameras can produce high quality data. However, the process of using these cameras with experimental data is significantly more intensive than using standard cameras, so these cameras were not used with the tomographic snake model experiment.

Finally, in Chapter 5 a summary of this work is given. The results of the projects are related to one-another and plans and recommendations for future work are given.

Chapter 2

Snake Model Tomographic PIV Experiment

2.1 Introduction

A genus of snakes native to Southeast Asia have evolved the ability to leap from trees, spread their ribs to flatten dorsoventrally to form an airfoil, and then glide to the ground or other trees while undulating in an 'S'-shape [58, 59]. This specific mode of flight is unique among animals and is interesting to investigate for this reason alone. Previous fluid dynamics studies of these snakes and the species *Chrysopelea paradisi* in particular, which is known for having the best flight capabilities in the genus, have primarily focused on two-dimensional flow. This work has shown interesting dynamics including a large jump in the lift-to-drag ratio of the snake at a 35 degree angle of attack and a very gentle stall behavior up to large angles of attack [25].

While these insights are inherently valuable, the fluid flow around the body of the snake is inherently three-dimensional due to the 'S'-shaped asymmetry of the body during flight, the undulatory motion of the snake, and the fact that the Reynolds number range observed of the snake during flight indicates that the flow is likely transitioning to turbulence for at least some flight cases [25]. Thus it is likely that some of the important dynamics governing the snake's flight could not be experimentally observed in the two-dimensional model case

and a fully three-dimensional, time-resolved experiment investigating the flight dynamics was developed.

This experiment was designed to measure the flow dynamics about a static model of the snake placed within a water tunnel with flow speed matching the Reynolds number of a snake in flight. Tomographic PIV data was collected to measure the three-dimensional flow field surrounding the full body of the snake model. By investigating the time-resolved behavior of flow structures, this data would ideally be able to address several open questions related to the snake flight. Does the 'S'-shape of the snake strongly affect the dynamics or can the dynamics be primarily characterized as a series of tandem air-foils? Does the shape of the snake inhibit vortex shedding, and would this positively or negatively affect flight? Does the lift augmentation at 35 degrees still exist in a fully three-dimensional model? Lastly, how does the snake flying in a turbulent transition region affect the flight dynamics?

The experiment was broken down into several components. First, the physical experiment had to be constructed. A 3D model of the snake had to be manufactured that was suitable for use in a water tunnel in addition to a system to securely and accurately position the snake model within the water tunnel. Since the cameras collecting the tomographic PIV data would be viewing the snake model at an angle, tilt-shift adapters had to be designed and manufactured to produce focused PIV images. Finally, a frame to hold the cameras needed to be created. Since creating accurate tomographic PIV data relies on very precise positioning, the camera frame needed to hold the cameras in position such that any motions due to vibrations, settling, et cetera would be on the order of 10 μm or less.

Once the data was collected, processing involved several steps. In tomographic PIV, the two-dimensional images from multiple cameras are combined together to reconstruct a three-dimensional image. Then cross-correlations are applied to windows within this three-dimensional image to yield an estimate of the velocity field. Since the 3D model of the snake is stationary

within the three-dimensional images, the model must be computationally masked to avoid introducing errors to the velocity field. Masking the model required estimating its precise position within the three-dimensional experimental volume. Additionally, since the calibration error of the cameras must be on the same order as the inherent PIV error to avoid introducing additional error in the velocity field, the calibration error must be corrected to approximately 0.1 pixels to yield accurate velocity measurements. Thus, a self-calibration procedure using the imaged particle positions was also applied prior to calculating the reconstructions and measuring the velocity field.

2.2 Experimental Design

2.2.1 3D Snake Model System

During gliding, the snake *Chrysopelea paradisi* undulates its body; however, the velocity of the undulations was much lower than the forward velocity of the snake, so it was assumed that a static model of the snake was appropriate for the experimental investigation [25]. The pose and geometry of the snake model was chosen as a typical case from photos and video of the snake during gliding [59]. A three-dimensional CAD model of the snake was created and 3D printed at a one-to-one scale to the real snake. To mount the snake model, a stainless steel support bar was added to the model protruding downstream so as to minimize any effects to the flow field.

The 3D printed model was slightly porous which could potentially detrimentally affect the experiments. Thus the surface of the model was sealed by coating it in a thin layer of acetone to dissolve the ABS plastic. Once the acetone evaporated, the ABS plastic re-solidified thus sealing the small pores on the surface of the model. Additionally, the laser light reflected from

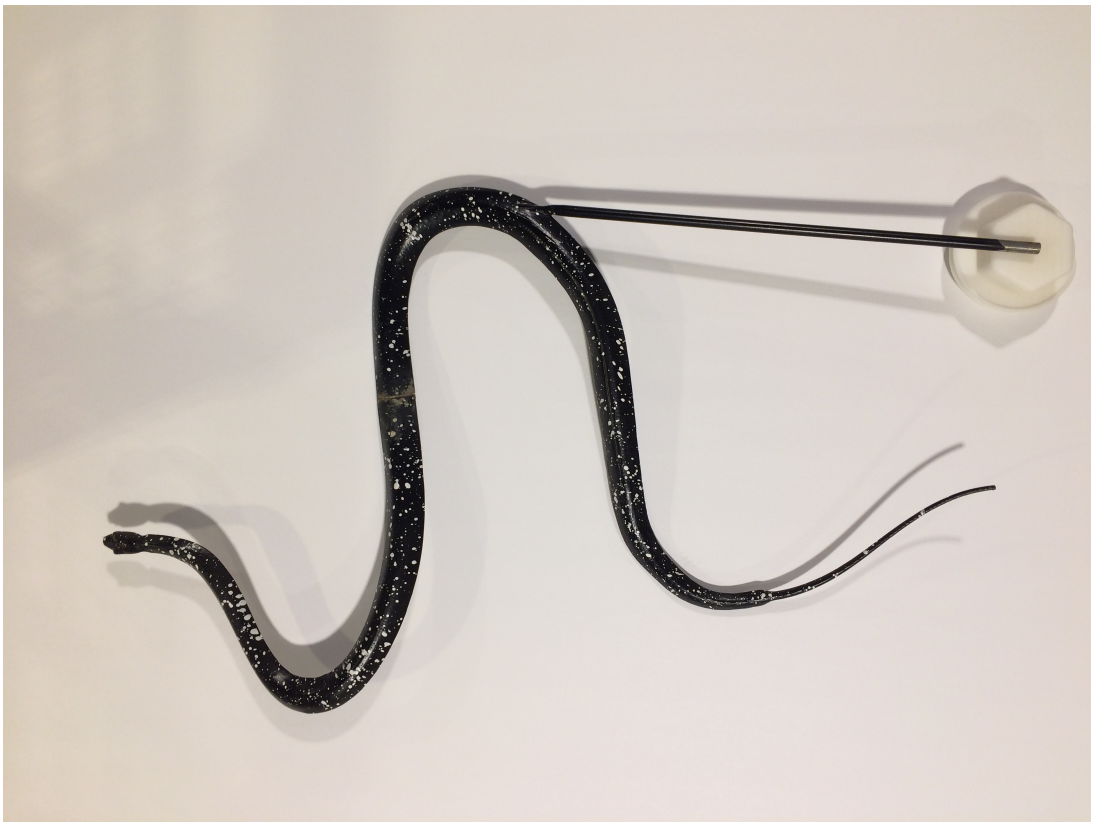


Figure 2.1: This is a photo of the 3D printed snake model showing the painted black background and the speckle painted calibration pattern. A stainless steel support cantilever beam holds the model in place during experiments.

the model body was assumed to be a large potential source of error in the measurements due to over saturation of the camera images, thus the model was painted with black water-proof paint to minimize reflections. Also, since knowing the exact coordinates of the surface of the model within the images is critical to understand the dynamics of the flow about the snake, the 3D printed model was splatter painted with white paint to provide reference points for reconstructing the surface position in the camera images. A photo of the snake model used in the experiments is shown in Figure 2.1.

The support cantilever was clamped to the end of a 25.4 mm diameter stainless steel cylinder to rigidly support the model during testing. The Reynolds number of the cylinder matches

the value of approximately 13000 that the snake model experiences during testing, so the cylinder by itself will induce turbulent vortex shedding. Additionally, the wake from the snake model may interact with the support rod affecting the snake model flow dynamics. Both of these interactions would detrimentally affect the measured flow dynamics about the snake model, so the vertical support cylinder was surrounded by a 3D printed NACA 0012 airfoil that decreased the net drag on the vertical support by approximately 90% while simultaneously minimizing any wake interactions or vibrations due to vortex shedding. While the airfoil would never be directly illuminated during the experiments, stray laser light could still detrimentally affect the measurements, so the airfoil was also painted with water-proof black paint to minimize any reflections.

The snake model system including the model, the support rod, and the surrounding airfoil were all suspended over the water tunnel and positioned by a 3-axis linear motor system that could be controlled to approximately 0.1 mm accuracy within the water tunnel. The snake model is approximately 60 cm long, so to collect high resolution data, the experimental volume was divided into regions approximately 12 cm on a side, and the snake model was moved within the water tunnel by the linear motors so that dynamical data could be collected over the whole model. A diagram showing the snake model experimental system is shown in Figure 2.2.

The water tunnel had a cross section of approximately 610 mm by 610 mm and was operated at a flow speed of approximately 0.5 m/s during the experiments. The width of the water tunnel limited the range of angle-of-attacks that could be measured with the snake model without edge effects interfering with the model. Due to this limitation, two angles-of-attack were chosen for the experiments: 20 degrees and 35 degrees. The 20 degree case allowed for the full body to be imaged. Data could only be collected near the center of the snake model for the 35 degree angle-of-attack case due to the model approaching the water tunnel walls,

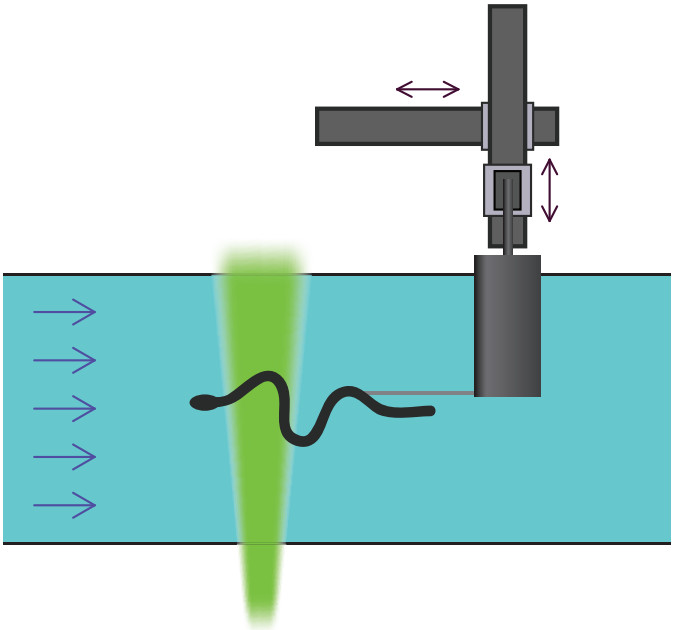
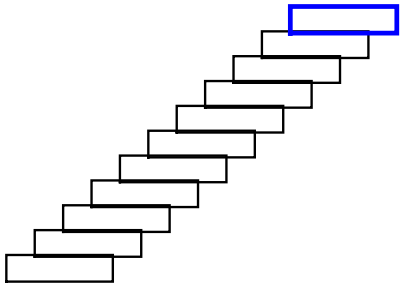


Figure 2.2: This shows a diagram of the snake model experimental system along with the X and Y-axis linear motors. The arrows in the water tunnel indicate the direction of flow while the green indicates the laser illumination.

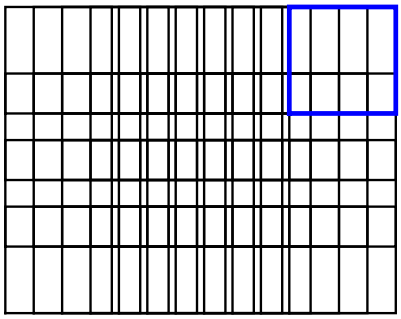
however, this case was chosen to investigate the lift augmentation that was seen to occur at this angle with the two-dimensional studies. The 20 degree case was divided into 44 imaging positions: 4 Y-axis positions and 11 XZ-plane positions. The 35 degree case was then divided into 16 imaging positions: 4 Y-axis positions and 4 ZX-plane positions. Diagrams showing the relative positions of the imaging regions-of-interest are shown in Figure 2.3.

2.2.2 PIV Imaging System

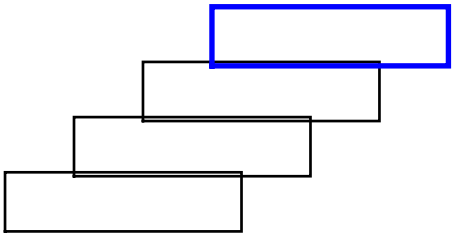
The snake model was illuminated from below using a New Wave Pegasus 20 W dual head 527 nm Nd:YLF laser with both heads simultaneously pulsed at 1000 Hz to produce 20 mJ per pulse. The system was imaged with four Photron APX-RS cameras with 1024 by 1024 pixel resolution. The expanded laser beam was aimed to illuminate an approximately 12 cm by 12 cm by 3 cm region located near the center of the water tunnel. Shutters were used to



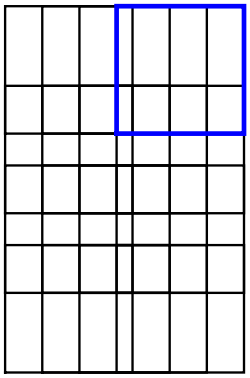
(a) 20 degree AOA XZ-plane ROIs



(b) 20 degree AOA XY-plane ROIs



(c) 35 degree AOA XZ-plane ROIs



(d) 35 degree AOA XY-plane ROIs

Figure 2.3: This diagram shows the relative position of the three-dimensional volumes that were measured during the tomographic PIV experiment in the XY and XZ planes. The volumes were approximately 12 cm by 12 cm by 3 cm in size in the X, Y, and Z directions respectively. The highlighted blue region shows the size of a single volume.

cut off the edge of the laser beam so that the illuminated region had a sharp decrease in the particle intensity along the camera axis (which decreases experimental noise while providing a metric of the quality of the tomographic reconstructions).

The cameras were positioned in a 2 by 2 arrangement surrounding the illuminated region and angled at approximately 35 degrees off of the laser sheet plane axis. A diagram of the experimental setup including the cameras is shown in Figure 2.4. Universal Scheimpflug adapters specifically designed and 3D printed for this experiment were attached to the cam-

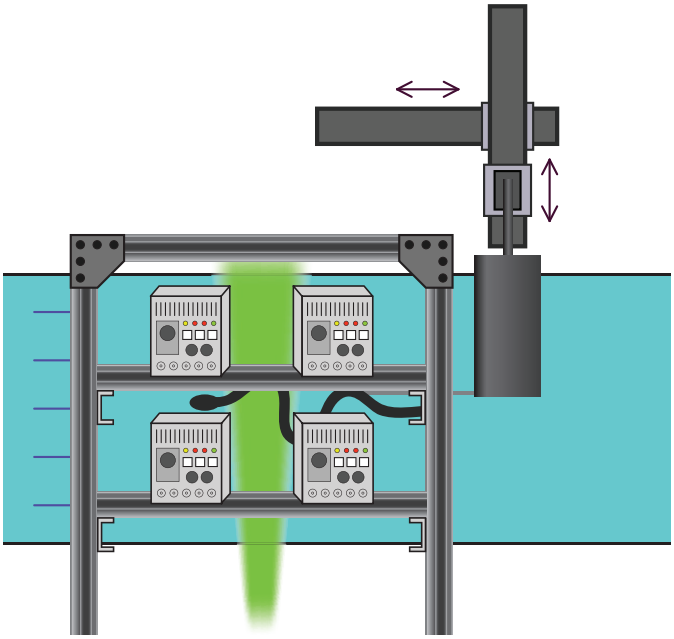


Figure 2.4: This diagram shows the relative position of the four imaging cameras connected to the support frame with respect to the snake model within the water tunnel.

eras to allow the 105 mm lenses to be focused at an off-axis angle. To ensure that the entire illuminated volume remained in focus, the lenses were stopped down to an aperture of $f/4.0$ resulting in a depth-of-field of approximately 4 cm.

A large amount of the laser light was expected to reflect from the snake model. This would likely over-saturate the camera images, resulting in lost data near the model. To eliminate this effect, particles were chosen that could absorb the laser light and then emit the light back at a different frequency. Then the laser light can be filtered out from the camera images. To accomplish this, the water was seeded with Thermo Scientific 13 μm diameter particles that had a peak absorbance wavelength of 542 nm and a peak emission wavelength of 612 nm. To remove the laser light reflected from the model, Edmund Scientific 590 nm long-pass filters were placed over the camera lenses. This successfully mitigated the reflected light; however, this also resulted in the particles being very dim in the camera images. Thus advanced filtering and processing methods will be used to process the data to account for

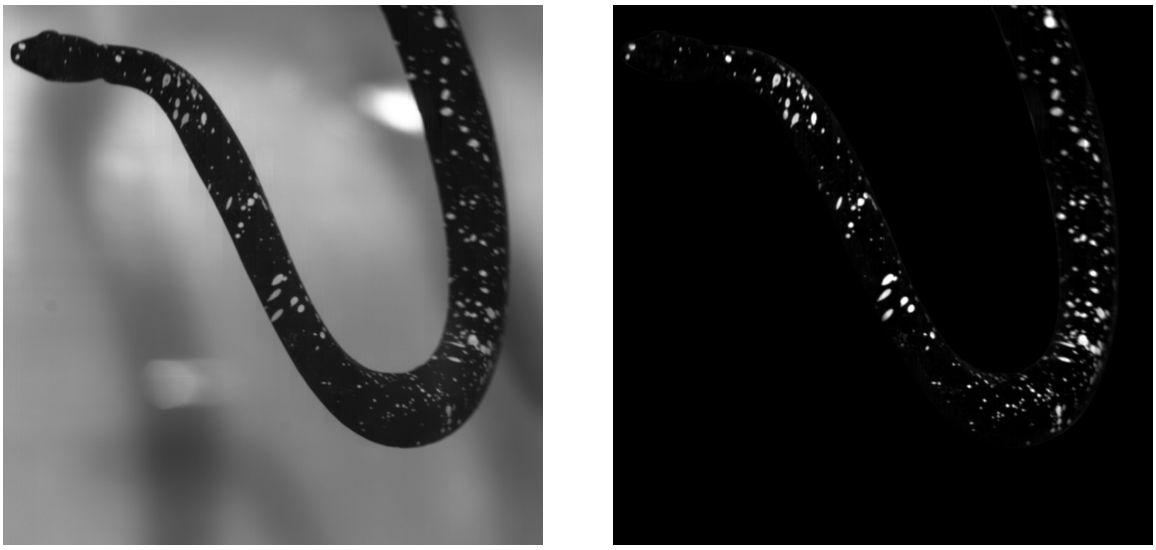
the low signal-to-noise ratio in the camera images.

Once the particles were added to the water tunnel, the particle density was measured using a time averaged particle identification algorithm. The particle density was initially measured to be between 0.07 and 0.08 particles per pixel. Typically tomographic PIV experiments are designed to have particle densities between 0.01 and 0.10 particles per pixel. It was expected that a large number of particles would be lost during the course of the experiment due to settling and unintentional filtering of the particles by the water tunnel flow-straightening system. The loss of particles was qualitatively apparent throughout the course of the experiment, so additional particles were continually added to the water over the course of the approximate week that it took to collect the data.

2.2.3 Data Collection

Four different types of data were collected from the tomographic PIV experiment. In addition to the particle field images surrounding the snake model, images were collected with bright field illumination of the snake model to determine its position. Additionally, particle field images without a model in the water tunnel were collected to use as a reference of the laser illumination. Finally, calibration images were collected of a calibration grid to determine the relative camera positions.

For each of the model positions, the full camera buffer of 6144 frames was recorded. The dominant oscillation frequency of the vortex shedding from the snake model was approximately $f = 0.47$ Hz, so this corresponds to about 13 oscillations per recording, ensuring that the full dynamics should be recorded in each data set. While the model was relatively stationary, the full camera buffer of data was recorded so that the specific motion and the extent of any oscillations of the model could also be measured. The full buffer of particle



(a) Bright field model photo (b) Masked model photo

Figure 2.5: (a) A photo showing an example bright field photo of the physical snake model. (b) The same photo, but with masking applied so that an image registration calculation can be performed. An intensity normalization was also applied to the iamge so that the speckle pattern spots all have roughly uniform intensity levels.

image data was recorded with the expectation that dominant flow features may be identified with relatively high resolution by performing a phase averaging of the flow oscillations.

Additionally, at each model position, the laser was shut off and room lights were used to illuminate the snake model within the water tunnel. Since the water tunnel flow caused the model to deform by approximately 1 cm, the tunnel was left flowing at the same speed as during the PIV tests to ensure that the model remained in the same position. While the snake model remained in the same position, sensor noise was still visible in the camera images, so 1000 bright field images of the model were collected so that an average of the images could be calculated to reduce the noise effects. An example bright field model image is shown in Figure 2.5a. During processing, these images were masked as in Figure 2.5b and an image registration process was formed to match the physical and computational models.

A series of particle images was also collected with the snake model removed from the water

tunnel both before and after the tomographic data was collected. Since the laser used to illuminate the volume has a Gaussian intensity profile, the illumination within the measurement volume also retains a Gaussian profile once the laser beam is optically expanded. Additionally, scratches or dust on any of the optical surfaces that the beam passes through may introduce additional inhomogeneities. The variation in particle intensity can introduce additional error in both the tomographic reconstruction and in the cross-correlations. Both of these effects will result in higher errors when the velocity field is calculated, so the average particle image was calculated and used as a normalization factor for rescaling the particle images that contained the model. This data was also used in the self-calibration procedure described later.

Calibration data for the cameras was recorded by collecting images of a LaVision type 31 dual level calibration grid traversed across a 42 mm distance in 3 mm Z-axis increments. Calibration grid data was collected both before and after the tomographic data collection. An example calibration grid image is shown in Figure 2.6. It is apparent in this image that this data will only result in a low resolution estimate of the calibration functions due to the error in estimating the center of the calibration grid circles and the large spacing between the circles. The circular grid points are spaced 15 mm apart, but due to the dual level grid pattern, the XY-plane grid resolution is 7.5 mm. While this resolution is relatively coarse compared to the approximately 0.17 mm per pixel resolution of the images at this magnification, the camera calibration functions are locally highly linear and can globally be approximated by a cubic polynomial function [3, 60].

The image coordinates of the calibration grid markers were determined to a sub-pixel accuracy using in-house calibration software written in the MATLAB programming language. Additionally, the errors in the calibration functions were later reduced by using the particle positions within the images as a higher resolution calibration data set in a process known as

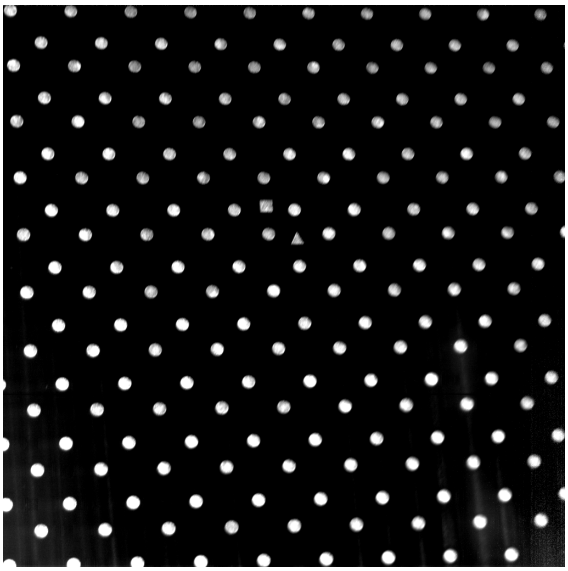


Figure 2.6: A photo of a single Z-axis position of the calibration grid from one camera. The calibration grid has two levels of grid points with the square and triangle patterns centered on each of these levels. The rectilinear grid pattern was offset by 7.5 mm in both the X and Y directions between the two levels.

self-calibration.

2.3 Data Processing

The data processing of the tomographic PIV data from the snake model experiment can be broken down into several steps. First, a self-calibration procedure is performed on the particle image data to yield higher accuracy calibration functions. Next, this newly calculated set of calibration functions is used along with the bright field model images to determine the position of the snake model within the measurement volumes. This model position data is then used to mask the three-dimensional reconstruction images. Finally, the velocity field is calculated by performing cross-correlations on the reconstructed images.

2.3.1 Self-Calibration

The calibration process that fits a cubic polynomial to the calibration grid coordinates is known to commonly have errors between 0.5 and 2.0 pixels [65]. However, to calculate accurate tomographic reconstructions, the calibration error generally needs to be less than about 0.4 pixels [8]. Therefore a way of refining the initial calibration function is needed.

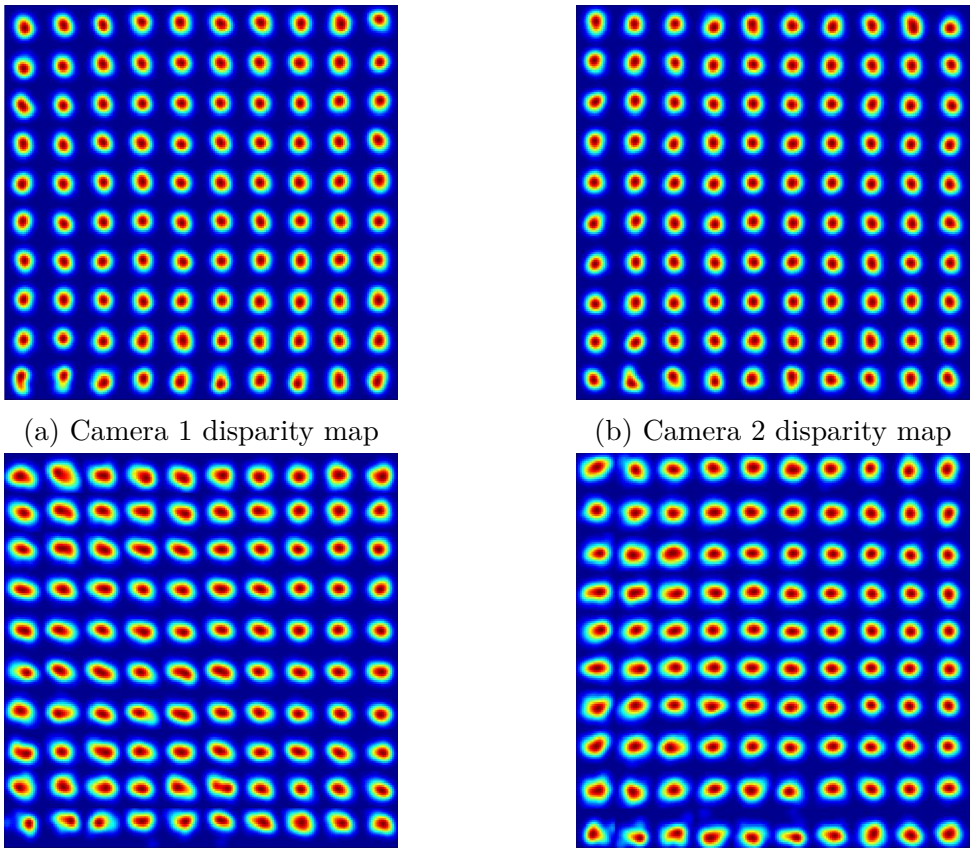
The camera calibration functions relate the world coordinates of the experiment to the pixel coordinates in the camera images. While the calibration grid data is relatively low resolution in both the world and camera image coordinates, the flow tracer particles may be used to produce high resolution calibration data since the particles will tend to densely mix throughout the experimental volume and the particle positions can be accurately determined to sub-pixel accuracy. To produce this calibration data, a triangulation procedure may be used to estimate the world coordinates of the particles from the image coordinates of the particles within the camera images. A similar procedure exists for using the particle data to self-calibrate stereo PIV data [37, 64]. However this procedure only refines the estimate of the location of the illuminated laser sheet and does not apply to volumetric data.

If the triangulation procedure worked with high accuracy for nearly every particle, then there would be no need to perform the computationally intensive tomographic reconstructions since the velocity field could be directly estimated from the three-dimensional particle tracking. In practice, due to the large number of particles in the camera images and the fact that there is likely error in the previously estimated calibration functions, uniquely and accurately triangulating the world coordinates of most particles tends to be impossible. However, the distribution of errors of the estimates of the particle world coordinates tends to be Gaussian with zero mean. Therefore, this particle triangulation procedure can be applied to a large number of particles to yield localized estimates of the camera calibration errors.

In practice, this self calibration procedure involves several steps: identifying particles in two-dimensional camera images, triangulating world coordinates for these identified particles, dividing the experimental volume into sub-regions, mapping all particles in each sub-region back to camera image coordinates, averaging the disparity between the mapped image coordinates and the original coordinates for each region, and updating the calibration functions with the measured disparities. This process is often repeated several times until the estimated disparities converge to zero. This procedure is described in depth in [65]. Typically the estimated disparity is calculated by summing Gaussian peaks centered on the disparity calculated from each particle. When the self-calibration procedure has converged, the summed peaks tend to be relatively narrow as is shown in the example dataset in Figure 2.7.

While the tomographic PIV dataset that was collected during the snake model experiments was very large and required the reconstruction and velocity field measurement processes to be very computationally fast, the self-calibration procedure only needed to be performed once. Therefore, the software to complete this process was written in MATLAB and was less optimized than the other tomographic PIV software written for processing this dataset.

The self-calibration procedure was run on the particle image data collected with the snake model not included in the water tunnel. The procedure was found to converge to measured disparities of approximately 0.1 pixels after three calibration function updates. The self-calibration processing was repeated for the calibration data that was collected after the tomographic PIV data and compared to the first calibration. Any discrepancies between the calibration functions would indicate that the cameras may have moved during the experiment and that the self-calibration may need to be repeated using the model tomographic particle data. It was found, however, that the calibration functions did not change over the course of the experiment, and thus the self-calibration functions could be used for the entire set of collected data.



(a) Camera 1 disparity map

(b) Camera 2 disparity map

(c) Camera 3 disparity map

(d) Camera 4 disparity map

Figure 2.7: These figures show the estimated disparity maps at one Z-axis depth for the four cameras used in the tomographic PIV experiment. The maps are constructed by adding a large number of Gaussian peaks together centered on the measured disparity of each triangulated particle. The narrow, well defined peaks shown here indicate that the self-calibration procedure has converged onto an accurate estimate of the calibration functions.

2.3.2 Model Masking

While the filters on the cameras remove the majority of the laser light reflected from the snake model, some reflected light is still apparent in the particle images. In particular, the white spots painted on the model are visible in many of the camera images. Unfortunately, these spots will act computationally similarly to the tracer particles and will appear as stationary regions within the time-resolved series of reconstructed three-dimensional images. These regions will result in a strong zero-displacement signal in the cross-correlations performed

during the velocity field measurements. This will tend to either bias the measured velocity (especially in low velocity regions) or result in zero velocity being incorrectly measured for some cross-correlation windows. Thus, the model must be computationally removed from the data.

While simply masking out the regions of the particle images corresponding to the model locations (by setting the pixel values in these regions equal to zero) would remove this effect from the reconstructions, this would also remove computationally valuable information from the data. Illuminated particles will pass between the model and the cameras, thus allowing the velocity field in this region to ideally be measured, but masking out the models in the camera images would also remove this data. Additionally, since the reconstruction calculation is effectively a multiplicative process, any region set to a value of zero in the original two-dimensional camera images, will result in the projection of this region being zero everywhere in the three-dimensional reconstructions. Thus, useful particle data will also be removed from 'behind' the model as well. Therefore, the masking operation must be performed in the three-dimensional computational domain.

Masking the computational domain requires determining the position of the snake model within the experimental volume. This is accomplished by creating a computational model representing the physical snake model and transforming this computational model in three-dimensional space until images generated from the computational model match the camera images of the physical model. This process involves several steps. First, a computational model of the snake needs to be created. This can be done by applying photographic images of the model to the triangulated mesh data that was used for 3D printing the model. Once this model is created, a ray-tracing algorithm can be used with the calibration functions to create simulated camera images of the computational model. The computational model is then translated and rotated in three-dimensional space to approximately match the images

from all camera and all model positions. Then since the physical model likely is a deformed version of the triangulated mesh (due to printing errors, the model absorbing moisture, and the water tunnel flow pushing on the model), the computational model is computationally deformed to match the physical model. After this process, the ray traced images of the computational model match the physical camera images to discrepancies of less than 1 mm. Finally, the computational domain is broken down into individual voxels and each voxel is tested to determine whether it lies inside or outside of the computational model. The voxels that lie inside the model are set equal to zero and those outside the model are set equal to one.

To create a computational model of the snake model used in the experiments, the physical model was photographed along with a calibration grid placed in the same location as the model. The grid points in the calibration photographs were used to determine a transformation to convert the image to an axis-aligned rectilinear coordinate system; this effectively removed any distortion from the camera lens or from the model being placed at an angle to the camera sensor. This transformed image was then aligned to the triangulated mesh that was used in 3D printing the model. The mesh was mapped onto the transformed image (by effectively removing the Z-axis component of the vertex data since the coordinate systems of the image and mesh are aligned) and each mesh triangle face was defined to have a color value equal to the mapped location of the center of the triangle on the transformed image. The result of this operation is shown in Figure 2.8 where the computational model is shown projected into the three-dimensional coordinate system.

Note that this operation results in the same paint patterns being apparent on both the dorsal and ventral sides of the computational model. However, since only the dorsal side of the physical model is visible to the cameras, as long as the ray tracing algorithm only visualizes 'visible' faces of the computational model, the ray traced images of the model will still be

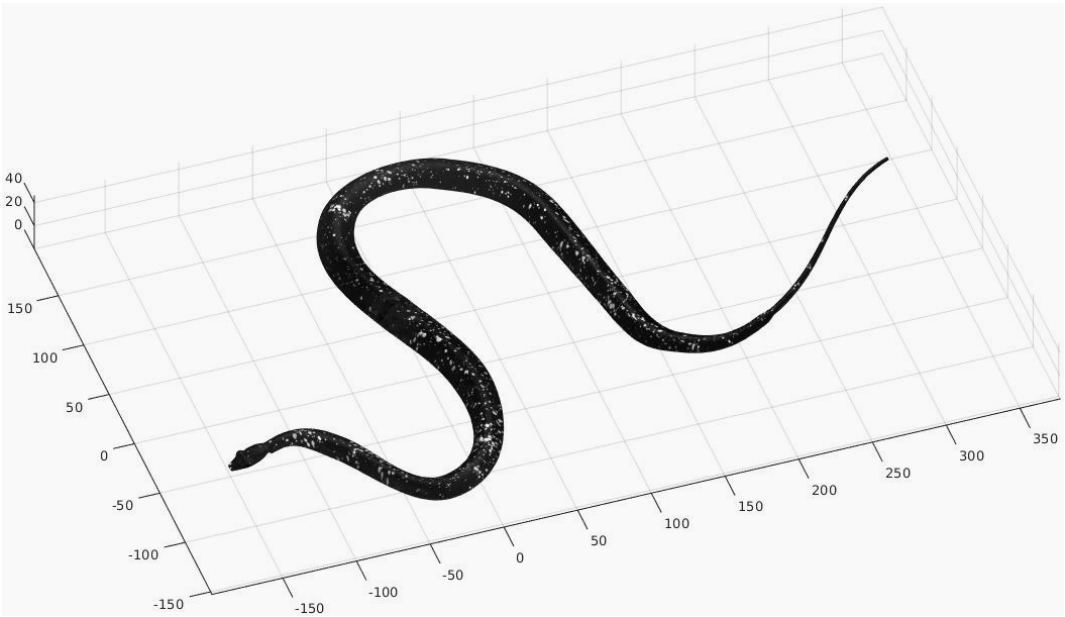


Figure 2.8: This shows the computational model of the snake including the mapped paint patterns. To determine the position of the model within the experimental measurement volume, ray traced images of this model were generated and compared to the physical camera images.

correct despite this simplification.

The ray tracing algorithm works by mapping the world coordinates of the triangular mesh vertices to the image coordinates in each camera using the calibration functions. Any mesh triangles that map to outside the camera image domain are excluded since these regions of the model won't be in the field of view of the camera. Additionally, any mesh faces whose normal vectors point away from the camera are not mapped since these faces will be on the 'back' side of the model from the perspective of the camera. The pixel values in the camera images are then set equal to the weighted sum of the mesh face color values where the weights are given by the proportion of the pixel that is covered by each triangular mesh face.

Calculating the weighting values involves estimating the intersected area between each of the simulated camera pixels and each of the mapped mesh triangular faces. Each simulated

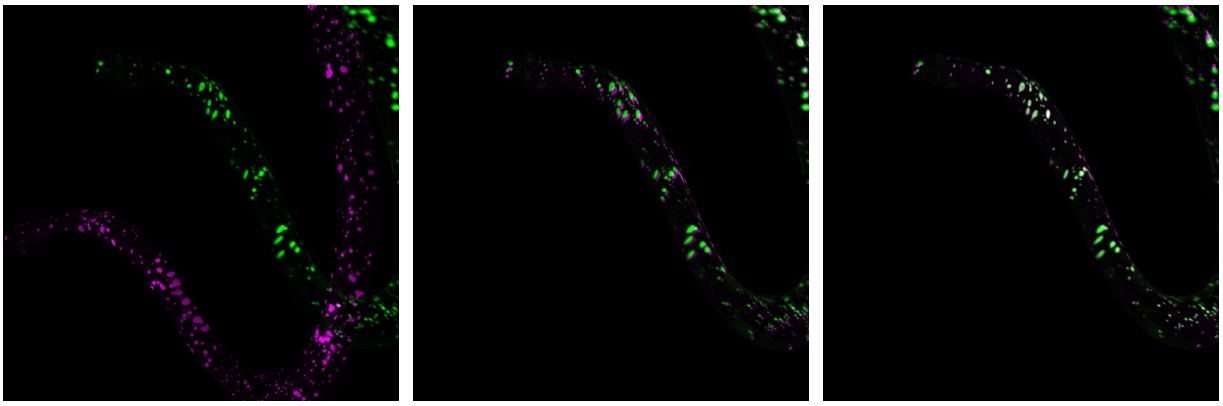
camera contained 1048576 pixels and the computational model consisted of 199632 faces. Despite the fact that many mesh faces will not map to the camera images due to being outside the field of view or facing away from the camera, this is still a very large number of intersections to calculate. For this reason, the ray tracing algorithm was written in C so that the code could run quickly and be easily parallelized.

To compare the ray traced images and the physical model images, the background of the camera images were masked by setting the pixel values to zero. An attempt to automate this process using image thresholding combined by edge detection algorithms was made, however there was too much variation in the images. It was determined that any attempt to automate the process would result in poor quality masking, so the masking was completed manually by tracing the edges of the snake model in all 176 camera images in which it appears. To ensure that any metric used to measure the similarity between the simulated and real images was relatively continuous as the position of the model was varied, a logistic function was applied to the mask so that the pixel values varied continuously from zero in the background to one over the model. An example masked image is shown in Figure 2.5b.

The zero-mean normalized cross-correlation (ZNCC) was chosen as the similarity metric between the physical model images I_P and the simulated computational model images I_C. This metric

$$Q^* = \frac{\sum_{x,y} I_P(x,y) \cdot I_C(x,y)}{\sqrt{\sum_{x,y} I_P(x,y)^2 \cdot \sum_{x,y} I_C(x,y)^2}}$$

was chosen due to its computational efficiency to calculate, the fact that its output is independent of intensity scaling on the images, and the fact that the function range $0 \leq Q^* \leq 1$, with a value of $Q^* = 1$ corresponding to a perfect match, is easy to interpret both mathematically and intuitively. For any particular computational model position, Q^* is calculated for all camera positions in which the model appears and the sum of these values is taken as



(a) Initial model position (b) Closely matched model (c) Converged model match

Figure 2.9: These figures show the solid-body transformation matching process between the physical camera images and the computational snake model. The physical camera images of the model are shown in green while the computational model is shown in magenta. When the two images overlap, a white color becomes visible as is shown in Figure 2.9c when the physical and computational models are closely aligned.

the overall similarity metric.

An initial guess of the model position and rotation angles based upon the known angle of attack and the rough position of the model was then used in gradient descent based algorithm to maximize the overall similarity metric. This processing was carried out in MATLAB using the 'fminsearch' function calling the C ray tracing program and a script to calculate the similarity metric sum. Several steps of this computational process are shown in Figure 2.9 where the computational model is progressively matched to the physical model in the images.

Once the gradient descent algorithm reached convergence (defined as further motions of the model resulting in less than 0.1 pixel changes in camera image positions), the physical model and the computational model were still separated by several millimeters in world coordinates in some locations along the body. The physical and computational models likely do not exactly match geometries exactly due to printing errors, the physical model absorbing moisture, and the model bending due the flow in the water tunnel. To account for these

discrepancies, the vertices of the computational model were transformed by small distances, essentially morphing the shape of the computational model to match the physical model.

To determine the motion necessary to transform the computational model, the gradient descent algorithm was applied to single sets of camera views instead of all camera views at once. Additionally, only translations (instead of translations and rotations) were considered. The initial estimate for the model position was taken as the converged location from the previous step. This resulted in a set of $(\Delta x, \Delta y, \Delta z)$ translation values for each of the model positions (44 for the 20 degree angle of attack and 16 for the 35 degree angle of attack).

Since these translation values applied equally to the entire model visible in each set of camera images and parts of the model appear in multiple different views, it was necessary to determine a way of smoothly parameterizing these translations. This would allow the translations to be uniquely defined in the cases where regions of the model are visible in multiple different views while avoiding discontinuities when the translations are applied to the computational model. The arc length along the center of the snake model body was chosen as a well defined parameterization that could be calculated for all vertices in the triangulation mesh.

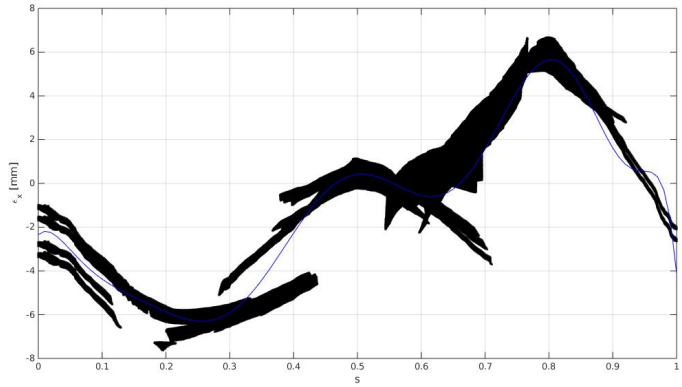
The translation values were then calculated for every vertex that was visible in each set of camera images. The translation values along each axis are plotted in Figure 2.10 as the black regions (which are in fact scatter plots, but appear as solid regions due the large number of vertices in the computational model). The arc length parameterization has a value of zero corresponding to the tip of the head with a value of one corresponding to the end of the tail. Since the translation values are not uniquely defined and tend to be relatively noisy, a least squares fit of a seventh order polynomial was applied along each of the axes. This function was chosen as a compromise between attempting to closely match the noisy data and overly smoothing the translations. The polynomial fits are plotted in blue in Figure

2.10. While the fit functions do not closely match all vertex translation values, it's apparent that transforming the computational model by the fit values will reduce the model position discrepancies to approximately 1 mm across most of the model.

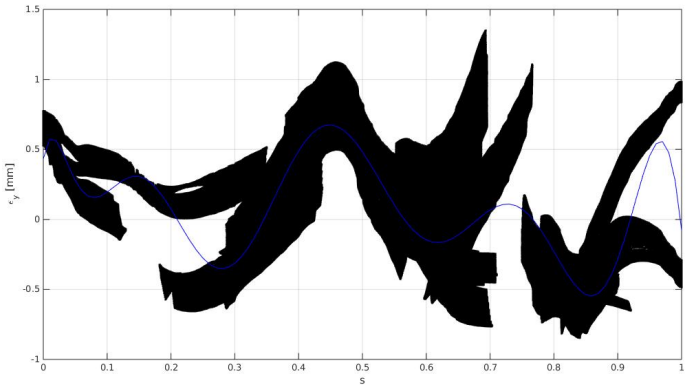
Once the polynomial coordinate transformation was applied to the vertices of the computational model, an estimate of the uncertainty of the computational model registration was desired to be calculated. This would give an idea of how closely the computational model matches the physical model. Ideally, the error between the two should be less than the vector field resolution of the calculated velocity field, since any errors smaller than this won't be apparent in the measured velocity field. The magnification of the camera system (as derived from the calibration functions) is approximately 0.17 mm/pixel, then assuming the smallest correlation window possible would likely be 16 pixels on a side, then the spatial resolution of the vector field would be 2.72 mm/vector. Therefore, if the error in the position of the computational model is less than approximately this distance, then it will have negligible effect on the measured velocity.

To estimate the uncertainty in the position of the computational model, the average ZNCC value was calculated across all camera views while the position of the model was shifted by small increments along each axis. The results of these tests are shown in Figure 2.11. It is apparent that the quality of the image similarities falls quickly with displacements more than 1 mm in the X and Y axes, thus implying that the overall position is likely of this magnitude. The Z-axis image similarity shows a broader peak implying that the error in this dimension is likely higher as well. The position error in this dimension likely approximately equals the minimum vector field resolution.

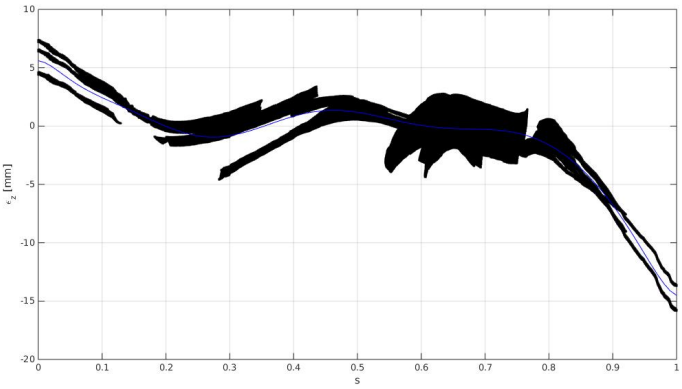
Once the position of the computational model was determined, this information needed to be translated to the reconstruction domain to properly mask the data as was discussed earlier. Since the tomographic reconstructions occur by iteratively refining the voxel values within a



(a) Model X-axis arc length parameterization fitting

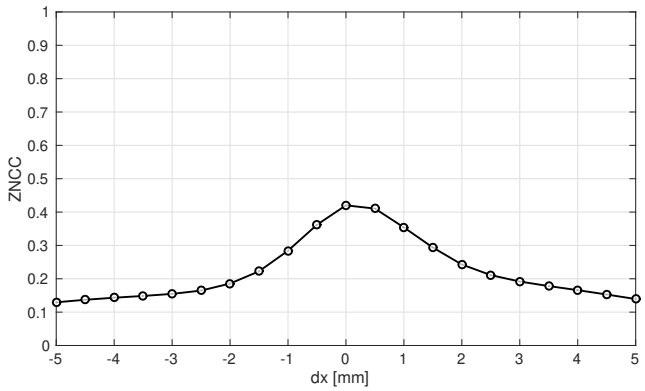


(b) Model Y-axis arc length parameterization fitting

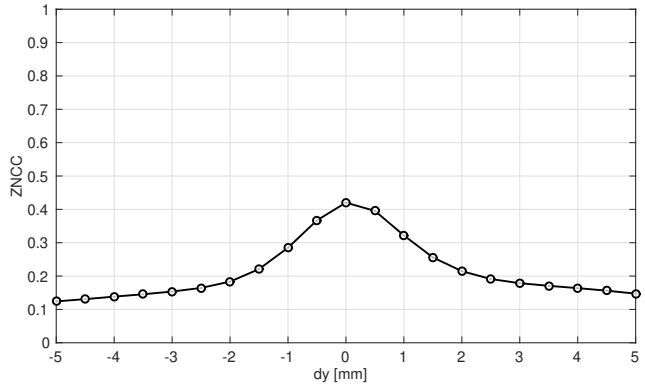


(c) Model Z-axis arc length parameterization fitting

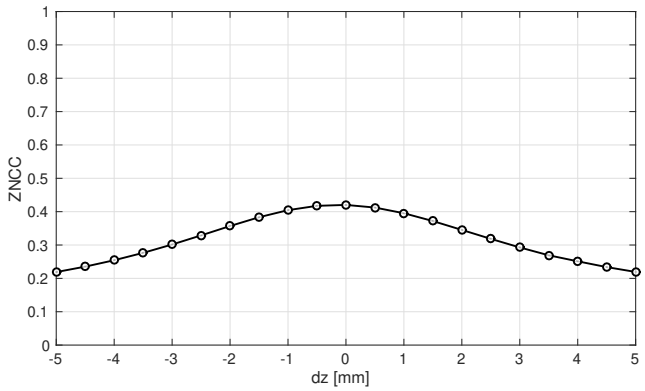
Figure 2.10: These figures show the deformation applied to the computational model in each axis parameterized by the arc length along the length of the snake. The blue curves represent the applided deformation while the black regions consist of the vertices of the computational model that are visible in each set of camera images.



(a) Mean ZNCC value as function of the X-axis displacement



(b) Mean ZNCC value as function of the Y-axis displacement



(c) Mean ZNCC value as function of the Z-axis displacement

Figure 2.11: These figures show the average zero-mean normalized cross-correlation values of all camera images as a function of the model displacement. These curves effectively give an estimate of the uncertainty of the final model position. Ideally the error in the position should be less than velocity vector field resolution.

three-dimensional image, a three-dimensional image mask that has the same dimensions as the tomographic reconstructions needed to be calculated.

The dimensions of the reconstruction volume, and thus the mask, were determined by finding the largest volume that was visible to all four cameras that included the entire 3 cm depth of the laser sheet. Additionally, the magnification of the voxels was set equal to that of the camera pixels since a one-to-one ratio is known to perform well during tomographic reconstructions. This resulted in a volume that was 800 by 900 by 200 voxels in size, which corresponded to 136 mm by 153 by 34 mm.

While spatial gradients would not need to be calculated across the image as was the case for the two-dimensional masking, it was still assumed that defining the mask with transitional values between zero and one at the border of the mask would decrease errors in later calculations. Leaving a sharp transition in the image could potentially increase the noise in vector field cross-correlations in particular. So voxels that were intersected by the mesh of the computational model were set to fractional values.

Ideally the value of these masking voxels would be set equal to the proportion of the voxel that lies inside the computational snake model. Unfortunately, this calculation is difficult to solve. Algorithms exist to efficiently calculate the intersection of two convex hulls, however the mesh defining the computational snake model is not a convex hull. Therefore, this calculation could only be completed by intersecting every face in the computational model with the faces of every voxel and then finding the intersection of the resultant halfspaces. The number of intersection points would grow exponentially - so this problem is computationally intractable in the general case. An approximate solution could be calculated by only considering faces in the computational model that are 'nearby' the voxel in consideration. In practice, this would likely work relatively well since the faces of the computational model are of the same length scale as the faces of the voxels and the computational model is locally smooth at this

length scale.

However, a computationally simpler method was used that is also mathematically well defined. The computational model is a manifold surface and therefore has a well defined 'inside' and 'outside'. Therefore, given any point in three-dimensional space, it is possible to efficiently test whether this point lies inside (or outside) the model by counting the number of times that a line from this point to infinity crosses a boundary of the model. Testing the center of the voxels in the reconstruction volume will still yield a simple binary mask. However, by dividing the voxels into subregions and testing whether each of these subregions lie within the model, any arbitrary precision can be reached in determining the intersection volume. Therefore, each voxel within the reconstruction was sub-divided into an 8 by 8 by 8 set of sub-regions and each of these regions was tested. This resulted in an estimate of the intersection volume that had an error on the order of 0.2%. This process was then completed for every voxel within the reconstruction volume to yield the full tomographic mask.

2.4 Particle Image Preprocessing

Due to spreading the laser light into a relatively large volume and the filters used to suppress the laser light reflections, the tomographic particle images that were collected were relatively low contrast with a low signal-to-noise ratio. Additionally, both random and time-dependent sensor noise (which appears as a flicker in time-resolved image sequences) was apparent in the camera images. To counteract these negative effects, multiple image preprocessing techniques were applied to the particle images prior to calculating the tomographic reconstructions.

The sequence of preprocessing steps included: calculating time-resolved image statistics over the entire image sequence, rescaling the images so that the intensities are roughly normalized by the overall sequence intensities, normalizing the images by the mean particle intensities

so that the brightest particles all roughly have the same intensity in all images, applying a linear-logistic scaling to the images to suppress low intensity noise while brightening particles, applying a Fourier based temporal low-pass filter to the image sequences to minimize time-dependent sensor noise, and finally thresholding the images to remove low intensity background noise.

2.4.1 Image Sequence Normalization

The intensity of the particles that is recorded by the cameras varies due to both the viewing angle of the cameras (caused by light scattering effects) and the individual sensitivities of the cameras. Moreover, different regions within each specific camera image will have different intensities, primarily due to the variation in the laser intensity within the illuminated volume. Unfortunately, the highest quality tomographic reconstructions and measured velocity fields will occur when all particles have approximately equal intensities. Therefore, it is necessary to normalize the images so that the particle intensity is roughly uniform amongst the cameras and across the individual camera images. To perform this normalization process, statistics were measured across the image sequences recorded by each camera.

In particular, the minimum values $I_{min}(x, y)$ and the median values $I_{med}(x, y)$ were measured for the entire image sequence of each camera. Then the sequence of images was rescaled by these values using the following formula

$$I_{norm}(x, y) = \frac{I(x, y) - I_{min}(x, y)}{I_{med}(x, y) - I_{min}(x, y) + \epsilon}$$

where the constant ϵ is defined as a small value equal to the one-percentile difference between the median and minimum values and is included to avoid division by zero errors. Once this scaling was performed, all images then had values roughly between zero and one across the

entire image sequence. However, this normalization process treated background noise from the snake model or from the camera image sensor noise the same as signals from particles, so a second normalization process was applied that accounted specifically for particle information.

2.4.2 Image Sequence Particle Scaling

Since the ideal tomographic particle image data will have particles of uniform intensities, the image sequence needed to be scaled by the particle intensity values as well. This could not be completed in the first step since the particle intensities varied dramatically across individual camera images and across the different cameras. Once the intensities were normalized, however, particles could be identified and tracked across multiple images to yield a more accurate normalization process.

In each image from each camera sequence, all local maximums were identified, and the sub-pixel location and fit intensity values were recorded. A three-point Gaussian intensity fitting function was used to estimate the peak location

$$x_{fit} = x + \frac{\log(I(x+1)) - \log(I(x-1))}{2 \cdot \log(I(x)) - \log(I(x+1)) - \log(I(x-1))}$$

where the pixel coordinate x is the integer coordinate of the local maxima. The peak intensity value was then linearly interpolated from the pixel values at the fit location. (Note that this is not a particularly robust or accurate method of determining the particle statistics, but a rough approximation is all that is needed for rescaling the images.) Once all local maxima were identified in the images, any maxima within one pixel of the image edge were removed from this list to avoid edge effects. Then only the 500 brightest maxima were used to calculate the particle statistics to avoid potentially including local maxima generated by background noise within the data.

The images were then normalized a second time using the median particle intensity P_{med} across all frames

$$I_{part}(x,y) = \frac{I_{norm}(x,y)}{P_{med}}$$

to yield images that had roughly constant particle intensities across the whole image and amongst all four cameras. The images at this point still contained significant noise, particularly at low intensities, so a transformation was performed to reduce low intensity values while keeping the higher intensity values constant.

2.4.3 Image Intensity Transformation

During the tomographic PIV reconstruction calculation, the high intensity particles primarily dominate the energy of the reconstructions. However, low intensity background noise tends to introduce noise to the reconstructions and thus the measured velocity fields. Therefore, reducing this background noise prior to the reconstruction process is advantageous.

Ideally any transformation applied to the images will suppress low level intensities, while leaving higher intensity values roughly constant. To accomplish this, a piecewise function that is logistic at low levels and linear at high levels was used

$$I_{tran} = \begin{cases} \frac{L}{1+\exp(-k\cdot(I_{part}-I_0))} - \frac{L}{1+\exp(k\cdot I_0)} & \text{if } I_{tran} < \mu \\ a \cdot I_{part} + b - \frac{L}{1+\exp(k\cdot I_0)} & \text{if } I_{tran} \geq \mu \end{cases}$$

where the values of the linear function are chosen to match the logistic function at the inflection point

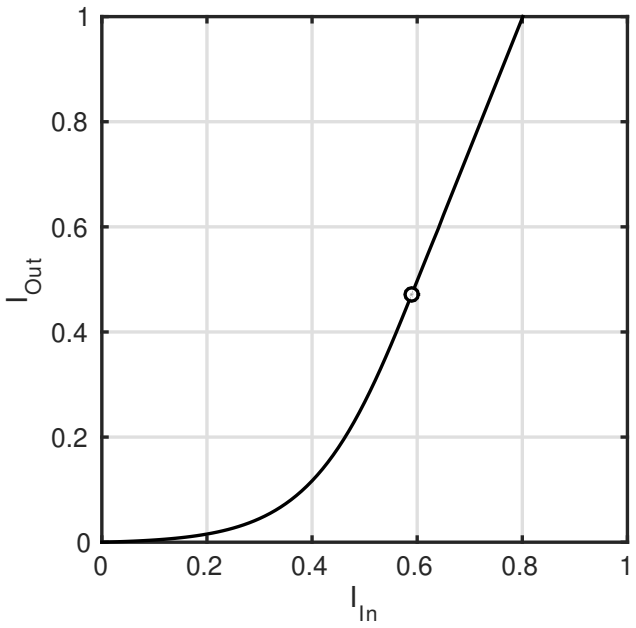


Figure 2.12: This plot shows an example intensity transformation function used to suppress low intensity noise while maintaining high intensity particle values. The inflection point of the function is denoted by the single marker and indicates the point below which noise will be suppressed.

$$a = \frac{k \cdot L \cdot \exp(-k \cdot (\mu - I_0))}{(1 + \exp(-k \cdot (\mu - I_0)))^2}$$
$$b = \frac{L}{1 + \exp(-k \cdot (\mu - I_0))} - a \cdot \mu$$

so that the intensity scaling remains continuous. An example of this intensity transformation function is plotted in Figure 2.12. Intensity values below the inflection point will tend to be pushed towards zero, while values above this point will be linearly scaled towards a value of one. This function will cause clipping of very high intensity values, so, in practice a less steep curve was used to minimize this effect.

The parameters L, k, I_0, and μ were chosen for each camera such that the output images

were all roughly similar and had values $L = 1$, $3 \leq k \leq 4$, $0.95 \leq I_0 \leq 1.05$, and $\mu = 1.5$. Different values were chosen for Figure 2.12 to accentuate the function features.

The intensity transformation tended to minimize intensity level noise fluctuations; however, the time-dependent sensor noise was still apparent in images at this point, so a temporal low-pass filter was applied to the images.

2.4.4 Temporal Low-pass Filter

While the laser intensity did noticeably fluctuate over time, this effect was negligible from one frame to the next and wasn't expected to negatively affect the tomographic reconstructions. However, the high-speed cameras tended to have obvious sensor noise that was time-dependent and apparent between individual frames of data. To minimize the effects that this might have on the reconstructions, a temporal low-pass filter was applied to the image sequences of all four cameras.

To minimize aliasing effects, the entire image sequence was transformed into the Fourier domain and the low pass filter was applied by multiplying the Fourier transformed image sequence with a super-Gaussian function

$$I(x,y,t) = \mathcal{F}^{-1}\left(\left(1 - \exp\left(-\frac{t^{\gamma}}{2\cdot\sigma^{\gamma}}\right)\right)\cdot\mathcal{F}\left(I(x,y,t)\right)\right)$$

where the Fourier transform is only applied in the time dimension. The parameters σ and γ control the range of frequencies that pass through the filter and how quickly the filter cuts off those frequencies. Since applying an overly narrow low-pass filter will tend to cause the particles to be blended across multiple images, a very wide filter was chosen that only cut off the highest frequencies.

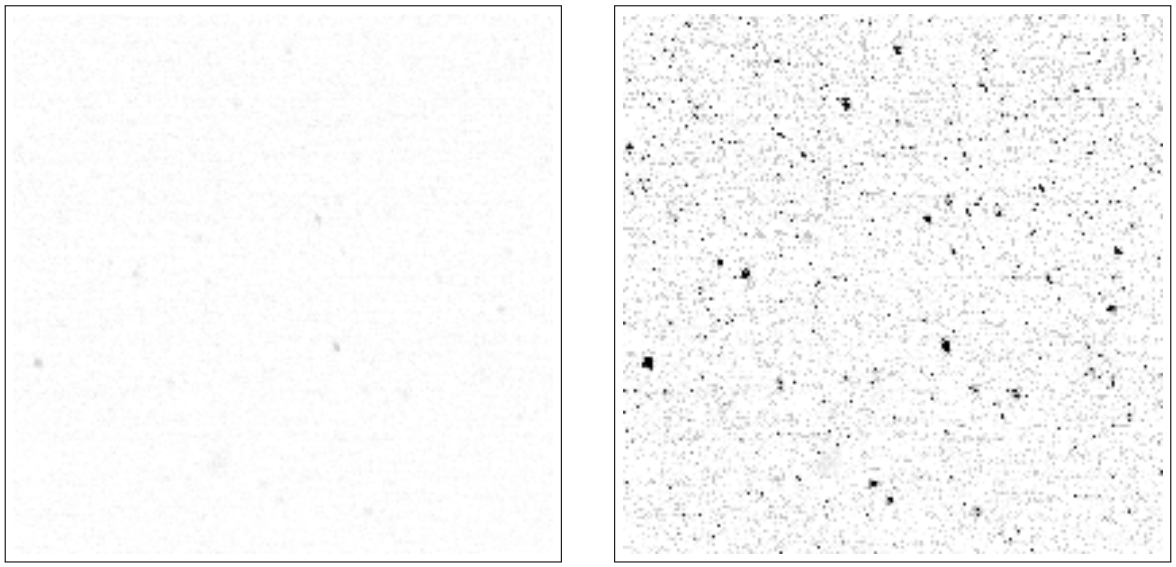
2.4.5 Threshold Filter

The final pre-processing step that was utilized before calculating the tomographic reconstructions was to apply a threshold to the images below which all intensity values are set equal to zero. This step is sufficiently common in tomographic PIV processing, that the threshold function was built into the tomographic reconstruction software and was applied in real time as the reconstructions were being calculated. A variety of threshold values were used during testing, but the value was typically about 0.08% to 0.15% of the maximum image intensity. This allowed nearly all particle information to be passed to the tomographic reconstruction while eliminating much of the remaining background noise.

2.4.6 Example Preprocessed Images

An example particle image in both original and preprocessed formats is shown in Figure 2.13. The images have been inverted to enhance the visibility of the particles. In the original image, the particles are very low intensity and generally only the largest particles are visible. However, after the preprocessing has been applied, all particles have similar intensity levels. Unfortunately, the preprocessing also enhances background noise that appears as many single pixels with high intensities. Some of the lower intensity or smaller diameter particles also appear as single illuminated pixels in individual frames, so this noise effect cannot be easily filtered out without removing some of the velocity field signal as well.

The exact parameters that were used in each step of image preprocessing were determined by calculating reconstructions from the preprocessed images and inspecting the quality of the resultant velocity field. A compromise was made between enhancing the contrast of the particle images and increasing the noise to unacceptable levels.



(a) Original PIV image (inverted) (b) Preprocessed PIV image (inverted)

Figure 2.13: This shows example particle images that were cropped to a 200 by 200 pixel region from a single camera. Both images have been inverted to enhance visibility and both are shown with the same colormap scaling. (a) The image here is an example image before any preprocessing has been applied. (b) This shows the same particle image after the various image preprocessing steps have been applied.

2.5 Tomographic Reconstruction

The tomographic reconstructions were performed using 10 iterations of the MART method [1, 2, 3, 8, 9, 10] using in-house processing code written in C. The three-dimensional reconstruction image was initialized using the mask calculated from the computational model. This causes any portion of the image corresponding to the snake model to be reconstructed with values of zero while regions outside this will be reconstructed normally. This should prevent any reconstruction artifacts from the model interfering with the velocity field calculations. The size of the reconstructed images was 800 by 914 by 229 voxels which corresponded to physical coordinates of 104.8 mm by 119.8 mm by 29.95 mm.

The camera images were threshold filtered with intensity values ranging from 0 to 200 (where the maximum intensity value of the images was 65535) to remove background noise in the

reconstructions. The MART relaxation parameter was set to $\mu = 1$ as is common practice [2, 8, 52]. After each reconstruction iteration, a mild Gaussian smoothing filter as well as a Laplacian sharpening filter were possibly applied. The strength of these filters decreased with the index of the iterations so that typically no filters were applied during the last iterations.

An example tomographic reconstruction time series is shown in Figure 2.14. This plot shows 10 frames of tomographic reconstruction data superimposed over one-another. The intensity of the isosurfaces in the plot are varied with the frame number to illustrate the motion of the particles. This dataset was collected without the snake model in the flow field to have a base dataset to determine the quality of the tomographic reconstructions.

While the flow field in this case is effectively a uniform flow field and is therefore not particularly complicated, this shows that the tomographic reconstruction process is working as expected since the particles appear to be temporarily coherent and traveling in a linear pattern as would be expected in a uniform flow. The threshold to calculate the isosurfaces was set relatively high to ensure that the plot did not become overly cluttered, so the actual number of particles in the reconstruction is higher than what is visually apparent here.

2.6 PIV Processing

Once the tomographic reconstructions were calculated, the reconstructions were processed using a variety of PIV processing methods to find the most optimal processing algorithm. The tomographic reconstructions appeared to produce high quality data; however, it was found that due to the low contrast particle camera images, the preprocessing operations carried out to enhance the particle contrast also significantly increased the background noise in the reconstructions causing significant contamination of the measured velocity fields.

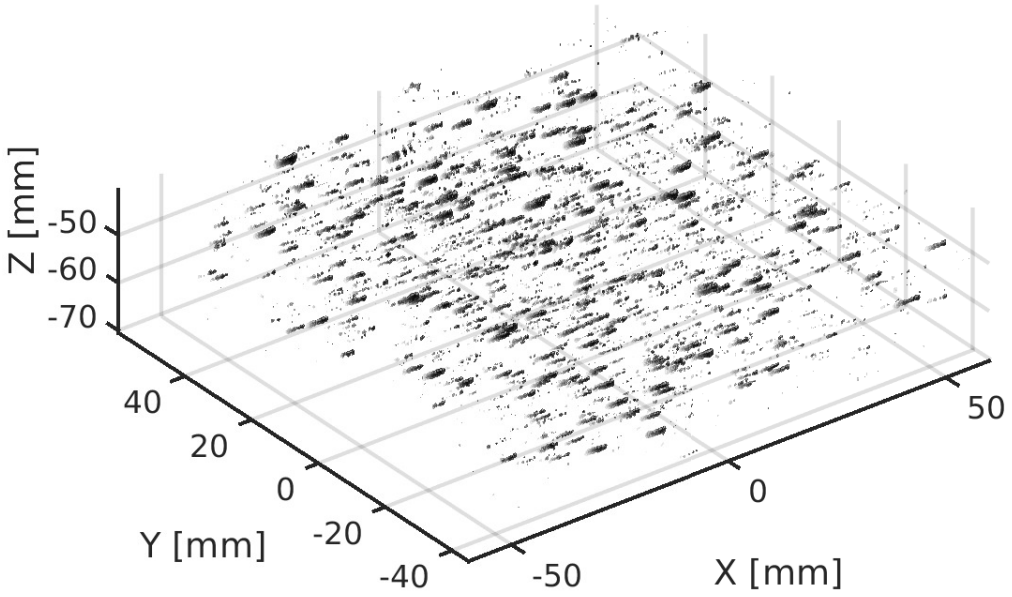


Figure 2.14: This plot shows a time sequence of tomographic reconstruction isosurfaces from multiple frames superimposed over one-another. The intensity of isosurfaces is varied with the frame numbers to illustrate the particle motion. This dataset was collected without the snake model present in the images.

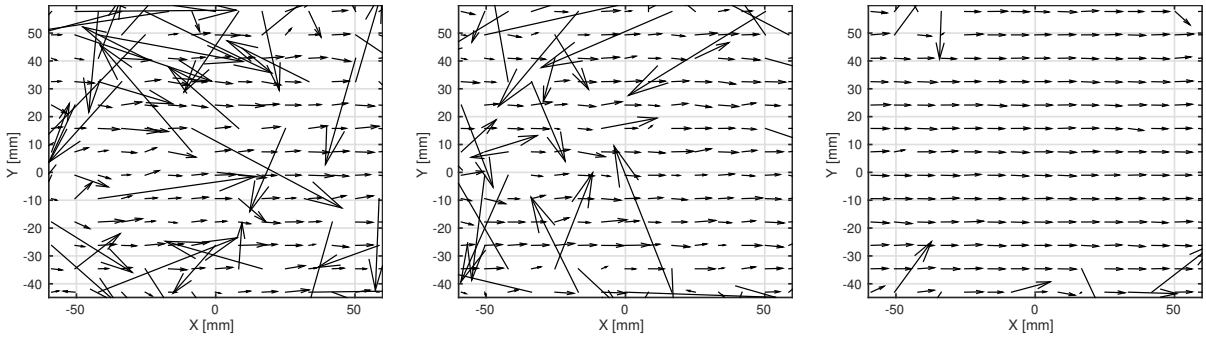
Many different processing methods were attempted to increase the signal-to-noise levels to acceptable levels.

2.6.1 Windowing Methods

Three different windowing methods were used in the processing: discrete window offset cross-correlations [67], window deformation cross-correlations [51], and pyramid correlations [57]. Basic testing of the reconstruction data was carried out using discrete window offset cross-correlations as these are relatively fast to process. Validation of the data was carried out by investigating the ratio of identified outliers in the measured vectors and looking at the variance of the velocity vectors within the uniform flow regions of the dataset. Using these metrics, it was determined that the first pass of the PIV processing required window sizes of 128 by 128 by 128 voxels with effective resolutions of 64 by 64 by 64 voxels to produce high quality velocity field data. Since the reconstructed volume was only 229 voxels across, this implied that only about 4 vectors could be measured across the Z-axis of the volume which is relatively low resolution.

Therefore, processing attempts were made using the higher accuracy window deformation and pyramid correlation methods. These methods involve deforming the reconstructed intensity field by the measured velocity field by performing interpolations at the deformed image coordinates. Pyramid correlations also add multiple time steps of frames together to increase both the sensitivity and the resolution of the measurements. These processes are relatively computationally intensive compared to the discrete window offset correlations, but they resulted in higher quality data.

It was found that using the window deformation method improved the quality of the measured velocity field, however the difference from the discrete window offset processing was



(a) Uniform flow discrete window offset cross-correlations (b) Uniform flow deformation cross-correlations (c) Uniform flow pyramid cross-correlations

Figure 2.15: These figures compare the three cross-correlation windowing methods tested in processing the snake model tomographic PIV data. The vector field should be uniform translation since the snake model was not included in this dataset. The methods were completed with two passes with no validation or smoothing applied on the second PIV pass. The window sizes were 64 by 64 by 64 voxels with 32 by 32 by 32 voxels effective resolutions. The same cross section from a single Z plane of the measured velocity field is shown in all three figures.

not significant. The pyramid correlations were found to reduce a large number of outlier vectors though. In Figure 2.15, the three different window cross-correlation methods are compared for the uniform flow field data with no snake model. The window size was intentionally set lower than what appeared to be required for producing high quality velocity fields to accentuate the effects of using the different window correlation methods.

It can be seen in the method comparison graphs that the discrete window offset cross-correlations produces a large number of outlier vectors. The number of outlier vectors is improved to a small extent when the window deformation cross-correlation method is used. However, only a large number of outliers are removed by using the pyramid cross-correlations, implying that using this method may be required for processing the snake model tomographic data.

Unfortunately, the higher order methods are considerably slower. For the vector fields shown in Figure 2.15, it took approximately 1 minute to process the data with the discrete window

offset correlations, about 31 minutes to process with the deformation correlations, and 231 minutes to process with the pyramid correlations. While the exact ratios are dependent upon the window sizes among other factors, this demonstrates that the pyramid correlations tend to be significantly slower to process.

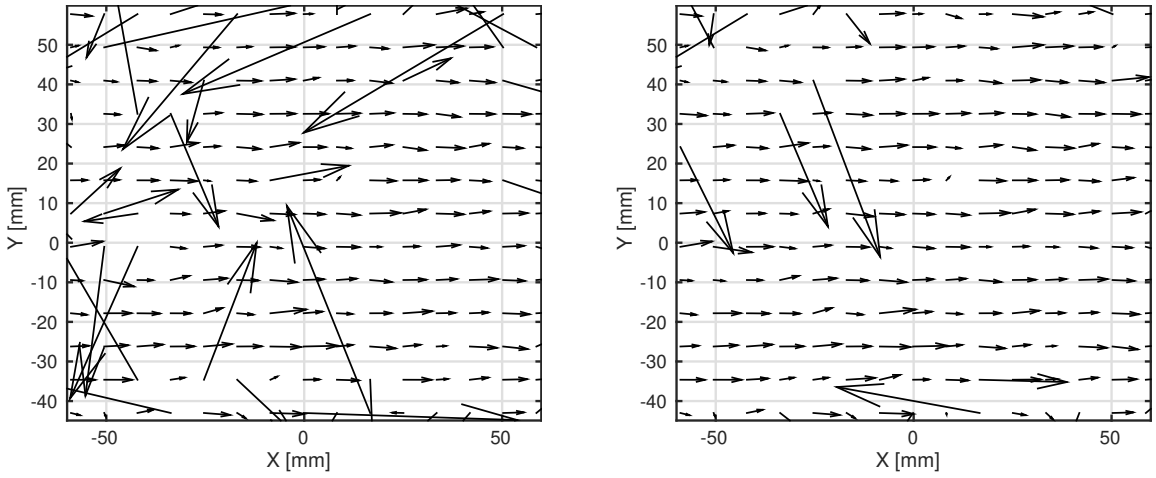
2.6.2 Cross-correlation Calculation

In addition to testing the windowing method used in the cross-correlations, the specific type of cross-correlation was tested as well. Both standard cross-correlations [67] and robust phase cross-correlations (RPC) [7] were tested as both methods are computationally fast and RPC method has been shown to be less sensitive to noise in the images. However, relatively little difference between the two cross-correlation calculation methods was found in the data.

2.6.3 Cross-correlation Peak Selection

When an outlier vector occurs in the measured velocity field, this is due to the highest magnitude peak in the cross-correlation volume corresponding to noise rather than the velocity signal generated by the particle motion. Generally, the true velocity will still correspond to a peak within the cross-correlation volume. To improve the probability that the true velocity peak vector was selected from the cross-correlation volumes, a method was developed that stored the largest n peaks from the cross-correlation volumes for all vectors in the velocity field and peaks that resulted in the lowest signal energy velocity field were selected as the most likely correct peaks from each cross-correlation. This tended to dramatically reduce the number of outliers in the measured velocity field as can be seen in Figure 2.16.

The algorithm effectively involves selecting a kernel size surrounding each vector within the velocity field and iterating through the n possible vectors while calculating the residuals to



(a) Maximum peak velocity field

(b) Least energy velocity field

Figure 2.16: These two velocity fields were both processed using discrete window offset cross-correlations, but the method to extract the cross-correlation peak was varied. (a) This field was calculated by setting the velocity equal to displacement associated with the maximum cross-correlation peak location. (b) This velocity field was estimated by storing the n largest cross-correlation peaks and choosing the peaks that would result in the lowest signal energy velocity field.

the adjacent vectors within the kernel. The vector with the lowest residual is then stored as the current most likely vector. This process is repeated for all vectors until an initial estimate of the lowest energy velocity field is calculated. Then during a second iterative process, the velocity vectors are iterated through and replaced with the cross-correlation peak vector that results in the lowest energy within the kernel radius compared to the initial estimate of the velocity field. This process is repeated until a convergence is reached and the vector field is no longer updated. The code describing the process is listed in Appendix A.1.

This algorithm has not been rigorously tested and it is unknown if there are conditions under which convergence may not be reached. However, in all studies completed so far, this algorithm has produced superior results to the typical maximum peak-based method in the presence of noise. When the algorithm was tested with low noise data, it yielded identical

results to the maximum peak algorithm. Note that this method would not be appropriate in any rapidly varying velocity field, such as across a shock boundary, though this should not be an issue for the snake model velocity field.

2.6.4 Cross-correlation Peak Fitting

Since the peak of the cross-correlation volume corresponds to integer particle displacement, a sub-voxel fit is applied to the peak. Early tests were completed using the standard three-point Gaussian fitting method [41] which fits a Gaussian function to the peak of the cross-correlation volumes using the three collinear points along each axis centered on the cross-correlation peak. However, it was expected that the apparent background noise in the data would potentially corrupt the fitting process using only three points. So least squares fitting procedures that fit the Gaussian function in all three dimensions simultaneously using the 3 by 3 by 3 voxel volume surrounding the cross-correlation peak were developed.

The first method assumed the Gaussian peak was only elongated in along the coordinate axes and had a functional form of

$$G(X,Y,Z) = C \cdot \exp\left(-\frac{(X-X_0)^2}{2\cdot\sigma_X^2} - \frac{(Y-Y_0)^2}{2\cdot\sigma_Y^2} - \frac{(Z-Z_0)^2}{2\cdot\sigma_Z^2}\right) \tag{2.1}$$

where the fitting parameters are the intensity C, the sub-voxel peak center (X_0, Y_0, Z_0), and the Gaussian function standard deviations σ_X, σ_Y, and σ_Z. For a 3 by 3 by 3 array of voxels this gives 27 equations with 7 variables and is thus an overdetermined system. If the coordinates of the Gaussian function in each dimension are taken as $\{-1, 0, +1\}$, then a linear system of equations with a constant matrix is produced for all input functions. The least squares fitting can then be calculated by taking the Moore-Penrose inverse [62] of this

constant matrix which yields the parameters as being equal to a matrix multiplication with rational coefficients.

When this is simplified for the center coordinates of the Gaussian function, the center coordinates are simple functions of the logarithms of the products of the 3 by 3 by 1 regions of the input array where the slices are taken in the direction of the dimension for which the center coordinate is being calculated. Thus, to calculate the sub-voxel coordinate of the peak in the X-axis direction when the discrete location of the cross-correlation occurs at (X_p, Y_p, Z_p), calculate the products of the cross-correlation values

$$\begin{aligned}
\Pi_{X-1} &= \prod_{Y=Y_p-1}^{Y_p+1} \prod_{Z=Z_p-1}^{Z_p+1} C(X_p - 1, Y, Z) \\
\Pi_{X} &= \prod_{Y=Y_p-1}^{Y_p+1} \prod_{Z=Z_p-1}^{Z_p+1} C(X_p, Y, Z) \\
\Pi_{X+1} &= \prod_{Y=Y_p-1}^{Y_p+1} \prod_{Z=Z_p-1}^{Z_p+1} C(X_p + 1, Y, Z)
\end{aligned}$$

and then the the sub-voxel peak location occurs at the location

$$X_0 = X_p + \frac{\log \Pi_{X-1} - \log \Pi_{X+1}}{2 \cdot \log \left(\Pi_{X-1} \cdot \Pi_{X+1}\right) - 4 \cdot \log \Pi_X} \tag{2.2}$$

where the values of the 3 by 3 by 3 region surrounding the cross-correlation peak must be positive due to the logarithms being calculated. Similar equations can be applied to calculate the values of Y_0 and Z_0. The same method can be applied to derive the peak intensity C as well as the Gaussian function standard deviations σ_X, σ_Y, and σ_Z. The code describing this peak fitting process is given in Appendix A.2.

The second sub-voxel calculation method uses a generalized non-axis aligned Gaussian function as the fitting function

$$G(X,Y,Z) = C \cdot \exp(-a_1 \cdot (X - X_0)^2 - a_2 \cdot (Y - Y_0)^2 - a_3 \cdot (Z - Z_0)^2 \\ - a_4 \cdot (X - X_0) \cdot (Y - Y_0) - a_5 \cdot (X - X_0) \cdot (Z - Z_0) - a_6 \cdot (Y - Y_0) \cdot (Z - Z_0)) \quad (2.3)$$

where the parameters a_i store both the direction of the principal axes of the Gaussian function as well as the standard deviation values. The least squares fitting of this function proceeds in a similar way to that of Equation 2.1, except that there are now 10 parameters to fit. The explicit least squares solution can then be calculated as a polynomial function of the logarithm of the cross-correlation values that is relatively easy to calculate. However, there are a large number of functional terms, so the explicit solution is not written here and is given in the C code form in Appendix A.3.

To measure the effectiveness of these two fitting methods in comparison to the three point Gaussian fit, three-dimensional particle images were simulated with varying amounts of Gaussian noise added onto the images. The results of these simulations are shown in Figure 2.17.

The two higher order fitting methods work better on average than the three point fitting method in the presence of noise. However, the confidence intervals in the graph indicate that there is a significant overlap amongst the three methods. The effectiveness of these methods needs further investigation, but this indicates that the higher order methods may be more suitable for particularly noisy data.

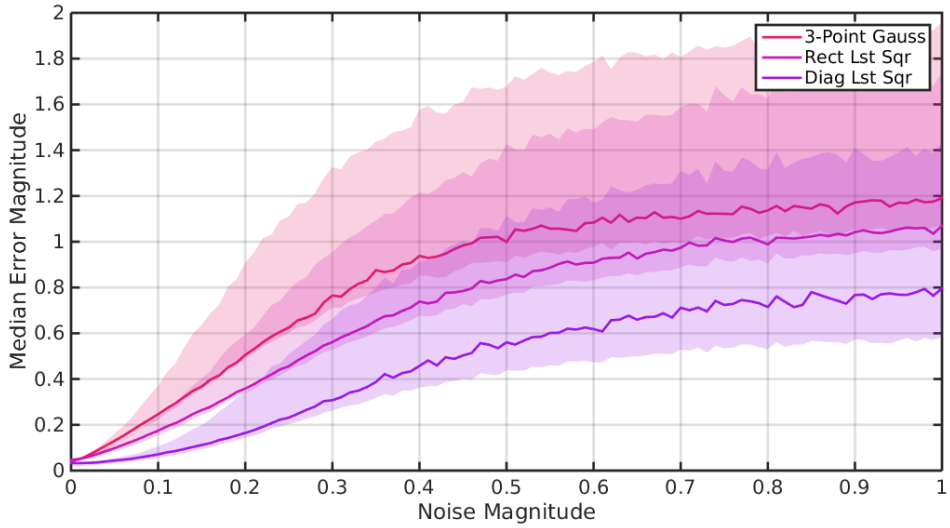


Figure 2.17: This plot compares the sensitivity of the three sub-voxel fitting techniques to additive noise in the volumetric images. The error is measured as the median voxel per frame error and the 95% confidence intervals are highlighted in the graph. The noise magnitude is measured as the standard deviation of the Gaussian distribution in comparison to the particle intensity.

2.6.5 Vector Field Validation and Filtering

Once the velocity fields were calculated, they were validated using both thresholding and the Universal Outlier Detector algorithm [63]. The identified outlier vectors are replaced using a Laplacian interpolation scheme calculated through a diffusion based process. After the vector field is validated, a Gaussian smoothing filter is applied to minimize high frequency noise, particularly when additional PIV passes will be completed and the previous passes are used as initial guesses in subsequent passes.

The vector fields were interpolated between PIV passes to use as initial estimates during the next PIV pass. Within the volume defined by the calculated vectors, a cubic interpolation scheme was used [29]. Outside of this volume, a novel extrapolation technique was used that applied a Taylor series expansion to a natural neighbor interpolation. This algorithm ensured that extrapolated vectors remained bounded and gave reasonable results.

To avoid overly filtering the data, the last pass of the PIV processing generally only applied very minor smoothing or no smoothing at all, depending upon the application of the data. Validation was sometimes also skipped during the last pass to examine the quality of the vector field.

2.7 Results

While a wide range of processing parameters were investigated, the following results were processed using a set of parameters that was found to be close to optimal for the snake model tomographic PIV data.

2.7.1 Processing Parameters

The tomographic reconstructions were calculated using ten passes of the MART algorithm with smoothing applied during the first eight iterations, but not on the last two iterations. No sharpening was applied during the reconstruction calculation. The camera image threshold value was set equal to an intensity level of 50. The reconstructions were stored with 16 bit precision using the 'zlib' compression in the HDF5 data file.

After the tomographic reconstructions were calculated, the three-dimensional images were processed with four PIV passes using the pyramid correlation windowing method. The first and second passes had window sizes of 128 by 128 by 128 voxels with window resolutions of 64 by 64 by 64 voxels. The third and fourth passes had window sizes of 64 by 64 by 64 voxels with window resolutions of 32 by 32 by 32 voxels. All passes used the RPC cross-correlation and stored 5 peaks to determine the least energy velocity field. The sub-voxel peak locations were determined using the three point Gaussian fitting method. Validation was applied on

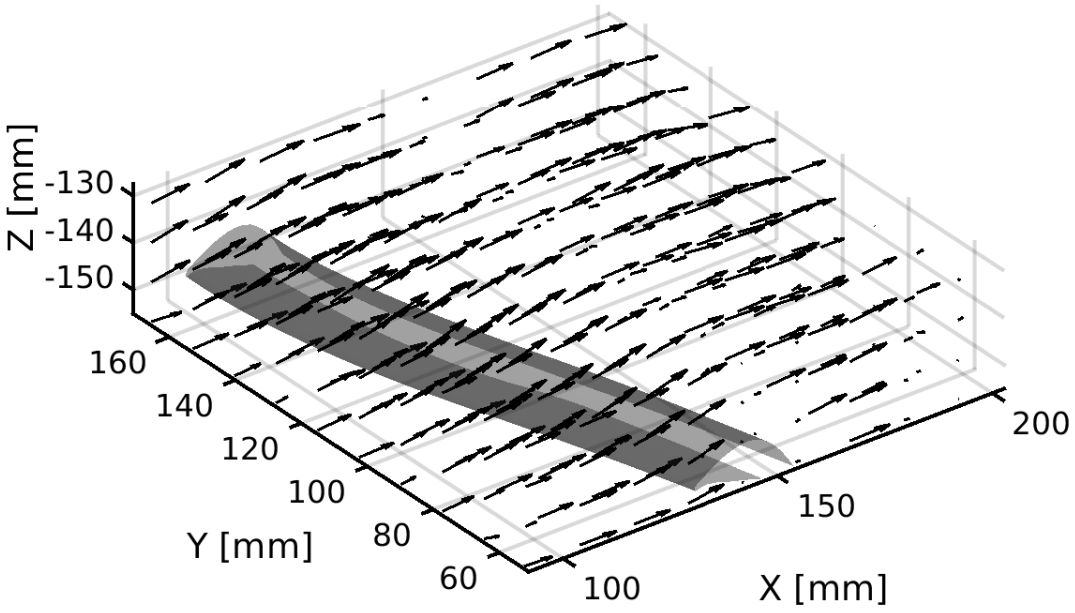


Figure 2.18: This graph shows the measured three-dimensional flow field around snake model. The location of the computational model mask is shown as the grey surface.

all four passes, however the vector field was only smoothed in the first three passes.

2.7.2 Reconstructions and Vector Fields

An example three-dimensional tomographic PIV velocity field is shown in Figure 2.18 along with the portion of the snake model superimposed onto the vector field. The general flow over the snake model can be seen in this graph, however it is difficult to discern specific flow field details.

A second velocity field is shown in Figure 2.19 that consists of a two-dimensional slice of the first vector field down the middle of the Y-axis. More detail can be seen in this graph. The

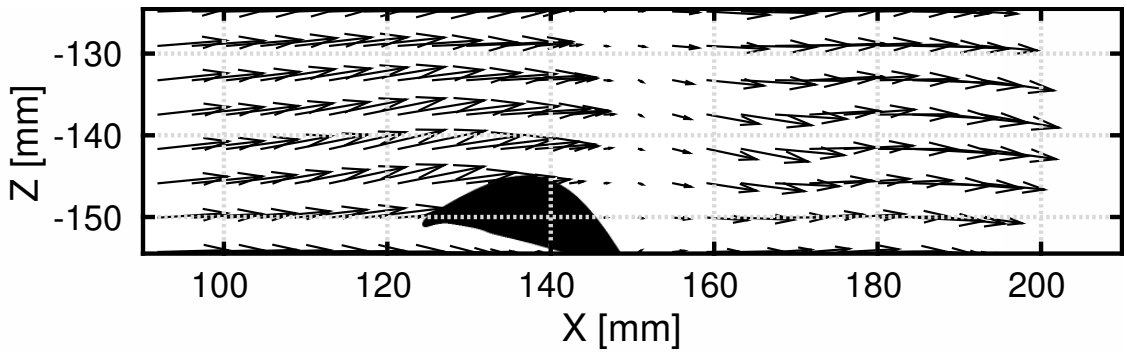
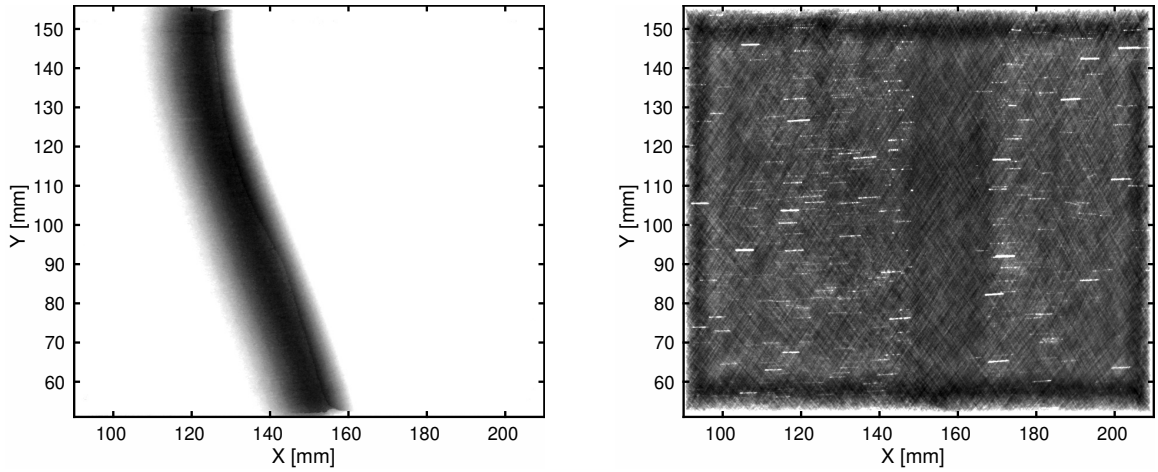


Figure 2.19: This graph shows a two-dimensional cross-section of the velocity field shown in Figure 2.18. The low resolution nature of the vector field is apparent in the flow near the snake model. Additionally, a shadow in the laser illumination is visible near the middle of the vector field where the vector magnitudes drop to nearly zero.

flow field appears to split into a flow above and below the snake model, similar to an airfoil as would be expected. However, little detail is apparent in the flow about the snake model. The vectors near the model have the same approximate magnitude as far-field vectors, indicating that the boundary layer forming over the model due to the no-slip condition could not be resolved. Additionally, the vectors near the model are only approximately parallel to the surface of the model as would also be expected. So the vector field shown here does not resolve the flow features about the snake model in sufficient detail to investigate the physics of the flow and the resolution of the processing needs to be increased.

A second prominent feature in the flow field is the group of low magnitude vectors in a vertical bar just down-stream of the snake model. In Figure 2.19, it appears as though this may be due to a shadow forming from the snake model. However, the laser illumination was parallel to the Y-axis (out of the page), so this was unlikely. To investigate this further, the three-dimensional reconstructions over a time series were investigated. In Figure 2.20a, the sum of all voxels greater than zero were calculated along the Z-axis. Since the model is masked out and set equal to zero, this shows the exact location of the model within the image. In Figure 2.20b, the reconstruction voxel values were summed along the Z-axis for



(a) Reconstruction model mask (b) Reconstruction time series

Figure 2.20: These graphs show the relationship between the location of the physical model and the laser illumination shadow. (a) The voxels greater than zero were summed along the Z-axis direction to highlight the zero-valued region corresponding to the snake model. (b) The same reconstruction images were summed across ten frames in the Z-axis direction. This highlights the tracer particles as well as the laser illumination shadow.

all frames in the time series. This shows particle tracks and a large vertical region across the reconstruction where the particles appear to lose illumination.

Thus, the region of the vector field corresponding to the low magnitude vectors does likely correspond to a shadow in the experiment, but it is in a different location than the snake model, so this likely was due to some other source. Since the particle illumination was already relatively low and significant preprocessing was required to bring out the particle contrast, it may be possible that this shadow was still in a relatively well illuminated region, but the illumination was just low enough that the particles were no longer visible above the noise threshold of the cameras. This further indicates that the laser illumination levels were too low to accurately resolve velocity field data.

2.7.3 Resolution Testing

Since after optimizing the processing parameters, the measured velocity field appeared to still have too low of resolution to accurately resolve the flow field features, the effective resolution of the tomographic data was investigated. Calculating the exact resolution requires knowing the correct solution to the velocity field, however since this is unknown, a proxy solution was used. The flow field sufficiently far upstream from the snake model should be approximately uniform; therefore the mean measured velocity in this region should be approximately equal to the true velocity.

To estimate the effective resolution of the tomographic PIV data, pyramid correlations were calculated with similar settings to the vector field calculated for Figure 2.18, except that only two passes were completed and the window size was varied from 16 vox^3 up to 64 vox^3. The vectors were only calculated for a small portion of the total reconstruction that corresponded to the upstream section from the model. Then the magnitude of the differences between each of the vectors and the mean vector was calculated. The result of these tests is shown in Figure 2.21.

At small window sizes, there will be an insufficient number of particles to generate an accurate displacement peak within the cross-correlation. This will result in the image noise covering the particle signal and highly erroneous vectors being measured across much of the vector field. As the window size increases, at some point the particle signal should dominate the image noise and the variance in the measured vectors will asymptotically approach zero for a uniform flow field. The point at which the variance starts converging will then effectively be the resolution of the particle image. Unfortunately, as can be seen in Figure 2.21, even for the largest window sizes, convergence is not reached.

This implies that the velocity field resolution shown in Figures 2.18 and 2.19 is essentially

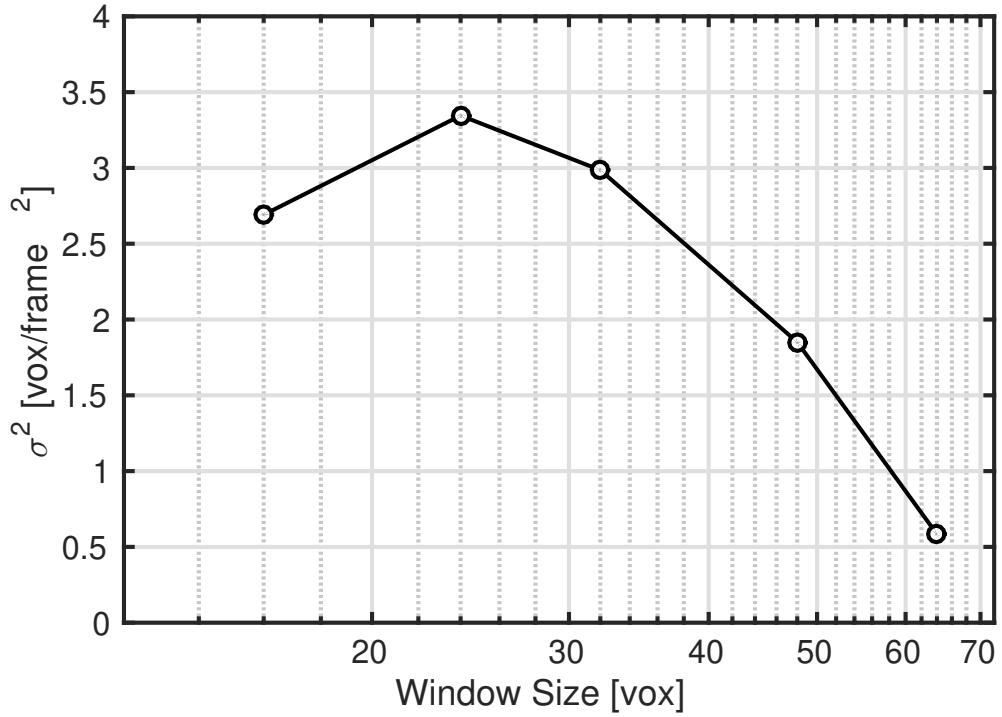


Figure 2.21: This shows the velocity field variance magnitude in the uniform flow region upstream of the snake model for a variety of different window sizes used with pyramid correlation processing. Since every vector should be nearly identical in this region, the variance should asymptotically approach zero when sufficient window resolution has been reached. Since even at the largest window resolution, the variance is still rapidly dropping, this means that the PIV windows must be at least 64 by 64 by 64 voxels in size.

the highest resolution that can be achieved with the collected tomographic PIV data since processing the reconstruction images with smaller window sizes will only increase the noise in the measured vector field without providing any additional refinement of the velocity field features.

The process described here is relatively linear, however a large number of different permutations of processing parameters, methods, and algorithms were attempted to yield the highest quality velocity field data. Some of these methodologies either did not work or worked more poorly than other processing methods that had been attempted. The methodology described here was found to produce the best data and thus was focused upon.

2.8 Recommendations

The primary reason that the tomographic PIV data did not yield velocity fields with sufficiently high resolution to investigate the physics of the flow around the snake model was that the signal-to-noise ratio in the collected camera particle images was too low. The two primary sources of error in tomographic PIV are due to calibration error and particle image noise. The calibration error was directly measured during the self-calibration process and found to be approximately 0.1 pixels, which falls well below the recommended maximum error tolerance of 0.4 pixels [8].

The effects of image noise on tomographic PIV measurements have not been thoroughly studied. This is likely due to the fact that modern lasers and high-speed cameras produce particle images with very high signal-to-noise ratios. Tomographic reconstruction has been largely replaced with particle tracking-based methodologies that require very high quality camera image data [54, 55, 56] where individual particle positions can be tracked over time. These methodologies would not be possible with particularly high noise particle images since it would be difficult to determine whether relatively high intensity regions on the camera images corresponded to either particles or noise, thus making particle tracking effectively impossible. Therefore it is difficult to determine based upon previous studies whether the data collected for the snake model experiment would be considered too low of a signal-to-noise ratio to produce high quality data.

However, the energy density of the laser illumination gives a metric which can be compared to previous tomographic PIV work. The snake model data collected used a dual head 10 mJ per pulse laser for illumination. Both laser heads were simultaneously pulsed and were used to illuminate an area of about 105 mm by 30 mm. This gives the illumination flux energy per pulse as $6.35 \cdot 10^{-3}\,\mathrm{mJ/mm^2}$. A tomographic PIV review paper from 2013 [52]

summarized 14 tomographic PIV experiments which had illumination flux energies between $2.66 \cdot 10^{-2}\,\mathrm{mJ/mm^2}$ and $1.35 \cdot 10^{0}\,\mathrm{mJ/mm^2}$. The median flux energy was $1.44 \cdot 10^{-1}\,\mathrm{mJ/mm^2}$ which was 22.7 times the energy of the snake model system. The laser and camera used to collect the snake model data had lower power and resolution than 12 of the 14 cases described in the paper. While the laser optical system, the sensitivity of the cameras, and the lens aperture diameters will also play a role in the overall image quality, this basic analysis implies that the illumination likely needed to be significantly higher to produce high quality tomographic PIV data.

Since the laser power is fixed, the area of illumination can be decreased to increase the energy density. To achieve the median energy density described above would require reducing the reconstruction volume to approximately 22 mm by 25 mm by 6 mm. Since the snake model is approximately 500 mm in length, this would require collecting data for far too many volumes to practically image the entire snake model. Therefore, a subsequent study would necessarily focus on small regions of the model for data collection.

Additionally, running preliminary processing on the tomographic PIV data at the start of the experimental data collection would be prudent to ensure that the data will produce high quality velocity fields. This would involve collecting calibration data, collecting particle image data, calculating the initial calibration, calculating the self-calibration, performing the reconstruction process, and measuring the velocity vector fields. This process would likely take six hours with in-house software and two to three with commercial software.

If the velocity field data was found to be too low of resolution, multiple techniques could be used to resolve the issue. First, the image magnification could be increased by moving the cameras closer to the volume or by using longer focal length lenses. The particle density could also be increased by adding additional seeding particles to the fluid. The illumination energy density could be increased by decreasing the illumination area and the corresponding

imaging volume. The camera sensitivity could also potentially be increased by opening the lens aperture or increasing the sensor gain on the camera, but both of these parameters may be effectively fixed by the experimental system. Without having the feedback process of examining the velocity field data to update the experimental configuration, it is difficult to estimate the best procedure for repeating the snake model experiment. More than likely it would involve a combination of all of these options, with the primary focus being on decreasing the volume size to increase illumination and image magnification.

Chapter 3

Tomographic PIV Processing Software

3.1 Introduction

Understanding the three-dimensional structure of fluid systems is essential for several applications. Practical applications include measuring the forces due to vortex shedding and turbulence, quantifying rates of mixing in industrial processes, determining energy flux in heat exchangers, as well as many other problems. Despite the importance of understanding the three-dimensional dynamics of flows, experimental techniques to collect three-dimensional velocity data remain relatively limited. There are several techniques for measuring three-dimensional velocity fields including stereo Particle Image Velocimetry (PIV) [37, 48, 53, 60], holographic PIV [24, 36], and defocused PIV [26, 46, 47], but currently tomographic PIV [1, 2, 3, 8, 9, 10] and its related Shake the Box method [54, 55, 56] are capable of resolving the largest volumes with the highest particle seeding densities.

3.2 Tomographic PIV

In tomographic PIV, multiple cameras record data from a three-dimensional particle field. A two-by-two or cross-shaped arrangement of four cameras is commonly used, although other configurations are possible. Generally, as additional cameras are added to the system, the error in the measured velocity field will decrease while the effective resolution will increase,

but this adds additional physical and computational complexity to the system.

The two-dimensional images from these cameras are then used to computationally reconstruct a three-dimensional best estimate approximation to the original particle field. Once this three-dimensional image is calculated, straightforward extensions of two-dimensional PIV algorithms are applied to this image to yield three-dimensional velocity fields. In practice, this process involves multiple steps.

3.2.1 Camera Calibration

First, a camera calibration function is determined for each camera. This function relates the three-dimensional world (or experimental) coordinates to the two-dimensional image coordinates in each camera. The calibration functions are primarily used during the tomographic reconstruction process, although the functions can also be used to measure the image magnification, perform three-dimensional particle tracking, and estimate the relative camera positions.

The data for determining the camera calibration are typically collected by traversing a precisely manufactured calibration grid through the measurement volume. The position of markers on the calibration grid and the traversal distances are generally known to a high degree of accuracy yielding a list of world coordinates (X_i, Y_i, Z_i). The position of the markers in the camera images (x_i, y_i) is then determined, typically by applying some type of sub-pixel fitting, to yield a list of point correspondences between the world coordinates and the image coordinates. This list of point correspondences $(X_i, Y_i, Z_i, x_i, y_i)$ is then used to fit a calibration function for each camera that can be used as a model relating the world coordinates to the image coordinates. The calibration functions may be further refined using a self-calibration procedure that uses the particle positions as a higher resolution calibration

dataset [65].

The imaging system of many cameras can be closely approximated by a model referred to as a pinhole, projective, or linear camera model. Mathematically this model is a homographic transformation between the world and image coordinates [21] and thus can be expressed as the straightforward linear relation

$$\begin{bmatrix} wx \\ wy \\ w \end{bmatrix} = \begin{bmatrix} P_{11} & P_{12} & P_{13} & P_{14} \\ P_{21} & P_{22} & P_{23} & P_{24} \\ P_{31} & P_{32} & P_{33} & P_{34} \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{3.1}$$

where the matrix $\mathbf{P}$ is referred to as the projective matrix and w is a dummy variable used to represent the homographic transform. The projective matrix $\mathbf{P}$ is calculated from the list of point correspondences $(X_i, Y_i, Z_i, x_i, y_i)$ using a singular value decomposition of a matrix constructed from the calibration coordinates [21]. In practice, many cameras have geometric distortions that cause significant errors if Equation 3.1 is used as a global calibration model across all world coordinates. However, since this model can be easily inverted to yield world coordinates as a function of image coordinates, the model can be used in calculations that do not require high accuracy.

Equation 3.1 can be modified by adding distortion terms to increase the accuracy of the calibration model. However, since it is difficult to account for all potential sources of distortion, a more generalized calibration model based upon a cubic polynomial is used in most

calculations. This calibration functions given by

$$\begin{aligned} x = a_0 &+ a_1X + a_2Y + a_3Z + a_4X^2 + a_5XY + a_6Y^2 \\ &+ a_7XZ + a_8YZ + a_9Z^2 + a_{10}X^3 + a_{11}X^2Y + a_{12}XY^2 \\ &+ a_{13}Y^3 + a_{14}X^2Z + a_{15}XYZ + a_{16}Y^2Z + a_{17}XZ^2 + a_{18}YZ^2 \end{aligned}$$

$$\begin{aligned} y = b_0 &+ b_1X + b_2Y + b_3Z + b_4X^2 + b_5XY + b_6Y^2 \\ &+ b_7XZ + b_8YZ + b_9Z^2 + b_{10}X^3 + b_{11}X^2Y + b_{12}XY^2 \\ &+ b_{13}Y^3 + b_{14}X^2Z + b_{15}XYZ + b_{16}Y^2Z + b_{17}XZ^2 + b_{18}YZ^2 \end{aligned} \quad (3.2)$$

was suggested in [60] and are now commonly used in PIV applications. The coefficients of these polynomials can be calculated using a least-squares fitting procedure. While Equation 3.2 can accurately model a large number of camera systems, including those with significant distortion, the calibration function cannot be directly inverted and numerical methods must be relied upon for these calculations.

3.2.2 Weighting Matrices

To calculate the tomographic reconstructions, the relationships between the world volume voxels and the camera image pixels must be determined. Specifically, the calibration function is used to calculate which voxels map to which pixels. This relationship can be expressed as a weighting matrix $\mathbf{W}$ that is multiplied by the three-dimensional particle field image vector $\mathbf{I}$ to yield each specific camera image vector $\mathbf{C}$, i.e.

$$\mathbf{C} = \mathbf{W} \cdot \mathbf{I} \tag{3.3}$$

where the image vectors $\mathbf{I}$ and $\mathbf{C}$ are simply one-dimensional copies of the three-dimensional particle field array $I(X, Y, Z)$ and the two-dimensional camera array $C(x, y)$. Thus, a weighting matrix must be calculated for each camera image which is the total number of camera pixels $nx_{pix} \cdot ny_{pix}$ rows by the total number of reconstruction voxels $nX_{vox} \cdot nY_{vox} \cdot nZ_{vox}$ columns in size.

For typical tomographic PIV setups, the weighting matrix is very large. For a system with four one megapixel cameras, this matrix will contain approximately 10^{15} elements. The large number of elements means that directly calculating the weighting matrix is computationally infeasible. Additionally, assuming that the matrix is stored with floating point precision, this would require thousands of terabytes of memory storage. However, in practice each voxel will only map to a small number of pixels, so the weighting matrix is largely sparse. Assuming that the magnification of the voxels is similar to that of the pixels, as is commonly done in tomographic PIV, this means that the number of non-zero values in the matrix is on the order of 10^{10} elements.

Thus, by calculating and storing only the elements that might be non-zero, the weighting matrix can be relatively efficiently calculated and stored. There are two primary methods that are used to calculate the weighting matrix elements: a cylinder-sphere intersection algorithm and a convex-hull intersection algorithm. The cylinder-sphere algorithm is computationally easier to calculate, however it is less accurate than the convex-hull algorithm.

The weighting matrix specifically stores the proportion of the volume of each voxel that intersects the line of sight of each pixel. Therefore every entry of the weighting matrix has a value $0 \leq W_{ij} \leq 1$. Since most voxels in the reconstruction do not intersect most camera

Algorithm 1 This algorithm is used to efficiently calculate the tomographic reconstruction weighting matrices from the calibration functions $x = f(X, Y, Z)$ and $y = g(X, Y, Z)$.

```
1: procedure WeightingMatrix(f(X,Y,Z), g(X,Y,Z))
2:     for X_min ≤ X ≤ X_max do
3:         for Y_min ≤ Y ≤ Y_max do
4:             for Z_min ≤ Z ≤ Z_max do
5:                 x_cam ← f(X,Y,Z)
6:                 y_cam ← g(X,Y,Z)
7:                 for ⌊x_cam⌋ ≤ x ≤ ⌈x_cam⌉ do
8:                     for ⌊y_cam⌋ ≤ y ≤ ⌈y_cam⌉ do
9:                         W_(X,Y,Z,x,y) ← IntersectVolume
10:                    end for
11:                end for
12:            end for
13:        end for
14:    end for
15: end procedure
```

pixels, the weighting matrix may be efficiently calculated by skipping a large number of potential calculations. Qualitatively, the calculation algorithm is described in Algorithm 1. Assuming that the voxel magnification approximately equals the pixel magnification, then only a small number of pixels need to be iterated over for each voxel. This effectively decreases the number of calculations by a factor of approximately 10^5 for typical tomographic PIV systems.

The $IntersectVolume$ function within the algorithm corresponds to the method used to intersect each pixel with each voxel. In the cylinder-sphere method, the pixel is back-projected into the volume as an infinitely long cylinder which is then intersected with the voxel represented as spheres. This calculation method is described in [2, 8, 9, 10]. An analytical solution to the cylinder-sphere intersection volume is described in [28] as functions of numerically evaluated elliptic integrals [49].

The convex hull intersection algorithm involves first representing the back-projected pixel as rectangular frustum. This frustum may then be converted to a convex hull triangular

mesh. The cubic (or more generally rectangular) voxel may also be converted to convex hull triangular mesh. Then since the intersection of two convex hulls is also a uniquely defined convex hull, these two geometries may be intersected. The volume of the resultant geometry (if it is not a null set) may be calculated by using an application of Gauss's theorem.

Both of these algorithms allow the weighting matrix to be calculated in a reasonable amount of time. However, the weighting matrices are still very large in terms of memory. Additionally, multiplying the reconstruction image by the weighting matrix is still computationally expensive if the non-zero elements are directly stored. Thus, a function was developed to convert the weighting matrix to a compressed sparse row (CSR) format which both decreases memory and increases matrix multiplication efficiency.

Storing the weighting matrices in CSR format decreases the memory requirements, however the memory is still quite extensive. So an additional function was developed to further compress the weighting matrix by converting the weighting matrix values from floating point to integer precision. Native HDF5 data compression is also used to store the weighting matrices on the hard disk with less memory as well.

3.2.3 Particle Images

Particle images for tomographic PIV appear similar to those used in two-dimensional PIV, however instead of a thin sheet, a volume of particles is illuminated. This illumination is typically accomplished by optically expanding a laser beam and using shutters to create well defined boundaries to the volume. Since the cameras need to keep a volume of particles in focus instead of a single plane, the camera lenses are commonly stopped down to increase the depth-of-field. This also has the effect of decreasing the light entering the cameras and will decrease the signal-to-noise ratio of the data. Therefore, the size of the volume that can

be imaged is partially limited by the available illumination.

The quality of the tomographic PIV velocity field data depends heavily on the number of particles imaged by the cameras. Sparse images with very few particles can result in high quality tomographic reconstructions. However, the low particle density results in significant errors in the measured velocity field due to few tracer particles to match in the cross-correlations. High particle densities reduce the quality of the tomographic reconstructions since particles will effectively obscure one-another, which also results in higher error velocity fields. This implies that there is generally an optimal particle density for tomographic PIV. This effect is investigated further in the software validation simulations.

Typically the camera particle images are pre-processed to increase the particle contrast and reduce any background noise prior to the reconstruction calculations. This generally involves normalizing the particle images so that the particle intensity is approximately constant across the images, applying a minimum background subtraction, and applying a thresholding operation that sets the images to zero below some defined threshold [52]. The thresholding operation is built into the tomographic PIV software described here, but additional camera image pre-processing would need to be carried out in other image processing software.

3.2.4 Tomographic Reconstruction

Once the two-dimensional particle images are collected by the cameras, they are then combined with the calibration data via the weighting matrices to calculate the tomographic reconstructions of the three-dimensional particle volume. The Multiplicative Algebraic Reconstruction Technique (MART) [1, 2, 8, 9, 10, 23] calculation method is described in Algorithm 2 where the reconstruction is repeated k_{max} iterations. The relaxation parameter μ controls how quickly the reconstruction process converges towards a solution and is typi-

Algorithm 2 This describes the MART calculation algorithm given the previously calculated weighting matrices and the camera images. The weighting matrix for the n^{th} camera of n_{cam} total cameras is denoted as $\mathbf{W}_n$ while the n^{th} camera image is denoted by $\mathbf{C}_n$. The number of MART iterations is given by k_{max} and μ is a relaxation parameter that is typically set equal to 1.

procedure MARTCALCULATION($\mathbf{W}, \mathbf{C}$)
  $\mathbf{R} \leftarrow \mathbf{1}$ ▷ Set all voxels equal to one
  for $1 \leq k \leq k_{max}$ **do** ▷ Iterate reconstruction k_{max} times
    for $1 \leq n \leq n_{cam}$ **do** ▷ Iterate over all cameras
      $\mathbf{R} \leftarrow \mathbf{R} \cdot \left(\frac{\mathbf{C}_n}{\mathbf{W}_n \cdot \mathbf{R}}\right)^{\mu \mathbf{W}_n}$
    end for
  end for
end procedure

cally set equal to 1. This method is effectively an iterative error-correction scheme that is applied to adjust the three-dimensional images so that their projections closely match the experimental images recorded by the cameras.

During testing of the reconstruction software, it was found that the process of raising the correction factor to the power of the weighting matrix was relatively slow. So the reconstruction equation was re-written to a logarithmically transformed version

$$\mathbf{R} \leftarrow \mathbf{R} \cdot \exp\left(\mu \cdot \log\left(\frac{\mathbf{C}_n}{\mathbf{W}_n \cdot \mathbf{R}}\right) \cdot \mathbf{W}_n\right)$$

which was found to run significantly faster. This is due to the combination of facts that the computational complexity for multiplications is lower than for calculating powers and that CSR matrix format is optimized for matrix multiplication.

Between each iteration of the MART reconstruction process, Gaussian smoothing and Laplacian sharpening filters may be applied to the reconstruction images. The smoothing operation produces higher quality reconstructions in particular when the particle diameters are one pixel or smaller since spreading the reconstructed particles out ensures that they fully

intersect amongst all cameras. The sharpening operation is then applied to ensure that the particles maintain relatively sharp peaks that give strong cross-correlation signals.

3.2.5 PIV Processing

The three-dimensional reconstructed images are processed using standard PIV algorithms. These algorithms break the images into a number of smaller windows that are then cross-correlated in time with other image frames. The cross-correlations are calculated using discrete Fourier transforms to increase speed. The location of the peaks in the correlation volumes is calculated with sub-voxel accuracy using a variety of fitting methods, which yield a velocity vector for each window. Several filtering methods are applied to the full vector field. This process is iteratively repeated to yield a velocity field for each reconstructed image frame.

Several different cross-correlation methods are built into the software. Standard cross-correlation simply extracts the windows and applies the cross-correlations without any processing applied to the windows. Window deformation uses the estimates of the velocity field from the previous PIV iterations to transform the fluid parcels corresponding to each window into an un-deformed state prior to the cross-correlation [51] which typically results in lower error, particularly in high-shear flows. Additionally, both standard cross-correlations and phase correlations may be applied in the software.

The measured velocity fields are validated using both thresholding and the Universal Outlier Detector algorithm [63]. The identified outlier vectors are replaced using a Laplacian interpolation scheme. Once the vector field has been validated, a Gaussian smoothing filter is applied to minimize high frequency noise, particularly when additional PIV passes will be completed and the previous passes are used as initial guesses in subsequent passes.

3.3 Software Description

The tomographic PIV software is designed to be relatively easy to install, cross-platform, and straightforward to call from other scientific programming languages. The software runs entirely from the command line and is controlled by passing parameter files created in the commonly used Hierarchical Data Format Version 5 (HDF5) scientific data format [20] as function arguments. All processed data is returned as additional HDF5 data files. This has the advantage that the software may be called as scripts to serially process large quantities of data with little user input required once the script begins running. The primary disadvantage of this processing method is that the software provides little feedback to the user by itself and the HDF5 data files must be opened by other software to analyze the software output.

When possible, publicly available libraries were used in the software to ensure that critical functions were efficient and thoroughly tested, while minimizing the amount of additional code that was necessary to write. At the same time, libraries were only chosen to be used in the software if they were known to be well supported both in terms of common usage and in terms of being regularly updated. These external libraries included the following:

- HDF5: Hierarchical Data Format Version 5 [20]
- BLAS: Basic Linear Algebra Subprograms [61]
- LAPACK: Linear Algebra Package [45]
- FFTW: Fastest Fourier Transform in the West [13]
- LibTIFF: Tag Image File Format Library [32]
- OpenMP: Open Multi-Processing [44]

Oftentimes, these libraries are used in multiple different functional parts. For example, HDF5 file format is used for communication between different functions, for passing processing parameters to functions, and for storing processed data. The HDF5 format was also chosen as it is widely supported by other research programming languages including Python, MATLAB, Mathematica, and Julia. This means that once the tomographic PIV software is compiled, its functions may be directly called from these languages.

Additionally, functions were written to be as universal as possible so that they could be used in multiple different locations across the processing pipeline. A cubic interpolation library [29] was developed that was used in both interpolating three-dimensional images as well as velocity fields as an example.

The software is divided into roughly two groups of functions. The primary group of functions performs the tomographic PIV processing. Given input data from a tomographic PIV experiment, these functions will output reconstructed three-dimensional particle field images along with calculated vector fields. The second group of functions were developed to create simulated tomographic PIV data that was used to validate the first group of functions. These functions all communicate by passing HDF5 parameters or data files to one another. A single function was developed to run a variety of simulation validation cases.

The tomographic PIV group of functions outputs two primary types of HDF5 data: tomographically reconstructed three-dimensional images of the particle fields and three-dimensional velocity vector fields. In addition to the reconstructed images, the reconstruction data files also include the voxel coordinates (in the calibration coordinate system). Since the reconstruction images can be up to a gigabyte in size, lossless compression, either ‘zlib’ or ‘szip’, may be applied to the saved images. Additionally, the images may be saved with lower numerical precision (8, 16, 32, or 64 bits per sample) to further save memory. The velocity vector fields also include the three-dimensional coordinates at which the vectors were

measured as well as a data set indicating whether the vectors were interpolated from the remaining vector field due to being flagged as outliers.

3.4 Software Validation

Testing the accuracy of the tomographic reconstructions and the measured velocity vector fields that are calculated from the tomographic PIV software requires having an input data set with a known true solution. Since even for well studied flow phenomena, it is impossible to know the true solution of experimental data with absolute certainty, software was developed to simulate arbitrary tomographic PIV data sets. This software simulates three-dimensional particle fields experiencing flows by a defined velocity field. Camera lenses, angles, and positions are specified to create calibration functions from which camera sensor images of the flow field are generated. Additionally, calibration grid data is created from these calibration functions. Noise can be added to the data at any of these steps to test the sensitivity of the tomographic PIV software to non-ideal experimental data.

3.4.1 Velocity Field

A variety of velocity fields can be simulated including uniform translation, shear flow, sinusoidal flow, and a homogeneous turbulence flow. The library defining the velocity fields was written to be highly portable so that additional velocity fields could be easily added to the library in the future.

The position of the cameras imaging the fluid volume in tomographic PIV will introduce an inherent geometrical bias to the tomographic reconstructions. This primarily manifests as the reconstructed particles effectively being blurred along the imaging axes of the cameras.

Due to this effect, the velocity field error tends to be higher along the imaging axis as well. To ensure that this effect could be accurately measured in the simulations, the homogeneous turbulence velocity field model was used in the validation simulations since the velocity field should not introduce a directional bias (as the shear flow would for example).

The homogeneous turbulence model used in the simulations was designed to provide an analytical model describing a flow field that satisfies continuity, exhibits a Kolgomorov scaling, causes advection of smaller scale vortices by larger vortices, and has independent Fourier modes with Gaussian distributions [15]. In addition to not introducing geometrical biases into the data, this flow will allow the software to be tested with a wide range of both spatial and temporal length scales. The velocity fields are analytically given by

$$\mathbf{u}\left(\mathbf{X},t\right)=\sum_{m=1}^{M_0}\sum_{n=1}^{N_k}\sum_{p=1}^{P_\omega}\left(\mathbf{a}_{mnp}\times\hat{\boldsymbol{\kappa}}_{mn}\right)\cos\left(\boldsymbol{\kappa}_{mn}\cdot\mathbf{X}+\omega_{np}\,t\right)+\left(\mathbf{b}_{mnp}\times\hat{\boldsymbol{\kappa}}_{mn}\right)\sin\left(\boldsymbol{\kappa}_{mn}\cdot\mathbf{X}+\omega_{np}\,t\right)$$

where $\hat{\boldsymbol{\kappa}}_{mn}$ are random vectors distributed on a sphere with a radius s. The radius s determines the spatial length scales of the flow with higher values resulting in a more rapidly varying velocity field. The vectors $\mathbf{a}_{mnp}$ and $\mathbf{b}_{mnp}$ are generated from normal distributions with zero-mean and a variance of one. The time based component ω_{np} also has zero-mean normal distribution with a variance of r which controls how quickly the velocity field changes in time. A single Z-axis plane of a velocity field used in the simulations is shown in Figure 3.1.

While only the uv velocity field is shown in the figure, the homogeneous nature of the flow and the range of length scales is apparent in the flow field. The temporal variance in the simulations was chosen such that the velocity field rapidly evolved and represented a range of different flow fields during any individual simulation case. Since the velocity field is

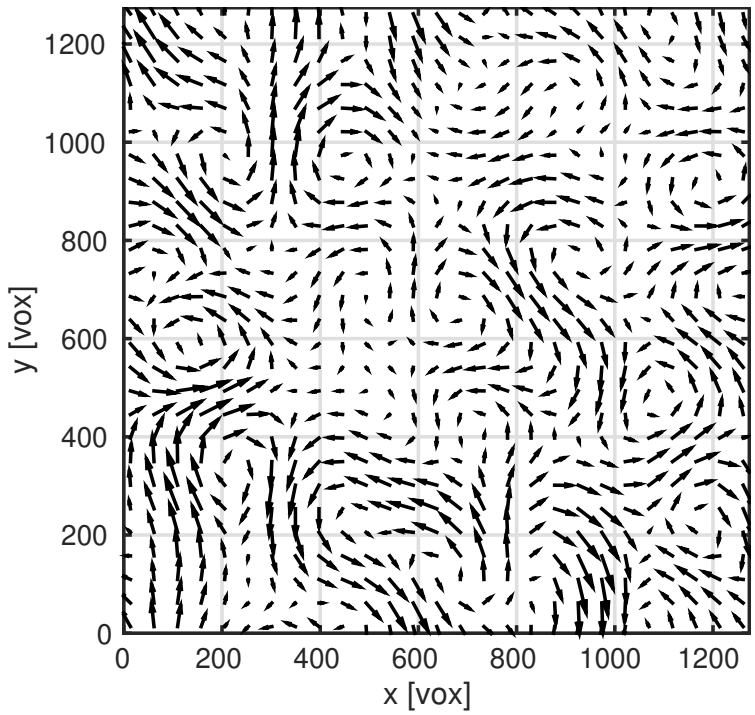


Figure 3.1: This plot shows the uv vector field of an example time instant of a single Z-axis plane of the simulated turbulence velocity field. The homogeneous nature of the flow along with the range of length scales is apparent in the flow.

non-stationary, this also means that any PIV measurements will be averaging the flow in both space and time and thus the measured velocity field error will be higher than for a non-changing flow field.

3.4.2 Particle Field

The three-dimensional particle field image was generated by simulating the particles as localized Gaussian functions with a range of different intensities and diameters. The intensity values and particle diameters were taken from a randomly generated log-normal distribution as is typically seen in PIV experiments. In practice, unless a very wide distribution is specified by the user, the distributions will be approximately normal.

The particles are uniformly 'seeded' within the simulated volumes and advected along the velocity field using a 4^{th} order adaptive Runge-Kutta-Fehlberg differential equation solver. The edge of the volume is 're-seeded' between every frame so that an arbitrary number of frames can be generated with constant particle densities. Any particles that leave the simulated volume are removed from the simulation and replaced by new randomly generated particles. This is experimentally unrealistic, but should not affect the results of the tomographic PIV simulations.

The number of particles within the volumes was determined based upon the particle density *ppp* which is defined in terms of the number of particles per pixel in the camera sensors. In tomographic PIV, this particle density is typically in the range $0.02 \leq ppp \leq 0.1$ [52]. However, this metric does not account for the diameter of the particles, so a second metric known as the source density N_s is also used. The source density is defined as

$$N_s = ppp \cdot \frac{\pi}{4} \cdot \left(\frac{d_\tau}{\mu_p}\right)^2$$

where d_τ is defined as the particle diameter on the image sensor and μ_p is the pixel pitch of the image sensor, so that d_τ/μ_p is the particle diameter in pixels. The source density is equivalent to the proportion of the area of the image that the particles in the image cover. For tomographic PIV the source density is typically in the range $0.1 \leq N_s \leq 0.3$ [52].

In the simulations completed here, the particle density was varied across the range $0.01 \leq ppp \leq 0.42$ with the mean particle diameter (defined in terms of the full width half maximum of the Gaussian intensity function) equal to $d_\tau/\mu_p = 2.25$ pixels. This resulted in a source density range of $0.04 \leq N_s \leq 1.68$. These ranges extend from below to above typical tomographic PIV ranges, but were chosen to test the software over a wide range of data. More thorough testing was completed at a narrower range of particle densities.

The simulated particle fields are stored in the same HDF5 data format as the reconstructions and additionally include the precise particle positions, intensities, and diameters so that the reconstruction particle statistics can be compared with the known true statistics.

3.4.3 Camera Simulation

The cameras are simulated in a rectilinear grid that can be an array of any size, however all simulations discussed here consisted of 2 by 2 camera grid since this is the most common configuration found in tomographic PIV experiments. The simulated cameras are offset from the Z-axis by a defined angle θ and are placed equidistant from the center of the simulated volume. The camera lens, sensor resolution, and sensor pixel size can all be specified. The simulations here were all completed using 105 mm lenses with the camera center placed at 1000 mm from the volume origin and using a 1024 by 1024 sensor with 17 μm pixel pitches. The angle θ was varied during the simulations, however the other parameters were held constant.

The camera sensor images may be calculated using one of two methods. The first method involves multiplying the simulated three-dimensional particle field images by the camera weighting matrices. And the second method maps the three-dimensional particle positions to the sensor images using the calibration functions and directly calculating the two-dimensional Gaussian functions. The second method was used in the simulations here since calculating the weighting matrices is computationally intensive, so this allowed the simulations to be run more efficiently.

Camera calibration grid data was also generated in the form of a list of point correspondences between the world coordinates and the image coordinates $(X_i, Y_i, Z_i, x_i, y_i)$ for each camera. The ΔX, ΔY, and ΔZ spacing of the calibration grid world coordinate data could be

specified and was set equal to $\Delta X = \Delta Y = 15$ mm and $\Delta Z = 5$ mm for the simulations as this represented a typical tomographic PIV experimental case.

3.4.4 Performance Metrics

The simulation software records a wide range of performance metrics including multiple image similarity metrics, particle reconstructions errors, and velocity field errors. However, to compare these results with other tomographic PIV studies, two metrics were focused on: the reconstruction normalized cross-correlation quality and the velocity field error.

The normalized cross-correlation quality factor Q is a metric that specifically investigates the fidelity of the tomographic reconstructions $R(X, Y, Z)$ compared to the original simulated three-dimensional particle intensity field $I(X, Y, Z)$ and is defined by

$$Q = \frac{\sum_{X,Y,Z} R(X,Y,Z) \cdot I(X,Y,Z)}{\sqrt{\sum_{X,Y,Z} R(X,Y,Z)^2 \cdot \sum_{X,Y,Z} I(X,Y,Z)^2}}.$$

This factor was first proposed for measuring tomographic PIV images in [8] due to being a convenient fidelity metric both in terms of being easy to evaluate and due to having simple to interpret output values. The factor will evaluate to exactly 1 when the two images are equal and approaches 0 as the images diverge. In practice, the value of Q will rarely drop below $Q \approx 0.1$ since even random noise will tend to have a non-zero correlation.

The velocity field error is defined as the mean magnitude of the difference between the simulated velocity field $\mathbf{u}(\mathbf{X}, t)$ and the measured velocity field $\mathbf{u}_p(\mathbf{X}, t)$ where the average is taken over all m calculated vectors in each of the n simulated frames.

$$|\epsilon| = \frac{1}{mn} \sum_{i=1}^{m} \sum_{j=1}^{n} |\mathbf{u}(\mathbf{X}_i, t_j) - \mathbf{u}_p(\mathbf{X}_i, t_j)|$$

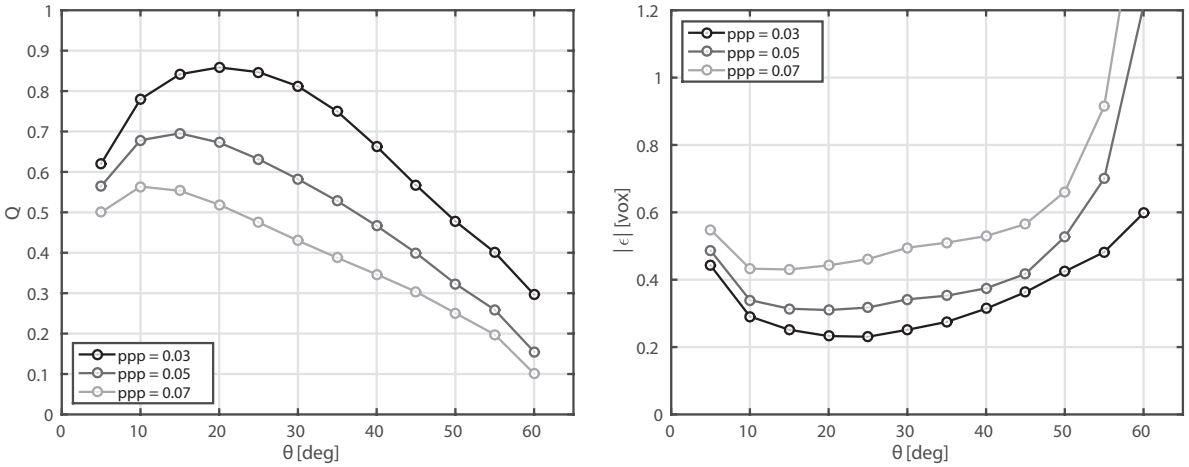
This metric was chosen since the velocity field is the output that will ultimately be taken from any tomographic PIV experiments. Additionally, it is known that in ideal scenarios, the minimum PIV error is typically about 0.1 voxel per frame. Therefore, if the results of the simulations approach this value, then it can be assumed that the software is performing correctly.

For the simulations described here, the flow fields were simulated across 50 frames of time with a 53 by 53 by 8 vector velocity field measured for each frame. A single PIV pass was completed for each frame using a 64 by 64 by 64 voxel window with an effective resolution of 32 by 32 by 32 voxels. Both velocity vector thresholding and the universal outlier detector were used to validate the vector fields with identified outlier vectors replaced using Laplacian interpolation.

3.4.5 Camera Angle

In many tomographic PIV studies, both experimental and simulated, four cameras are placed in either a square or a cross shape that is offset from the experimental axis by an angle [52]. The optimal angle depends upon the source density, the thickness of the imaged volume, and the lens system (and corresponding depth-of-field) of the cameras. However, it has been generally found that the optimal reconstruction quality results are achieved when the camera angle is approximately $20° \leq \theta \leq 30°$ [8, 52]. Since this is a well studied problem, this was chosen as an example case to investigate and validate the tomographic PIV software.

In these simulations, a two by two camera array views the simulated particle field volume

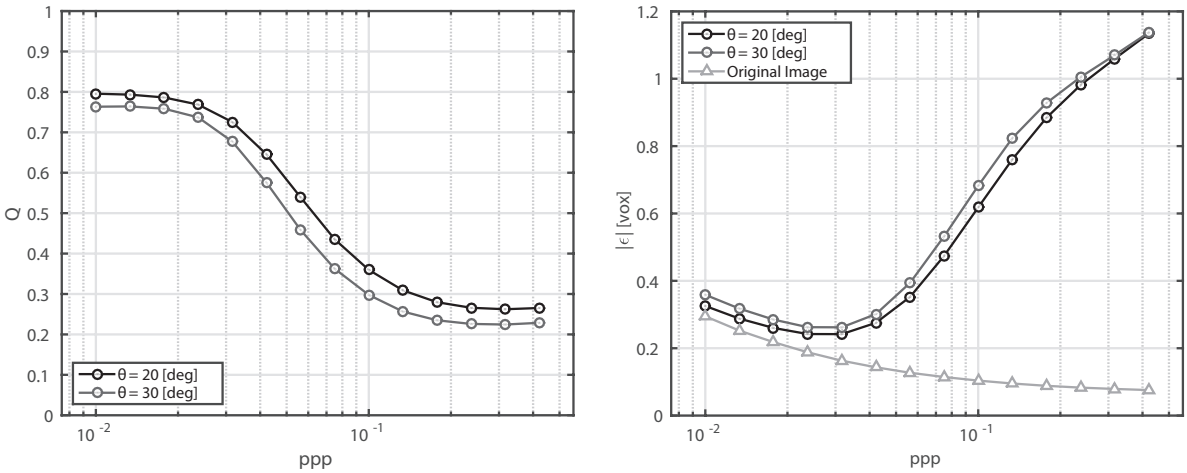


(a) Quality as a function of camera angle (b) Velocity error as a function of camera angle

Figure 3.2: This figure shows the effect of camera angle on the reconstruction quality and velocity error. It can be seen that the optimal camera angle is a function of particle density since higher camera angles will require viewing the volume through longer lines of sight and thus more particles will be imaged by the cameras.

with an angle varying from 5° to 60°. Particle densities of 0.03, 0.05, and 0.07 *ppp* are simulated which correspond to source densities of 0.12, 0.20, and 0.28. The reconstructed volume had a thickness of 192 voxels, but the width of the volume varied with the camera angle and ranged from 986 by 986 voxels down to 526 by 526 voxels at the highest camera angle. The results of the simulations are shown in Figure 3.2.

The maximum reconstruction quality factor can be seen to occur for lower camera angles than the minimum velocity field error across all three particle densities tested. This indicates that previous simulation studies that have primarily relied on the reconstruction quality factors in determining the effectiveness of tomographic PIV may have suggested lower camera angles than are truly optimal for measuring the velocity fields. Additionally, it was found that while increasing the particle density resulted in substantially lower reconstruction qualities, the negative effect was less detrimental to the measured velocity field. However, overall these results are in close agreement with previous tomographic PIV simulations.



(a) Reconstruction quality as a function of particle density

(b) Velocity error as a function of particle density

Figure 3.3: This figure shows the effect of particle density on the reconstruction quality and velocity error. It can be seen that there is an optimal particle density for measuring the velocity field since at low particle densities, there are not enough particles to sufficiently resolve the velocity field. But at higher particle densities, the reconstruction noise starts negatively affecting the velocity measurement.

3.4.6 Particle Density

Since particle density is known to strongly affect the measurements of tomographic PIV, this was chosen as a second case to investigate with simulations. In these simulations, the camera angle was held constant at two values of $\theta = 20°$ and $\theta = 30°$ while the particle density was logarithmically varied from $ppp = 0.01$ to $ppp = 0.42$. The reconstructed volume was 888 by 888 by 192 voxels for the 20° case and 720 by 720 by 192 for the 30° case. The results of these simulations are shown in Figure 3.3.

The reconstruction quality monotonically decreases with the particle density. This would seem to indicate that any high seeding density will result in poor quality PIV measurements. However, error in the measured vector field decreases up to a seeding density of $ppp \approx 0.03$. The reason for this can be seen in Figure 3.3b where the measured PIV error of the original

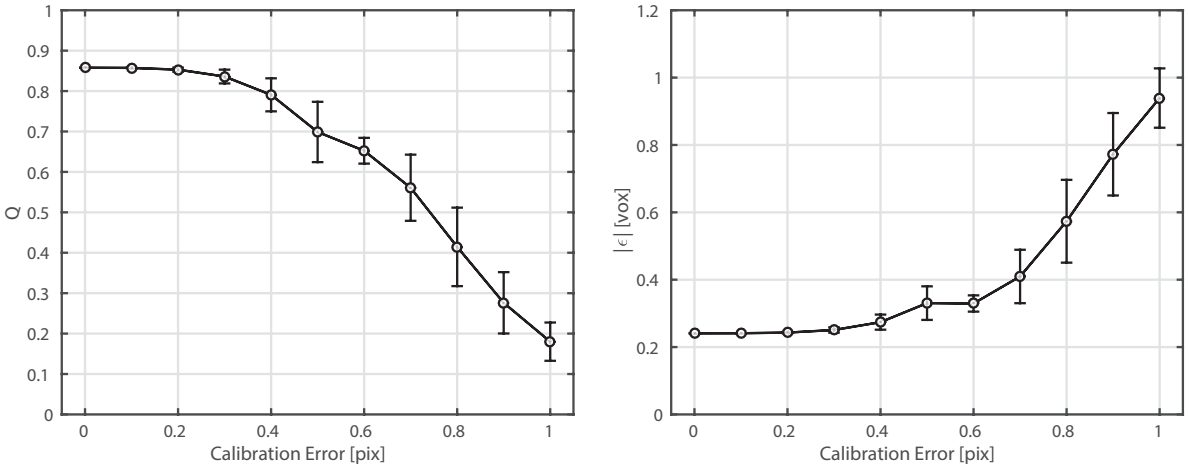
simulated particle image is shown in addition to the error of the reconstructed image. The PIV error of the original simulated image continues to decrease across the whole range of particle densities simulated here. So at lower densities, the additional particles contribute more information to the velocity measurement. But eventually the additional noise in the reconstructed image will overwhelm the additional velocity information and the PIV error will begin to increase.

The velocity field error measured here appears to be in agreement with previous work that has shown that optimal tomographic PIV measurements will typically occur in the range $0.02 \leq ppp \leq 0.1$. The PIV measurements on the original simulated volume also indicate that the PIV measurements are performing as expected as the error approaches the typical lower value of 0.1 voxels.

3.4.7 Calibration Error

It is known that relatively small errors in the estimated calibration functions can cause significant errors in the reconstructed data. For this reason, the effect of calibration error was tested using the simulations. Gaussian noise was added to the image coordinates of the simulated grid points prior to calculating the calibration functions. This closely approximates what would occur during a physical experiment where the sub-pixel position of the grid points would be measured with some non-zero error. The standard deviation of the Gaussian noise was varied from 0 to 1 pixels in 0.1 pixel increments. Four trials were conducted at each error level with 887 by 887 by 192 voxel volumes. The results of these simulations are shown in Figure 3.4.

Only four trials were run for each error level since it was necessary to calculate new weighting matrices for each new calibration function and calculating the weighting matrix is the slowest



(a) Reconstruction quality as a function of the calibration error

(b) Velocity error as a function of the calibration error

Figure 3.4: The data in these plots was calculated by adding Gaussian noise to the simulated calibration coordinate data prior to calculating the calibration functions. The standard deviation of the Gaussian noise is plotted along the horizontal axis of the graphs. Confidence intervals showing the 99% confidence range are plotted here since only a small number of trials could be run and the uncertainty is higher than for the other simulations.

part of the simulation process. This means that the results from this simulation had higher statistical uncertainty than the other simulations. Therefore, the 99% confidence interval is plotted for the calibration error simulations. (For the other simulations, the confidence interval was smaller than the markers in the plots.)

The results of the calibration error simulation show that errors up to about 0.5 pixels can occur before the error in the reconstruction and velocity field start rapidly increasing. This is in close agreement with other work that has examined the effects of calibration errors [8].

Interestingly, the reconstruction quality and the velocity field error are very closely correlated in this simulation in contrast to the results investigating camera angle and particle density effects. It is speculated that this due to the fact that with the camera angle and particle density cases, artifacts are generated during the reconstruction process known as ghost particles [2, 8, 11], but these ghost particles tend to be correlated with nearby par-

ticles and thus tend to flow with the actual tracer particles to an extent, so the measured velocity is less affected by the ghost particles. However, in this case since the error is directly affecting the calibration function, the geometry of the reconstruction is being changed and thus the potential correlation between the tracer particle and the ghost particles is lost. This indicates that calibration error may be more important than other sources of error in tomographic PIV, however this effect should be investigated further.

3.5 Conclusions

This work has shown that the developed tomographic PIV software functions well and produces results that are similar to previous tomographic PIV studies. The software was primarily developed in Linux and currently can be compiled and installed with an included Makefile that should directly work in all Linux and Apple operating systems. A standalone executable file could also be produced that would work in Windows based systems with relatively little modification to the source code. The software will likely be distributed via Github [19] with an open source license.

There are also several updates that should be included in future versions of the software. First of all, additional parallelization should be added to the software, in particular to the process of calculating the weighting matrix as this is currently the slowest operation. Adding the ability to process across multiple nodes using the Open MPI library [39] should also be relatively straightforward and has the potential to dramatically increase the speed of the software.

Since the particle positions correlate in time, but the noise in the reconstructed images tends to lose correlation very quickly across multiple frames, the quality of time resolved data can be improved by combining the data from multiple frames together in a technique known as

Motion Tracking Enhanced (MTE) tomography [43]. This technique works by deforming the reconstructed particle fields by the measured velocity fields and then combining multiple frames of deformed data together. The combined reconstructions are then used to initialize new tomographic reconstructions. All the components of this process already exist in the tomographic PIV software, so implementing this feature should be simple as well.

Adding multiple frame correlation methods such as pyramid correlations [57] or fluid trajectory correlation [34] would also increase the accuracy of the velocity field measurements and could utilize previously developed tools within the software. Some additional tomographic PIV methods such as volume self-calibration [65] ideally need a graphical user interface could also be developed, but this would change the paradigm that the software can be easily called by other software through the command line.

Chapter 4

Multi-camera Plenoptic PIV

In many fluid systems, measuring the full three-dimensional velocity field is essential to understanding the flow dynamics. However, due to the increased complexity and computational costs typically associated with three-dimensional measurement techniques, these experimental methods still remain much less common than planar techniques. Many different experimental techniques have been proposed for measuring three-dimensional velocity fields using Particle Image Velocimetry (PIV) including defocussing PIV [26, 46, 47], holographic PIV [24, 36], and tomographic PIV, among several others. In particular, tomographic PIV has emerged as the standard three-dimensional imaging technique currently used in fluid dynamics. This method involves imaging the fluid volume from several different directions simultaneously and then combining these images together to produce a three-dimensional reconstruction. Several tomographic reconstruction algorithms have been proposed including the Multiplied Line of Sight (MLOS) technique [2, 38] and the Multiplicative Algebraic Reconstruction Technique (MART) [1, 2, 8, 9, 10, 23] which has become the standard three-dimensional imaging method in fluid dynamics.

Recently, so-called plenoptic or lightfield cameras have been proposed for performing single camera three-dimensional fluid velocity field measurement [6, 12, 35, 42]. Conventional cameras only record the spatial variation of light intensity, but plenoptic cameras have the unique capability of additionally measuring the propagation direction of the incoming light [16, 17, 40]. The combination of both the spatial and angular information of the light, which

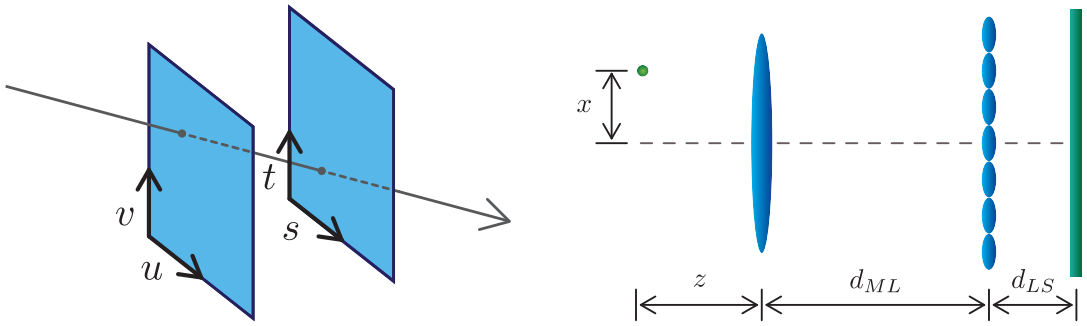
can be measured by a single camera, yields an approximation to the lightfield function that can be used to create three-dimensional reconstructions [5, 6, 12, 35, 42]. Unfortunately PIV reconstructions created using a single plenoptic camera suffer from the same low angular resolution issues as do the single camera defocussing PIV [26, 46, 47] and holographic PIV techniques.

While a camera sensor ostensibly records only two-dimensional data, the lightfield function is in fact seven-dimensional. This function was first described by Arun Gershun as the quantity of light L, of a particular wavelength λ, passing through a point in space (x, y, z), traveling in a specific direction (θ, ϕ), at a precise moment in time t [18]. However, the lightfield function $L(x, y, z, \theta, \phi, \lambda, t)$ may be described by a four-dimensional approximation to enable measurement with standard camera sensors. For typical PIV measurements, the color of the light λ may be neglected. Further assuming that attenuation of the light is negligible, the radiance along a particular lightray may then be assumed to be constant, thus eliminating the need for three spatial dimensions. Finally, since the speed of light is much greater than processes found in typical PIV, the temporal dimension may be assumed to be constant for a particular image frame. Therefore the lightfield may be described by the simplified function $L(x, y, \theta, \phi)$ without loss of generality. This function may be parameterized by several different methods, however in this analysis, the function will be represented by the intersection of the light rays with two parallel planes. This may be interpreted as being the two points of intersection (u, v) and (s, t) within the planes. A diagram illustrating this parameterization is shown in Figure 4.1a.

A variety of experimental techniques have been developed to record lightfield data. Stereo camera setups used to extract depth information from images were the first approximations to lightfield measurements [4, 14, 48]. However stereo systems do not allow for computational refocusing or synthesis of new imaging viewpoints. The first practical lightfield measurements

were made using cameras moving on gantries [31], but since the angular information of these systems is encoded in time, they are impractical for fluid measurements. To enable time-resolved measurements of lightfields, arrays of cameras were designed in which each camera measures a different set of lightfield angles [66]. This technique was applied to experimental fluid measurements to develop the Synthetic Aperture PIV technique which allows the fluid particle lightfield to be computationally refocused to produce a volumetric reconstruction [5]. To develop the plenoptic camera system to measure lightfields, an array of micro-lenses was placed directly in front of the sensor in a standard camera, allowing for both spatial and angular information to be recorded by a single camera [40].

A plenoptic camera measures both the position and the angle of incoming light rays to produce a four-dimensional radiance function [16, 17, 30, 40]. This function can be used to computationally refocus the camera to a range of different focal lengths to produce a focal stack. In PIV applications, this focal stack consists of a three-dimensional intensity field of tracer particles upon which standard cross-correlation may be performed to yield a fluid velocity field. There are currently two primary designs of lightfield cameras that are referred to as plenoptic 1.0 cameras and plenoptic 2.0 cameras. Plenoptic 1.0 cameras densely sample the angular information of the lightfield while sampling the spatial information at a relatively low resolution. In contrast, plenoptic 2.0 cameras sample the angular information at a low resolution while sampling the spatial information at a high resolution [33]. Though high spatial resolution images have benefits for PIV measurements, the angular resolution is vital for collecting volumetric PIV data. For this reason, the cameras simulated in this work are plenoptic 1.0 cameras. In Figure 4.1b the design of a typical plenoptic camera is shown, consisting of a main lens and an array of microlenses in front of the camera sensor. By varying the distances d_{ML} and d_{LS}, the camera may be switched between plenoptic 1.0 and plenoptic 2.0 configurations. These microlenses produce an array of images in which



(a) Lightfield parameterization (b) Plenoptic camera design

Figure 4.1: (a) A diagram showing a lightray intersecting the (u, v) and (s, t) planes. (b) A diagram showing the design of a plenoptic camera including a particle in free-space in front of the camera, the main lens, the lenslet array, and the camera sensor. Additionally the relative distances between the different camera components are shown.

each image corresponds to a different set of angles of the lightfield. Typically the set of lightfield angles for each lenslet is relatively small compared to the total angular resolution of the camera and can thus be assumed to be constant. The spatial information of the lightfield is encoded by the position of the pixels on which each microlens image is projected. Three-dimensional reconstructions may then be produced by extracting and integrating the angular and spatial information from the sensor.

A single lightfield camera can produce a three-dimensional reconstruction that is suitable for performing PIV analysis. However the quality of this reconstruction will likely be severely limited by the low angular resolution of the single camera. The maximum angular resolution that a single camera can attain is limited by the diameter of the primary lens. Additionally for performing volumetric reconstructions of a fluid volume, the diameter of the fluid volume must be small in comparison to the main camera lens. For example, a camera with a very large aperture lens that is placed close to a fluid volume will have a much higher angular resolution over the volume than the angular resolution produced by a camera with a narrow aperture lens placed far from the fluid volume.

For this reason, the reconstruction fidelity of the light field cameras will be strongly limited

by the available angular resolution of the cameras. Thus the cameras should perform better when the imaged fluid volume is closer to the camera. However, since the depth-of-field of cameras decreases with the focal distance of the cameras, we speculate that below some distance, too many of the particles will become unfocused and the reconstruction quality will decrease. Therefore there should be a focal distance that is optimal in terms of the reconstruction fidelity.

Since the primary lens limits the angular resolution of the individual cameras, we also hypothesize that the reconstruction fidelity will dramatically increase by adding a single additional lightfield camera to the imaging system. The additional angular resolution provided by the second camera will be controlled by the baseline distance between the cameras rather than the individual cameras lenses. Thus placing the two cameras in a stereo imaging configuration will provide a large increase to the available angular resolution. However since the individual cameras already produce fully three-dimensional reconstructions, we speculate that adding more than a single additional camera to the system will only marginally increase the system fidelity.

To prove these hypotheses, a mathematical model is derived to describe the lightfield camera and a series of simulations is then completed to test the performance of the lightfield cameras. The optical properties of the plenoptic camera are used to develop a camera calibration function relating the world coordinates to the lightfield imaged on the camera sensor. Previous work studying plenoptic cameras has primarily focused on producing computationally refocused planes, which are then combined to yield a focal stack. In contrast, the equations derived here can be used to directly perform volumetric refocusing. During this process the images from each of the individual lenslets are combined together to yield the reconstructed volume. This process has typically involved adding the lenslet images together, which we refer to as the additive reconstruction process. However, in this work we derive a single

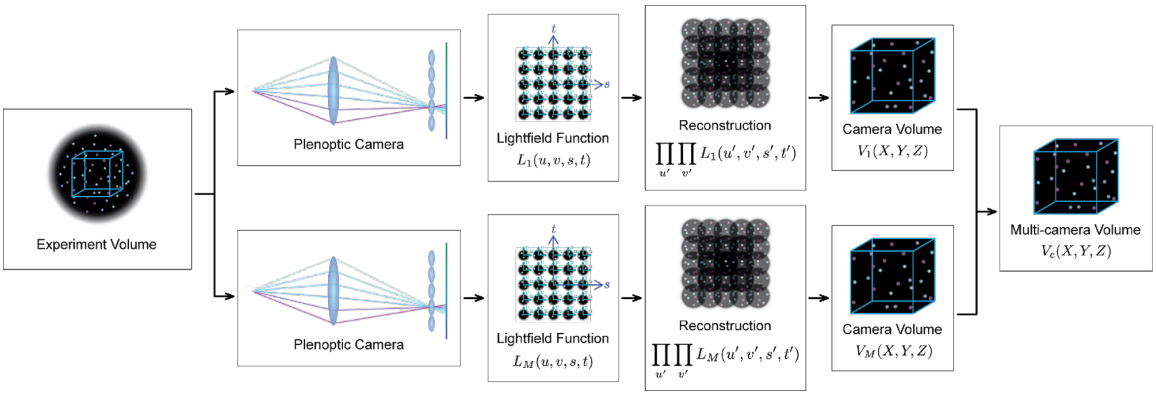


Figure 4.2: This diagram schematically shows the multi-camera plenoptic reconstruction algorithm. First, several plenoptic cameras image the PIV volume. The images from these cameras are next used to calculate the lightfield function. Then the individual reconstruction from each camera is calculated. Finally, the reconstructions from each camera are combined together to form the full volumetric reconstruction.

camera multiplicative reconstruction algorithm and we show that this algorithm produces higher quality reconstructions than the additive refocusing algorithms. We then expand this algorithm to allow multiple plenoptic cameras to be used in performing volumetric reconstructions. This is in contrast to previous works, which have only investigated the use of a single plenoptic camera for PIV measurements [6, 12, 35, 42]. Figure 4.2 shows a diagram illustrating the multiple camera plenoptic reconstruction process.

This work first describes the optical system used to model the lightfield camera. These models are then used to derive the volumetric refocusing algorithm for a single and multiple plenoptic cameras. Finally, the results of the computational simulations are analyzed to compare the performance of multiple plenoptic cameras with that of single plenoptic cameras as well as with the established tomographic PIV MART approach [1, 2, 8, 9, 10].

4.1 Methods

4.1.1 Lightray Simulation

We created a computational model of the optical and imaging systems using a lightray approach and geometrical optics [22]. In these simulations, we modeled combinations of one to four cameras. The simulated cameras each consist of a single large primary lens that focuses the incoming lightrays onto an array of small lenses, referred to as the lenslet array. Figure 4.1b shows the design of the simulated cameras. The individual lenslets are focused at infinity and thus produce out-of-focus images upon the simulated camera sensor. However, these images are generally non-uniform and still contain spatial information. The tracer particles are simulated by computationally creating a series of lightrays emanating from points surrounding each tracer particle. The intensity profile of the lightrays is Gaussian with respect to the radial coordinate from the tracer particle; this is based upon the assumption that the illuminated tracer particles will produce approximately Gaussian shaped intensity distributions upon the image sensor due to diffraction of the light [50]. The lightray source points have a uniform spatial distribution within a radius corresponding to 1% and greater of the peak intensity; no lightrays are simulated that emanate from outside of this radius.

The standard deviation of the Gaussian particles is set equal to 0.7 times the voxel diameter to produce particle images that are consistent in diameter with traditional PIV/PTV measurements. The direction of the lightrays is randomly selected from a uniform distribution that exactly covers the primary lens. This increases the computational efficiency of the simulations since lightrays that do not enter the simulated camera are not created.

The lightrays are propagated through free-space and through the camera lenses using standard optical matrix operations. In this system, the lightray is represented by the vector

$\vec{L}(x, y, \theta, \phi)$ where x and y are the position of the lightray from the z-axis and θ and ϕ are the angles of the lightray to the z-axis in the x and y directions, respectively. All optical operations then correspond to a simple matrix operation $\vec{L}' = \mathcal{M} \cdot \vec{L}$ on the lightray vector $\vec{L}$ where the matrix $\mathcal{M}$ is determined by the type of optical operation. Using this notation, the simplest operation is given by a lightray propagating through a free-space of length d which is described by the transformation

$$\vec{L}' = \mathcal{T} \cdot \vec{L} = \begin{bmatrix} 1 & 0 & d & 0 \\ 0 & 1 & 0 & d \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ \theta \\ \phi \end{bmatrix}$$

which can be seen to change the position of the lightray, but not the angle of the lightray as would be expected from a non-refracting medium. This matrix operation is applied three times for each lightray: first, when the lightray passes from the source particle to the primary lens; second, when the lightray passes from the primary lens to the lenslet array; and finally, when the lightray passes from the lenslet array to the camera sensor. The second matrix operation corresponds to a lightray refracting through a thin lens with a focal length f and is given by

$$\vec{L}' = \mathcal{R} \cdot \vec{L} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -1/f & 0 & 1 & 0 \\ 0 & -1/f & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ \theta \\ \phi \end{bmatrix}$$

which changes the angle of the lightray, but not the position of the lightray. This operation is used when the lightrays pass through the main lens that lies on the optical axis. However since the lenslets do not lie on the optical axis, a modified form of this operation must be

used to calculate the transformation of a ray passing through the lenslets. Thus for a thin lens with a focal length f that is centered at the location (s_x, s_y) the ray transformation is given by

$$\vec{L}' = \mathcal{R} \cdot \vec{L} + \vec{S} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -1/f & 0 & 1 & 0 \\ 0 & -1/f & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ \theta \\ \phi \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ s_x/f \\ s_y/f \end{bmatrix}$$

which can be seen to be equivalent to a refracting lens followed by a prism such that the position and direction of the lightrays changes due to the transformation [16]. The center locations of the lenslets (s_x, s_y) are set to lie on a rectilinear grid with the spacing between the lenslets equal to the lenslet's pitch. In practice, the precise value of (s_x, s_y) must be calculated individually for each lightray. The total lightray propagation transformation is given by

$$\vec{L}' = \mathcal{T}_{LS} \cdot (\mathcal{R}_L \cdot \mathcal{T}_{ML} \cdot \mathcal{R}_M \cdot \mathcal{T}_{EM} \cdot \vec{L} + \vec{S}_L) \tag{4.1}$$

where $\mathcal{T}_{EM}$ is the propagation matrix exterior to the camera from the source point to the main lens, $\mathcal{R}_M$ is the refraction matrix for the main lens, $\mathcal{T}_{ML}$ is the propagation matrix from the main lens to the lenslet array, $\mathcal{R}_L$ is the refraction matrix of the lenslet array, $\vec{S}_L$ is a shifting vector due to the lenslets being off-axis, and $\mathcal{T}_{LS}$ is the propagation matrix from the lenslet array to the sensor. This operation is applied to every simulated lightray. Additionally, the shifting vector $\mathcal{T}_{LS}$ depends upon the particular lenslet that the lightray intersects and thus needs to be calculated after propagating the lightray to the lenslets.

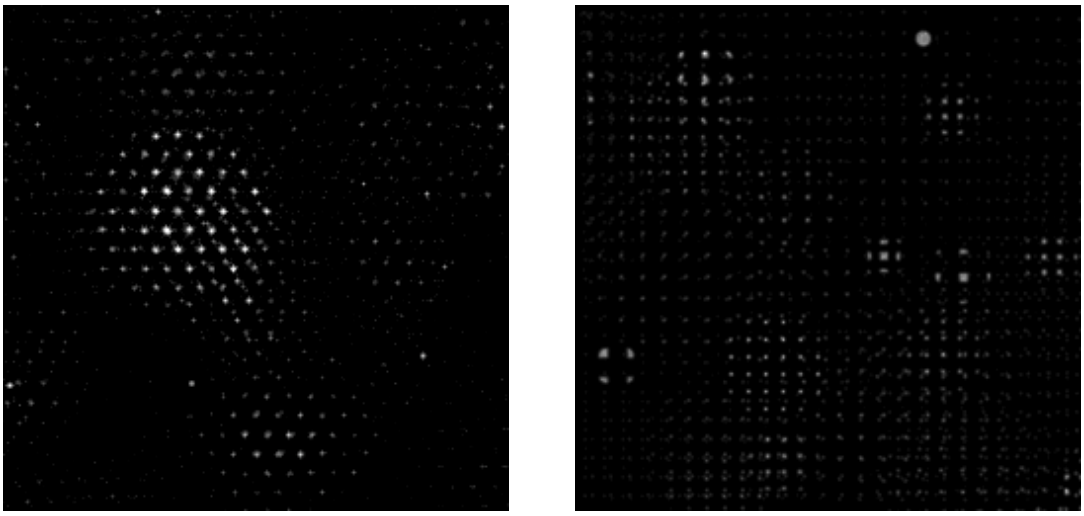
During the computational propagation of the lightrays, it was assumed that there was no attenuation of the lightray intensity due to absorption by either the free-space or the lenses. However, the rays that do not intersect the optical elements are assumed to be lost as though the interior of the camera was coated with 100% absorbing paint. Stops are not specifically

simulated within the camera, but the lenses act as stops by programming the simulation to remove all the lightrays that do not intersect the lenses. The intensity field produced on the simulated camera sensor is calculated by integrating all the lightrays that intersect the sensor.

The simulated sensor consists of a rectilinear grid of square pixels that return a weighted sum of the lightrays' irradiance. The lightrays intersect the simulated sensor at relatively small angles with respect to the optical axis, so a paraxial approximation is used to assume that the irradiance is independent of the incident angle. Since the lightrays typically intersect the pixels at non-integral positions, the intensity produced by each lightray must be interpolated. This is accomplished by assuming that the lightrays have square profiles with dimensions equal to the dimensions of a single pixel. Then the intensity of the lightray is split between the nearest four pixels to the point of intersection based upon the area of overlap. The total intensity of each pixel in the simulated camera sensor is equal to the sum of the intensities of all the interpolated lightrays that intersect the particular pixel. The final sensor image is saved both as a double precision array as well as a 16 bit unsigned integer array. To accurately model discretization effects upon the volumetric reconstruction, the 16 bit images were used in all subsequent processing. Figure 4.3 shows the qualitative agreement between a simulated plenoptic camera image and a real plenoptic camera image acquired from a commercial lightfield camera.

4.1.2 Volumetric Image Reconstruction

To perform the volumetric reconstructions from the plenoptic camera data, a four-dimensional lightfield function is extracted from the camera sensor and integrated over a specified domain. The lightfield function may then be used to produce images that are computationally



(a) Real sensor image (b) Simulated sensor image

Figure 4.3: (a) The sensor image from a photograph taken of a PIV particle field using a commercial lightfield camera. This camera uses a hexagonal array of lenslets, which is apparent in the particle images. (b) A simulated lightfield camera sensor image showing a simulated PIV particle field. The simulated cameras used a rectilinear lenslet array for computational ease.

refocused on different regions by varying the integration domain of the lightfield function. The matrix optics transformations described in the previous section are used to analytically derive the reconstruction algorithm.

Qualitatively, the plenoptic reconstruction algorithm is equivalent to estimating the image that would have been formed in the camera at a new position either in front of the lenslet array or behind the lenslet array. To computationally refocus the lightfield onto the new focal plane requires knowing both the spatial and angular information of the lightrays. This knowledge can be extracted from the camera sensor image produced by the lenslets. In plenoptic 2.0 cameras, the spatial information of the lightfield is encoded by the position of the lenslets, while the angular information of the lightfield is encoded within the images produced by each lenslet. This intuitively makes sense, since the main lens is focused onto the lenslet array; the sensor in a standard camera is found where the lenslet array is in a plenoptic

camera, so the lenslets are essentially acting as large pixel sensors. Additionally since the lenslets produce images of the main lens, the spatial variation within each lenslet image corresponds to the angular variation of the light entering the camera. For example, the light that falls on the pixels on the right side of a lenslet image may be predominantly from the left side of the main lens (the actual orientation will depend upon where the lightrays originated with respect to the focal plane of the main lens). In practice, computationally refocusing the lightfield involves extracting the images under each lenslet and then overlapping these images to combine them. The new focal distance of the refocused image is controlled by varying the amount of overlap of the individual lenslet images. In order to emulate the physical process that occurs in a standard camera, the lenslet images may be added together.

Single Camera Additive Reconstruction Algorithm

In order to add, or integrate, the images from the individual lenslets together, the lightfield function must be extracted from the sensor image in terms of the system coordinates. Once this is complete, the lightfield function is interpolated onto new coordinates and integrated to computationally refocus the camera.

Analytically, refocusing the lightfield may be represented by re-parameterizing the spatial and angular information from the sensor into (u, v, s, t) coordinates. In this parameterization the (u, v) coordinates correspond to the location that a lightray intersects the main lens, while the (s, t) coordinates give the location that the same lightray intersects the lenslet plane. All coordinates are measured with respect to the optical axis of the camera. Using this parameterization, the irradiance on the (s, t) plane, or equivalently the image produced by a standard camera, is given by integrating the lightfield across the (u, v) coordinates [40]

$$E(s,t) = \frac{1}{D^2} \iint L(u,v,s,t) \cdot A(u,v) \cdot \cos^4(\alpha)\, du\, dv$$

where D is the distance from camera aperture to the (s,t) plane, $A(u,v)$ is an aperture function returning one within the aperture and zero elsewhere, and α is the angle the lightray makes with the (s,t) plane. Since only the relative magnitude within the refocused volume is important for measuring velocities, the $1/D^2$ factor may be eliminated. Additionally, by assuming that the main lens acts as the camera aperture stop, the aperture function $A(u,v)$ may be assumed to be uniformly equal to one. While in general a camera may have multiple lenses as well as a separate aperture stop, the camera's optical system can be assumed to be equivalent to a single thin lens with its outer edge acting as the aperture stop without loss of generality. Finally, by assuming that the lightrays arrive on the (s,t) plane with a small angle to the optical axis, the paraxial approximation can be used to assume that $\cos^4(\alpha) \approx 1$. With these assumptions, the measured image irradiance equation on the (s,t) plane becomes

$$E(s,t) = \iint L(u,v,s,t)\, du\, dv. \tag{4.2}$$

However, to volumetrically reconstruct the image requires integrating the lightfield over a range of different domains, so a relationship between the (u,v,s,t) coordinates and the lab coordinates (x,y,z) must be calculated. This is accomplished by using the optical matrix transformations to propagate a lightray emanating from the coordinate (x,y,z) first to the main lens to find the relationship with the (u,v) coordinates and second, to the lenslet array to find the relationship with the (s,t) coordinates. A system of equations is then constructed from these relationships to transform the lab coordinates into the lightfield coordinates. Propagation of a lightray to the main lens is given by the transformation

$$\begin{bmatrix} u \\ v \\ \theta_{(u,v)} \\ \phi_{(u,v)} \end{bmatrix} = \begin{bmatrix} x + \theta z \\ y + \phi z \\ \theta \\ \phi \end{bmatrix} = \begin{bmatrix} 1 & 0 & z & 0 \\ 0 & 1 & 0 & z \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ \theta \\ \phi \end{bmatrix},$$

which gives the relationship between the lab coordinates and the (u, v) coordinates. Additionally the propagation of the lightray to the lenslet array is given by the operation

$$\begin{aligned} \begin{bmatrix} s \\ t \\ \theta_{(s,t)} \\ \phi_{(s,t)} \end{bmatrix} &= \begin{bmatrix} \left(1 - \frac{d_{ML}}{f_M}\right) x + \left(d_{ML} + \left(1 - \frac{d_{ML}}{f_M}\right) z\right) \theta \\ \left(1 - \frac{d_{ML}}{f_M}\right) y + \left(d_{ML} + \left(1 - \frac{d_{ML}}{f_M}\right) z\right) \phi \\ -\frac{x}{f_M} + \left(1 - \frac{z}{f_M}\right) \theta \\ -\frac{y}{f_M} + \left(1 - \frac{z}{f_M}\right) \phi \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & d_{ML} & 0 \\ 0 & 1 & 0 & d_{ML} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -1/f_M & 0 & 1 & 0 \\ 0 & -1/f_M & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & z & 0 \\ 0 & 1 & 0 & z \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ \theta \\ \phi \end{bmatrix}, \end{aligned}$$

which gives the relationship between the lab coordinates and the (s, t) coordinates. Then the system of equations relating the lab coordinates and the lightfield coordinates is

$$\begin{cases} u = x + \theta z \\ v = y + \phi z \\ s = \left(1 - \frac{d_{ML}}{f_M}\right) x + \left(d_{ML} + \left(1 - \frac{d_{ML}}{f_M}\right) z\right) \theta \\ t = \left(1 - \frac{d_{ML}}{f_M}\right) y + \left(d_{ML} + \left(1 - \frac{d_{ML}}{f_M}\right) z\right) \phi \end{cases},$$

which may then be solved for the original lab coordinates lightray vector to yield

$$\begin{cases} x = u \cdot \left(1 - \frac{z}{f_M}\right) + \frac{z \cdot (u-s)}{d_{ML}} \\ y = v \cdot \left(1 - \frac{z}{f_M}\right) + \frac{z \cdot (v-t)}{d_{ML}} \\ \theta = \frac{s \cdot f_M + u \cdot (d_{ML} - f_M)}{d_{ML} \cdot f_M} \\ \phi = \frac{t \cdot f_M + v \cdot (d_{ML} - f_M)}{d_{ML} \cdot f_M} \end{cases} \tag{4.3}$$

To perform the volumetric reconstructions, the relationship between the (u, v, s, t) coordinates of the lightfield and the pixel coordinates on the sensor must also be known. To calculate this transformation, the lab coordinates in equation (4.3) are substituted into the full camera lightray equation (4.1) to yield the sensor pixel coordinates

$$\begin{cases} x_S = \dfrac{(s-u) f_L + s\, d_{ML}}{d_{ML}} \\ y_S = \dfrac{(t-v) f_L + t\, d_{ML}}{d_{ML}} \end{cases} . \tag{4.4}$$

Then if the camera sensor image is denoted by the function $I_S(x, y)$, the coordinates in equation (4.4) may be used to extract the lightfield from the sensor according to the function $L(u, v, s, t) = I_S(x_S, y_S)$. Since u and v correspond to the coordinates at which the lightrays intersect the main lens, they must satisfy $p_M/2 \geq \sqrt{u^2 + v^2}$ where p_M is the pitch of the main lens. Additionally, since the lenslets are focused at infinity, the images they form on the camera sensor are of the main lens. Thus each pixel under an individual lenslet corresponds to one (u, v) coordinate. The s and t coordinates correspond to the locations of the centers of the individual lenslets. The lightfield function $L(u, v, s, t)$ is then calculated by varying the (u, v, s, t) coordinates over their respective domains to produce the sensor coordinates (x_S, y_S). Since these coordinates likely do not correspond to integral pixel values in the

sensor, cubic interpolation is used to calculate $L(u, v, s, t)$.

A calibration process is required to refocus an actual plenoptic camera. This process involves several steps. First, the location of the individual lenslet images on the sensor must be determined to calculate the (s, t) coordinates. This process is relatively straightforward since the images projected by the lenslets appear as periodic circles on a uniformly dark background and this format is commonly used in standard camera calibration procedures. The second step in performing the calibration process is to calculate estimates for the values of main lens focal length f_L and the main lens to lenslet array distance d_{ML}. Both of these values should be approximately known from the camera design; however, precise estimates may be determined by refocusing the plenoptic camera on a calibration grid. The parameters are then calculated by finding the distance from the plenoptic camera at which the calibration grid becomes focused. Additionally higher order calibration terms may be accounted for by using the positional information in the calibration grid.

While the camera produces a lightfield with a focused image on the (s, t) plane, by shifting this plane to a new position on the z-axis denoted by (s', t'), an image focused on a different depth may be produced. The image at this new plane may be calculated from equation (4.2). To computationally refocus the image on an arbitrary lab coordinate (x, y, z) requires finding a relationship to the computational focal plane (s', t'). Using the first two equations in the system of equations (4.3), the transformed (s', t') coordinates may be calculated as

$$\begin{cases} s' = u + d_{ML}\left(\frac{u-x}{z} - \frac{u}{f_M}\right) \\ t' = v + d_{ML}\left(\frac{v-y}{z} - \frac{v}{f_M}\right) \end{cases}.$$

The (u, v) plane that is coincident with the main lens may also be transformed to a new location along the optical axis. However, this increases the complexity of the volumetric

reconstruction algorithm while not conferring a particular advantage, so the transformed coordinates are set equal to the original coordinates $(u', v') = (u, v)$.

Once the transformed coordinates (u', v', s', t') are calculated, equation (4.2) may be used to computationally refocus the image to the lab coordinate (x, y, z) by the function

$$V(x, y, z) = \iint L\left(u, v, u + d_{ML}\left(\frac{u - x}{z} - \frac{u}{f_M}\right), v + d_{ML}\left(\frac{v - y}{z} - \frac{v}{f_M}\right)\right) du\, dv. \quad (4.5)$$

Evaluating the integral in (4.5) requires interpolating the lightfield from the discretely sampled function $L(u, v, s, t)$. However, it was found that the results produced by using a simple discrete sum were nearly identical to the results produced by using numerical quadrature, so the lightfield function only needs to be interpolated at a limited number of points for each evaluation.

Using the additive refocusing algorithm given by Equation (4.5) is suitable for performing computational refocusing that closely resembles the focusing process in a standard camera; however, the quality of the volumetric reconstructions can be improved by taking into account the unique nature of fluid measurement images. PIV imaging data is generally sparse with the large majority of the imaged volume consisting of empty space while the remaining space is filled with small, high intensity particles. Additionally the primary concern in three-dimensional PIV processing is to correctly measure the position of the particles within the volume. Assuming that there are no obstructions, this measurement can be made using just two images from different angles. However since higher seeding densities yield higher fidelity velocity measurements, often there is a sufficiently large number of particles present to cause obstructions from at least one viewing angle. Thus views from several different angles are typically needed to finely resolve the particle positions within the imaged volume, but generally only a small number of viewing angles is necessary.

For tomographic PIV using the MLOS and MART reconstruction algorithms, only four viewing angles are typically needed to fully resolve the particle field, while the lightfield camera commonly images particles from over ten different angles. For this reason, we employed the MLOS reconstruction algorithm to computationally refocus the lightfield cameras.

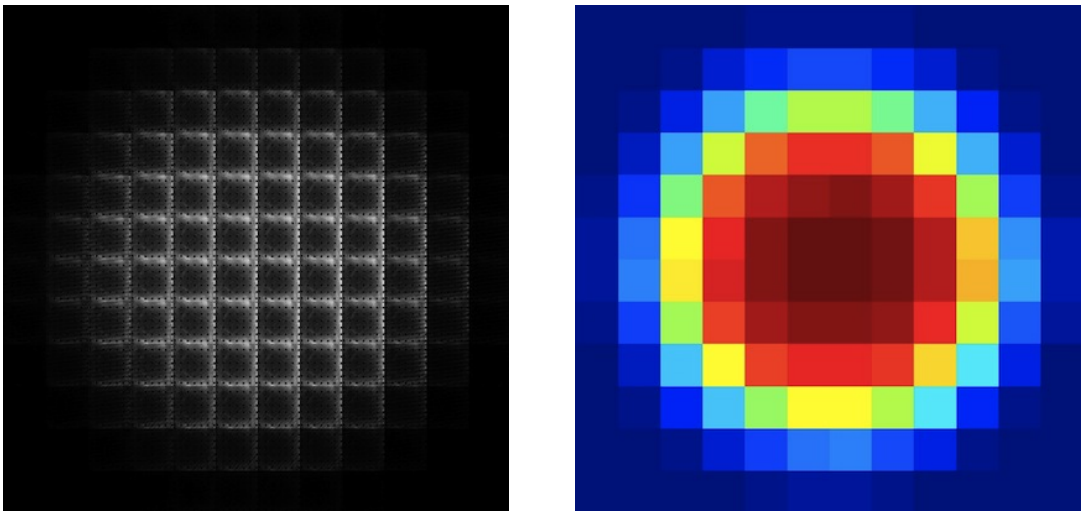
Single Camera Multiplicative Reconstruction Algorithm

Directly implementing the MLOS reconstruction technique by multiplying the different lenslet images together is not possible due to the particular manner in which data is produced by lightfield cameras. Each of the lenslet images has a roughly uniform intensity in the center of the image, but strong vignetting occurs near the edge of the images. However, the pixels near the edge of the lenslet images still contain useful information for computationally refocusing the camera. This information may be retained during the MLOS reconstruction process by calculating a renormalization factor.

To illustrate the lenslet image vignetting issue, the same pixel under every lenslet image can be extracted and combined together to form what is known as a sub-aperture image. These images can be extracted from the lightfield $L(u, v, s, t)$ by evaluating the function over the domain of (s, t) while holding (u, v) constant [40]. A collection of sub-aperture images is shown in Figure 4.4a.

In this figure it can be seen that the vignetting causes some of the sub-aperture images to have nearly uniformly zero intensities. Thus a simple multiplication of these images together would yield a uniformly zero volumetric reconstruction. To overcome the effect of the vignetting, a weighting function is introduced.

To overcome the effect of the vignetting, a weighting function is introduced. To calculate the weighting function, an image of a uniformly bright background is taken with the plenoptic



(a) Sub-aperture images (b) Weighting function

Figure 4.4: (a) This image shows a collection of sub-aperture images extracted from the lightfield $L(u, v, s, t)$. Each sub-aperture image corresponds to a single pixel under each lenslet. The camera was focused on a calibration grid for this lightfield. The vignetting on the edge of the lenslet images is clearly visible in the radial intensity decrease of the sub-aperture images. (b) This image shows the corresponding weighting function calculated for the lightfield.

camera and the lightfield $L(u, v, s, t)$ is extracted. The weighting function is set equal to the sum of the intensities of each sub-aperture image $w(u, v) = \sum_{s,t} L(u, v, s, t)$. Then the weighting function is linearly scaled to lie in the range $0 \leq w(u, v) \leq 1$. This results in the weighting function being approximately equal to one for the sub-aperture images taken from the center of the lenslet images and approximately zero for the sub-aperture images taken from the edge of the lenslet images. Figure 4.4b shows the weighting function calculated from the sub-aperture image in Figure 4.4a. The weighting function only needs to be calculated once for a particular plenoptic camera configuration since it is independent of the scene being imaged by the camera.

The MLOS plenoptic camera reconstruction is then performed by raising each sub-aperture image to the power of the weighting function. This scaling results in the low intensity sub-aperture images being nearly uniformly equal to ones and as a result, contributing relatively

little to the total reconstruction. The specific formula used to calculate the MLOS volumetric reconstruction is

$$V(x,y,z) = \left(\prod_{u_{min}}^{u_{max}} \prod_{v_{min}}^{v_{max}} \left(L(u,v,u + d_{ML}\left(\frac{u-x}{z} - \frac{u}{f_M}\right), v + d_{ML}\left(\frac{v-y}{z} - \frac{v}{f_M}\right)\right)^{w(u,v)} \right)^{\gamma_r} \tag{4.6}$$

where γ_r is used to rescale the reconstruction to ensure that the intensity histogram of the reconstruction approximately equals the histogram of the individual sub-aperture images. The value of the exponent γ_r will vary with the lightfield camera configuration and focal distance, but typically γ_r will be approximately equal to the reciprocal of the number of pixels under each lenslet.

The MLOS reconstruction algorithm has advantages over the previously used additive reconstruction algorithms for PIV data, however due to the limited angular resolution of a single lightfield camera, we hypothesized that combining the data from multiple lightfield cameras would produce higher fidelity reconstructions. Thus, we also developed a multiple lightfield camera reconstruction algorithm.

4.1.3 Multiple Camera Reconstruction Algorithm

Since the computational refocusing of a single lightfield camera directly yields a three-dimensional intensity field, a basic process for combining the data from multiple cameras is a simple extension of the MLOS algorithm to three-dimensional data. It is possible that other algorithms may exist that could yield better reconstructions, but for the purposes of demonstrating the potential of the method, focusing on the MLOS algorithm is sufficient.

In a similar manner to tomographic reconstruction algorithms, the first step in the multiple camera reconstruction is to perform an intensity normalization of the individual camera

reconstructions so that the intensity histograms of the individual reconstructions are approximately equal. This ensures that each camera contributes equally to the composite reconstruction and no information is lost. Then by denoting each plenoptic camera's reconstruction as $V_i(x, y, z)$, the composite reconstruction is given by

$$V_c(x, y, z) = \prod_{i=1}^{M} V_i(x, y, z)^{\gamma_c}$$

where M is the number of lightfield cameras and γ_c is a rescaling exponent. Since in equation 4.6 the single camera reconstruction is raised to the exponent γ_r, the two exponents are dependent upon one-another. Thus only one exponent needs to be controlled during the reconstruction process. For this reason, the multiple camera reconstruction exponent is set equal to the reciprocal of the camera number $\gamma_c = 1/M$ as is typically done in MLOS reconstructions [2].

Once the lightfield camera data is used to produce a volumetric reconstruction, the quality of these reconstructions must be measured and compared to the known true solution. This allows us to directly compare the validity of the different volumetric reconstruction techniques.

4.1.4 Reconstruction Fidelity Metrics

A variety of different quality measurements are used to evaluate the accuracy of the lightfield reconstructions and to compare them to standard tomographic reconstructions. The fidelity of the reconstructions was measured using the following metrics: the zero mean reconstruction quality factor, the error in the reconstructed particle positions, the RMS error of the measured velocity field with respect to the true velocity field, and the percentage of outlier vectors as defined by the Universal Outlier Detector (UOD) [63].

The zero mean reconstruction quality factor is a modified form of the reconstruction quality factor defined by Elsinga, et al [2, 8, 10]. The quality factor as defined by Elsinga is

$$Q = \frac{\sum_{x,y,z} V(x,y,z) \cdot T(x,y,z)}{\sqrt{\sum_{x,y,z} V(x,y,z)^2 \cdot \sum_{x,y,z} T(x,y,z)^2}}$$

where the summation is taken over all coordinates in the reconstructed field. This metric is essentially a normalized cross-correlation between the reconstructed intensity field $V(x,y,z)$ and the true intensity field $T(x,y,z)$. This function will produce a value of Q approximately equal to one for nearly perfect reconstructions, while poorer quality reconstructions will yield values approaching zero. However, it was found that this quality factor produced artificially high values for high seeding densities due to the fact that as the seeding density increased, the functions $V(x,y,z)$ and $T(x,y,z)$ approached nearly uniform values of one. Thus regardless of the difference between the true field and the reconstructed field, the correlation would yield a value nearly equal to one; hence, this metric cannot properly characterize high seeding density reconstructions.

To overcome this effect a zero mean reconstruction quality is used in analyzing the lightfield reconstruction results. This quality factor is defined as

$$Q^* = \frac{\sum_{x,y,z} \tilde{V}(x,y,z) \cdot \tilde{T}(x,y,z)}{\sqrt{\sum_{x,y,z} \tilde{V}(x,y,z)^2 \cdot \sum_{x,y,z} \tilde{T}(x,y,z)^2}}$$

where $\tilde{V}(x,y,z)$ and $\tilde{T}(x,y,z)$ are given by zero-mean reconstruction field and zero-mean true intensity field respectively. The effect this has on the reconstruction quality is shown in Figure 4.5 where the standard reconstruction quality Q and the zero-mean reconstruction quality Q^* are plotted as functions of particle density in particles per image pixels (ppp) for standard tomographic reconstructions. Below a particle density of approximately ppp = 0.1

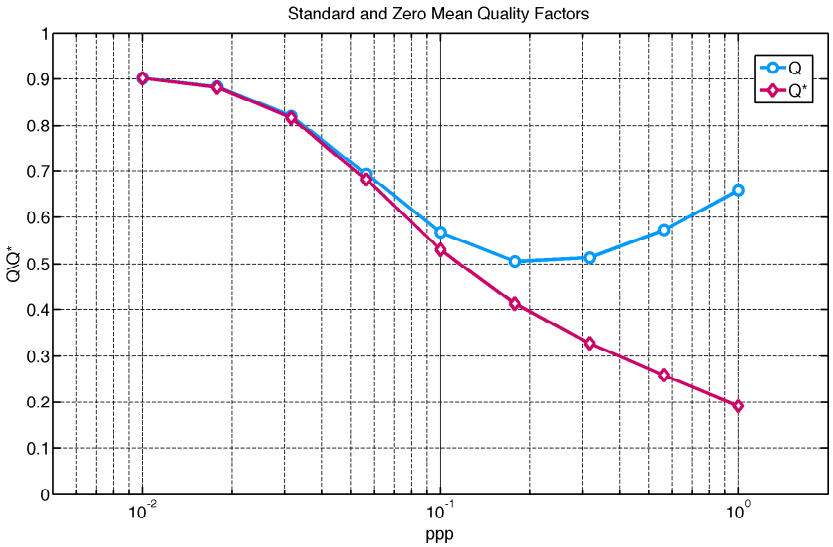


Figure 4.5: A comparison between the standard tomographic reconstruction quality factor Q and the zero mean reconstruction quality factor Q^* showing the artificially high values produced by Q at very high particle seeding densities.

the two reconstruction quality factors differ by less than 10%. For higher particle densities, the zero mean quality factor Q^* continues to decrease while the standard quality factor increases beyond a particle density of approximately ppp = 0.2. Intuitively, higher particle densities will make volumetric reconstructions more difficult due to the rapidly increasing number of particles overlapping in the camera images, thus the zero-mean reconstruction quality factor produces more reasonable values at high densities. In all following analysis the zero-mean reconstruction quality factor Q^* will be used.

The error in the reconstructed particle positions is measured by fitting a Gaussian intensity profile to all reconstructed particles and then comparing these fits to the known true particle positions. Typically ghost particles that have no corresponding true particles will be created during the reconstruction process, so the nearest true particle to each reconstructed particle is identified after performing the Gaussian fit. Then the particle position error is measured for the nearest reconstructed particle to the true particle. This process is repeated for the

set of all true particles to produce a distribution of particle position errors. Typically the position errors followed a normal, zero-mean distribution as can be seen in Figure 4.6 which shows a scatter plot of the reconstructed particle position errors for the x and z axes.

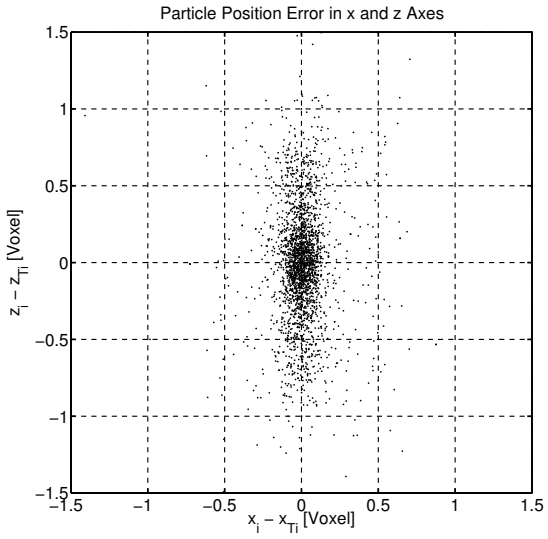


Figure 4.6: A scatter plot showing the reconstructed particle position error distribution for the x and z axes. The reconstruction was performed with 2,500 particles and two simulated lightfield cameras placed 25° off the volume axis.

The uncertainty in the reconstructed particle positions is used as the primary position-based metric to assess the quality of the lightfield reconstructions. The uncertainty is reported as the distance in voxels from the center of the reconstructed particles in which there is a 95% chance that the true particle is located assuming a normal distribution of errors. The z-axis uncertainty is specifically reported since this tends to be larger than either the x or y axis uncertainty and sets an upper bound in the particle position error.

The simulated particle positions are advected using an analytical solution for a vortex ring described in [68]. The simulated vortex ring translates through the particle volume with a constant velocity without experiencing dissipation or changing shape. Three-dimensional PIV measurements are made on the reconstructed particle volume to return a measured

velocity field that can be compared to the analytical solution of the vortex ring to produce an RMS velocity field error defined by

$$\epsilon_{RMS} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left((u_i - u_{T_i})^2 + (v_i - v_{T_i})^2 + (w_i - w_{T_i})^2\right)}$$

where (u_i, v_i, w_i) are the velocity vectors measured using the PIV correlations. Two passes are completed using the Robust Phase Correlation algorithm [7]; during the first pass, the UOD is used to locate likely outlier vectors. These vectors are then subsequently replaced before the second PIV pass is processed. The final reconstruction quality metric is given by the ratio of vectors flagged by the UOD in the second pass using the UOD residual threshold $r_0^* > 2$.

4.1.5 Simulation Parameters

Due to the complex design of plenoptic cameras and the nature of the reconstruction algorithms, the parameter space that can be studied to analyze the use of plenoptic cameras for fluid measurements is very large. For this reason, the simulations are designed to approximate the design of the commercial Lytro plenoptic camera design. The results of the simulations will then be used to design a complementary series of experiments with a camera designed using the same parameters. Table 4.1 lists the parameters used in the lightfield simulations. The commercial plenoptic cameras use zoom lenses that cover a large range of angles of view, so a fixed focal length was chosen for the simulations that has an angle of view roughly equivalent to a 70 mm lens on an SLR camera.

The Lytro cameras use a hexagonal lenslet array, however a rectangular lenslet array was used in the simulations to simplify the algorithm. All remaining parameters were chosen to

Simulation Parameters			
	Main Lens	Lenslets	Sensor
Focal Length	$f_M = 10\ mm$	$f_L = 28\ \mu m$	—
Pitch	$p_M = 5\ mm$	$p_L = 14\ \mu m$	$p_S = 1.4\ \mu m$
Aperture	$f_M/\# = 2$	$f_L/\# = 2$	—
Number	—	$n_{x_L} \times n_{y_L} = 328 \times 328$	$n_{x_S} \times n_{y_S} = 3280 \times 3280$

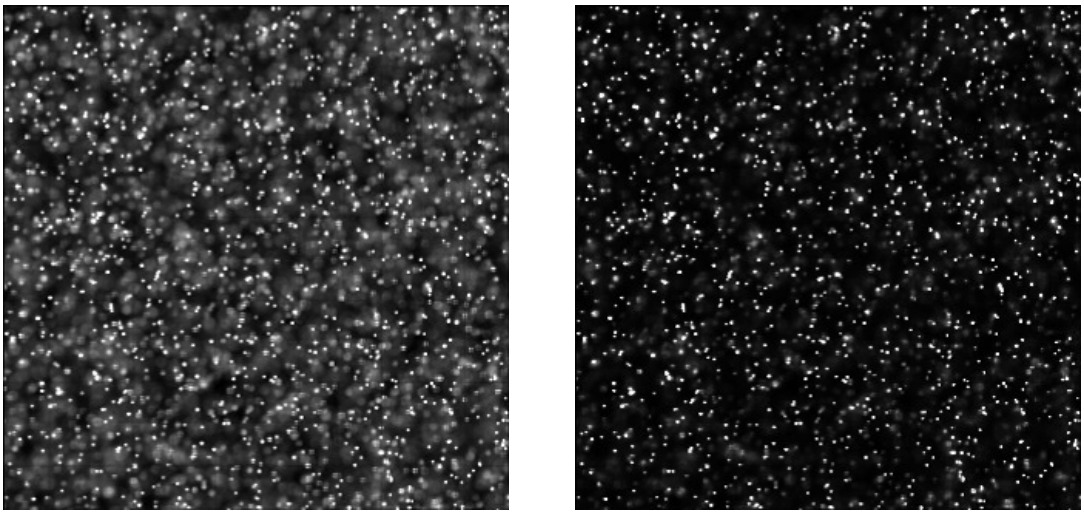
Table 4.1: A table listing the plenoptic camera parameters used in the lightfield simulations. These parameters were held constant for all completed simulations.

closely match those found in the Lytro cameras. These differences to the physical cameras are minor and the results will still apply to the actual cameras.

4.2 Results

Before investigating the reconstruction fidelity of the lightfield cameras over a large parameter range, the reconstruction algorithm was specifically studied. To validate the multiplicative reconstruction algorithm given by Equation 4.6, images from both simulated and experimental lightfields were computationally refocused. The experimental data were collected using a single Lytro camera viewing a field of fluorescent particles illuminated with a 1 cm thick 532 nm laser sheet.

The data were computationally refocused using both the additive refocusing algorithm and the multiplicative refocusing algorithm, as is shown by the volume slices in Figures 4.7a and 4.7b. The background noise in the multiplicatively reconstructed field appears much lower than the noise in the additively reconstructed field. The simulations comparing the additive and multiplicative refocusing algorithms demonstrated clearly improved performance for the multiplicative method.



(a) Additive refocusing (b) Multiplicative refocusing

Figure 4.7: Computationally refocused images taken using the Lytro lightfield camera during a PIV experiment. (a) The PIV field refocused using the standard additive refocusing algorithm described in Equation (4.5). There is relatively high-magnitude background noise in the image due to out-of-focus particles. (b) The same PIV field refocused using the multiplicative refocusing algorithm described in Equation (4.6). The background noise level in this image is much lower than the noise level produced by the additive reconstruction.

A range of different parameters are studied in the lightfield camera simulations including varying (1) the distance of the cameras from the particle volume, (2) the thickness of the particle volume, (3) the number of simulated cameras, (4) the angle of the simulated cameras to the particle volume, and (5) the density of the particles inside the volume. The simulations varying the camera distance and the particle volume thickness were completed to test the hypothesis that an optimal distance exists for using the lightfield cameras to collect PIV measurements. Additionally the simulations varying the number and angles of the simulated cameras were completed to test the theory that using two cameras should produce very high quality reconstructions, while using more than two lightfield cameras provides only a small additional benefit. For the sake of comparison to standard tomographic reconstruction techniques, two, three, and four standard cameras were also simulated in these tests. These cameras were configured to have the same magnification as the lightfield

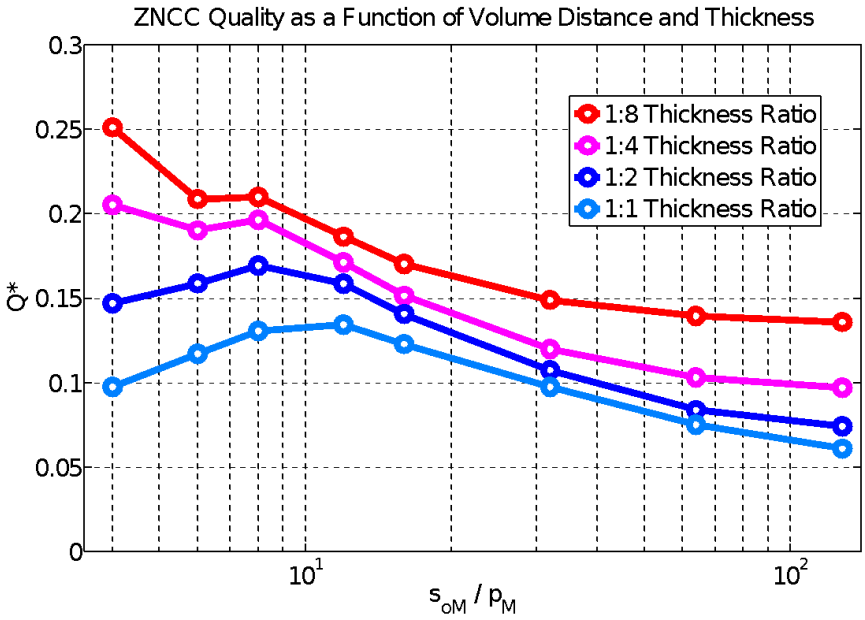


Figure 4.8: A graph showing the zero-mean normalized cross correlation quality factor as a function of the dimensionless camera focal distance for a range of different volume width to thickness ratios. The camera focal distance s_{oM} is normalized by the aperture of the main lens p_M. The quality factor has a peak at s_{oM}/p_M due to the camera's angular resolution decreasing with distance while the depth-of-field increases.

cameras and imaged the same particle volume from the same point in space as the light-field cameras. The reconstructions from the standard cameras were calculated using the MART tomographic reconstruction algorithm [2, 8, 9, 10]. Additional tests also compared the different reconstruction techniques for a range of particle densities.

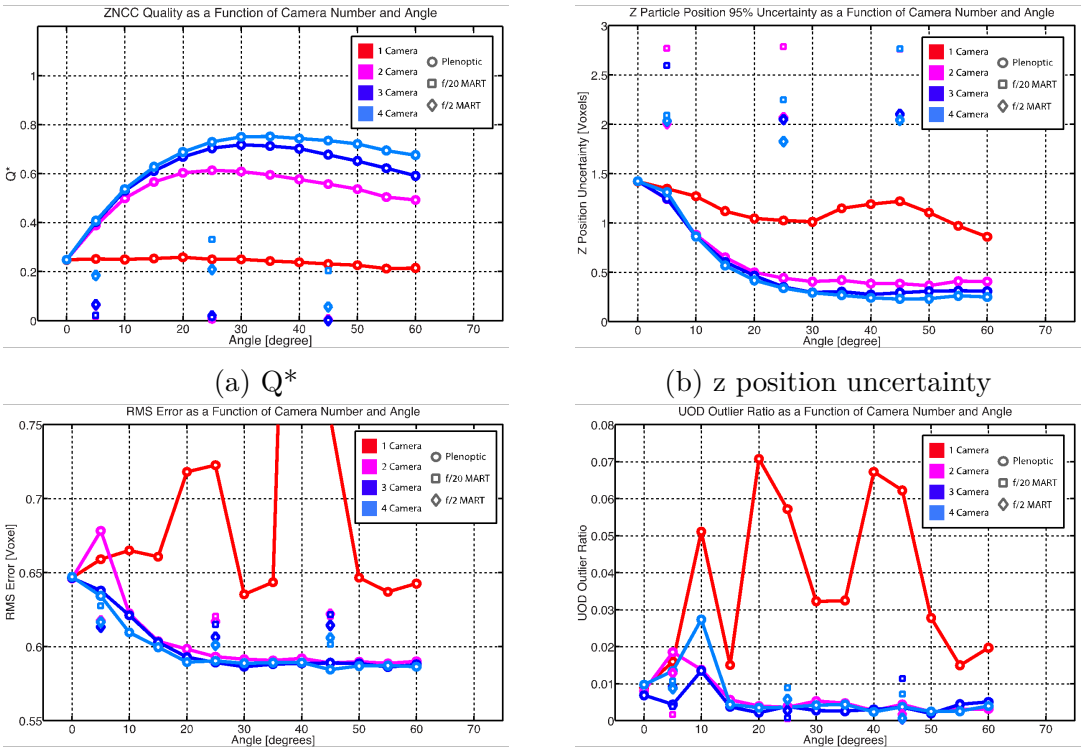
Since a lightfield camera's depth-of-field increases with the focal distance of the camera, but the angular resolution of the camera decreases, it was speculated that there might be an optimal distance to perform volumetric measurements with the lightfield cameras. To test this hypothesis, a series of simulations were performed for a range of different camera focal distances and particle volume thicknesses. Figure 4.8 shows the results of these simulations. In these tests the zero-mean normalized cross correlation quality factor is measured for reconstructions produced by a single lightfield camera located on the particle volume axis.

The camera focal distance s_{oM} is normalized by the aperture of the main lens p_M. so that the results may be scaled to arbitrarily sized primary lenses. Additionally the particle volume was scaled to fill the entire field-of-view of the camera at each different focal distance.

Due to the resolution trade-off, the quality factor Q^* has a local maximum near $8 \leq s_{oM}/p_M \leq 10$. The quality factor also decreases as the particle volume increases due to the particles near the edge of the volume becoming more out-of-focus. For all subsequent lightfield camera simulations a volume thickness-to-width ratio of 1:4 is used as this is a common ratio in tomographic PIV experiments; additionally, a dimensionless focal distance of $s_{oM}/p_M = 8$ is also used since the quality factor experiences a peak at this relative distance.

Once an optimal distance was determined for completing volumetric measurements with the lightfield camera, the number of cameras and their configurations wer investigated. Between one and four lightfield and standard cameras are simulated in these tests. The cameras are located equal distances from one another with angles measured from the z-axis of the particle volume. Using this system, two cameras are located along a line on the x-axis, three cameras are located at the vertices of an equilateral triangle on the xy-plane, and four cameras are located at the vertices of a square on the xy-plane. The angles of the lightfield cameras were varied from 0 to 60 degrees in five degree increments. The standard cameras were simulated at 5 degrees, 25 degrees, and 45 degrees. The standard camera images are used to perform tomographic reconstructions using the MART algorithm.

The results of these simulations are shown in Figure 4.9. The single plenoptic cameras perform poorly using all four quality metrics. However, reconstruction fidelity dramatically increased by adding a single additional plenoptic camera. The z particle position uncertainty in particular is between two and three times as high for a single camera as for the cases using multiple cameras. Additionally, from this data it is apparent that using more than two lightfield cameras to perform volumetric reconstructions only marginally increases the



(a) Q*

(b) z position uncertainty

(c) RMS velocity error

(d) UOD outlier ratio

Figure 4.9: (a) The quality factor as a function of the camera configuration. (b) The z particle position uncertainty as a function of the camera configuration. Two or more cameras dramatically increase the z resolution. (c) The RMS velocity error as a function of the camera configuration. Two or more cameras perform equally well in reconstructing the velocity field. (d) The UOD outlier ratio as a function of the camera configuration.

fidelity of the reconstructions. This data also shows that the fidelity of the reconstructions created using multiple plenoptic cameras has a maximum value around 25 to 35 degrees, agreeing well with the results found in tomographic PIV simulations [2, 8].

The tomographic MART reconstruction qualities created using the simulated standard camera are also shown in Figure 4.9. For these tests, two different apertures on the standard cameras were simulated: an aperture of $f/2$, which is the same aperture used on the plenoptic cameras, and an aperture of $f/20$, which is the aperture necessary to have an equivalent depth-of-field as the plenoptic cameras. The reconstruction quality of the images suffers from a low depth-of-field for the large aperture case. The depth-of-field is large for the small aper-

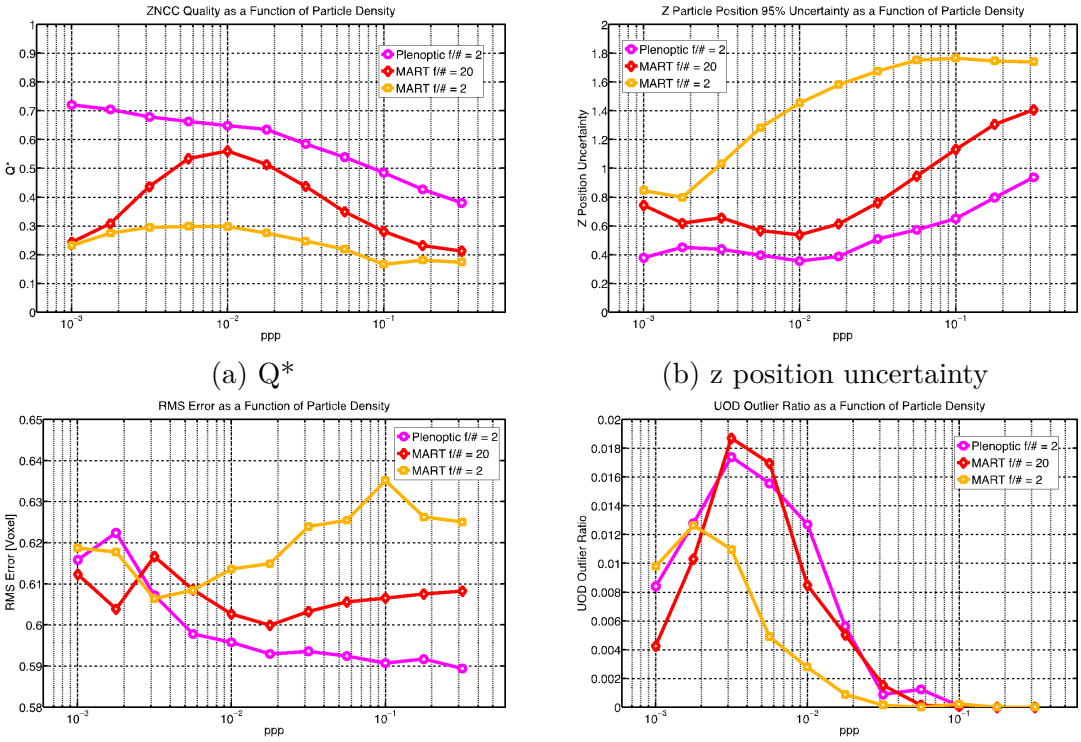
ture case, but diffraction effects start causing uniform blurring of the particles. The quality of the MART reconstructions is generally poorer than the plenoptic camera reconstructions that used two or more cameras.

The low quality of the MART reconstructions, in comparison to the plenoptic reconstructions, can be explained by several factors. First, the standard cameras with $f/2$ apertures have a very low depth-of-field which will result in a high level of noise in the reconstructions. Second, the equivalent depth-of-field $f/20$ aperture cameras will have significant diffraction effects (which will not be apparent in the lightfield camera reconstructions).

The final series of simulations investigated the effects of particle density on the lightfield camera reconstruction quality. In these tests, the particle density of the simulated volume was varied over a range typically used in volumetric PIV experiments. The particle density ppp is generally measured as the ratio of the number of particles imaged to the total number of pixels covered by the particle volume in the camera image. However, the number of used pixels is poorly defined for a plenoptic camera since the images produced by the lenslet array typically do not cover 100% of the sensor. Additionally for Plenoptic 1.0 cameras, the final image produced by the camera is based upon the number of lenslets rather than the number of pixels on the sensor. So to provide a fair comparison to particle densities reported in other volumetric PIV papers, the particle density ppp is measured as the ratio of the number of particles to the number of reconstructed pixels.

The particle density tests were performed using two plenoptic cameras and two standard cameras positioned 25 degrees off the particle volume axis. The reconstruction qualities are measured for particle densities over the domain $1 \cdot 10^{-3} \leq ppp \leq 3 \cdot 10^{1}$. The standard cameras used to create the MART reconstructions have simulated apertures of $f/2$ and $f/20$.

The particle density simulations showed that the reconstruction fidelities generally decreased



(a) Q*

(b) z position uncertainty

(c) RMS velocity error

(d) UOD outlier ratio

Figure 4.10: Graphs showing several different reconstruction fidelity metrics for a range of particle densities. Tomographic MART reconstructions using both $f/2$ and $f/20$ apertures of the same particle field are shown for comparison. (a) A graph showing the quality factor as a function of the particle density. (b) A graph showing the particle position uncertainty as a function of the particle density. (c) The RMS velocity error as a function of the particle density. (d) The UOD outlier ratio as a function of the particle density.

as the particle density increased. However, the PIV velocity metrics showed an increase in quality with the particle density as is shown in Figure 4.10. The decrease in the measured velocity field error with increasing particle density is likely due to the relatively simple nature of the prescribed velocity field. While the vortex ring does produce three-dimensional motion relative to the simulated cameras, the motion has sufficiently large scales when compared to the total particle volume, so it is likely that the ghost particles produced in the reconstructions only contribute relatively little noise to the PIV correlations. At the same time, increasing the particle density results in the true particles increasing the signal in the

correlations. It is thus possible that increasing the particle density will only decrease the measured velocity field noise in cases where the flow is relatively simple, but determining these cases may be difficult. This effect needs to be investigated in further studies.

4.3 Conclusions

This work presents advancements to the field of volumetric PIV developing novel tools based on lightfield (plenoptic) imaging. First, by using ray optics, we developed a volumetric reconstruction algorithm for refocusing single plenoptic cameras. Then we show that a multiplicative refocusing algorithm may be used to improve the quality of the refocused images due to the unique properties of PIV data. We then modify this multiplicative algorithm so that data from several plenoptic cameras may be combined together. This reconstruction algorithm is next used to calculate volumetric reconstructions from artificial sensor data created using lightray simulations of an illuminated particle field. These reconstructions are analyzed in terms of several metrics measuring both the reconstruction fidelity to the true particle intensity field and the accuracy with which the particle motion is measured.

From the results of the simulations, we show that multiple camera plenoptic reconstructions can yield higher quality data than traditional volumetric measurement techniques under some circumstances. Single plenoptic camera reconstructions are shown to yield relatively low quality data, but we show that high quality volumetric data may be taken by simultaneously using two plenoptic cameras. The simulations show that there is only a small benefit from using more than two cameras, however. Additionally, we show that under the optimal conditions for plenoptic cameras, standard tomographic cameras perform relatively poorly.

Future work on the plenoptic camera PIV system must focus on several areas. First, the parameter range of the simulations will be expanded with the end goal of designing plenoptic

cameras that are optimized in terms of performing volumetric particle measurements. Additionally, by using the information from these simulations, a complementary set of experiments will be performed to validate the simulation results. Finally, more advanced reconstruction algorithms will be developed and validated using the simulation tools that were developed for this work.

Chapter 5

Conclusions

5.1 Snake Model Experimental Data

Due to the low effective resolution of the snake model experiment, the primary conclusions must be drawn from the process of performing the experiment and processing the data. In particular, the results of this experiment highlighted the importance of collecting high signal to noise ratio data for tomographic PIV.

Ensuring that the particle illumination is sufficiently high to ensure high contrast images is crucial to successful tomographic PIV. Additionally, since it is intuitively more difficult to interpret the quality of tomographic data by simply visually examining the collected image data, calculating tomographic reconstructions of the data during the experiment is also vital to ensuring that the data is high quality.

The experimental snake model data also highlighted the ability of extracting high quality velocity vector data using more advanced tomographic PIV techniques. Processing the data using a variety of techniques showed that basic PIV algorithms would have been incapable of yielding nearly any useful data from the highly noisy particle images. So using techniques like the pyramid correlations and the novel processing algorithms that were described are necessary for noisy data and may dramatically improve the quality of the output velocity field.

5.2 Software Development

High speed tomographic PIV processing software was developed along with software to generate simulated tomographic PIV datasets. In Section 3.4 simulations were ran to validate the tomographic PIV software. These tests focused on ensuring that the basic algorithms behind the software produced reasonable results that were in agreement with previously published work.

In all simulations, the tomographic PIV software closely replicated the results from previous studies, showing that the software performs correctly. This ensured that using the software to process the snake model tomographic dataset would produce accurate results (within the limitations of the collected data). This also demonstrated that the software functions sufficiently well to be released to the general community.

5.3 Plenoptic PIV

The results of the plenoptic PIV simulations were compared with the results of tomographic PIV ran on the same data cases. These results showed that using multiple plenoptic cameras can yield better reconstruction data than a similar number of standard cameras processing the data with tomographic reconstruction algorithms. However, due to the additional complexity of plenoptic cameras, performing a calibration to match multiple cameras together is experimentally difficult. Therefore, unless more advanced calibration techniques are developed, experimental work using plenoptic cameras will continue to focus on single camera systems. This additional complexity suggests that standard cameras will likely continue to be used for three-dimensional fluid flow measurement experiments.

Bibliography

[1] C. H. Atkinson and J. Soria. Algebraic reconstruction techniques for tomographic particle image velocimetry. In *Proceedings of the 16th Australasian Fluid Mechanics Conference, 16AFMC*, pages 191–198, 2007. ISBN 9781864998948.

[2] Callum Atkinson and Julio Soria. An efficient simultaneous reconstruction technique for tomographic particle image velocimetry. *Experiments in Fluids*, 47:553–568, 2009.

[3] Callum H Atkinson, Craig J Dillon-Gibbons, Sophie Herpin, and Julio Soria. Reconstruction Techniques for Tomographic PIV (Tomo-PIV) of a Turbulent Boundary Layer. In *14th Int Symp on Applications of Laser Techniques to Fluid Mechanics*, Jul 2008.

[4] Stephen T Barnard and Martin A Fischler. Computational Stereo. Technical Report 261, SRI International, mar 1982.

[5] Jesse Belden, Tadd T Truscott, Michael C Axiak, and Alexandra H Techet. Three-dimensional synthetic aperture particle image velocimetry. *Measurement Science and Technology*, 21, nov 2010.

[6] Antonio Cenedese, Claudia Cenedese, Francesco Furia, Marco Marchetti, Monica Moroni, and Luca Shindler. 3D particle reconstruction using light field imaging. In *International Symposium on Applications of Laser Techniques to Fluid Mechanics*, jul 2012.

[7] Adric Eckstein and Pavlos P Vlachos. Digital particle image velocimetry (DPIV) robust phase correlation. *Measurement Science and Technology*, 2009.

[8] G E Elsinga, F Scarano, B Wieneke, and B W van Oudheusden. Tomographic particle image velocimetry. *Experiments in Fluids*, 41:933–947, 2006.

[9] G E Elsinga, B W van Oudheusden, and F Scarano. Experimental assessment of Tomographic-PIV accuracy. In *13th Int Symp on Applications of Laser Techniques to Fluid Mechanics*, 2006.

[10] G. E. Elsinga, B. Wieneke, F Scarano, and A. Schröder. Tomographic 3D-PIV and Applications. *Topics in Applied Physics*, 112:103–125, 2008.

[11] G. E. Elsinga, J. Westerweel, F Scarano, and M. Novara. On the velocity of ghost particles and the bias errors in Tomographic-PIV. *Experiments in Fluids*, 2010.

[12] Timothy W Fahringer and Brian S Thurow. Tomographic Reconstruction of a 3-D Flow Field Using a Plenoptic Camera. In *AIAA Fluid Dynamics Conference and Exhibit*, jun 2012.

[13] FFTW, Jul 2022. URL https://www.fftw.org/.

[14] Pascal Fua. A parallel stereo algorithm that produces dense depth maps and preserves image features. *Machine Vision and Applications*, 6:35–49, 1993.

[15] J. C. H. Fung, J. C. R. Hunt, N. A. Malik, and R. J. Perkins. Kinematic simulation of homogeneous turbulence by unsteady random Fourier modes. *Journal of Fluid Mechanics*, pages 281–318, Aug 1992.

[16] Todor Georgeiv and Chintan Intwala. Light Field Camera Design for Integral View Photography. Technical report, Adobe Systems Incorporated, 2003.

[17] Todor Georgeiv, Ke Colin Zheng, Brian Curless, David Salesin, Shree Nayar, and Chintan Intwala. Spatio-Angular Resolution Tradeoff in Integral Photography. In *Eurographics Association*, 2006.

[18] A Gershun. The light field. *Mathematical Physics*, 18, 1939.

[19] Github, Aug 2022. URL https://github.com/.

[20] The HDF Group, Jun 2022. URL https://www.hdfgroup.org/.

[21] Richard Hartley and Andrew Zisserman. *Multipe View Geometry in Computer Vision*. Cambridge University Press, 2000.

[22] E. Hecht. *Optics*. Pearson, 2002.

[23] Gabor T Herman and Arnold Lent. Eij2. (3). *Computers in biology and medicine*, 6(3): 273–294, 1976.

[24] K D Hinsch. Holographic particle image velocimetry. *Measurement Science and Technology*, 13:R61–R72, 2002.

[25] Daniel Holden, John J Socha, Nicholas D Cardwell, and Pavlos P Vlachos. Aerodynamics of the flying snake Chrysopelea paradisi: how a bluff body cross-sectional shape contributes to gliding performance. *The Journal of Experimental Biology*, 217:382–394, 2014.

[26] L. Kajitani and D. Dabiri. A full three-dimensional characterization of defocusing digital particle image velocimetry. *Measurement Science and Technology*, 16:790–804, 2005.

[27] Anush Krishnan, John J. Socha, Pavlos P. Vlachos, and L. A. Barba. Lift and wakes of flying snakes. *Physics of Fluids*, 26(3):031901, 2014. doi: 10.1063/1.4866444. URL https://doi.org/10.1063/1.4866444.

[28] Francois Lamarche and Claude Leroy. Evaluation of the Volume of Intersection of a Sphere with a Cylinder by Elliptic Integrals. *Computer Physics Communications*, 59: 359–369, 1990.

[29] F. Lekien and J. Marsden. Tricubic interpolation in three dimensions. *International Journal for Numerical Methods in Engineering*, pages 455–471, Mar 2005.

[30] Marc Levoy. Light Fields and Computational Imaging. *IEEE Computer Society*, pages 46–55, aug 2006.

[31] Marc Levoy and Pat Hanrahan. Light Field Rendering. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques ACM*, pages 31–42, 1996.

[32] LibTIFF TIFF Library and Utilities, Jun 2022. URL http://www.libtiff.org/.

[33] Andrew Lumsdaine and Todor Georgeiv. Focused plenoptic camera and rendering. *Journal of Electronic Imaging*, 19(2):021106, 2010. ISSN 1017-9909. doi: 10.1117/1.3442712.

[34] Kyle Lynch and Fulvio Scarano. A high-order time-accurate interrogation method for time-resolved PIV. *Measurement Science and Technology*, 24, 2013.

[35] Kyle Lynch, Tim Fahringer, and Brian Thurow. Three-dimensional particle image velocimetry using a plenoptic camera. In *50th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*. AIAA Aerospace Sciences Meeting, jan 2012. doi: 10.2514/6.2012-1056.

[36] Hui Meng, Gang Pan, Ye Pu, and Scott H Woodward. Holographic particle image velocimetry: from film to digital recording. *Measurement Science and Technology*, 15: 673–685, 2004.

[37] D. Michaelis and B. Wieneke. Comparison between Tomographic PIV and Stereo PIV. In *14th Int Symp on Applications of Laser Techniques to Fluid Mechanics*, 2008.

[38] D. Michaelis, M. Novara, F. Scarano, and B. Wieneke. Comparison of volume reconstruction techniques at different particle densities. In *15th Int Symp on Applications of Laser Techniques to Fluid Mechanics*, 2010.

[39] Open MPI, Jul 2022. URL https://www.open-mpi.org/.

[40] Ren Ng, Marc Levoy, Mathieu Bredif, Gene Duval, Mark Horowitz, and Pat Hanrahan. Light Field Photography with a Hand-held Plenoptic Camera. Technical report, Standford University, 2005.

[41] Holger Nobach, Nils Damaschke, and Cam Tropea. High-precision sub-pixel interpolation in piv/ptv image processing. In *Proceedings of the 12th international symposium on applications of laser techniques to fluid mechanics, Lisbon, Portugal*, 2004.

[42] T Nonn, J Kitzhofer, D Hess, and Ch. Bruker. Measurements in an IC-engine Flow using Light-field Volumetric Velocimetry. In *Internation Symposium on Applications of Laser Techniques to Fluid Mechanics*, jul 2012.

[43] Matteo Novara1, Kees Joost Batenburg, and Fulvio Scarano. Motion tracking-enhanced MART for tomographic PIV. *Measurement Science and Technology*, 21, 2010.

[44] OpenMP, Jul 2022. URL https://www.openmp.org/.

[45] LAPACK — Linear Algebra PACKage, Jul 2022. URL https://netlib.org/lapack/.

[46] F. Pereira and M. Gharib. Defocusing digital particle image velocimetry and the three-dimensional characterization of two-phase flows. *Measurement Science and Technology*, pages 683–694, 2002.

[47] F. Pereira, M. Gharib, D. Dabiri, and D. Modarress. Defocusing digital particle image velocimetry: a 3-component 3-dimensional DPIV measurement technique. Application to bubbly flows. *Experiments in Fluids*, pages S78–S84, 2000.

[48] A. K. Prasad. Stereoscopic particle image velocimetry. *Experiments in Fluids*, 29: 103–116, 2000.

[49] William H. Press and Saul A. Teukolsky. Elliptic Integrals. *Computers in Physics*, pages 92–96, 1990.

[50] C. E. W. Markus Raffel, Steve T. Wereley, and Jurgen Kompenhans. *Particle Image Velocimetry: A Practical Guide*. Springer, 1998.

[51] F. Scarano. Iterative image deformation methods in PIV. *Measurement Science and Technology*, pages R1–R19, 2002.

[52] F. Scarano. Tomographic PIV: principles and practice. *Measurement Science and Technology*, 2013.

[53] F. Scarano, L. David, M. Bsibsi, and D. Calluaud. S-PIV comparative assessment: image dewarping+misalignment correction and pinhole+geometric back projection. *Experiments in Fluids*, pages 257–266, 2004.

[54] D. Schanz, S. Gesemann, A. Schröder, D. Michaelis, and B. Wieneke. 'Shake The Box': A highly efficient and accurate Tomographic Particle Tracking Velocimetry (TOMO-PTV) method using prediction of particle positions. In *Proceedings of the 10th International Symposium on Particle Image Velocimetry, Delft, The Netherlands*, 2013.

[55] D. Schanz, S. Gesemann, and A. Schröder. Shake-The-Box: Lagrangian particle tracking at high particle image densities. *Experiments in Fluids*, 2016.

[56] Daniel Schanz, Andreas Schröder, and Sebastian Gesemann. 'Shake The Box' - a 4D PTV algorithm: Accurate and ghostless reconstruction of Lagrangian tracks in densely seeded flows. In *10th International Symposium on Particle Image Velocimetry - PIV13*, 2014.

[57] Andrea Sciacchitano, Fulvio Scarano, and Bernhard Wieneke. Multi-frame pyramid correlation for time-resolved PIV. *Experiments in Fluids*, 53:1087–1105, 2012.

[58] John J Socha. Gliding flight in the paradise tree snake. *Nature*, 418:603,604, 2002.

[59] John J Socha. A 3-D kinematic analysis of gliding in a flying snake, Chrysopelea paradisi. *The Journal of Experimental Biology*, 208:1817–1833, 2005.

[60] S M Soloff, R J Adrian, and Z-C Liu. Distortion compensation for generalized stereoscopic particle image velocimetry. *Measurement Science and Technology*, 8:1441–1454, 1997.

[61] BLAS (Basic Linear Algebra Subprograms), Jun 2021. URL https://netlib.org/blas/.

[62] E. W. Weisstein. *CRC Concise Encyclopedia of Mathematics*. CRC Press LLC, 2003.

[63] Jerry Westerweel and Fulvio Scarano. Universal outlier detection for PIV data. *Experiments in Fluids*, 39:1096–1100, 2005.

[64] B Wieneke. Stereo-PIV using self-calibration on particle images. *Experiments in Fluids*, 39:267–280, 2005.

[65] B Wieneke. Volume self-calibration for 3D particle image velocimetry. *Experiments in Fluids*, 45:549–556, 2008.

[66] Bennett Wilburn, Neel Joshi, Vaibhav Vaish, Eino-Ville Talvala, Emilio Antunez, Adam Barth, Andrew Adams, Mark Horowitz, and Marc Levoy. High Performance Imaging Using Large Camera Arrays. *ACM Transactions on Graphics (TOG)*, 24(3):765–776, 2005.

[67] C. E. Willert and M. Gharib. Digital particle image velocimetry. *Experiments in Fluids*, 10:181–193, 1991.

[68] Jie-Zhi Wu, Hui-Yang Ma, and J Z Zhou. *Vorticity and vortex dynamics*. Springer, 2005.

Appendices

Appendix A

C Code Functions

A.1 Least Energy Velocity Field

This code takes arrays of possible u, v, and w three-dimensional vector fields that are extracted from multiple cross-correlation peaks and calculates an approximation to the vector field that has the lowest signal energy. Since for n stored cross-correlation peaks and m vectors within the vector field, the number of vector field permutations is n^m, determining the exact solution is computationally infeasible. So the algorithm finds local approximate solutions and iteratively updates the vector field until convergence is reached. In preliminary testing, this algorithm always produces more accurate vector fields than simply selecting the largest cross-correlation peak. Only basic testing has been completed examining if the selected vector field is always the unique least energy vector field.

This code takes the three components of the velocity field as input arrays 'ii_velocity_peaks', 'jj_velocity_peaks', and 'kk_velocity_peaks' which are one-dimensional arrays with lengths equal to the number of stored cross-correlation peaks 'peak_vector_number' multiplied by the total number of vectors which is given by the product of the elements of the length 3 'vector_number'. The length 3 array 'peak_energy_kernal_size' stores the size of the kernel used to calculate the local vector field energy. The kernel size must be odd in each dimension.

The output of the least energy calculation is stored in the velocity field arrays 'ii_velocity',

'jj_velocity', and 'kk_velocity' which are of lengths equal to the product of the elements of the length 3 'vector_number'. Upon success, the function will return with a value of 0. If the function fails for any reason, it will return with a value of -1.

Listing A.1: Least Signal Energy Velocity Field Function

```
int least_signal_energy_velocity_field( double ii_velocity_peaks[], double
    jj_velocity_peaks[], double kk_velocity_peaks[],
                    int vector_number[], int peak_vector_number, int
                        peak_energy_kernal_size[],
                    double ii_velocity[], double jj_velocity[], double
                        kk_velocity[] )
{
  // This function extracts the velocity vectors that correspond to the least
  // energy combination of the possible velocity vectors stored in the 4th
  // dimension of the input arrays (which come from additional peaks in the
  // correlation volumes).

  ////////////////////////////////////////////////////////////////////////
  // Lowest Energy Calculation Parameters                                //
  ////////////////////////////////////////////////////////////////////////

  // This initialize variables to store the radius of the neighborhood to
  // compare vectors in each dimension
  int ii_neighborhood_radius;
  int jj_neighborhood_radius;
  int kk_neighborhood_radius;

  // This is the radius of the neighborhood to compare the vectors in each
  // dimension
  ii_neighborhood_radius = ( peak_energy_kernal_size[ 0 ] - 1 ) / 2;
```

```
23  jj_neighborhood_radius = ( peak_energy_kernal_size[ 1 ] - 1 ) / 2;
24  kk_neighborhood_radius = ( peak_energy_kernal_size[ 2 ] - 1 ) / 2;
25
26  // This initializes variables to store the number of windows (ie vectors)
27  // in each dimension
28  int ii_vector_number;
29  int jj_vector_number;
30  int kk_vector_number;
31
32  // This is the number of windows (ie vectors) in each dimension
33  ii_vector_number = vector_number[ 0 ];
34  jj_vector_number = vector_number[ 1 ];
35  kk_vector_number = vector_number[ 2 ];
36
37  ////////////////////////////////////////////////////////////////////////////
38  // Initializing Variables                                                 //
39  ////////////////////////////////////////////////////////////////////////////
40
41  // This initializes variables to index the windows (ie vectors) in each
42  // dimension
43  int ii_vector_index;
44  int jj_vector_index;
45  int kk_vector_index;
46
47  // This initializes variables for the first dimension range to load for
48  // calculating the local signal energy
49  int ii_region_min;
50  int ii_region_max;
51  // This initializes variables for the second dimension range to load for
52  // calculating the local signal energy
53  int jj_region_min;
```

```
54    int jj_region_max;
55    // This initializes variables for the third dimension range to load for
56    // calculating the local signal energy
57    int kk_region_min;
58    int kk_region_max;
59
60    // This initializes a variable for indexing through the identified peaks
61    int peak_vector_index;
62
63    // This initializes a variable to store the linear index of the center
64    // element of the local region of the vector fields
65    int center_element_linear_index;
66
67    // This initializes a variable to store the linear index of the adjacent
68    // elements of the local region of the vector field
69    int adjacent_element_linear_index;
70
71    // This initializes variables to store the value of the center velocity
72    // vector in each extracted region
73    double ii_velocity_center;
74    double jj_velocity_center;
75    double kk_velocity_center;
76
77    // This initializes indexing variables for extracting the vectors in each
78    // velocity region
79    int ii_region_index;
80    int jj_region_index;
81    int kk_region_index;
82    // This intializes an indexing variable for extracting all possible vectors
83    // for each of the peaks within the velocity region
84    int peak_region_index;
```

```
85
86  // This initializes a variable to store the velocity residuals for each of
87  // the possible velocity field peaks
88  double *velocity_residuals;
89
90  // This allocates memory to the velocity field residuals
91  velocity_residuals = ( double * ) malloc( peak_vector_number * sizeof(
        double ) );
92  // If the memory could not be allocated, this prints an error and then
93  // exits the function
94  if ( velocity_residuals == NULL )
95  {
96    // This prints a memory allocation error message
97    error_memory_allocation( "velocity_residuals", "
          least_signal_energy_velocity_field" );
98    // This returns from the function with an error
99    return -1;
100 }
101
102 // This initializes a variable to store the minimum velocity residual value
103 double min_velocity_residual_value;
104 // This initializes a variable to store the minimum velocity residual index
105 int min_velocity_residual_index;
106
107 // This is a linear index into the output velocity arrays
108 int output_linear_index;
109
110 // This initializes a variable to store the number of elements in each
111 // adjacent region of the velocity field
112 int adjacent_element_number;
113 // This calculates the number of adjacent velocity elements in each region
```

```
114   adjacent_element_number = peak_energy_kernal_size[ 0 ] *
          peak_energy_kernal_size[ 1 ] * peak_energy_kernal_size[ 2 ] *
          peak_vector_number;
115
116   // This initializes arrays to store the possible velocity vector values in
117   // each neighborhood region
118   double *ii_velocity_neighborhood;
119   double *jj_velocity_neighborhood;
120   double *kk_velocity_neighborhood;
121   // This allocates memory to the arrays storing the adjacent possible
122   // velocity vectors in each region
123   ii_velocity_neighborhood = ( double * ) malloc( adjacent_element_number *
          sizeof( double ) );
124   jj_velocity_neighborhood = ( double * ) malloc( adjacent_element_number *
          sizeof( double ) );
125   kk_velocity_neighborhood = ( double * ) malloc( adjacent_element_number *
          sizeof( double ) );
126   // If the memory could not be allocated, this prints an error and then
127   // exits the function
128   if ( ii_velocity_neighborhood == NULL )
129   {
130     // This prints a memory allocation error message
131     error_memory_allocation( "ii_velocity_neighborhood", "
            least_signal_energy_velocity_field" );
132     // This returns from the function with an error
133     return -1;
134   }
135   // If the memory could not be allocated, this prints an error and then
136   // exits the function
137   if ( jj_velocity_neighborhood == NULL )
138   {
```

```
139     // This prints a memory allocation error message
140     error_memory_allocation( "jj_velocity_neighborhood", "
            least_signal_energy_velocity_field" );
141     // This returns from the function with an error
142     return -1;
143   }
144   // If the memory could not be allocated, this prints an error and then
145   // exits the function
146   if ( kk_velocity_neighborhood == NULL )
147   {
148     // This prints a memory allocation error message
149     error_memory_allocation( "kk_velocity_neighborhood", "
            least_signal_energy_velocity_field" );
150     // This returns from the function with an error
151     return -1;
152   }
153
154   // This initializes an array to store the index of the peak that is
155   // calculated to yield the lowest energy velocity field
156   int *peak_index_array;
157   // This allocates memory to the array giving the index of the peaks
158   // corresponding to the lowest energy velocity field
159   peak_index_array = ( int * ) malloc( ii_vector_number * jj_vector_number *
          kk_vector_number * sizeof( int ) );
160   // If the memory could not be allocated, this prints an error and then
161   // exits the function
162   if ( peak_index_array == NULL )
163   {
164     // This prints a memory allocation error message
165     error_memory_allocation( "peak_index_array", "
            least_signal_energy_velocity_field" );
```

```
166     // This returns from the function with an error
167     return -1;
168   }
169
170   // This initializes a variable to both index into the adjacent peaks and to
171   // store the total number of peaks in each region (which might be less than
172   // the total adjacent_element_number if the region is on the edge of the
173   // vector field)
174   int current_neighborhood_element_number;
175
176   ////////////////////////////////////////////////////////////////////////////
177   // Initial Estimate of Lowest Energy Field                                //
178   ////////////////////////////////////////////////////////////////////////////
179
180   // This iterates through the first dimension of the velocity field
181   for ( ii_vector_index = 0; ii_vector_index < ii_vector_number;
          ii_vector_index++ )
182   {
183
184     // This calculates the neighborhood over which to compare the
185     // vectors in the first dimension
186     ii_region_min = ii_vector_index - ii_neighborhood_radius;
187     ii_region_max = ii_vector_index + ii_neighborhood_radius;
188
189     // If the minimum of the neighborhood region is less than zero, this
190     // sets it equal to zero
191     if ( ii_region_min < 0 )
192     {
193       // This sets the region minimum equal to zero
194       ii_region_min = 0;
195     }
```

```
196
197     // If the maximum of the neighborhood region is greater than the number
198     // of vectors, this sets it equal to vector number (minus one)
199     if ( ii_region_max > ( ii_vector_number - 1 ) )
200     {
201       // This sets the region minimum equal to the vector number (minus
202       // one)
203       ii_region_max = ( ii_vector_number - 1 );
204     }
205
206     // This iterates through the second dimension of the velocity field
207     for ( jj_vector_index = 0; jj_vector_index < jj_vector_number;
            jj_vector_index++ )
208     {
209
210       // This calculates the neighborhood over which to compare the
211       // vectors in the second dimension
212       jj_region_min = jj_vector_index - jj_neighborhood_radius;
213       jj_region_max = jj_vector_index + jj_neighborhood_radius;
214
215       // If the minimum of the neighborhood region is less than zero,
216       // this sets it equal to zero
217       if ( jj_region_min < 0 )
218       {
219         // This sets the region minimum equal to zero
220         jj_region_min = 0;
221       }
222
223       // If the maximum of the neighborhood region is greater than the
224       // number of vectors, this sets it equal to vector number (minus
225       // one)
```

```
226       if ( jj_region_max > ( jj_vector_number - 1 ) )
227       {
228         // This sets the region minimum equal to the vector number
229         // (minus one)
230         jj_region_max = ( jj_vector_number - 1 );
231       }
232
233       // This iterates through the third dimension of the velocity field
234       for ( kk_vector_index = 0; kk_vector_index < kk_vector_number;
              kk_vector_index++ )
235       {
236
237         // This calculates the neighborhood over which to compare the
238         // vectors in the third dimension
239         kk_region_min = kk_vector_index - kk_neighborhood_radius;
240         kk_region_max = kk_vector_index + kk_neighborhood_radius;
241
242         // If the minimum of the neighborhood region is less than zero,
243         // this set it equal to zero
244         if ( kk_region_min < 0 )
245         {
246           // This sets the region minimum equal to zero
247           kk_region_min = 0;
248         }
249
250         // If the maximum of the neighborhood region is greater than
251         // the number of vectors, this sets it equal to vector number
252         // (minus one)
253         if ( kk_region_max > ( kk_vector_number - 1 ) )
254         {
255           // This sets the region minimum equal to the vector number
```

```
256           // (minus one)
257           kk_region_max = ( kk_vector_number - 1 );
258         }
259
260         ////////////////////////////////////////////////////////////////
261         // Extracting Vectors in Neighborhood                         //
262         ////////////////////////////////////////////////////////////////
263
264         // This initializes the number of adjacent velocity vector
265         // peaks in the current region to zero (so that it can be used
266         // as an indexing variable and for recording the total number
267         // of elements in the current region)
268         current_neighborhood_element_number = 0;
269
270         // This iterates through the first dimension of the vector
271         // field region
272         for ( ii_region_index = ii_region_min; ii_region_index <=
                ii_region_max; ii_region_index++ )
273         {
274
275           // This iterates through the second dimension of the
276           // vector field region
277           for ( jj_region_index = jj_region_min; jj_region_index <=
                  jj_region_max; jj_region_index++ )
278           {
279
280             // This iterates through the third dimension of the
281             // vector field region
282             for ( kk_region_index = kk_region_min; kk_region_index <=
                    kk_region_max; kk_region_index++ )
283             {
```

```
284
285                 // This iterates through the peaks of the
286                 // vector field region
287                 for ( peak_region_index = 0; peak_region_index <
                        peak_vector_number; peak_region_index++ )
288                 {
289
290                   // This calculates the linear index of the
291                   // vector adjacent to the center vector in the
292                   // current region with the current peak index
293                   adjacent_element_linear_index = ii_region_index *
                          jj_vector_number * kk_vector_number * peak_vector_number +
294                   jj_region_index * kk_vector_number * peak_vector_number +
295                   kk_region_index * peak_vector_number +
296                   peak_region_index;
297
298                   // This extracts the adjacent velocity vector
299                   // in the current region
300                   ii_velocity_neighborhood[ current_neighborhood_element_number
                          ] = ii_velocity_peaks[ adjacent_element_linear_index ];
301                   jj_velocity_neighborhood[ current_neighborhood_element_number
                          ] = jj_velocity_peaks[ adjacent_element_linear_index ];
302                   kk_velocity_neighborhood[ current_neighborhood_element_number
                          ] = kk_velocity_peaks[ adjacent_element_linear_index ];
303
304                   // This increments the index (and number) of
305                   // adjacent velocity vectors in the current
306                   // region
307                   current_neighborhood_element_number++;
308
309                 }
```

```
310
311             }
312
313           }
314
315         }
316
317         ////////////////////////////////////////////////////////////////
318         // Calculating Vector Residuals for Potential Peaks           //
319         ////////////////////////////////////////////////////////////////
320
321         // This iterates through the peak values of the current vector
322         for ( peak_vector_index = 0; peak_vector_index < peak_vector_number;
                peak_vector_index++ )
323         {
324
325           // This calculates the linear index of the vector centered
326           // in the current region with the current peak index
327           center_element_linear_index = ii_vector_index * jj_vector_number *
                kk_vector_number * peak_vector_number +
328                         jj_vector_index * kk_vector_number *
                            peak_vector_number +
329                         kk_vector_index * peak_vector_number +
330                         peak_vector_index;
331
332           // This extracts the velocity vector centered in the current
333           // region with the current peak index
334           ii_velocity_center = ii_velocity_peaks[ center_element_linear_index
                ];
335           jj_velocity_center = jj_velocity_peaks[ center_element_linear_index
                ];
```

```
336             kk_velocity_center = kk_velocity_peaks[ center_element_linear_index
                    ];
337
338             // This initializes the velocity field residual for the
339             // current peak to zero (before the residuals are added
340             // onto this value)
341             velocity_residuals[ peak_vector_index ] = 0.0;
342
343             // This iterates through the adjacent velocity vectors,
344             // calculating the velocity field residual for the current
345             // peak velocity value
346             for ( adjacent_element_linear_index = 0;
                    adjacent_element_linear_index <
                    current_neighborhood_element_number;
                    adjacent_element_linear_index++ )
347             {
348
349               // This adds the residual of the current
350               // adjacent velocity vector to the sum of all
351               // adjacent vector residuals
352               velocity_residuals[ peak_vector_index ] += pow( ii_velocity_center
                      - ii_velocity_neighborhood[ adjacent_element_linear_index ],
                     2.0 ) +
353                                  pow( jj_velocity_center -
                                      jj_velocity_neighborhood[
                                      adjacent_element_linear_index ], 2.0 )
                                      +
354                                  pow( kk_velocity_center -
                                      kk_velocity_neighborhood[
                                      adjacent_element_linear_index ], 2.0 );
355
```

```
356           }
357
358         }
359
360         ////////////////////////////////////////////////////////////////
361         // Determining Minimum Residual Peak                          //
362         ////////////////////////////////////////////////////////////////
363
364         // This initializes the velocity residual minimum value equal
365         // to the first velocity residual element
366         min_velocity_residual_value = velocity_residuals[ 0 ];
367
368         // This initializes the velocity residual minimum index equal
369         // to zero
370         min_velocity_residual_index = 0;
371
372         // This iterates through the possible peak options finding the
373         // index of the peak corresponding to the lowest velocity
374         // residual sum
375         for ( peak_vector_index = 0; peak_vector_index < peak_vector_number;
                peak_vector_index++ )
376         {
377
378           // If the current value of the velocity residual sum is
379           // less than the recorded minimum value, this resets the
380           // minimum value and the corresponding index
381           if ( velocity_residuals[ peak_vector_index ] <
                  min_velocity_residual_value )
382           {
383
384             // This sets the minimum velocity residual value eqaul
```

```
385             // to the current velocity residual sum value
386             min_velocity_residual_value = velocity_residuals[
                    peak_vector_index ];
387
388             // This sets the index of the minimum velocity residual
389             // sum equal to the current index
390             min_velocity_residual_index = peak_vector_index;
391
392           }
393
394         }
395
396         ////////////////////////////////////////////////////////////////
397         // Saving Minimum Energy Velocity Vector                      //
398         ////////////////////////////////////////////////////////////////
399
400         // This calculates the linear index of the vector
401         // corresponding to the minimum velocity residual sum
402         center_element_linear_index = ii_vector_index * jj_vector_number *
                kk_vector_number * peak_vector_number +
403                       jj_vector_index * kk_vector_number *
                            peak_vector_number +
404                       kk_vector_index * peak_vector_number +
405                       min_velocity_residual_index;
406
407         // This extracts the velocity vector centered in the current
408         // region with the current peak index
409         ii_velocity_center = ii_velocity_peaks[ center_element_linear_index ];
410         jj_velocity_center = jj_velocity_peaks[ center_element_linear_index ];
411         kk_velocity_center = kk_velocity_peaks[ center_element_linear_index ];
412
```

```
413         // This calculates the linear index of the current output
414         // array vector
415         output_linear_index = ii_vector_index * jj_vector_number *
                kk_vector_number +
416                   jj_vector_index * kk_vector_number +
417                   kk_vector_index;
418
419         // This sets the current output velocity vector equal to the
420         // vector corresponding to the minimum energy value
421         ii_velocity[ output_linear_index ] = ii_velocity_center;
422         jj_velocity[ output_linear_index ] = jj_velocity_center;
423         kk_velocity[ output_linear_index ] = kk_velocity_center;
424
425         // This sets the current element of the peak index array equal
426         // to the index that was identified as yielding the lowest
427         // energy velocity field
428         peak_index_array[ output_linear_index ] = min_velocity_residual_index;
429
430       }
431
432     }
433
434   }
435
436   ////////////////////////////////////////////////////////////////////////
437   // Initializing Iterative Lowest Energy Variables                     //
438   ////////////////////////////////////////////////////////////////////////
439
440   // This initializes a variable to count the number of vectors that were
441   // updated during each iteration (which when equal to zero will be used to
442   // exit the while loop)
```

```
443   int updated_vector_number;
444
445   // This initializes arrays to store the updated lowest energy velocity
446   // fields
447   double *ii_velocity_updated;
448   double *jj_velocity_updated;
449   double *kk_velocity_updated;
450   // This allocates memory to the arrays storing the updated velocity fields
451   ii_velocity_updated = ( double * ) malloc( ii_vector_number *
          jj_vector_number * kk_vector_number * sizeof( double ) );
452   jj_velocity_updated = ( double * ) malloc( ii_vector_number *
          jj_vector_number * kk_vector_number * sizeof( double ) );
453   kk_velocity_updated = ( double * ) malloc( ii_vector_number *
          jj_vector_number * kk_vector_number * sizeof( double ) );
454   // If the memory could not be allocated, this prints an error and then
455   // exits the function
456   if ( ii_velocity_updated == NULL )
457   {
458     // This prints a memory allocation error message
459     error_memory_allocation( "ii_velocity_updated", "
          least_signal_energy_velocity_field" );
460     // This returns from the function with an error
461     return -1;
462   }
463   // If the memory could not be allocated, this prints an error and then
464   // exits the function
465   if ( jj_velocity_updated == NULL )
466   {
467     // This prints a memory allocation error message
468     error_memory_allocation( "jj_velocity_updated", "
          least_signal_energy_velocity_field" );
```

```
469     // This returns from the function with an error
470     return -1;
471   }
472   // If the memory could not be allocated, this prints an error and then
473   // exits the function
474   if ( kk_velocity_updated == NULL )
475   {
476     // This prints a memory allocation error message
477     error_memory_allocation( "kk_velocity_updated", "
          least_signal_energy_velocity_field" );
478     // This returns from the function with an error
479     return -1;
480   }
481
482   ////////////////////////////////////////////////////////////////////////
483   // Iterative Approximation to Lowest Energy Field                     //
484   ////////////////////////////////////////////////////////////////////////
485
486   // Now that an approximation to the lowest energy velocity field has been
487   // calculated, this repeatedly iterates through the velocity field
488   // replacing any vectors that result in a lower energy field until
489   // stability is reached
490   while ( 1 )
491   {
492
493     // This initializes the number of updated vectors to zero for the
494     // current iteration of the while loop
495     updated_vector_number = 0;
496
497     // This iterates through the first dimension of the velocity field
498     for ( ii_vector_index = 0; ii_vector_index < ii_vector_number;
```

```
        ii_vector_index++ )
499     {
500
501       // This calculates the neighborhood over which to compare the
502       // vectors in the first dimension
503       ii_region_min = ii_vector_index - ii_neighborhood_radius;
504       ii_region_max = ii_vector_index + ii_neighborhood_radius;
505
506       // If the minimum of the neighborhood region is less than zero, this
507       // sets it equal to zero
508       if ( ii_region_min < 0 )
509       {
510         // This sets the region minimum equal to zero
511         ii_region_min = 0;
512       }
513
514       // If the maximum of the neighborhood region is greater than the number
515       // of vectors, this sets it equal to vector number (minus one)
516       if ( ii_region_max > ( ii_vector_number - 1 ) )
517       {
518         // This sets the region minimum equal to the vector number (minus
519         // one)
520         ii_region_max = ( ii_vector_number - 1 );
521       }
522
523       // This iterates through the second dimension of the velocity field
524       for ( jj_vector_index = 0; jj_vector_index < jj_vector_number;
            jj_vector_index++ )
525       {
526
527         // This calculates the neighborhood over which to compare the
```

```
528         // vectors in the second dimension
529         jj_region_min = jj_vector_index - jj_neighborhood_radius;
530         jj_region_max = jj_vector_index + jj_neighborhood_radius;
531
532         // If the minimum of the neighborhood region is less than zero,
533         // this sets it equal to zero
534         if ( jj_region_min < 0 )
535         {
536           // This sets the region minimum equal to zero
537           jj_region_min = 0;
538         }
539
540         // If the maximum of the neighborhood region is greater than the
541         // number of vectors, this sets it equal to vector number (minus
542         // one)
543         if ( jj_region_max > ( jj_vector_number - 1 ) )
544         {
545           // This sets the region minimum equal to the vector number
546           // (minus one)
547           jj_region_max = ( jj_vector_number - 1 );
548         }
549
550         // This iterates through the third dimension of the velocity field
551         for ( kk_vector_index = 0; kk_vector_index < kk_vector_number;
                kk_vector_index++ )
552         {
553
554           // This calculates the neighborhood over which to compare the
555           // vectors in the third dimension
556           kk_region_min = kk_vector_index - kk_neighborhood_radius;
557           kk_region_max = kk_vector_index + kk_neighborhood_radius;
```

```
558
559             // If the minimum of the neighborhood region is less than zero,
560             // this set it equal to zero
561             if ( kk_region_min < 0 )
562             {
563               // This sets the region minimum equal to zero
564               kk_region_min = 0;
565             }
566
567             // If the maximum of the neighborhood region is greater than
568             // the number of vectors, this sets it equal to vector number
569             // (minus one)
570             if ( kk_region_max > ( kk_vector_number - 1 ) )
571             {
572               // This sets the region minimum equal to the vector number
573               // (minus one)
574               kk_region_max = ( kk_vector_number - 1 );
575             }
576
577             ////////////////////////////////////////////////////////////
578             // Extracting Vectors in Neighborhood                     //
579             ////////////////////////////////////////////////////////////
580
581             // This initializes the number of adjacent velocity vector
582             // peaks in the current region to zero (so that it can be used
583             // as an indexing variable and for recording the total number
584             // of elements in the current region)
585             current_neighborhood_element_number = 0;
586
587             // This iterates through the first dimension of the vector
588             // field region
```

```
589             for ( ii_region_index = ii_region_min; ii_region_index <=
                    ii_region_max; ii_region_index++ )
590             {
591
592               // This iterates through the second dimension of the
593               // vector field region
594               for ( jj_region_index = jj_region_min; jj_region_index <=
                      jj_region_max; jj_region_index++ )
595               {
596
597                 // This iterates through the third dimension of the
598                 // vector field region
599                 for ( kk_region_index = kk_region_min; kk_region_index <=
                        kk_region_max; kk_region_index++ )
600                 {
601
602                   // This calculates the linear index of the
603                   // vector adjacent to the center vector in the
604                   // current region with the current peak index
605                   adjacent_element_linear_index = ii_region_index *
                          jj_vector_number * kk_vector_number +
606                   jj_region_index * kk_vector_number +
607                   kk_region_index;
608
609                   // This extracts the adjacent velocity vector
610                   // in the current region
611                   ii_velocity_neighborhood[ current_neighborhood_element_number
                          ] = ii_velocity[ adjacent_element_linear_index ];
612                   jj_velocity_neighborhood[ current_neighborhood_element_number
                          ] = jj_velocity[ adjacent_element_linear_index ];
613                   kk_velocity_neighborhood[ current_neighborhood_element_number
```

```
                    ] = kk_velocity[ adjacent_element_linear_index ];
614
615                 // This increments the index (and number) of
616                 // adjacent velocity vectors in the current
617                 // region
618                 current_neighborhood_element_number++;
619
620               }
621
622             }
623
624           }
625
626           ////////////////////////////////////////////////////////////
627           // Calculating Vector Residuals for Potential Peaks       //
628           ////////////////////////////////////////////////////////////
629
630           // This iterates through the peak values of the current vector
631           for ( peak_vector_index = 0; peak_vector_index < peak_vector_number;
                    peak_vector_index++ )
632           {
633
634             // This calculates the linear index of the vector centered
635             // in the current region with the current peak index
636             center_element_linear_index = ii_vector_index * jj_vector_number *
                    kk_vector_number * peak_vector_number +
637             jj_vector_index * kk_vector_number * peak_vector_number +
638             kk_vector_index * peak_vector_number +
639             peak_vector_index;
640
641             // This extracts the velocity vector centered in the current
```

```
642             // region with the current peak index
643             ii_velocity_center = ii_velocity_peaks[
                    center_element_linear_index ];
644             jj_velocity_center = jj_velocity_peaks[
                    center_element_linear_index ];
645             kk_velocity_center = kk_velocity_peaks[
                    center_element_linear_index ];
646
647             // This initializes the velocity field residual for the
648             // current peak to zero (before the residuals are added
649             // onto this value)
650             velocity_residuals[ peak_vector_index ] = 0.0;
651
652             // This iterates through the adjacent velocity vectors,
653             // calculating the velocity field residual for the current
654             // peak velocity value
655             for ( adjacent_element_linear_index = 0;
                    adjacent_element_linear_index <
                    current_neighborhood_element_number;
                    adjacent_element_linear_index++ )
656             {
657
658               // This adds the residual of the current
659               // adjacent velocity vector to the sum of all
660               // adjacent vector residuals
661               velocity_residuals[ peak_vector_index ] += pow(
                      ii_velocity_center - ii_velocity_neighborhood[
                      adjacent_element_linear_index ], 2.0 ) +
662               pow( jj_velocity_center - jj_velocity_neighborhood[
                      adjacent_element_linear_index ], 2.0 ) +
663               pow( kk_velocity_center - kk_velocity_neighborhood[
```

```
                adjacent_element_linear_index ], 2.0 );
664
665           }
666
667         }
668
669         ////////////////////////////////////////////////////////////
670         // Determining Minimum Residual Peak                      //
671         ////////////////////////////////////////////////////////////
672
673         // This initializes the velocity residual minimum value equal
674         // to the first velocity residual element
675         min_velocity_residual_value = velocity_residuals[ 0 ];
676
677         // This initializes the velocity residual minimum index equal
678         // to zero
679         min_velocity_residual_index = 0;
680
681         // This iterates through the possible peak options finding the
682         // index of the peak corresponding to the lowest velocity
683         // residual sum
684         for ( peak_vector_index = 0; peak_vector_index < peak_vector_number;
                peak_vector_index++ )
685         {
686
687           // If the current value of the velocity residual sum is
688           // less than the recorded minimum value, this resets the
689           // minimum value and the corresponding index
690           if ( velocity_residuals[ peak_vector_index ] <
                min_velocity_residual_value )
691           {
```

```
692
693             // This sets the minimum velocity residual value eqaul
694             // to the current velocity residual sum value
695             min_velocity_residual_value = velocity_residuals[
                    peak_vector_index ];
696
697             // This sets the index of the minimum velocity residual
698             // sum equal to the current index
699             min_velocity_residual_index = peak_vector_index;
700
701           }
702
703         }
704
705         ////////////////////////////////////////////////////////////
706         // Saving Minimum Energy Velocity Vector                  //
707         ////////////////////////////////////////////////////////////
708
709         // This calculates the linear index of the vector
710         // corresponding to the minimum velocity residual sum
711         center_element_linear_index = ii_vector_index * jj_vector_number *
                kk_vector_number * peak_vector_number +
712         jj_vector_index * kk_vector_number * peak_vector_number +
713         kk_vector_index * peak_vector_number +
714         min_velocity_residual_index;
715
716         // This extracts the velocity vector centered in the current
717         // region with the current peak index
718         ii_velocity_center = ii_velocity_peaks[ center_element_linear_index
                ];
719         jj_velocity_center = jj_velocity_peaks[ center_element_linear_index
```

```
            ];
720         kk_velocity_center = kk_velocity_peaks[ center_element_linear_index
            ];
721
722         // This calculates the linear index of the current output
723         // array vector
724         output_linear_index = ii_vector_index * jj_vector_number *
            kk_vector_number +
725         jj_vector_index * kk_vector_number +
726         kk_vector_index;
727
728         // This sets the current output velocity vector equal to the
729         // vector corresponding to the minimum energy value
730         ii_velocity_updated[ output_linear_index ] = ii_velocity_center;
731         jj_velocity_updated[ output_linear_index ] = jj_velocity_center;
732         kk_velocity_updated[ output_linear_index ] = kk_velocity_center;
733
734         // If the index of the peak is different from that which
735         // was previously calculated, this increments the number of
736         // updated vectors and updates the array of peak index
737         // values
738         if ( min_velocity_residual_index != peak_index_array[
            output_linear_index ] )
739         {
740
741           // This increments the number of updated vectors
742           updated_vector_number++;
743
744           // This stores the new index of the peak that was
745           // calculated to have the lowest energy
746           peak_index_array[ output_linear_index ] =
```

```
                min_velocity_residual_index;
747
748           }
749
750         }
751
752       }
753
754     }
755
756     // This copies the updated velocity field arrays into the output
757     // velocity field arrays
758     memcpy( ii_velocity, ii_velocity_updated, ii_vector_number *
            jj_vector_number * kk_vector_number * sizeof( double ) );
759     memcpy( jj_velocity, jj_velocity_updated, ii_vector_number *
            jj_vector_number * kk_vector_number * sizeof( double ) );
760     memcpy( kk_velocity, kk_velocity_updated, ii_vector_number *
            jj_vector_number * kk_vector_number * sizeof( double ) );
761
762     // If no vectors have been updated in the current loop, this exits the
763     // while loop since convergence has been reached
764     if ( updated_vector_number == 0 )
765     {
766
767       // This breaks the while loop since convergence of the lowest
768       // energy vector field has been reached
769       break;
770
771     }
772
773   }
```

```
774
775     ////////////////////////////////////////////////////////////////////////////
776     // Freeing Memory and Exiting                                             //
777     ////////////////////////////////////////////////////////////////////////////
778
779     // This frees the array of velocity residuals from memory
780     free( velocity_residuals );
781
782     // This frees the array storing the indices of the peaks that correspond to
783     // the lowest energy velocity field from memory
784     free( peak_index_array );
785
786     // This frees the arrays storing the adjacent possible velocity vectors
787     // from memory
788     free( ii_velocity_neighborhood );
789     free( jj_velocity_neighborhood );
790     free( kk_velocity_neighborhood );
791
792     // This frees the updated velocity field arrays from memory
793     free( ii_velocity_updated );
794     free( jj_velocity_updated );
795     free( kk_velocity_updated );
796
797     // This succesfully returns from the function
798     return 0;
799
800 }
```

A.2 Rectilinear Gaussian Fitting

This code takes a cross-correlation array as input and calculates the sub-voxel fit coordinate to the specified peak within the cross-correlation. The fitting is completed using an axis-aligned three-dimensional Gaussian function that is fit using a least squares algorithm calculated from the 3 by 3 by 3 region surrounding the specified peak.

The input array 'cross_correlation' is a one-dimensional array that has a length equal to the product of the length 3 array 'window_size'. The variables 'ii_peak_max', 'jj_peak_max', and 'kk_peak_max' give the subscripted index location of the peak that is being fit. The sub-voxel fit coordinates of the peak are stored in the output variables 'ii_max_subpixel', 'jj_max_subpixel', and 'kk_max_subpixel'. This function will always produce an output, although if any of the 3 by 3 by 3 regions surrounding the peak have zero or negative values, the sub-voxel fit variables will have NaN values.

This function references the 'integer_modulus' function which simply calculates the modulus function for all integer inputs. This is used instead of the remainder '%' operator in C since the remainder operator only functions with non-negative inputs. This allows the periodic boundary conditions of the cross-correlation array (as applied by the discrete Fourier transform) to be utilized for fitting peaks along the boundary of the cross-correlation.

Listing A.2: Rectilinear Gaussian Fitting Function

```
1 void least_squares_rectilinear_gaussian_fit( double cross_correlation[], int
     window_size[], int ii_peak_max, int jj_peak_max, int kk_peak_max, double *
     ii_max_subpixel, double *jj_max_subpixel, double *kk_max_subpixel )
2 {
3
4   // This function calculates the sub-voxel center of the Gaussian function
```

```
5   // given by the 3 x 3 x 3 function 'gaussian_function' using a least
6   // squares fitting method.
7   //
8   // The least squares fitting can be calculated by taking the logarithm of
9   // both sides of a ellipsoidal axis-aligned Gaussian function of the form
10  //
11  //  f(x,y,z)=C*exp(-((x-x0)^2)/(2*sigma_x^2)-((y-y0)^2)/(2*sigma_y^2)-((z-z0
        )^2)/(2*sigma_z^2))
12  //
13  // to create a linear summation.  For a 3 x 3 x 3 array of voxels this
14  // gives 27 equations with 7 variables and is thus an over-determined
15  // system.  If the coordinates of the Gaussian function in each dimension
16  // are taken as { -1, 0, +1 }, then the 'A' matrix of the linear equation
17  // is constant for all input functions.  The least squares fitting can then
18  // be calculated by taking the Moore-Penrose inverse of 'A'; this inverse
19  // matrix consists of simple rational expression coefficients.
20  //
21  // Then since the logarithm was taken of both sides of the Gaussian
22  // function, this means that the matrix summation can be converted to the
23  // logarithm of the product of the function values.
24  //
25  // When this is simplified for the center coordinates of the Gaussian
26  // function, the center coordinates are simple functions of the logarithms
27  // of the products of 3 x 3 x 1 regions of the input array where the slices
28  // are taken in the direction of the dimension for which the center
29  // coordinate is being calculated.
30
31  ////////////////////////////////////////////////////////////////////////////
32  // Initializing Calculation Variables                                     //
33  ////////////////////////////////////////////////////////////////////////////
34
```

```
35  // This initializes relative subscripted indices into the voxels (which may
36  // be above or below the actual array range)
37  int ii_array_rel_index;
38  int jj_array_rel_index;
39  int kk_array_rel_index;
40  // This initializes absolute subscripted indices into the voxels (which
41  // will always lie in the array range)
42  int ii_array_abs_index;
43  int jj_array_abs_index;
44  int kk_array_abs_index;
45  // This initializes a linear index into the voxels
46  unsigned int linear_array_index;
47
48  // This initializes a variable to store the product of the voxels in the
49  // negative direction
50  double product_ngtv;
51  // This initializes a variable to store the product of the voxels in the
52  // zero direction
53  double product_zero;
54  // This initializes a variable to store the product of the voxels in the
55  // positive direction
56  double product_pstv;
57
58  ////////////////////////////////////////////////////////////////////////
59  // First Dimension Least Squares Fitting                              //
60  ////////////////////////////////////////////////////////////////////////
61
62  // This initializes the negative direction voxel product to one
63  product_ngtv = 1.0;
64  // This initializes the zero direction voxel product to one
65  product_zero = 1.0;
```

```
66    // This initializes the positive direction voxel product to one
67    product_pstv = 1.0;
68
69    // This iterates through the second dimension of the array calculating the
70    // voxel product
71    for ( jj_array_rel_index = ( jj_peak_max - 1 ); jj_array_rel_index <= (
          jj_peak_max + 1 ); jj_array_rel_index++ )
72    {
73
74      // This calculates the absolute subscripted matrix index (since the
75      // relative index can go below or above the range of the array)
76      integer_modulus( jj_array_rel_index, window_size[ 1 ], &jj_array_abs_index
            );
77
78      // This iterates through the third dimension of the array calculating
79      // the voxel product
80      for ( kk_array_rel_index = ( kk_peak_max - 1 ); kk_array_rel_index <= (
            kk_peak_max + 1 ); kk_array_rel_index++ )
81      {
82
83        // This calculates the absolute subscripted matrix index (since the
84        // relative index can go below or above the range of the array)
85        integer_modulus( kk_array_rel_index, window_size[ 2 ], &
              kk_array_abs_index );
86
87        // This sets the index of the first dimension absolute array index
88        // equal to the center voxel index minus one
89        integer_modulus( ii_peak_max - 1, window_size[ 0 ], &ii_array_abs_index
              );
90        // This calculates the linear index into the array
91        linear_array_index = ii_array_abs_index * window_size[ 1 ] * window_size
```

```
            [ 2 ] + jj_array_abs_index * window_size[ 2 ] + kk_array_abs_index;
92        // This multiplies the current voxel value into the total product
93        product_ngtv *= cross_correlation[ linear_array_index ];
94
95        // This sets the index of the first dimension absolute array index
96        // equal to the center voxel index
97        ii_array_abs_index = ii_peak_max;
98        // This calculates the linear index into the array
99        linear_array_index = ii_array_abs_index * window_size[ 1 ] * window_size
              [ 2 ] + jj_array_abs_index * window_size[ 2 ] + kk_array_abs_index;
100       // This multiplies the current voxel value into the total product
101       product_zero *= cross_correlation[ linear_array_index ];
102
103       // This sets the index of the first dimension absolute array index
104       // equal to the center voxel index plus one
105       integer_modulus( ii_peak_max + 1, window_size[ 0 ], &ii_array_abs_index
              );
106       // This calculates the linear index into the array
107       linear_array_index = ii_array_abs_index * window_size[ 1 ] * window_size
              [ 2 ] + jj_array_abs_index * window_size[ 2 ] + kk_array_abs_index;
108       // This multiplies the current voxel value into the total product
109       product_pstv *= cross_correlation[ linear_array_index ];
110
111     }
112
113   }
114
115   // This calculates the sub-voxel least squares fit in the first dimension
116   *ii_max_subpixel = ( log( product_ngtv ) - log( product_pstv ) ) / ( 2 * log
          ( product_ngtv * product_pstv ) - 4 * log( product_zero ) ) + ( double )
           ii_peak_max;
```

```
117
118   ////////////////////////////////////////////////////////////////////////////
119   // Second Dimension Least Squares Fitting                                 //
120   ////////////////////////////////////////////////////////////////////////////
121
122   // This initializes the negative direction voxel product to one
123   product_ngtv = 1.0;
124   // This initializes the zero direction voxel product to one
125   product_zero = 1.0;
126   // This initializes the positive direction voxel product to one
127   product_pstv = 1.0;
128
129   // This iterates through the first dimension of the array calculating the
130   // voxel product
131   for ( ii_array_rel_index = ( ii_peak_max - 1 ); ii_array_rel_index <= (
          ii_peak_max + 1 ); ii_array_rel_index++ )
132   {
133
134     // This calculates the absolute subscripted matrix index (since the
135     // relative index can go below or above the range of the array)
136     integer_modulus( ii_array_rel_index, window_size[ 0 ], &ii_array_abs_index
            );
137
138     // This iterates through the third dimension of the array calculating
139     // the voxel product
140     for ( kk_array_rel_index = ( kk_peak_max - 1 ); kk_array_rel_index <= (
          kk_peak_max + 1 ); kk_array_rel_index++ )
141     {
142
143       // This calculates the absolute subscripted matrix index (since the
144       // relative index can go below or above the range of the array)
```

```
145         integer_modulus( kk_array_rel_index, window_size[ 2 ], &
                kk_array_abs_index );
146
147         // This sets the index of the second dimension absolute array index
148         // equal to the center voxel index minus one
149         integer_modulus( jj_peak_max - 1, window_size[ 1 ], &jj_array_abs_index
                );
150         // This calculates the linear index into the array
151         linear_array_index = ii_array_abs_index * window_size[ 1 ] * window_size
                [ 2 ] + jj_array_abs_index * window_size[ 2 ] + kk_array_abs_index;
152         // This multiplies the current voxel value into the total product
153         product_ngtv *= cross_correlation[ linear_array_index ];
154
155         // This sets the index of the second dimension absolute array index
156         // equal to the center voxel index
157         jj_array_abs_index = jj_peak_max;
158         // This calculates the linear index into the array
159         linear_array_index = ii_array_abs_index * window_size[ 1 ] * window_size
                [ 2 ] + jj_array_abs_index * window_size[ 2 ] + kk_array_abs_index;
160         // This multiplies the current voxel value into the total product
161         product_zero *= cross_correlation[ linear_array_index ];
162
163         // This sets the index of the second dimension absolute array index
164         // equal to the center voxel index plus one
165         integer_modulus( jj_peak_max + 1, window_size[ 1 ], &jj_array_abs_index
                );
166         // This calculates the linear index into the array
167         linear_array_index = ii_array_abs_index * window_size[ 1 ] * window_size
                [ 2 ] + jj_array_abs_index * window_size[ 2 ] + kk_array_abs_index;
168         // This multiplies the current voxel value into the total product
169         product_pstv *= cross_correlation[ linear_array_index ];
```

```
170
171     }
172
173   }
174
175   // This calculates the sub-voxel least squares fit in the second dimension
176   *jj_max_subpixel = ( log( product_ngtv ) - log( product_pstv ) ) / ( 2 * log
          ( product_ngtv * product_pstv ) - 4 * log( product_zero ) ) + ( double )
           jj_peak_max;
177
178   ////////////////////////////////////////////////////////////////////////////
179   // Third Dimension Least Squares Fitting                                  //
180   ////////////////////////////////////////////////////////////////////////////
181
182   // This initializes the negative direction voxel product to one
183   product_ngtv = 1.0;
184   // This initializes the zero direction voxel product to one
185   product_zero = 1.0;
186   // This initializes the positive direction voxel product to one
187   product_pstv = 1.0;
188
189   // This iterates through the first dimension of the array calculating the
190   // voxel product
191   for ( ii_array_rel_index = ( ii_peak_max - 1 ); ii_array_rel_index <= (
          ii_peak_max + 1 ); ii_array_rel_index++ )
192   {
193
194     // This calculates the absolute subscripted matrix index (since the
195     // relative index can go below or above the range of the array)
196     integer_modulus( ii_array_rel_index, window_size[ 0 ], &ii_array_abs_index
            );
```

```
197
198     // This iterates through the second dimension of the array calculating
199     // the voxel product
200     for ( jj_array_rel_index = ( jj_peak_max - 1 ); jj_array_rel_index <= (
            jj_peak_max + 1 ); jj_array_rel_index++ )
201     {
202
203       // This calculates the absolute subscripted matrix index (since the
204       // relative index can go below or above the range of the array)
205       integer_modulus( jj_array_rel_index, window_size[ 1 ], &
              jj_array_abs_index );
206
207       // This sets the index of the third dimension absolute array index
208       // equal to the center voxel index minus one
209       integer_modulus( kk_peak_max - 1, window_size[ 2 ], &kk_array_abs_index
              );
210       // This calculates the linear index into the array
211       linear_array_index = ii_array_abs_index * window_size[ 1 ] * window_size
              [ 2 ] + jj_array_abs_index * window_size[ 2 ] + kk_array_abs_index;
212       // This multiplies the current voxel value into the total product
213       product_ngtv *= cross_correlation[ linear_array_index ];
214
215       // This sets the index of the third dimension absolute array index
216       // equal to the center voxel index
217       kk_array_abs_index = kk_peak_max;
218       // This calculates the linear index into the array
219       linear_array_index = ii_array_abs_index * window_size[ 1 ] * window_size
              [ 2 ] + jj_array_abs_index * window_size[ 2 ] + kk_array_abs_index;
220       // This multiplies the current voxel value into the total product
221       product_zero *= cross_correlation[ linear_array_index ];
222
```

```
223         // This sets the index of the third dimension absolute array index
224         // equal to the center voxel index plus one
225         integer_modulus( kk_peak_max + 1, window_size[ 2 ], &kk_array_abs_index
                );
226         // This calculates the linear index into the array
227         linear_array_index = ii_array_abs_index * window_size[ 1 ] * window_size
                [ 2 ] + jj_array_abs_index * window_size[ 2 ] + kk_array_abs_index;
228         // This multiplies the current voxel value into the total product
229         product_pstv *= cross_correlation[ linear_array_index ];
230
231       }
232
233     }
234
235     // This calculates the sub-voxel least squares fit in the third dimension
236     *kk_max_subpixel = ( log( product_ngtv ) - log( product_pstv ) ) / ( 2 * log
          ( product_ngtv * product_pstv ) - 4 * log( product_zero ) ) + ( double )
           kk_peak_max;
237
238 }
```

A.3 Diagonal Gaussian Fitting

This code takes a cross-correlation array as input and calculates the sub-voxel fit coordinate to the specified peak within the cross-correlation. The fitting is completed using a three-dimensional Gaussian function that may be aligned along any axis and is fit using a least squares algorithm calculated from the 3 by 3 by 3 region surrounding the specified peak.

The input array ‘cross_correlation’ is a one-dimensional array that has a length equal to the

product of the length 3 array 'window_size'. The variables 'ii_peak_max', 'jj_peak_max', and 'kk_peak_max' give the subscripted index location of the peak that is being fit. The sub-voxel fit coordinates of the peak are stored in the output variables 'ii_max_subpixel', 'jj_max_subpixel', and 'kk_max_subpixel'. This function will always produce an output, although if any of the 3 by 3 by 3 regions surrounding the peak have zero or negative values, the sub-voxel fit variables will have NaN values. This function references the 'integer_modulus' function which simply calculates the modulus function for all integer inputs. This is used instead of the remainder '%' operator in C since the remainder operator only functions with non-negative inputs. This allows the periodic boundary conditions of the cross-correlation array (as applied by the discrete Fourier transform) to be utilized for fitting peaks along the boundary of the cross-correlation.

Listing A.3: Diagonal Gaussian Fitting Function

```
1  void least_squares_diagonal_gaussian_fit( double cross_correlation[], int
       window_size[], int ii_peak_max, int jj_peak_max, int kk_peak_max, double *
       ii_max_subpixel, double *jj_max_subpixel, double *kk_max_subpixel )
2  {
3
4    // This function fits an ellipsoidal Gaussian function to the peak of the
5    // input cross-correlation peak 3 x 3 x 3 region.  The Gaussian function
6    // may be aligned along any axis (as opposed to being aligned along the x,
7    // y, z axes as is more standard).
8    //
9    // The least squares fitting can be calculated by taking the logarithm of
10   // both sides of the non-axis-aligned ellipsoidal Gaussian function of the
11   // form
12   //
13   //  f(x,y,z)=C*exp(-a1*(x-x0)^2-a2*(y-y0)^2-a3*(z-z0)^2
```

```
14  //            -a4*(x-x0)*(y-y0)-a5*(x-x0)*(z-z0)-a6*(y-y0)*(z-z0))
15  //
16  // to create a linear summation.  For a 3 x 3 x 3 array of voxels this
17  // gives 27 equations with 10 variables and is thus an over-determined
18  // system.  If the coordinates of the Gaussian function in each dimension
19  // are taken as { -1, 0, +1 }, then the 'A' matrix of the linear equation
20  // is constant for all input functions.  The least squares fitting can then
21  // be calculated by taking the Moore-Penrose inverse of 'A'; this inverse
22  // matrix consists of simple rational expression coefficients.
23  //
24  // Then since the logarithm was taken of both sides of the Gaussian
25  // function, this means that the matrix summation can be converted to the
26  // logarithm of the product of the function values.  This yields 9
27  // equations related to the center coordinates (the 10th equation gives the
28  // peak intensity value C - which isn't needed).  The coefficient vector b
29  // is a function of the coefficient vector a and the center coordinates by
30  // the following equations
31  //
32  //  b1  = -a1
33  //  b2  = -a2
34  //  b3  = -a3
35  //  b4  = -a4
36  //  b5  = -a5
37  //  b6  = -a6
38  //  b7  = 2*a1*x0+a4*y0+a5*z0
39  //  b8  = a4*x0+2*a2*y0+a6*z0
40  //  b9  = a5*x0+a6*y0+2*a3*z0
41  //  b10 = log(C)-a1*x0^2-a2*y0^2-a3*z0^2-a4*x0*y0-a5*x0*z0-a6*y0*z0
42  //
43  // Finally, these 9 equations can be related to the center coordinates by a
44  // system of three linear equations which can then be solved for the 3
```

```
45  // center coordinates.
46
47  ////////////////////////////////////////////////////////////////////////////
48  // Initializing Calculation Variables                                     //
49  ////////////////////////////////////////////////////////////////////////////
50
51  // This initializes a variable to store the 3 x 3 x 3 region surrounding
52  // the cross-correlation peak
53  double g[ 27 ];
54
55  // This initializes an array to store the fit polynomial coefficients
56  double b[ 9 ];
57
58  // This initializes relative subscripted indices into the voxels (which may
59  // be above or below the actual array range)
60  int ii_array_rel_index;
61  int jj_array_rel_index;
62  int kk_array_rel_index;
63  // This initializes absolute subscripted indices into the voxels (which
64  // will always lie in the array range)
65  int ii_array_abs_index;
66  int jj_array_abs_index;
67  int kk_array_abs_index;
68  // This initializes a linear index into the voxels
69  unsigned int linear_cross_correlation_index;
70
71  // This initializes an index into the 3 x 3 x 3 Gaussian vector
72  unsigned int linear_peak_index = 0;
73
74  // This initializes a variable to store the maximum value of the
75  // cross-correlation to normalize the intensity values so that they are all
```

```
76    // approximately one to minimize numerical error
77    double cross_correlation_max_inverse;
78
79    // This initializes a variable equal to 1/18 for quick multiplication
80    // during the coefficient calculation
81    double one_eighteenth = 1.0 / 18.0;
82    // This initializes a variable equal to 1/12 for quick multiplication
83    // during the coefficient calculation
84    double one_twelfth = 1.0 / 12.0;
85
86    ////////////////////////////////////////////////////////////////////////////
87    // Extracting and Normalizing 3 x 3 x 3 Peak Region                       //
88    ////////////////////////////////////////////////////////////////////////////
89
90    // This calculates the linear index of the cross-correlation peak
91    linear_cross_correlation_index = ii_peak_max * window_size[ 1 ] *
          window_size[ 2 ] + jj_peak_max * window_size[ 2 ] + kk_peak_max;
92    // This extracts the value of the cross-correlation peak for normalizing
93    // the intensity values
94    cross_correlation_max_inverse = 1.0 / cross_correlation[
          linear_cross_correlation_index ];
95
96    // This sets the index into the 3 x 3 x 3 Gaussian vector as zero
97    linear_peak_index = 0;
98
99    // This iterates through the first dimension of the array calculating the
100   // voxel product
101   for ( ii_array_rel_index = ( ii_peak_max - 1 ); ii_array_rel_index <= (
          ii_peak_max + 1 ); ii_array_rel_index++ )
102   {
103
```

```
104     // This calculates the absolute subscripted matrix index (since the
105     // relative index can go below or above the range of the array)
106     integer_modulus( ii_array_rel_index, window_size[ 0 ], &ii_array_abs_index
            );
107
108     // This iterates through the second dimension of the array calculating
109     // the voxel product
110     for ( jj_array_rel_index = ( jj_peak_max - 1 ); jj_array_rel_index <= (
            jj_peak_max + 1 ); jj_array_rel_index++ )
111     {
112
113       // This calculates the absolute subscripted matrix index (since the
114       // relative index can go below or above the range of the array)
115       integer_modulus( jj_array_rel_index, window_size[ 1 ], &
              jj_array_abs_index );
116
117       // This iterates through the third dimension of the array
118       // calculating the voxel product
119       for ( kk_array_rel_index = ( kk_peak_max - 1 ); kk_array_rel_index <= (
              kk_peak_max + 1 ); kk_array_rel_index++ )
120       {
121
122         // This calculates the absolute subscripted matrix index (since
123         // the relative index can go below or above the range of the
124         // array)
125         integer_modulus( kk_array_rel_index, window_size[ 2 ], &
                kk_array_abs_index );
126
127         // This calculates the linear index into the array
128         linear_cross_correlation_index = ii_array_abs_index * window_size[ 1 ]
                * window_size[ 2 ] + jj_array_abs_index * window_size[ 2 ] +
```

```
            kk_array_abs_index;
129
130         // This copies the current value of the cross-correlation array
131         // into the gaussian vector
132         g[ linear_peak_index ] = cross_correlation_max_inverse *
                cross_correlation[ linear_cross_correlation_index ];
133
134         // This increments the linear peak index for the next voxel
135         linear_peak_index++;
136
137       }
138
139     }
140
141   }
142
143   ////////////////////////////////////////////////////////////////////////////
144   // Calculating the Guassian Function Coefficients                         //
145   ////////////////////////////////////////////////////////////////////////////
146
147   // This calculates the 1st fit polynomial coefficient of the least squares
148   // diagonal Gaussian function
149   b[ 0 ] = one_eighteenth * ( log( g[ 0 ] * g[ 1 ] * g[ 2 ] * g[ 3 ] * g[ 4 ]
          * g[ 5 ] * g[ 6 ] * g[ 7 ] * g[ 8 ] * g[ 18 ] * g[ 19 ] * g[ 20 ] * g[
          21 ] * g[ 22 ] * g[ 23 ] * g[ 24 ] * g[ 25 ] * g[ 26 ] )
150   - 2 * log( g[ 9 ] * g[ 10 ] * g[ 11 ] * g[ 12 ] * g[ 13 ] * g[ 14 ] * g[ 15
          ] * g[ 16 ] * g[ 17 ] ) );
151   // This calculates the 2nd fit polynomial coefficient of the least squares
152   // diagonal Gaussian function
153   b[ 1 ] = one_eighteenth * ( log( g[ 0 ] * g[ 1 ] * g[ 2 ] * g[ 6 ] * g[ 7 ]
          * g[ 8 ] * g[ 9 ] * g[ 10 ] * g[ 11 ] * g[ 15 ] * g[ 16 ] * g[ 17 ] * g[
```

```
      18 ] * g[ 19 ] * g[ 20 ] * g[ 24 ] * g[ 25 ] * g[ 26 ] )
154   - 2 * log( g[ 3 ] * g[ 4 ] * g[ 5 ] * g[ 12 ] * g[ 13 ] * g[ 14 ] * g[ 21 ]
         * g[ 22 ] * g[ 23 ] ) );
155   // This calculates the 3rd fit polynomial coefficient of the least squares
156   // diagonal Gaussian function
157   b[ 2 ] = one_eighteenth * ( log( g[ 0 ] * g[ 2 ] * g[ 3 ] * g[ 5 ] * g[ 6 ]
         * g[ 8 ] * g[ 9 ] * g[ 11 ] * g[ 12 ] * g[ 14 ] * g[ 15 ] * g[ 17 ] * g[
          18 ] * g[ 20 ] * g[ 21 ] * g[ 23 ] * g[ 24 ] * g[ 26 ] )
158   - 2 * log( g[ 1 ] * g[ 4 ] * g[ 7 ] * g[ 10 ] * g[ 13 ] * g[ 16 ] * g[ 19 ]
         * g[ 22 ] * g[ 25 ] ) );
159   // This calculates the 4th fit polynomial coefficient of the least squares
160   // diagonal Gaussian function
161   b[ 3 ] = one_twelfth * ( log( g[ 0 ] * g[ 1 ] * g[ 2 ] * g[ 24 ] * g[ 25 ] *
          g[ 26 ] )
162   - log( g[ 6 ] * g[ 7 ] * g[ 8 ] * g[ 18 ] * g[ 19 ] * g[ 20 ] ) );
163   // This calculates the 5th fit polynomial coefficient of the least squares
164   // diagonal Gaussian function
165   b[ 4 ] = one_twelfth * ( log( g[ 0 ] * g[ 3 ] * g[ 6 ] * g[ 20 ] * g[ 23 ] *
          g[ 26 ] )
166   - log( g[ 2 ] * g[ 5 ] * g[ 8 ] * g[ 18 ] * g[ 21 ] * g[ 24 ] ) );
167   // This calculates the 6th fit polynomial coefficient of the least squares
168   // diagonal Gaussian function
169   b[ 5 ] = one_twelfth * ( log( g[ 0 ] * g[ 8 ] * g[ 9 ] * g[ 17 ] * g[ 18 ] *
          g[ 26 ] )
170   - log( g[ 2 ] * g[ 6 ] * g[ 11 ] * g[ 15 ] * g[ 20 ] * g[ 24 ] ) );
171   // This calculates the 7th fit polynomial coefficient of the least squares
172   // diagonal Gaussian function
173   b[ 6 ] = one_eighteenth * ( log( g[ 18 ] * g[ 19 ] * g[ 20 ] * g[ 21 ] * g[
         22 ] * g[ 23 ] * g[ 24 ] * g[ 25 ] * g[ 26 ] )
174   - log( g[ 0 ] * g[ 1 ] * g[ 2 ] * g[ 3 ] * g[ 4 ] * g[ 5 ] * g[ 6 ] * g[ 7 ]
          * g[ 8 ] ) );
```

```
175   // This calculates the 8th fit polynomial coefficient of the least squares
176   // diagonal Gaussian function
177   b[ 7 ] = one_eighteenth * ( log( g[ 6 ] * g[ 7 ] * g[ 8 ] * g[ 15 ] * g[ 16
          ] * g[ 17 ] * g[ 24 ] * g[ 25 ] * g[ 26 ] )
178   - log( g[ 0 ] * g[ 1 ] * g[ 2 ] * g[ 9 ] * g[ 10 ] * g[ 11 ] * g[ 18 ] * g[
          19 ] * g[ 20 ] ) );
179   // This calculates the 9th fit polynomial coefficient of the least squares
180   // diagonal Gaussian function
181   b[ 8 ] = one_eighteenth * ( log( g[ 2 ] * g[ 5 ] * g[ 8 ] * g[ 11 ] * g[ 14
          ] * g[ 17 ] * g[ 20 ] * g[ 23 ] * g[ 26 ] )
182   - log( g[ 0 ] * g[ 3 ] * g[ 6 ] * g[ 9 ] * g[ 12 ] * g[ 15 ] * g[ 18 ] * g[
          21 ] * g[ 24 ] ) );
183
184   ////////////////////////////////////////////////////////////////////////////
185   // Calculating the Guassian Function Center Coordinate                    //
186   ////////////////////////////////////////////////////////////////////////////
187
188   // This calculates the least squares fit Gaussian first dimension peak
189   // center coordinate
190   *ii_max_subpixel = ( pow( b[ 5 ], 2 ) * b[ 6 ] + 2 * b[ 2 ] * b[ 3 ] * b[ 7
          ] - b[ 5 ] * ( b[ 4 ] * b[ 7 ] + b[ 3 ] * b[ 8 ] ) + 2 * b[ 1 ] * (  b[
          4 ] * b[ 8 ] - 2 * b[ 2 ] * b[ 6 ] ) )
191   / ( 8 * b[ 0 ] * b[ 1 ] * b[ 2 ] - 2 * b[ 2 ] * pow( b[ 3 ], 2 ) - 2 * b[ 1
          ] * pow( b[ 4 ], 2 ) + 2 * b[ 3 ] * b[ 4 ] * b[ 5 ] - 2 * b[ 0 ] * pow(
          b[ 5 ], 2 ) ) + ( double ) ii_peak_max;
192   // This calculates the least squares fit Gaussian second dimension peak
193   // center coordinate
194   *jj_max_subpixel = ( pow( b[ 4 ], 2 ) * b[ 7 ] + 2 * b[ 2 ] * ( b[ 3 ] * b[
          6 ] - 2 * b[ 0 ] * b[ 7 ] ) + 2 * b[ 0 ] * b[ 5 ] * b[ 8 ] - b[ 4 ] * (
          b[ 5 ] * b[ 6 ] + b[ 3 ] * b[ 8 ] ) )
195   / ( 8 * b[ 0 ] * b[ 1 ] * b[ 2 ] - 2 * b[ 2 ] * pow( b[ 3 ], 2 ) - 2 * b[ 1
```

```
    ] * pow( b[ 4 ], 2 ) + 2 * b[ 3 ] * b[ 4 ] * b[ 5 ] - 2 * b[ 0 ] * pow(
    b[ 5 ], 2 ) ) + ( double ) jj_peak_max;
196 // This calculates the least squares fit Gaussian third dimension peak
197 // center coordinate
198 *kk_max_subpixel = ( 2 * b[ 0 ] * b[ 5 ] * b[ 7 ] - b[ 3 ] * ( b[ 5 ] * b[ 6
     ] + b[ 4 ] * b[ 7 ] ) + pow( b[ 3 ], 2 ) * b[ 8 ] + 2 * b[ 1 ] * ( b[ 4
     ] * b[ 6 ] - 2 * b[ 0 ] * b[ 8 ] ) )
199 / ( 8 * b[ 0 ] * b[ 1 ] * b[ 2 ] - 2 * b[ 2 ] * pow( b[ 3 ], 2 ) - 2 * b[ 1
    ] * pow( b[ 4 ], 2 ) + 2 * b[ 3 ] * b[ 4 ] * b[ 5 ] - 2 * b[ 0 ] * pow(
    b[ 5 ], 2 ) ) + ( double ) kk_peak_max;
200
201 }
```