The Pennsylvania State University

The Graduate School

Department of Mechanical and Nuclear Engineering

DEVELOPMENT OF A HIGH FRAME RATE, BRIGHT FLASH NEUTRON IMAGING METHOD FOR RAPID, TRANSIENT PROCESSES

A Thesis in

Nuclear Engineering

by

Chad A. Lani

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2019

The thesis of Chad A. Lani was reviewed and approved* by the following:

Marek Flaska
Assistant Professor of Nuclear Engineering
Thesis Co-advisor

Robert Zboray
Group Leader X-ray Imaging at Swiss Federal Laboratories for Materials Science and Technology
Thesis Co-advisor

William Walters
Assistant Professor of Nuclear Engineering

Arthur T. Motta
Professor and Chair of Nuclear Engineering

*Signatures are on file in the Graduate School

ABSTRACT

The objective of this work was to design and assess a novel digital high-speed imaging system in conjunction with neutron pulses from the Penn State Breazeale Reactor (PSBR) called bright flashes. These pulses are tens of milliseconds long with a peak power up to 761.0 ± 12.44 MW. This technique is like x-ray imaging by being capable of imaging items that cannot be capture with visual imaging but separates itself because the contrast is not dependent on material density but rather hydrogen content. The imaging system is capable of acquiring images up to 4000 frames per second (fps) frame rates and is thus capable of capturing rapid transient processes without motion blur. The process used to test the bright flash neutron imaging technique was air-water two-phase flow contained in a bubbler. To determine and to put into context the capabilities of this bright flash imaging system, the data acquired from Penn State were compared to images that were taken by using a continuous neutron source, at the CG-1D beamline of the High Flux Isotope Reactor (HFIR) of Oak Ridge National Laboratory. Images acquired ranged from acquisition speeds of 60 fps to 4,000 fps. The images were evaluated based on calculating the signal-to-noise and the contrast-to-noise ratios to determine their quality. The quality for the bright flash images was on average four to five times greater than that of the HFIR images and was determined that at higher frame rates bright flash imaging is more beneficial. Bright flash imaging has a limit on the duration of the acquisition being limited to tens of milliseconds of the pulse duration, nevertheless this is no limitation for short, rapid transient processes. In conclusion, both imaging methods have applications for numerous scenarios, but for rapid transient processes and frame rates greater than 500 fps the bright flash imaging is a superior method.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Many thanks to all who helped aid the work in this thesis. Thank you to the Radiation Science and Engineering Center staff including Dr. Ünlü and Alison Portanova who graciously allowed the work to be conducted at the Penn State Breazeale Reactor and helped with the data acquisition. Additional thanks to the scientists at the CG-1D beamline at the High Flux Isotope Reactor for their help. Special thanks to Jean Bilheux for his help with the normalization code and to Hassina Bilheux for her help with the imaging equipment and for hosting for duration of testing at Oak Ridge National Laboratory. Finally, thank you to Dr. Zboray who introduced this project and helped with every step on the way.

Chapter 1

Introduction

The neutron, a subatomic particle of no electric charge that stabilizes nuclei, was first discovered by James Chadwick in 1932. Within three years of the neutron discovery, the first neutron radiograph, an image created with the aid of radiation, was generated [1]. The first series of neutron images were taken by H. Kallmann and E. Kuhn [2]. Kallmann and Kuhn took images with both x-rays and neutrons in order to compare the differences that the two methods can result in. It was not long until neutron imaging became a more broadly utilized method for nondestructive analysis (NDA) due to its ability to penetrate materials of high density and show the internal structure of sensitive equipment that cannot be dismantled [3].

Today neutron imaging is still being utilized as a method of NDA and as a tool for research. Some of the main NDA industrial uses of neutron imaging are looking at lithium ion batteries, ensuring that the coolant channels in turbine blades of airplanes are clear of obstructions and verifying that microcracks are not forming in equipment. In the area of research, neutron imaging has been used in horticulture to observe how plants absorb water through their roots, and in hydrogen fuel cells to study how water flows through and potentially freezes within the fuel cell [4]. Neutron imaging in general is complementary to X-ray imaging enabling a different contrast due to different elemental sensitivity. Unlike X-ray imaging, neutron imaging gives high contrast for several light elements while it can penetrate heavy metals such as lead relatively easily.

Even though neutron imaging has advanced significantly since its inception, research continues to expand the process and capabilities of this form of imaging. High speed neutron imaging, defined in this thesis by an acquisition speed of greater than 100 frames per second (fps), is an area of neutron imaging that has not been developed as extensively as static imaging due to

the lack of refined technology that exists today. High speed imaging is useful for rapid transient processes that could not be captured with lower frame rates.

One method that has been used to circumvent the need for high frame imaging is phase lock imaging. Phase lock imaging is a technique that can be used when the subject that is being imaged is a fast-moving cyclical process, such as an engine piston or a rotating fan [32]. The imaging setup is made to trigger when the subject is in a specific phase of the cycle. Many images over many cycles are obtained and combined to form a statistically robust image of that phase in the cycle. The triggering point can be altered to obtain a different phase point. After sufficient phase points are captured, the entire cycle is effectively imaged. Limitations to phase lock imaging are that the rapid process must be cyclical, and that the triggering mechanism must have a sufficiently small time uncertainty or the image combination process will fail. This thesis focuses on expanding the functionality of neutron imaging to capture rapid non-repeating processes such as those observed in two-phase flows.

High frame rate imaging has multiple benefits over static or low frame rate imaging. The major benefit is the capability of capturing faster processes with minimal motion artifacts, features that were not present in the original object. In a rapid process, if the frame rate used to image the process is too slow, then there will be significant artifacts in the image. These artifacts are usually in the form of ghosting or blurring of the objects as they move across the field of view (FOV). The blurring of the object movement can be reduced as the frame rate is increased. At a certain frame rate the artifacts can be smaller than the native resolution of the imaging system.

High speed imaging has not been utilized often due to the major limitations that have hindered its applicability in the past. The first limitation was that the technology was insufficient to capture frames at high speeds, while maintaining a resolution with useful results. This limitation has been mitigated by the advent of high efficiency low light cameras and more efficient scintillators that can allow for greater neutron conversion to light. The second limitation was due

to the fluxes of available neutron sources. The neutron flux is important since the number of neutrons interacting with the scintillator directly correlates to the amount of light that is produced. As the frame rate of a camera is increased, the amount of light per frame is reduced. Consequently, images can become too dark or noisy to discern features. In summary, an increase in the number of neutrons produces more light, which permits the higher frame rate to be used. This thesis looks at a pulsed TRIGA reactor and a high flux continuous source to counteract these limitations.

Literature Review

High frame rate imaging has been investigated as early as 1972 by Robinson and Barton [5]. Robinson and Barton the Oregon State University TRIGA Mark II reactor to generate a neutron pulse that reached a peak power of 3.8 GW and a peak central flux of 10^{17} n/cm^2-s in the reactor core. This method of reactor pulsing for the sake of neutron imaging has been since informally named “bright flash” imaging. The object of interest was a strip of cadmium moving at a speed of approximately 60 mph. The cadmium strip was measured at 10,000 fps with a film-based system and an image intensifier.

Several years later, Bossi continued the work on high-speed neutron imaging with Robinson and Barton [6]. Bossi utilized the Oregon State University TRIGA Mark II reactor to create a neutron pulse of 4.2 x 10^{11} n/cm^2-s at the exit of the beam port. Bossi also utilized a film based analog imaging system and an image intensifier to obtain neutron images as high as 10,000 fps. Unfortunately, such film needs a longer exposure time and the images contained high statistical errors and were of mediocre quality. However, Bossi et al. did prove that the bright flash imaging has a potential for the future.

Continuing the research at Oregon State University in 1981, Shou-Kong Wang performed his studies using the TRIGA Mark II reactor to take radiographs of two-phase flows [7]. Wang used

much of the same equipment that Bossi used with the method of bright flash imaging. Wang imaged boiling water resulting from a heating element placed inside a water bath. The frame rates at which the flow was imaged was 250 fps to 5,000 fps. Due to the analog imaging setup, Wang was not capable of performing enhancement and editing techniques of eliminating imaging artifacts and normalizing.

In Japan, high frame rate neutron imaging was being utilized to view molten metal-water interactions and the development of new reactor safety systems that involved water injectors aimed at a molten core [8, 9]. Both Nakamura and Sibamoto used the same high-speed video camera, recorder, and neutron source. Unlike a bright flash from a TRIGA or other pulsing reactors, the Japan Research Reactor 3M of the Japan Atomic Energy Research Institute had a high flux steady state neutron beam, meaning that high frame rate imaging can be accomplished without the varying intensity of the neutron source. Nakamura was focused on better understanding possible severe accident scenarios that may occur during reactor operation. This work used a maximum frame rate of 500 fps and looked at how molten metal acts as it is injected into a pool of water. Sibamoto was investigating a new safety feature that could be utilized in next generation reactors. The safety device was a water injector that could act as an additional heat exchanger in the case of core melting. Sibamoto used a maximum frame rate of 1,125 fps to obtain his results. Both Nakamura and Sibamoto showed that high frame rate imaging can be acquired with a continuous neutron source and that there are more applications for this technology.

In 2014 Tresmin et al. [10] and Lerche et al. [11] continued the work of bright flash imaging. Both Tresmin and Lerch used the pulsing capabilities of the McClellan Nuclear Research Center's (MNRC) TRIGA reactor. Tresmin and Lerche did not perform high-frame rate imaging, but rather imaged static objects to show that the method was still feasible. They utilized a high-resolution neutron counting detector along with microchannel plates in order to demonstrate the imaging capabilities of the MNRC facility.

Some of the latest publications of high-frame rate neutron imaging have been performed by Zboray and Trtik at Paul Scherrer Institute (PSI) in Switzerland [12]. Zboray and Trtik used a scientific complementary metal-oxide-semiconductor (sCMOS) digital camera to reach a frame rate of 800 fps. The neutron source that was used at PSI was the ICON cold neutron imaging beam line which is part of the continuous spallation neutron source SINQ. Zboray and Trtik focused on imaging air-water two phase flow that was created inside an aluminum bubbler. The resulting images were capable of showing the coalescence and breakup of the bubbles with minimal motion artifacts.

The researches that have worked on high-frame rate imaging have proved that there are viable methods to achieve the frame rates and neutron fluxes required. In addition to the feasibility of the method, there are many applications for the method. However, the researchers of high-frame rate neutron imaging have also shown that the method is not perfect and that additional research is required for further improvements.

Goals and Methodology

The objective of this work is to develop a new high speed neutron imaging system, and evaluate its functionality using the neutrons produced from the thermalized bright flash of the Penn State Breazeale Reactor and the continuous cold source of the High Flux Isotope Reactor at the Oak Ridge National Laboratory.

These experiments at both research facilities utilized air-water two-phase flow in a flat aluminum bubbler that will be described in greater detail in the subsequent section. The bubbler is a simple apparatus that generates the two-phase flow conditions required to demonstrate this new technique of high speed imaging and to be able to analyze and quantify the abilities of the imaging system.

Potential Applications

In general, the method can be used for any rapid, transient process where the combination of the penetrating power of neutrons and their sensitivity for specific elements is advantageous. Any element that has a high interaction cross section with neutrons can provide good contrast. These materials include but not limited to hydrogen, boron, lithium and some isotopes of uranium and xenon.

In the case of two-phase flows, the uses of high speed imaging are many. The method can be used to further research to better understand the dynamics of gas-liquid interactions or to enable observation of the dynamics of the two-phase flow under pressurized conditions. Other specific applications for two-phase flows include capturing the rapid transients water hammers in piping or steam explosions upon molten metal-water interaction, relevant for nuclear energy safety. The advantage of this imaging technique is that the experimental apparatus is no limited to the use of specialized transparent piping.

Chapter 2

Pulsing the Penn State Breazeale Reactor

To obtain the high neutron fluxes that are required for high-frame rate imaging, the Penn State Breazeale Reactor (PSBR) was utilized. The PSBR is a 1MW TRIGA pool type reactor that is capable of producing a neutron pulse. The specifics of how the pulse is made, the nuclear physics, the PSBR, and the imaging beam line are described later in this chapter.

The Penn State Breazeale Nuclear Reactor

The PSBR was first licensed in 1955 as a 100 kW MTR pool type reactor and was the first reactor that was licensed by the Atomic Energy Commission [15]. By 1960 the operating power was increased to 200 kW. Within five years, the MTR core was replaced with a new 1 MW TRIGA core that was rated for pulses up to 2000 MW. The TRIGA core required fuel that was less than 20% enriched rather than the 93% enrichment of the MTR core. The reactor core was capable of moving within the coolant pool and was upgraded to allow X, Y, and angular movements. Due to the core exchange from the MTR to the TRIGA, several of the beam ports, “holes” in the reactor containment to allow radiation, in the form of a beamline, through for various uses, were no longer aligned due to the new height of the TRIGA core. In the summer of 2018, renovations to the reactor facility initiated to create a larger D_2O moderator tank with new beam ports that are focused on the core center [16]. Today, the PSBR is available for a variety of experiments and research such as neutron activation analysis, neutron radiography, food and plant irradiation, radioisotope production, and much more.

Pulsing Physics of the TRIGA Reactor

Reactor pulses are most frequently caused by a control rod ejection, removal of neutron absorbing material that protects the reactor, which is a hazard if it is present in a commercial reactor. In some research reactors, a reactor pulse is just another way that the reactor can be utilized as tool. During the ejection, a large amount of reactivity in inserted by removal of the control rod and the reactor becomes prompt critical, meaning that the reactor can be self-sustaining with prompt neutrons alone, which are neutrons created directly from fission. Since delayed neutrons, neutrons that are created by decaying radionuclides, are also part of a nuclear reaction, the number of free neutrons in the chain reaction increase exponentially causing the power produced to increase. With increased power, the temperature of the fuel increases and because of negative feedback, mechanism that cause negative reactivity, from the increase in temperature, the power is reduced and the reactivity inserted is reduced to or below zero [13].

To approximate the conditions of the core during a transient the Fuchs-Nordheim model is useful as an adiabatic approximation for a reactor transient that works well when a large amount of energy is deposited and heat transfer from the fuel to the coolant is slow:

$$\delta T(t) = C_{QT} \int_0^t P(t') - P_0 dt', \quad \textbf{(2-1)}$$

, where $\delta T(t)$ is the average fuel temperature rise after the onset of a transient, C_{QT} is the conversion factor between energy and temperature, and *P* is the total power in the core [13]. The model assumes that heat transfer out of the rod does not increase, as opposed to no heat transfer at all. From the Fuchs-Nordheim model, various quantities of interest can be derived, such as the maximum power of the transient and the pulse width itself. Respectively, the maximum power and pulse width can be calculated by:

$$P_m = -\frac{(\rho-\beta)^2}{2\Lambda\gamma} P_n \text{ and} \quad \textbf{(2-2)}$$

$$\delta t = \frac{4\Lambda}{\rho - \beta}, \quad \textbf{(2-3)}$$

where P_m is the maximum power, ρ is the reactivity inserted, β is the delayed neutron fraction, Λ is the neutron generation time, γ is the fuel temperature feedback coefficient, P_n is the nominal power before the reactivity insertion, and δt is the pulse width. The pulsing physics is explained in greater detail in [14].

During a control rod ejection, such as the one used by the TRIGA reactor, the subsequent pulse is named for the amount of reactivity that is inserted into the reactor core during the transient. For this thesis, 13 pulses were utilized with two different pulse types, 2-dollar and 2.5-dollar. The term "dollar" is utilized as a measure of how much reactivity is added to a nuclear reactor. Traditionally, reactivity [ρ] in the amount of β is called 1 $:

$$\rho[\$] = \frac{\rho[\,]}{\beta}, \quad \textbf{(2-4)}$$

where β is the effective fraction of delayed neutrons [13]. This measurement is normalized to the amount of reactivity that is needed to make a reactor prompt critical, 1-dollar. The 2-dollar pulse should be shorter and wider than the 2.5-dollar pulse as per the Fuchs-Nordheim model. Using two of the pulses captured by a compensated ion chamber neutron detector, the 2-dollar pulse has a peak value of 284.0 ± 6.006 MW and the full width at half maximum (FWHM) of 24.47 ± 2.573 ms, while the latter has a peak value of 761.0 ± 12.44 MW and a FWHM of 15.38 ± 0.2727 ms. Knowing the neutron flux of the TRIGA reactor while operating, it can be assumed that the neutron flux is directly proportional to the reactor power. During normal operations the maximum steady flux is approximately $1.7 * 10^7$ neutrons-cm^{-2}-s^{-1}, consequently the 2-dollar pulse results in a peak neutron flux of $4.75*10^9$ n-cm^{-2}-s^{-1} and the 2.5-dollar pulse results in a flux of $1.3*10^{10}$ n-cm^{-2}-s^{-1}. The neutron pulsing can be repeated every 12-15 minutes and slight variations in the peak power and FWHM occur due to the non-exact reproduction of the initial conditions. Figure 2-1 shows the two different pulse types plotted as normalized power amplification against time.

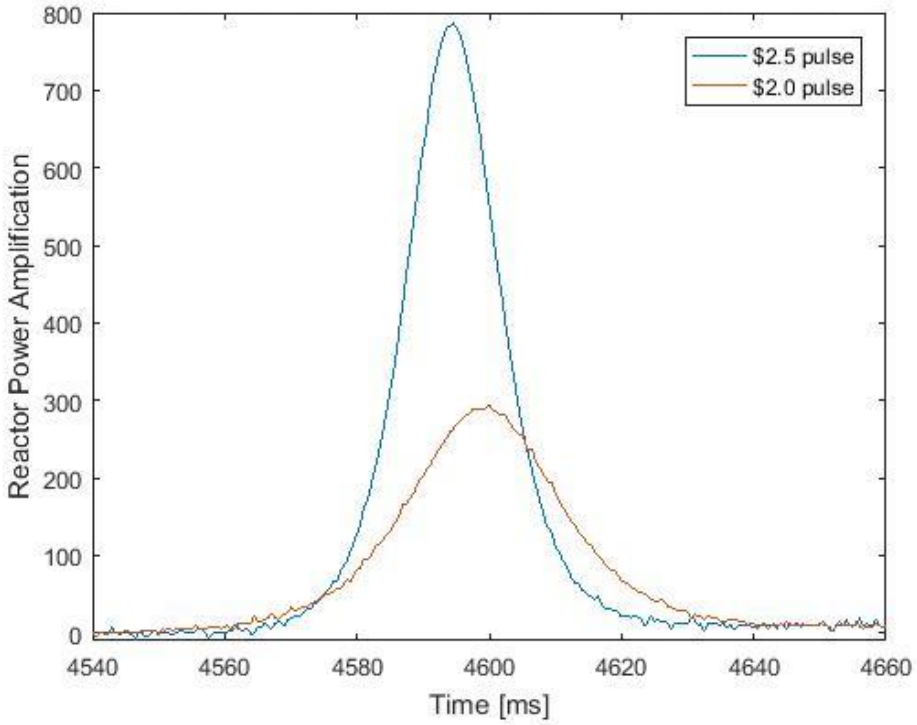

Figure **2-1**: 2 $ and 2.5 $ neutron pulses measured by a compensated ion chamber (CIC) neutron detector placed in the vicinity of the reactor core.

Imaging Beam Line at Penn State

The imaging beamline that is used at the PSBR corelates to an ASTM E 544 Category 1 facility. The beam port exits into a structure made of lead, called a beam cave, with a beam dump against the back wall that absorbs the remaining radiation. Figure 2-2 shows a schematic of the beam cave and the positions of the sample and detector. The beam enters the cave through a 19.05 cm aperture. During normal operations, the maximum steady flux is approximately $1.7 * 10^7$ n-cm^{-2}-s^{-1}. Most items that are to be imaged are placed approximately 2 m from the beam guide allowing for a L/D ratio from 115-150 [17]. Both the aperture and distance from the source are parameters that go into calculating this ratio. The L/D ratio describes how parallel the neutron beam can be.

The larger the L/D the more parallel the beam. A parallel beam allows for better edge resolution and less artifacts due to scattering. Further details of the beam line can be found in [17] and [18].

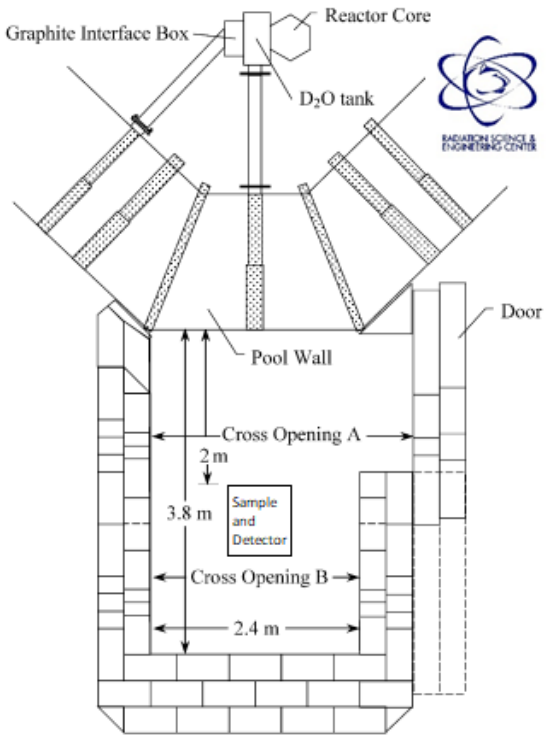

Figure **2-2**: Schematic diagram of the neutron beam lab including the sample and detector location at the Penn State Breazeale Nuclear Reactor adapted from Cimbala et al. [17].

Chapter 3

Development of a Bright Flash Thermal Neutron Imaging Setup at PSBR

High-frame rate neutron imaging would not be possible without the neutron source or the acquisition system, i.e. an imaging detector. There are three main factors that need to be taken into account when creating a new imaging setup. These three parameters are time resolution, spatial resolution, and the signal-to-noise ratio (SNR). The parameters identify a triangle of triple constraint that require different setups to focus on certain aspects of the triangle. In order to get good time resolution, some aspects of spatial resolution and SNR must be sacrificed. One main feature of the system that can allow for great time resolution and good SNR is light production and collection. This chapter will detail the neutron imaging camera, camera lens, and the software utilized in the acquisition of the images, post-processing, and the initial calibration of the system [31]. The general schematic of a camera-based neutron imaging detector is shown in Figure 3-1.

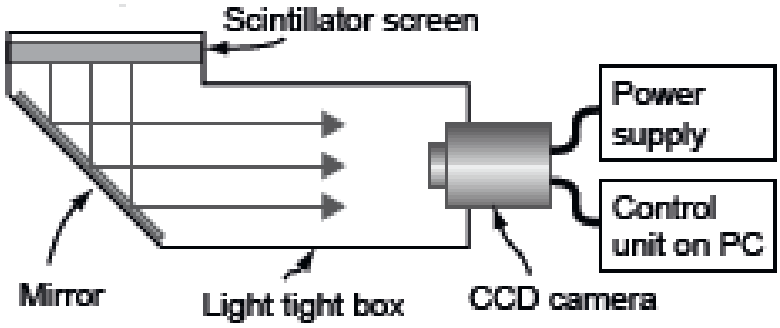

Figure **3-1**: General schematic diagram of a camera-based neutron imaging detector. The camera type can be CCD or CMOS depending on the application.

High-Speed Camera

The camera that was used for the bright flash imaging was a Photron FASTCAM-Ultima 512. The Ultima 512 is a complementary metal-oxide-semiconductor (CMOS) camera. CMOS cameras readout data row by row as opposed to pixel by pixel as in a charge-coupled device (CCD) camera. Line readout is much quicker and allows CMOS cameras to capture images faster than a CCD camera. Figure 3-2 shows the readout mechanisms between CMOS and CCD cameras.

The Ultima 512 has a chip with 512 x 512 pixels with a grayscale readout bit depth of 10. The internal memory that can save up to 2048 images if utilizing the full chip. By allowing partial readout, e.g. 512 x 32 pixels, the camera can reach frame rates up to 40,000 fps. The camera is connected to a desktop computer and is interfaced by using Photron FASTCAM Viewer (PFV) software that will be discussed later in this chapter. Based on the 530 nm green emission, the camera has about 40% quantum efficiency at the 530 nm emission peak of the LiF/ZnS(Cu) scintillator according to specifications produced by the manufacturer [19]. For more information on the hardware of the Photron FASTCAM-Ultima 512 see [20].

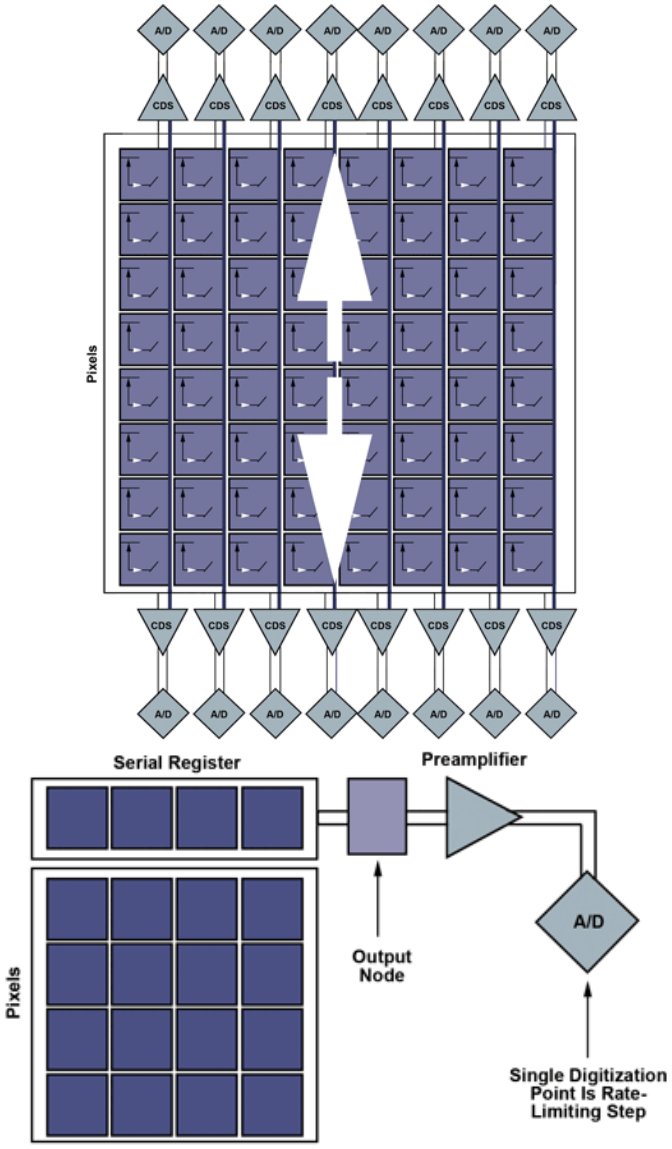

Figure **3-2**: (Top) CMOS camera chip showing how two rows are read at the same time. The arrows designate which direction the pixels are read. (Bottom) CCD camera chip showing that the data is read by a single output node and read pixel by pixel.

Camera Lens

The lens properties, such as light collection, focus length, and magnification, make the lens as critical as the camera. Light collection is important because as the frame rate of the camera

increases, the average number of photons collected per pixel decreases. Therefore, a greater intensity of light allows for faster acquisition. Three separate lenses were tested with the camera in order to produce the best results. In the end the lens of choice was a Fujinon f = 50 mm/0.7 lens. This lens allowed for the greatest light collection out of the prospective lenses by a factor of two. In addition to the light collection, the Fujinon lens provided the largest field of view at 70 x 70 mm. This lens is a closed-circuit television lens that predated the Ultima 512 digital camera so it was not optimized to work in a camera/lens combination with the CMOS. Consequently, the lens was focused to its optimum that allowed the average pixel size to be 0.13 mm/pixel by using a Siemens star and a checkerboard calibration pattern.

Dark Box and Scintillator

The dark box was purchased from Grenoble Scientific and is made from 3 mm thick aluminum. The dark box is to ensure that no external light is introduced to these light sensitive experiments. The material was chosen since aluminum is very “transparent” to neutrons and it is only slightly activated when exposed to a neutron beam for extended periods of time and decays to clearance levels within a few hours typically. Featured in Figure 3-3, the dark box is in an L-shape with a front window size of 220 mm high and 270 mm wide. The horizontal arm is 400 mm long and the vertical arm is 350 mm high.

The scintillator is screwed onto the front window of the dark box with aluminum screws and the seams are then covered with black imaging tape. The scintillator’s function is to capture and convert incident neutrons in to visible light. The scintillator used was LiF/ZnS screen doped with Cu from Scintacor [19]. Lithium is a strong thermal neutron absorber frequently used as a neutron detection medium and releases and energetic triton and an alpha particle (Q-value 4.78 MeV) upon neutron capture. These energetic charged particles create visible scintillation light in

ZnS:Cu with is mixed into the screen as a scintillating medium. The light is directed into the camera by a mirror as shown in Figure 3-1. This mirror is used to avoid placing the camera into the direct neutron beam and the reduce the chances of fast and early degradation of the camera. The screen material is typically brought on an aluminum substrate to add structure with minimal neutron attenuation.

Standard LiF/ZnS screen thickness usually does not exceed 250 μm as it is assumed to start strongly attenuating its own scintillation light above that thickness due to diffuse light scatter by the granular and polycrystalline structure of the ZnS. However, in this work it was anticipated that additional thickness above 250-300 μm can improve detection efficiency, based on the experiments by Baechler et al. in [21]. Physically the improved efficiency can be explained by the relatively large range, 50 μm, of the triton produced in the screen material due to the Li reaction. Tritons produced in the first 50 μm of the screen can still be absorbed and create light in the last 250-300 μm section of the screen where part of the light can still escape. Since this system was optimized for light collection, a 400 μm scintillator was used to enhance the light output. Scintillator properties are discussed further by Gnezdilov et al. in [22].

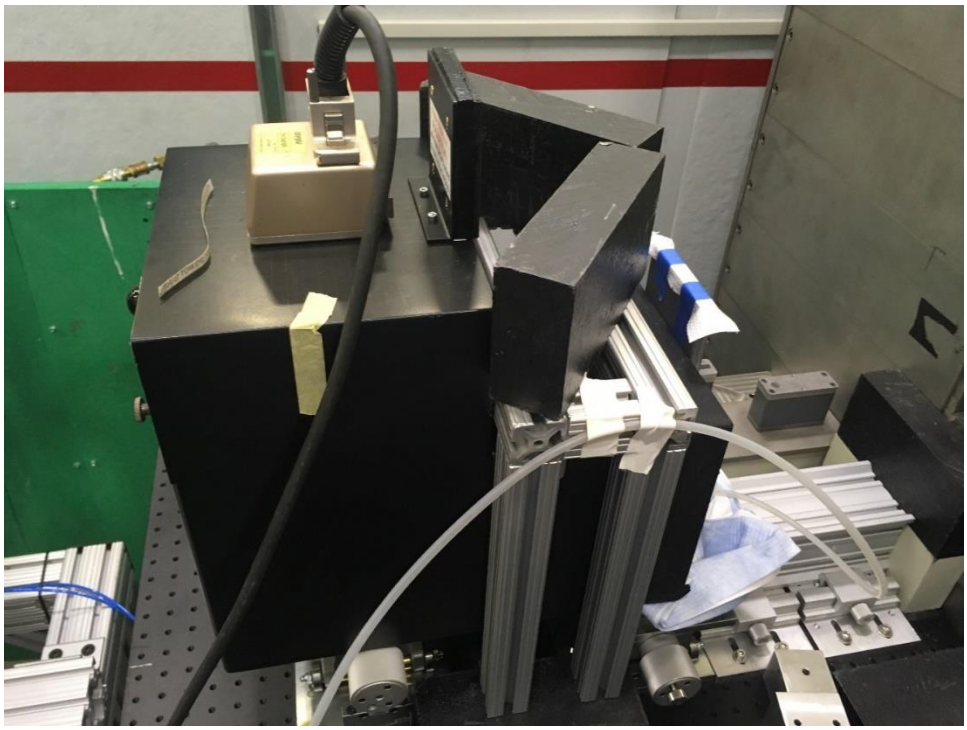

Figure **3-3**: Camera and dark box setup at the High Flux Isotope Reactor.

Two-Phase Flow Bubbler

The key component of the test apparatus that generated the two-phase flow conditions was an aluminum bubbler. The bubbler consisted of an aluminum container with dimension of 300 mm high, 98 mm wide, and 5 mm depth in the beam direction. An inlet nozzle introduced pressurized air toward the bottom creating bubbles. Aluminum, as stated previously for the dark box, was chosen to reduce the amount of neutron attenuation and to limit the amount of activated material. The bubbler was filled to approximately 3/4 of the way with water and was operated at an air flow rate of one to two cubic feet per hour. To reduce the penumbra image blurring effect due to the finite L/D ratio, the bubbler was placed as close to the scintillator as physically allowable.

Software

Due to the multiple tasks that needed to be accomplished during the acquisition and processing of the neutron images, several different types of software were utilized. The main software to capture the images was PFV, while ImageJ, Python, and MATLAB were used in post processing.

PFV is a software supplied by Photron to interact with Photron's cameras. The software has many features such as frame rate selection, gain settings, triggering, and much more. All features of the PFV software can be found in [23] and [24]. The original calibration for the camera was completed by utilizing this software.

ImageJ was used for the initial preparation of the images [25]. Using ImageJ's built-in features, data outliers were removed. An outlier in a gray scale image is defined as a pixel that has a value that deviates from the surrounding pixel values by a predetermined percent. The outlier is then replaced by an average of the surrounding pixel values. Due to the age of the camera, there were dead pixels, pixels that constantly have a value of zero. In addition to these pixels, there were bright pixels that had a maximum value which were most likely caused by gamma rays from the reactor or shielding imparting their energy directly onto the camera's chip. After the removal of the outliers, the images were cleaned with a striped filter to remove the readout bias [26]. Since a CMOS camera reads out line by line, alternating lines tend to have a bias and cause stripes across the image.

Once the images were processed, they were normalized utilizing a python script. Normalization is a process in radiography where the sample image is compared with a dark field image, an image taken with no light source, and a flat-field image, and image taken with a light source but no sample. The objective of normalization is to remove systematic errors of the imaging

system. These bias effects can be caused by defects in the scintillator, biases of the camera, and uneven profiles of incident radiation source.

Most normalizations are performed by the following equation:

$$p = \frac{I - I_{DC}}{I_0 - I_{DC}}, \quad \textbf{(3-1)}$$

where p is the normalized projection, I is the intensity of the sample image, I_{DC} is the intensity of the dark field image, and I_0 is the intensity of the flat-field image.

However, when dealing with two-phase flow, two flat-field images must be taken to estimate the void fraction of the sample. The two flat-field images have the container filled with liquid and the container empty. With the additional image, the normalization equation becomes:

$$\varepsilon(x,y) = \frac{\ln(\frac{I(x,y)}{I_w(x,y)})}{\ln(\frac{I_g(x,y)}{I_w(x,y)})}, \quad \textbf{(3-2)}$$

where I is the dark-filed compensated intensity of the image taken for the two-phase flow, ε is the void fraction and the subscripts w and g are for flat-field intensities for the bubbler filled with water and filled with air respectively. The void fraction is equal to the normalized image in the case of two-phase flow and it allows computation of the liquid to gas ratio for this work.

A problem that was evident in the pulsed images was that there was no synchronization from data set to data set causing an inability to normalize the images. There are three different image sets that are needed to normalize two-phase flow images, sample images, flat-field images with water, and flat-field images without water. Unfortunately, due to manual data acquisition, each of the three sets have different numbers of images and different times in which the pulse is present.

To rectify this problem, Python code was created. The Python code first queues all the folders of interest that contain the three sets of data and determines the largest array possible with those data sets. The data sets can then be “centered” to each other by taking the mean intensity of each image and taking the maximum. Since the intensity directly relates to the incident neutron flux, then the maximum image intensity corresponds to the peak of the neutron pulse. Now that the

images are centered, the three sets of data are run through a loop that truncates the images that correspond to empty values in the other data sets ensuring that each data set has the same number of images with the maximum intensity at the middle. Consequently, the data sets are properly edited and prepared so that they may be combined in order to normalize the sample image. The code used for the normalization can be found in Appendix A and the supplementary functions in Appendix B.

Calibration

With a system of multiple components, calibration of each becomes important because the error can become compounded throughout the system. The main items that needed particular attention in their calibration are the camera lens and the scintillator.

It was stated earlier that multiple lenses were tested in order to maximize the amount of light collected from the lens. Another parameter of the lens is its focus, or how small of a feature that can be discerned with the camera-lens-scintillator setup. To determine this parameter, a Fujinon f = 25 mm/ 1.4 lens was placed onto the camera to maximize the resolution of the system. When assuming that the camera itself is the limiting factor for this setup, the innate resolution of the camera with the 400 μm scintillator was determined to be roughly 0.5 mm, meaning that two objects would need to be at least 0.5 mm apart in order to determine they are in fact separate. The scintillator was then replaced with a paper Siemens star and the Fujinon f = 25 mm/ 1.4 was replaced with the Fujinon f = 50 mm/ 0.7 lens. Natural light was used to take an image and the maximum resolution determined with the high light capture lens was calculated to be 0.3 mm. Assuming that the total resolution is a convolution of the screen resolution and the lens resolution, the true resolution can be determined by taking the root mean square of both of the resolutions. In doing so, the total system resolution is determined to be approximately 0.6 mm.

Spatial resolution is not the only feature that needs to be accounted for when taking digital images. The camera utilized and the exposure time can introduce bias effects that alter the data that is recorded. Low count bias is an effect that occurs during short exposure times and the associated low number of counts in the camera. The low count bias is caused by the non-linearity of the Lambert-Beer law for attenuation for measurements with a low total number of counts. The low count bias is introduced specifically by the logarithmic term [27]. For the experimental set up for this thesis, the low count bias was determined to be less than 5% at a void fraction of 0.2 and decreases with an increase in the void fraction but increases greatly with a decrease in void fraction. This low count bias is significantly lower than the bias of the void fraction normalization that is utilized in these experiments.

In addition to the low count bias, the individual pixels are also subject to stochastic fluctuations. A study on chord-averaged of void fractions and its statistical uncertainty was completed by Munshi and Jayakumar [28]. Utilizing the error equation in [28], the statistical error of the experimental setup is 15% at a void fraction of 0.3 and decreased to 8% for a void fraction close to one.

Chapter 4

Results of Bright Flash Imaging

In order to attempt a holistic approach at determining the abilities of this high-speed neutron imaging method, a test matrix was developed to ensure that all of the images were acquired, as detailed in Table 4-1. This work at the PSBR looks at frame rates from 500 to 4,000 fps and utilized a frame rate of 1,000 fps to take a series of images of a step wedge in order to obtain the edge spread function.

Table **4-1**: Test matrix of data acquired with thirteen neutron pulses.

Frame Rate	2 $ Two-Phase	2 $ Air Flat-Field	2 $ Water Flat-Field	2.5 $ Two-Phase	2.5 $ Air Flat-Field	2.5 $ Water Flat-Field
500	Y	N	Y	N	N	N
1,000	Y	Y	N	N	Y	N
2,000	Y	Y	N	Y	Y	Y
4,000	N	N	N	Y	Y	Y

There are several values that contribute to the quality of an image besides the line spread function. Two of the key values that can be calculated are the signal-to-noise ratio (SNR) and the contrast-to-noise ratio (CNR). The SNR is a comparison of the desired information of an image compared to the undesirable noise and is calculated by:

$$SNR = \frac{mean(I)}{standarddeviation(I)} = \frac{\mu}{\sigma}. \quad \textbf{(4-1)}$$

Equation 4-1 can be reduced by assuming Poisson particle counting and rewritten as:

$$SNR = \frac{N}{\sqrt{N}} = \sqrt{N}. \quad \textbf{(4-2)}$$

The CNR is a quantification of how well a feature can still be distinguished in an image above the background noise and is calculated by:

$$CNR = \frac{mean(C)}{standarddeviation(C)} = \frac{\mu(I_a - I_b)}{\sqrt{{\sigma_{I_a}}^2 + {\sigma_{I_b}}^2}}, \quad \textbf{(4-3)}$$

where $C = I_a - I_b$ and I_a and I_b are intensities of two separate features. With these two parameters, a basis can be established to compare two different images in terms of image quality. All images were taken with the camera having a gain setting of four and an air flow rate of two standard cubic feet of air per second into the bubbler.

500 fps Results

The objective of the first tests that were conducted were to ensure that the neutron flux was of sufficient intensity to capture the two-phase flow. These tests utilized a frame rate of 500 fps one of the lowest capture rates of the Ultima 512. Due to pulsing limitations on the day of the 500 fps experiment, only 2 $ pulses were utilized. Figure 4-3 shows the image of the air-water mixture at the peak of the neutron flux and Figure 4-4 shows the image normalized. During the tests an empty flat-field images for 500 fps was not obtained and normalization could not be completed. The 500 fps images were normalized by taking the 1,000 fps empty flat-field image and combining every two images to simulate 500 fps.

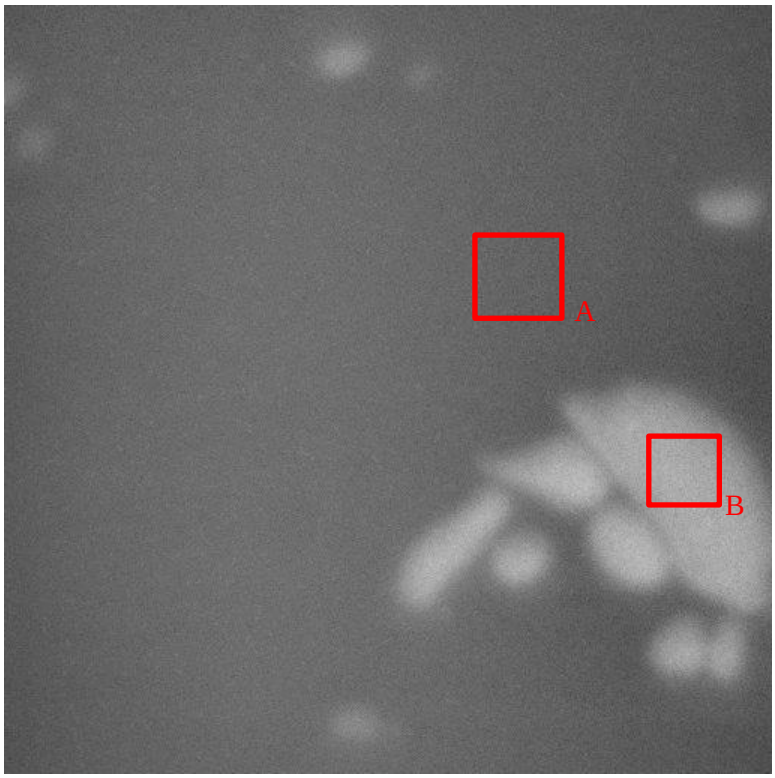

Figure **4-1**: Raw image of two-phase flow taken at 500 fps with a 2 $ neutron pulse.

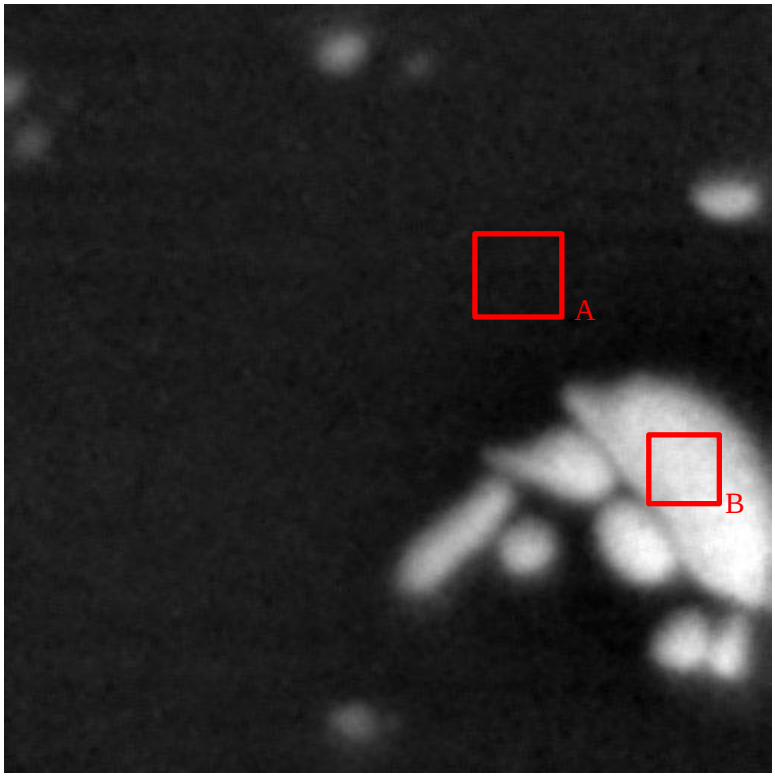

Figure **4-2**: Normalized image of two-phase flow taken at 500 fps with a 2 $ neutron pulse.

By using the software ImageJ, the mean intensity and the standard deviation can be extracted from the image. For the image in Figure 4-3, the mean and standard deviation are determined to be 425.433 and 74.731 respectively. Utilizing Equation 4-1, the SNR is calculated to be 5.692. To calculate the CNR of Figure 4-3, regions A and B are utilized to sample the background and the bubble. The mean intensities for region A and B were determined to be 404.499 and 660.323, respectively and the standard deviations were calculated to be 19.116 and 36.375. The CNR is calculated by using Equation 4-3 and results in a value of 6.222. Unfortunately, as only the 2 $ pulse condition was run for the 500 fps images, no comparison can be made with the SNR or CNR between 2 $ and 2.5 $ pulses.

Using the same methodology as with Figure 4-3, the normalized image Figure 4-4 was determined to have a mean value of 1.050 and a standard deviation of 0.190. These values translated to a SNR of 5.526. For Region A, the mean and standard deviation were determined to be 0.989 and 0.010 respectively and Region B had a mean of 1.801 and a standard deviation of 0.057, leading to a CNR of 14.031.

1,000 fps Results

After determining that a frame rate of 500 fps was not a problem for the imaging system, the camera speed was increased to 1,000 fps. As in the 500 fps test, only a 2 $ pulse was utilized for the 1,000 fps images. Figure 4-3 shows the image taken at the maximum pulse height and the regions of interest used for determining the CNR. Figure 4-4 shows the normalized image. According to Table 4-1, there was not a water filled flat filled image. This problem was rectified by taking the 500 fps water filled flat filled images and halving each image to roughly approximate what the 1,000 fps test experienced.

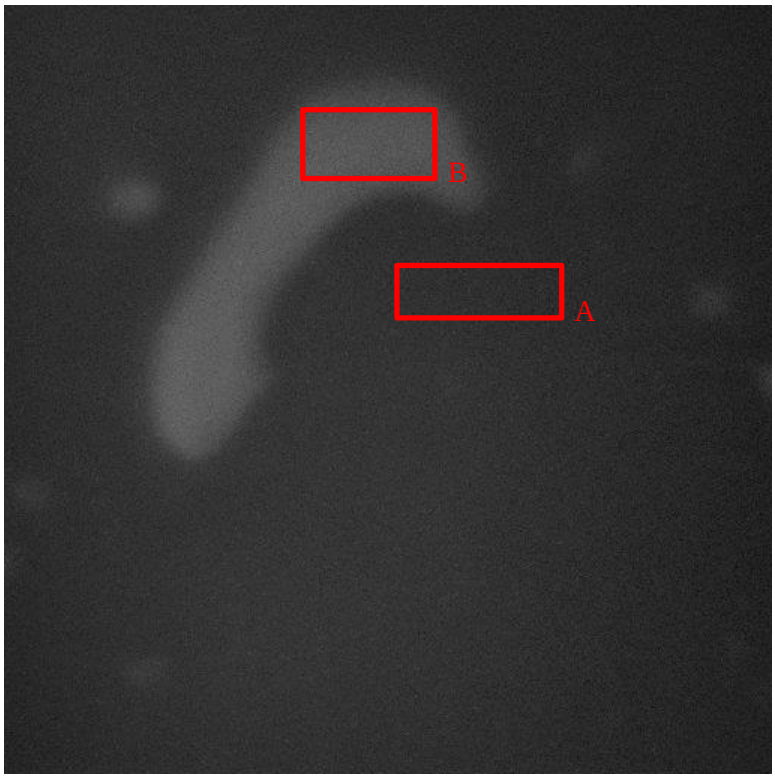

Figure **4-3**: Raw image of two-phase flow taken at 1,000 fps with a 2 $ neutron pulse.

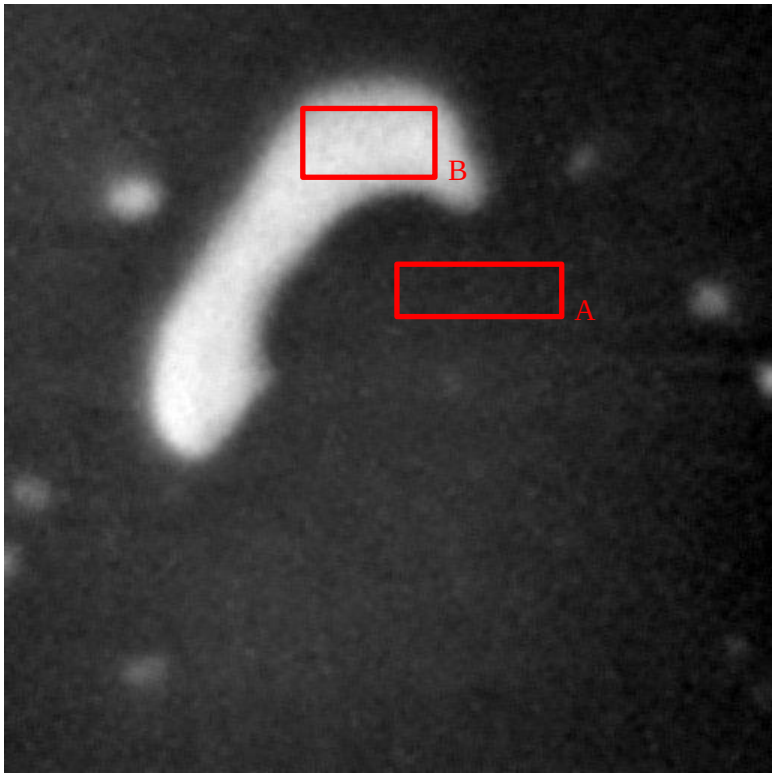

Figure **4-4**: Normalized image of two-phase flow taken at 1,000 fps with a 2 $ neutron pulse.

It is expected that the SNR and the CNR decrease as the frame rate increases because each pixel on average will have a lower intensity. Using the same methodology that was utilized for the 500 fps images to calculate the SNR and the CNR, the results for Figures 4-3 and 4-4 are shown in Table 4-2.

Table **4-2**: Comparison of SNR and CNR from images taken at 1,000 fps with 2 $ pulse.

	Figure 4-9 (raw)		Figure 4-10 (normalized)	
	Mean	StdDev	Mean	StdDev
Full Image	196.327	51.031	0.868	0.172
Region A	198.938	16.203	1.517	0.026
Region B	345.306	18.819	0.857	0.013
SNR	3.847		5.047	
CNR	5.894		24.562	

2,000 fps Results

Once the pulsing rate of the reactor was resolved and the 2.5 $ pulse achieved, the imaging rate was increased to 2,000 fps. At this imaging speed, both 2 $ and 2.5 $ pulses were used to compare the resolution of the imaging method. Maintaining the same methodology for analysis, the images for both sets of pulses were normalized to show the difference between raw and normalized images. The CMOS chip readout was reduced to 256 x 512 pixels instead of 512 x 512 pixels due to the limited memory of the camera. Figure 4-5 shows the raw image taken at the maximum pulse height for 2 $ pulse. Figures 4-6 and 4-7 show the images from the 2.5 $ pulse. The 2 $ pulse was not normalized because the flat-field images were not obtained during the experiment. Note during the acquisition of the water filled flat-field image a bubble was present. After the normalization the bubble that was present in the flat-field image caused a black spot in Figure 4-7 that is located to the left of region B.

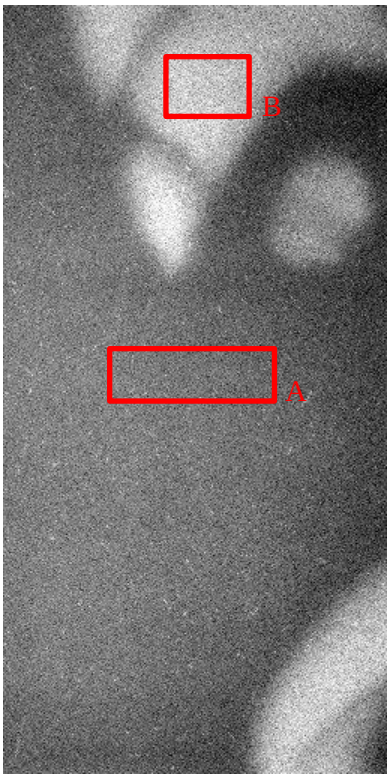

Figure **4-5**: Raw image of two-phase flow taken at 2,000 fps with a 2 $ neutron pulse.

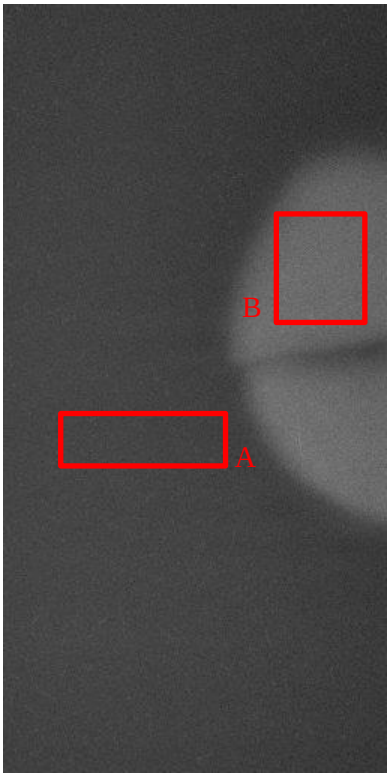

Figure **4-6**: Raw image of two-phase flow taken at 2,000 fps with a 2.5 $ neutron pulse.

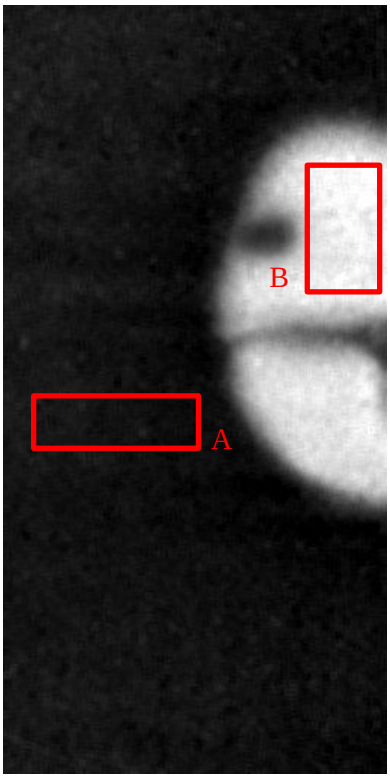

Figure **4-7**: Normalized image of two-phase flow taken at 2,000 fps with a 2.5 $ neutron pulse.

Table 4-3 shows the comparison of the SNR and CNR of Figures 4-5 – 4-7. As expected, both the SNR and CNR values of the raw images increase with the larger neutron pulse. With respect to the 2.5 $ raw and normalized images, the SNR is reduced yet the normalization allows for a much greater CNR due to the values being limited between zero and one. To visualize the bubble motion, Appendix D contains Figure D-1 which shows a sequence of images taken from the raw data and Figure D-2 which shows the same images but normalized to show the motion of the bubble frame by frame.

Table **4-3**: Comparison of SNR and CNR of images taken at 2,000 fps with a 2 $ and a 2.5 $ pulse.

	Figure 4-15 (raw)		Figure 4-16 (raw)		Figure 4-17 (normalized)	
	2 $		2.5 $		2.5 $	
	Mean	StdDev	Mean	StdDev	Mean	StdDev
Full Image	107.939	24.672	280.474	58.330	0.954	0.240
Region A	107.514	13.188	279.160	14.197	0.848	0.019
Region B	148.468	14.428	442.679	16.664	1.616	0.033
SNR	4.374		4.808		3.975	
CNR	2.095		7.469		20.169	

4,000 fps Results

The highest frame rate that was reached with the manually synchronized detection system was 4,000 fps. Once again due to the limited memory and acquisition duration of the camera, the chip readout area was reduced to a 256 x 256 pixel area from the center of the chip at this frame rate to be able to capture the pulse using the manual synchronization with high probability. At this frame rate if the entire chip was utilized, the acquisition time would be 0.5 seconds, by reducing the readout this time is increased to about 2 seconds. Only a 2.5 $ pulse was used due to the belief that a 2 $ pulse would not produce enough neutrons for reasonable results. Figures 4-8 and 4-9 show the raw and normalized images at the maximum pulse height.

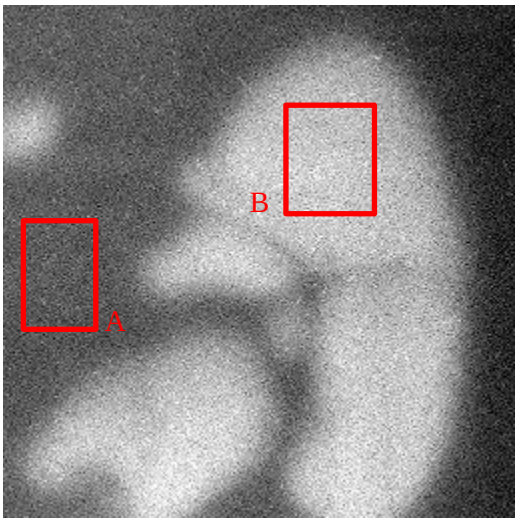

Figure **4-8**: Raw image of two-phase flow taken at 4,000 fps with a 2.5 $ neutron pulse.

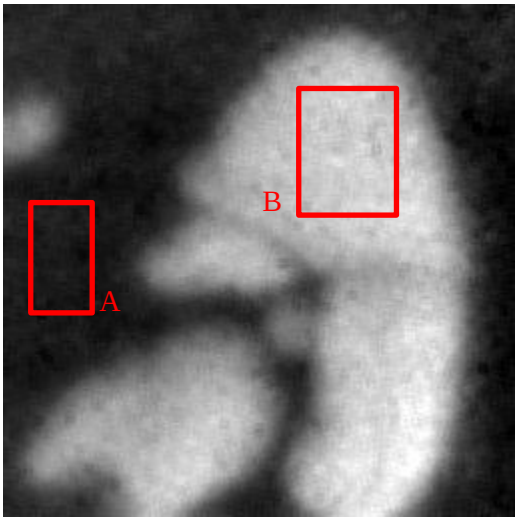

Figure **4-9**: Normalized image of two-phase flow taken at 4,000 fps with a 2.5 $ neutron pulse.

The SNR and CNR of Figures 4-8 and 4-9 can be seen in Table 4-4 calculated from Equations 4-1 and 4-3. To aid in the visualization of motion, Appendix E contains Figures E-1 and E-2 which present a series of images both raw and normalized.

Table **4-4**: Comparison of SNR and CNR of images taken at 4,000 fps with a 2.5 $ pulse.

	Figure 4-24 (raw)		Figure 4-25 (normalized)	
	Mean	StdDev	Mean	StdDev
Full Image	164.461	44.079	1.000	0.248
Region A	132.853	12.072	0.733	0.018
Region B	213.987	14.631	1.335	0.032
SNR	3.731		4.032	
CNR	4.277		16.397	

Pulsing Analysis

From 500 fps to 4,000 fps there are multiple trends that can be determined from the data that was acquired from the Penn State Breazeale Reactor. One item to note is the decrease in the SNR and CNR values as the frame rate increased. Figures 4-10 – 4-17 show the SNR and CNR values for like pulses. Figures 4-10 – 4-13 correspond to the 2 $ pulses while Figures 4-14 – 4-17 are the values of the 2.5 $ pulses. Note that the pulses are not centered on each other due to the manual acquisition of the imaging system. The calculations of the SNR and CNR were completed by a python script that is located in Appendix C.

Assuming that each neutron pulse of the same reactivity insertion produces the around the same number of neutrons, higher frame rates yield less neutrons per frame. Less neutrons directly relates to less light produced and signal obtained. The reduction of the signal leads to the lower SNR and CNR values. One outlier of this trend can be seen in Figure 4-10, where the 2,000 fps SNR values are higher than the 1,000 fps values. The reason for this could be caused by the reduction of the readout size or the size of the bubbles. Later in this section it will be discussed how the void fraction, air to volume ratio, effects the quality of the images.

When comparing the SNR curves to those of the CNR values, it is apparent that the SNR curves are smoother. This smoothness is a result from several phenomenon. The first reason that the SNR curves are smoother is because the SNR calculation uses a larger sample size than the CNR calculation. The larger sample size leads to a less dependency of SNR on small changes of the images. A second reason why the CNR is less smooth than the SNR curves is that static regions of interest were used to calculate the CNR values. Due to the static regions, bubbles can enter and exit the region and drastically affect the contrast of the image. This problem is the reason why there are some sharp changes in the CNR values, see Figure 4-13, and could be mitigated by a dynamic/tracking region.

A trend that is present between the SNR values of the raw and normalized images is the flattening of the peak. This flattening is expected as the images are normalized. The normalization process is an attempt to make all of the images as relatable as possible during the transient of the reactor core. This normalization leads to less of a difference between each image throughout the entire stack.

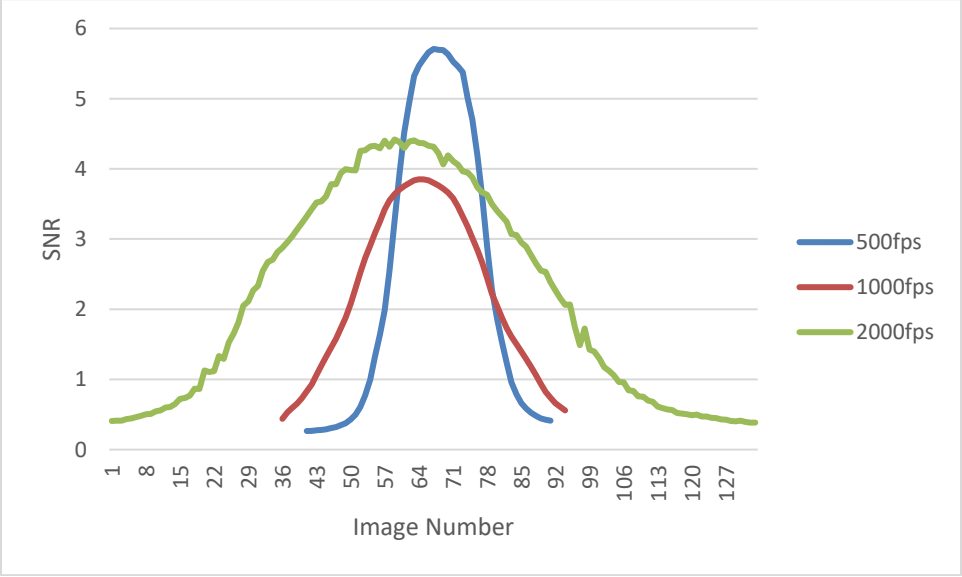

Figure **4-10**: SNR values of raw PSBR image stacks taken with a 2 $ pulse.

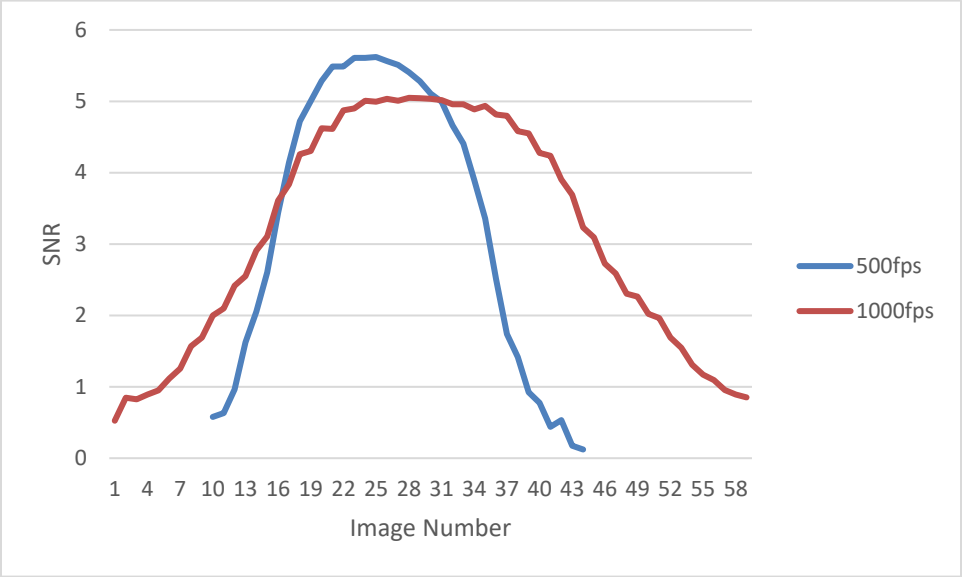

Figure **4-11**: SNR values of normalized PSBR image stacks taken with a 2 $ pulse.

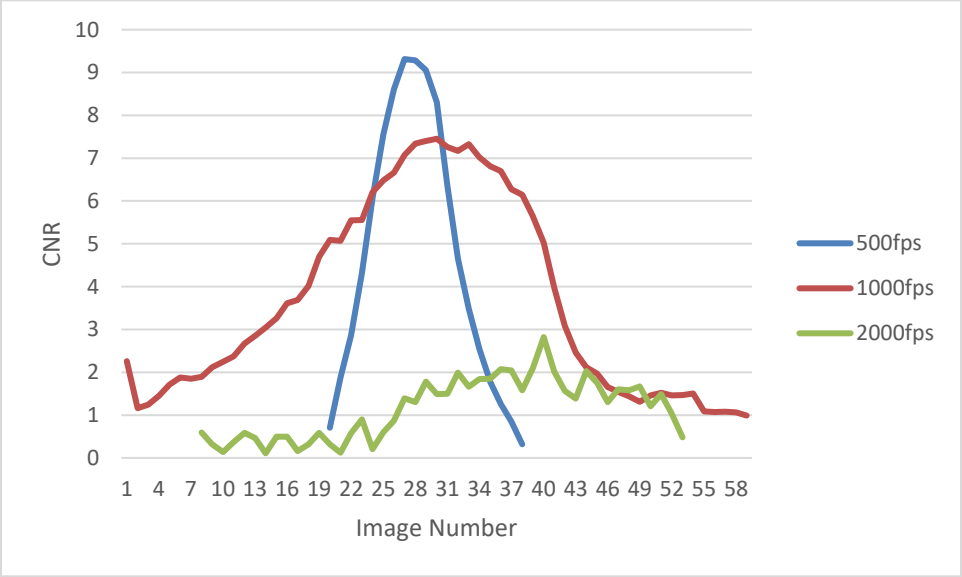

Figure **4-12** CNR values of raw PSBR image stacks taken with a 2 $ pulse.

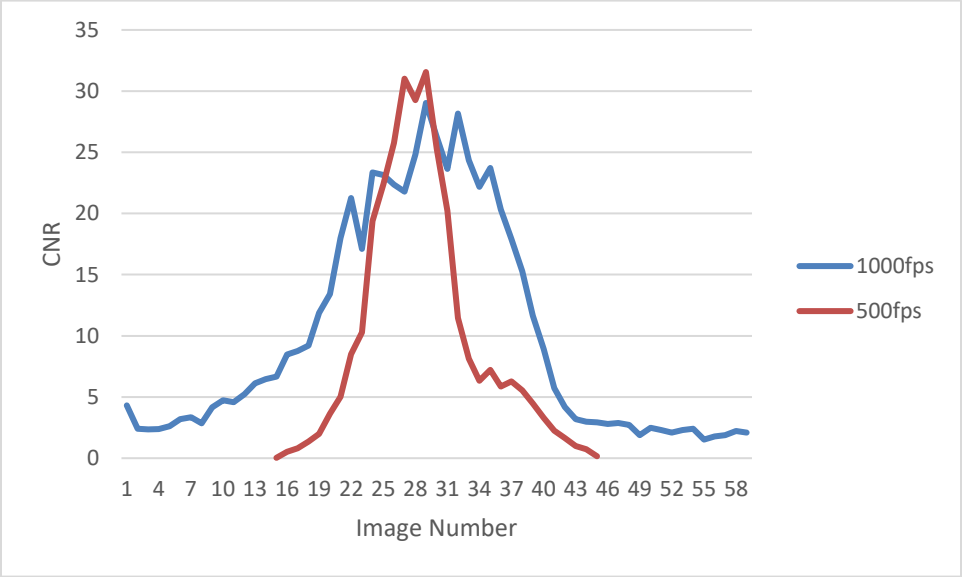

Figure **4-13**: CNR values of normalized PSBR image stacks taken with a 2 $ pulse.

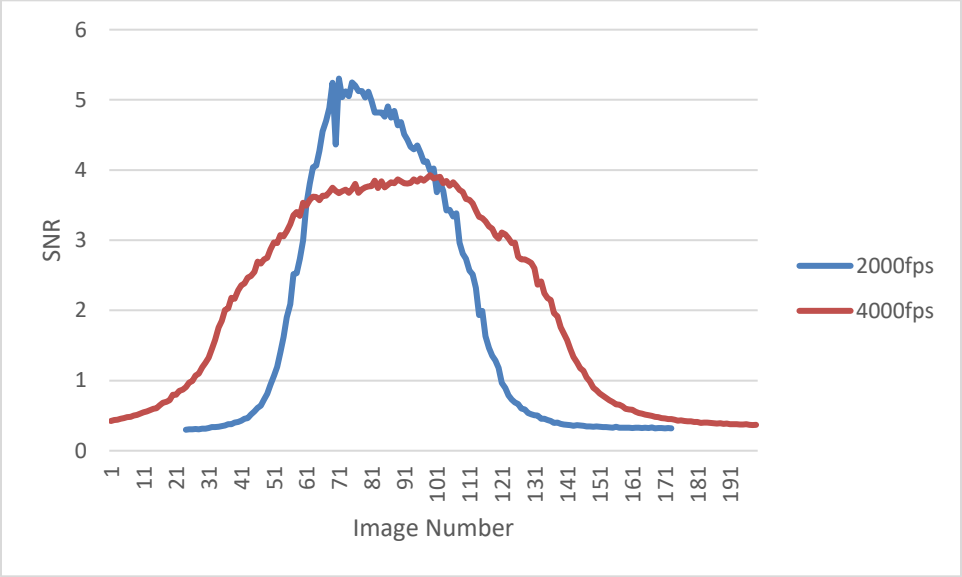

Figure **4-14**: SNR values of raw PSBR image stacks taken with a 2.5 $ pulse.

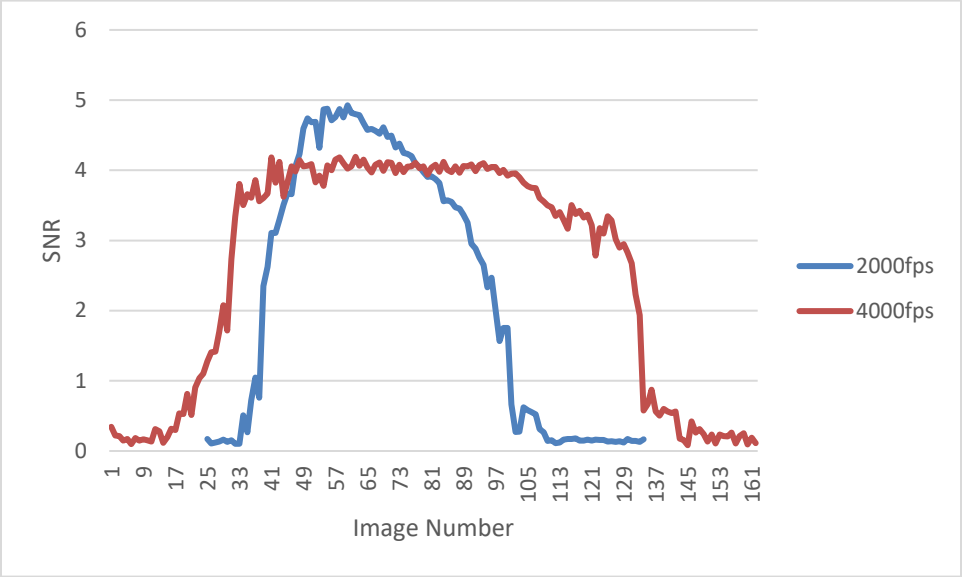

Figure **4-15**: SNR values of normalized PSBR image stacks taken with a 2.5 $ pulse.

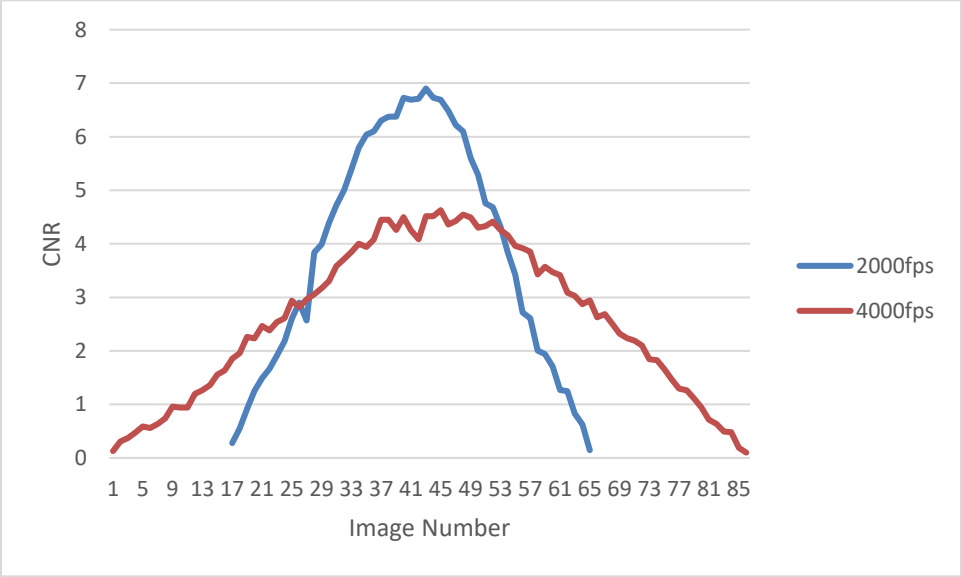

Figure **4-16**: CNR values of raw PSBR image stacks taken with a 2.5 $ pulse.

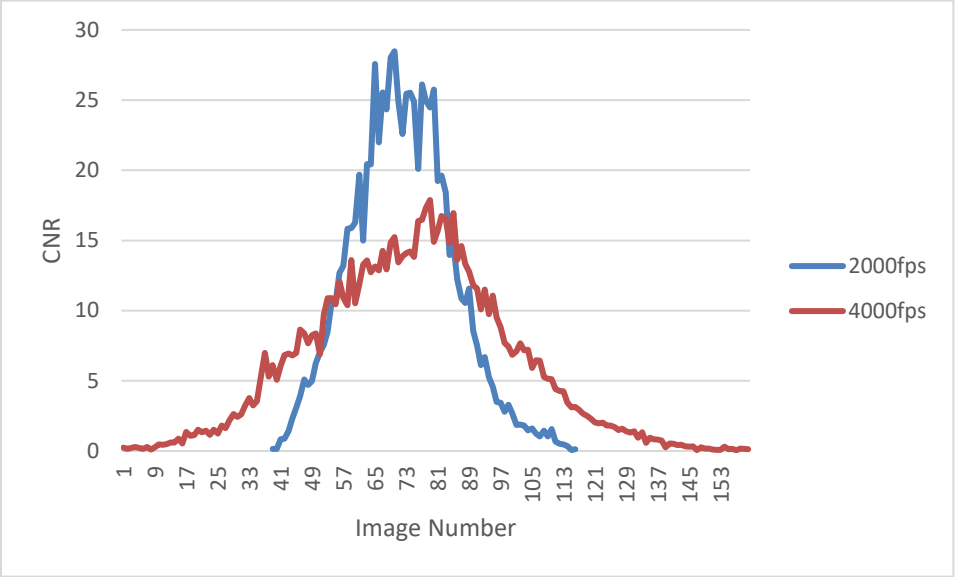

Figure **4-17**: CNR values of normalized PSBR image stacks taken with a 2.5 $ pulse.

Table 4-5 summarizes the peak SNR and CNR values of the image sets acquired at the PSBR. The table facilitates the ability to see some of the main trends that occur as the frame rates and the reactivity is increased. Starting at 500 fps and increasing to 2,000 fps, the SNR drops. From 1,000 fps to 2,000 fps the SNR increases and as stated prior in this section is due to the decrease in readout size and the size of the bubbles.

Table **4-5**: Maximum SNR and CNR values from raw and normalized images obtained at the PSBR.

Frame Rate	500 fps		1,000 fps		2,000 fps			4,000 fps	
Readout	512 x 512		512 x 512		512 x 256			256 x 256	
Image Type	Raw 2 $	Norm 2 $	Raw 2 $	Norm 2 $	Raw 2 $	Raw 2.5 $	Norm 2.5 $	Raw 2.5 $	Norm 2.5 $
SNR	5.692	5.526	3.847	5.047	4.374	4.808	3.975	3.731	4.032
CNR	6.222	14.031	5.894	24.562	2.095	7.469	20.169	4.277	16.397

The reduction of the signal is not the only parameter that is affecting the quality of the images that are obtained. Biases, such as the low count bias and the statistical error, begin playing a more significant role. As the frame rates increase and the signal decrease, these biases tend to increase. Figures 4-30 and 4-31 illustrate the change in the statistical error as both void fraction and frame rate change. It can be seen that as a bubble exits the FOV, reducing the void fraction, the

statistical error becomes quite significate reaching a max at about 22% for an image taken at 4,000 fps utilizing a 2 $ pulse. Additionally, from Figures 4-18 and 4-19, the statistical error decreases as the frame rate decreases due to the amount of signal acquired for each frame.

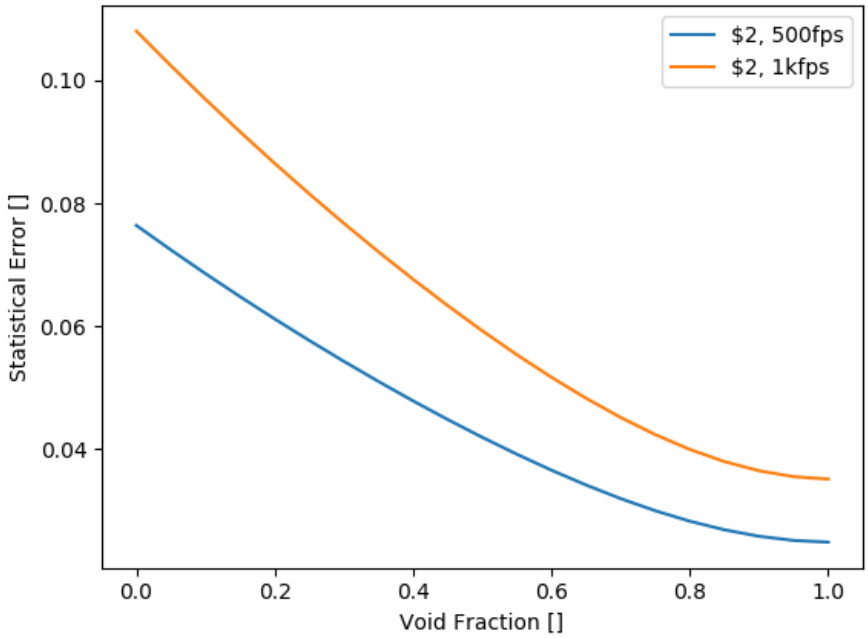

Figure **4-18**: Statistical error as a function of void fraction for a 2 $ pulse at 500 and 1,000 fps.

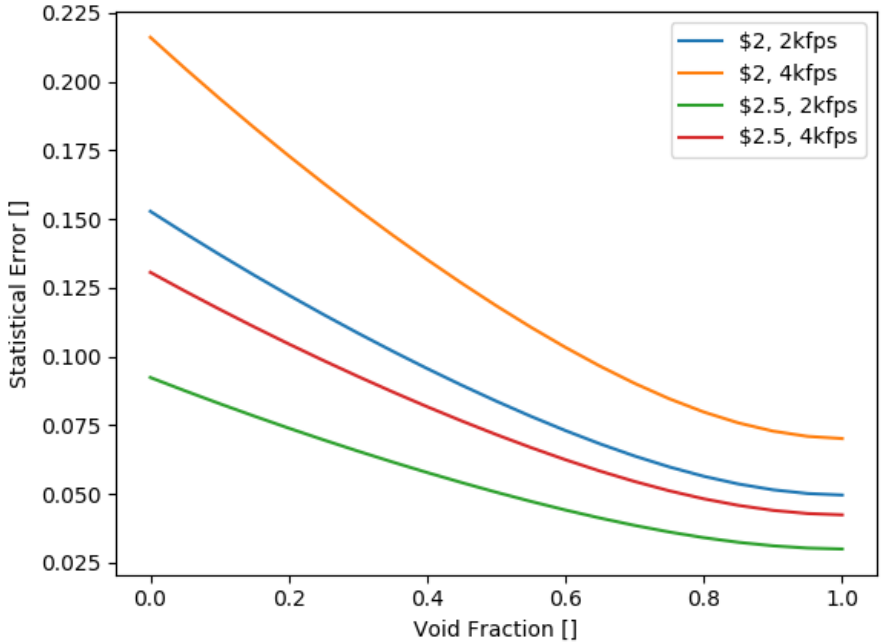

Figure **4-19**: Statistical error as a function of void fraction for a 2 $ and 2.5 $ pulse at 2,000 and 4,000 fps.

The low count bias follows almost the same patterns that the statistical error exhibits. Figures 4-20 and 4-21 present the low count bias values as a function of void fraction utilizing different pulse sizes and frame rates. The low count bias was calculated by utilizing the equations from [27]. Similar to the statistical error, the low count bias increases with low void fractions. However, the low count bias becomes nearly negligible as it passes beyond a void fraction of 0.4. The low count bias also decreases as the frame rate is reduced due to the greater signal obtained per frame. Note that random numbers with a Poisson distribution were used to generate the low count bias values and that Figures 4-20 and 4-21 are the results of multiple iterations.

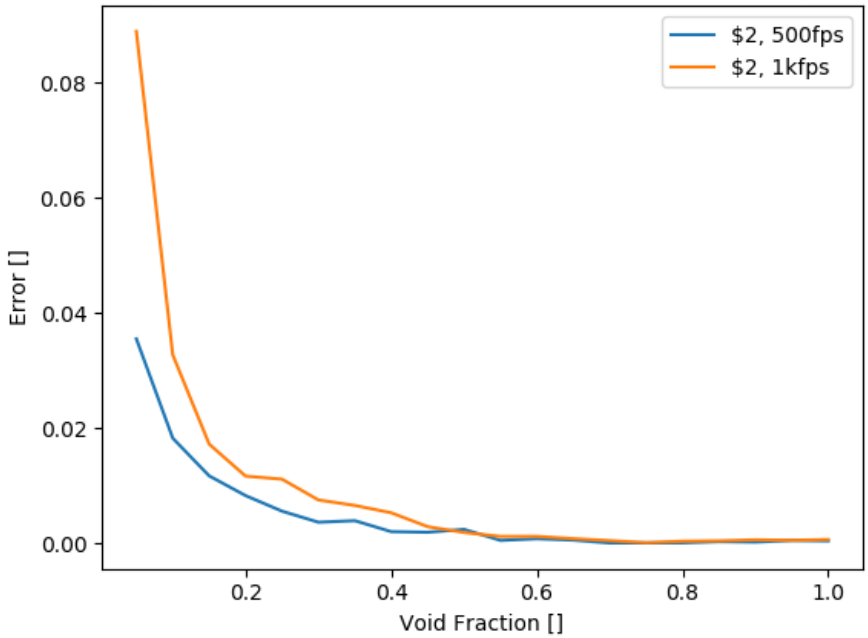

Figure **4-20**: Low count bias as a function of void fraction for a 2 $ pulse at 500 and 1,000 fps.

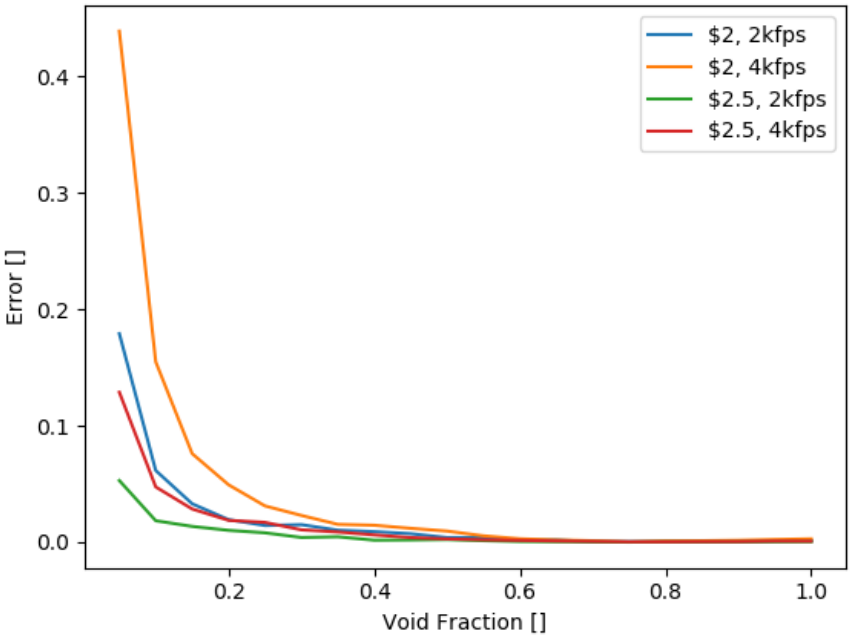

Figure **4-21**: Low count bias as a function of void fraction for a 2 $ and 2.5 $ pulse at 2,000 and 4,000 fps.

Chapter 5

High Speed Imaging at the Continuous HFIR Beamline

In order to compare bright flash imaging as done at the Penn State Breazeale Reactor to steady neutron flux reactors, the High Flux Isotope Reactor of Oak Ridge National Laboratory was utilized, featured in Figure 5-1. HFIR generates the highest steady-state neutron fluxes of any other neutron source in the United States and one of the highest in the world.

Figure **5-1**: The High Flux Isotope Reactor facilities at Oak Ridge National Laboratory from [29].

HFIR is a 100 MW reactor that was constructed in 1965. The original mission of HFIR was to produce elements such as curium and plutonium. As time passed, new facilities were added to HFIR expanding the utility of the reactor and its radiation. Today HFIR operates at 85 MW and its used for research on cold and scattered neutron along with producing californium-252. HFIR has a core that lasts approximately 23 days and is completely replaced at the end of each cycle. Over the years that HFIR has operated, over 480 cores have been utilized. HFIR has been designated as a Nuclear Historic Landmark by the American Nuclear Society in 2014 and as an International R&D Hub by the International Atomic Energy Agency in 2017 [29].

The CG-1D cold neutron beamline was the specific neuron guide utilized in the user facilities of HFIR. The CG-1D beamline is the imaging beamline at HFIR that is moderated by liquid hydrogen. At the traditional detector setup, the flight path from the aperture is 6.59 m. Depending on the aperture size used, ranging from 3.3 mm to 16 mm, the L/D ratio of the beamline ranges from 400 to 2,000 where L is the 6.59 m length of the neutron flight. Since the maximization of neutron flux is important for this experiment, the imaging setup was placed about 0.125 m from the exit of the beam port and using the largest aperture size. This setup leads to a lower L/D than a traditional setup at the CG-1D beamline. From initial experiments for this work, the neutron flux was approximated to be 6.5×10^{8} neutrons-cm^{-2}-s^{-1}. This neutron flux value was from comparing the intensities the images of those acquired from the PSBR and HFIR. Since the image intensity is directly proportional to the neutron flux, the HFIR neutron flux was calculated based on the relative intensities which was approximately 20 times less than PSBR. The CG-1D beamline has many other amenities that were of not pertinent to this work but can be found in the ORNL documentation [30].

Table 5-1 shows a comparison of different parameters that are pertinent to neutron imaging. The aperture sized and the distance to the beam ports are several of the parameters that go into calculating the L/D ratio. A higher L/D ratio means that the neutron beam will be more parallel, leading to a decrease in edge blurring. The neutron flux is important since it is the light source for the imaging system. The higher the flux, the greater the light output, allowing faster frame rates to be utilized.

Table **5-1**: Comparison PSBR and HFIR pertinent qualities.

	PSBR	HFIR
L/D Ratio	115-150	400-2,000
Aperture size (cm)	19.05	0.33-1.6
Neutron Flux (n-cm^{-2}-s^{-1})	1.3*10^{10} peak for 2.5 $ pulse	~6.5*10^{8}
Distance to beam port (m)	2	0.125

In the experiments at HFIR, the same imaging system was utilized and the setup was adjusted to be almost identical. In order to maximize the number of neutrons at the CG-1D imaging beamline at HFIR, the high-speed detector was setup at the exit from the beam port. The detector setup was the same setup that was utilized at the PSBR. Figure 3-2 shows the camera setup that was used for the acquisition of data from HFIR and Figure 5-2 shows the bubbler centered in front of the scintillator and dark box.

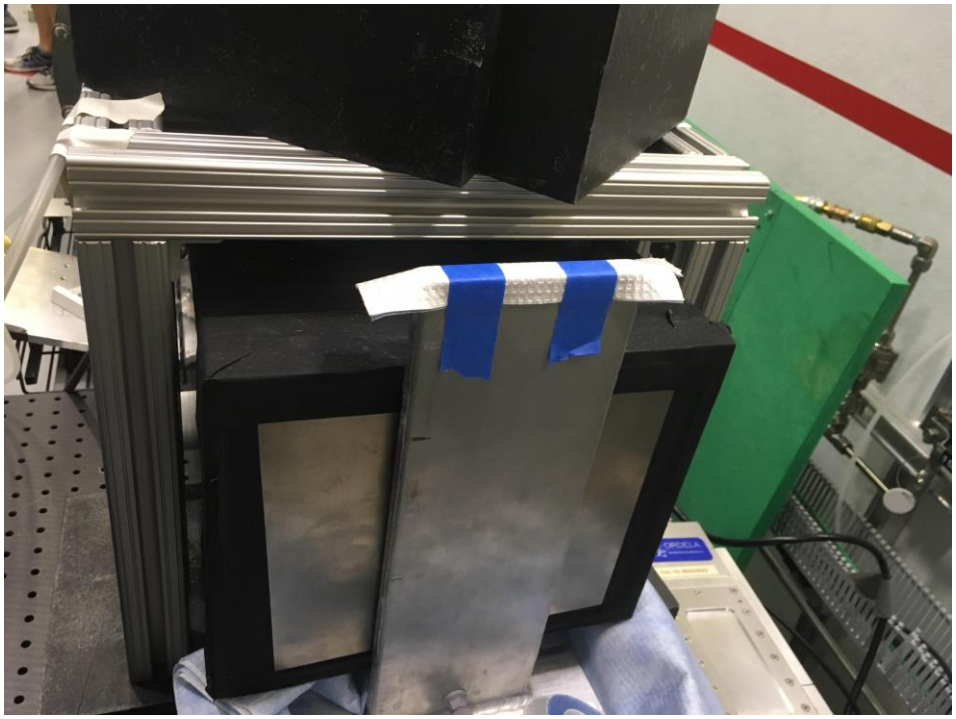

Figure **5-2**: Bubbler, scintillator, and dark box setup for HFIR experiments.

Chapter 6

Results of High-Speed Imaging at HFIR

Due to the steady neutron flux of the High Flux Isotope Reactor, no synchronization was required for taking the neutron images. Therefore, all 2,048 images that the camera can save can have data. However, due to the lower flux of the HFIR facility, approximately 20 times less than the PSBR, frame rates of up to only 1,000 fps could be acquired. Through the testing at HFIR, images were taken at 60, 125, 250, 500, and 1,000 fps. These frame rates were chosen so that the motion blurring could be witnessed and shown how it is reduced as the frame rate increases. Additionally, due to the inconsistency of bubble position throughout the set of images, only the signal-to-noise ratio was calculated to determine the quality of the images through the duration of the exposure, the individual contrast-to-noise ratios were still calculated for the selected images. All images were taken with the camera having a gain setting of four and an air flow rate of two standard cubic feet of air per second into the bubbler.

60 fps Rresults

The slowest frame rate used at HFIR was 60 fps. Figure 6-1 shows one of the images of a bubble passing though the FOV and Figure 6-2 shows the normalized version of that image. The FOV of the camera setup at HFIR is much greater than the diameter of the neutron beam, which is limited by the neutron guide, the approximate beam circumference is highlighted by the green circle on Figure 6-1.

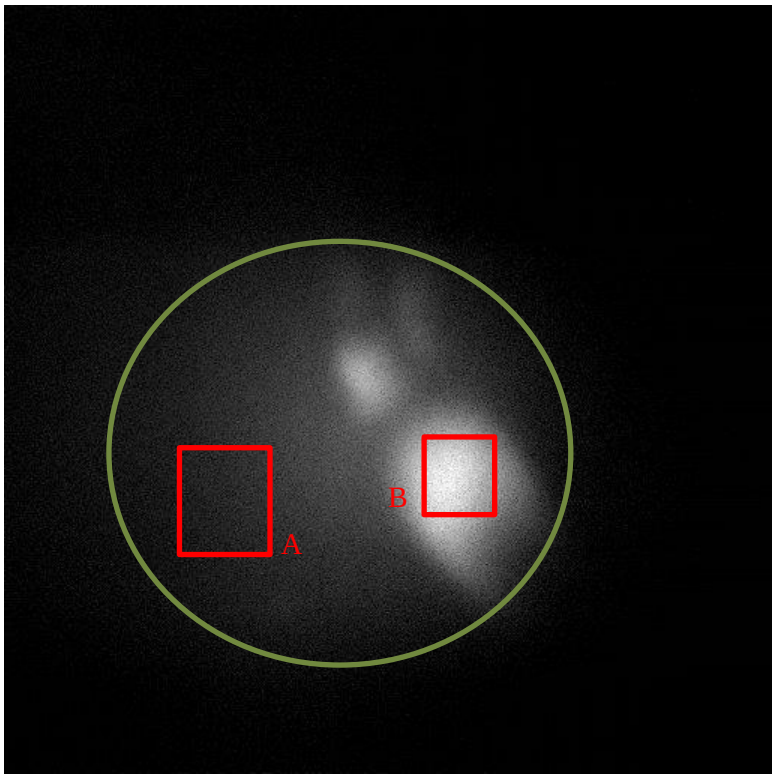

Figure **6-1**: Raw image taken at 60 fps at HFIR.

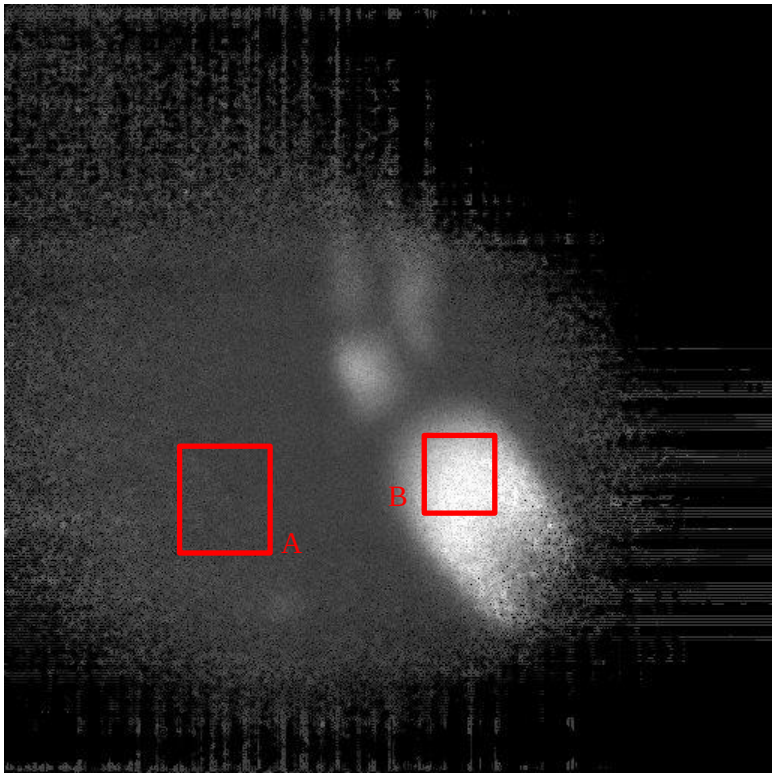

Figure **6-2**: Normalized image taken at 60 fps at HFIR.

Motion artifacts can be seen in Figure 6-1 and 6-2 at the bottom of the bubbles. These artifacts are blurring caused by the bubbles moving quickly through the FOV and leaving trails of higher intensities. As the frame rate increases to 125 fps and 250 fps, one can notice that these trails shrink until they become almost non-existent. This reduction of motion artifacts is one of the main advantages of high frame rate imaging.

Due to the reduced beam size, the SNR would not be representative of the image if the entire FOV was used. Instead the pixels used for the SNR are those that are in the green circle of Figure 6-1. The SNR and CNR of Figures 6-1 and 6-2 can be found in Table 6-1 whereas the SNR of the image stacks of the raw and normalized 60 fps images can be found in Figures 6-3 and 6-4, respectively.

Table **6-1**: Comparison of SNR and CNR from images taken at 60 fps from HFIR.

	Figure 6-1 (raw)		Figure 6-2 (normalized)	
	Mean	StdDev	Mean	StdDev
Full Image	118.109	79.365	1.077	0.578
Region A	109.312	17.923	0.758	0.055
Region B	312.190	57.309	2.491	0.267
SNR	1.488		1.863	
CNR	3.379		6.357	

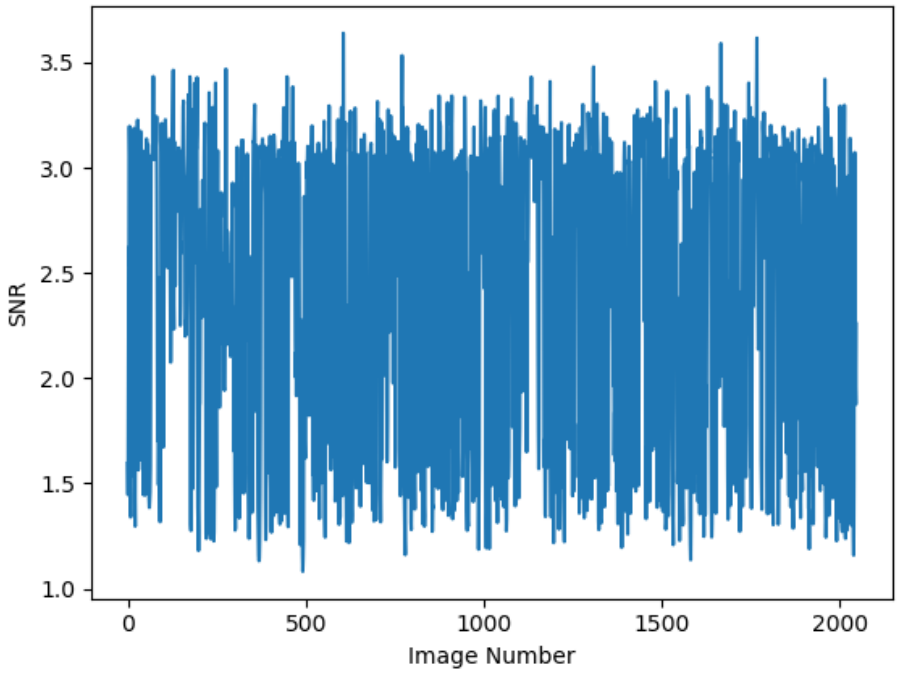

Figure **6-3**: SNR of raw 60 fps image stack from HFIR.

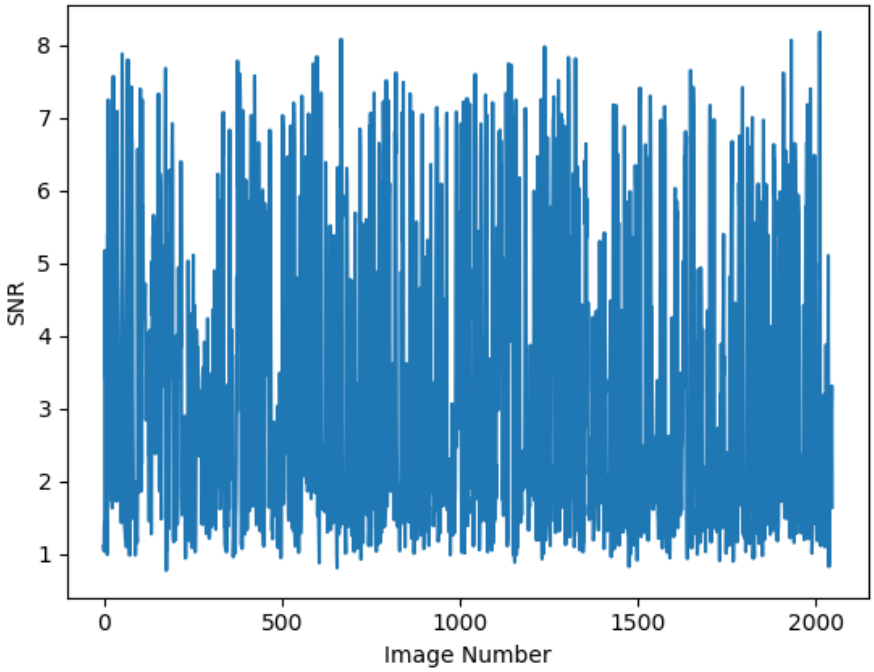

Figure **6-4**: SNR of normalized 60 fps image stack form HFIR.

It can be seen in Figures 6-3 and 6-4 that the SNR does not follow a pulse like the PSBR data does. Instead the SNR is heavily dependent on whether a bubble is present in the FOV or not. If a bubble is present then the SNR reaches a maximum, yet if there is no bubble, then the FOV is only background leading to a low SNR. With the lack of a bubble, the void fraction of the image is approximately zero. With an extremely small void fraction, the statistical errors and the low signal bias become an increasing problem. These biases for the HFIR experiments are discussed in greater detail in a later section. However, due to the continuous source, the SNR does not vary greatly over time. Comparing Figures 6-3 and 6-4 to the SNR values of higher frame rates, it can be seen that the spread of the SNR is reduced.

125 fps Results

At 125 fps, the motion artifacts are still present. The blending of the bubbles together is a result of the speed at which they are moving. However, when one compares Figure 6-5 to Figure 6-1, it is apparent that the motion artifacts are reduced due to the higher frame rate. Figure 6-6 is the normalized image of Figure 6-5 and makes the blurring of the bubbles slightly more noticeable.

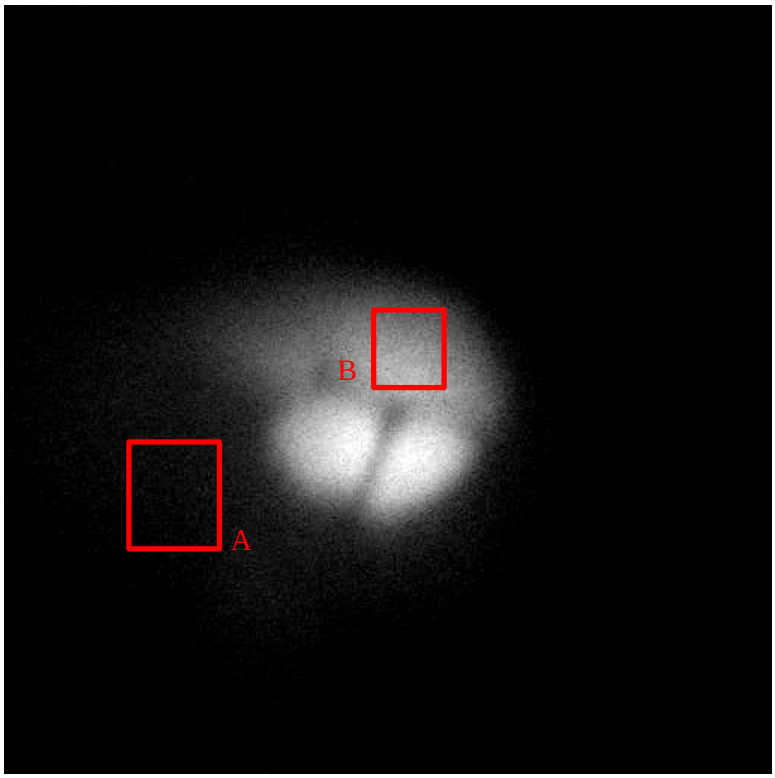

Figure **6-5**: Raw image taken at 125 fps at HFIR.

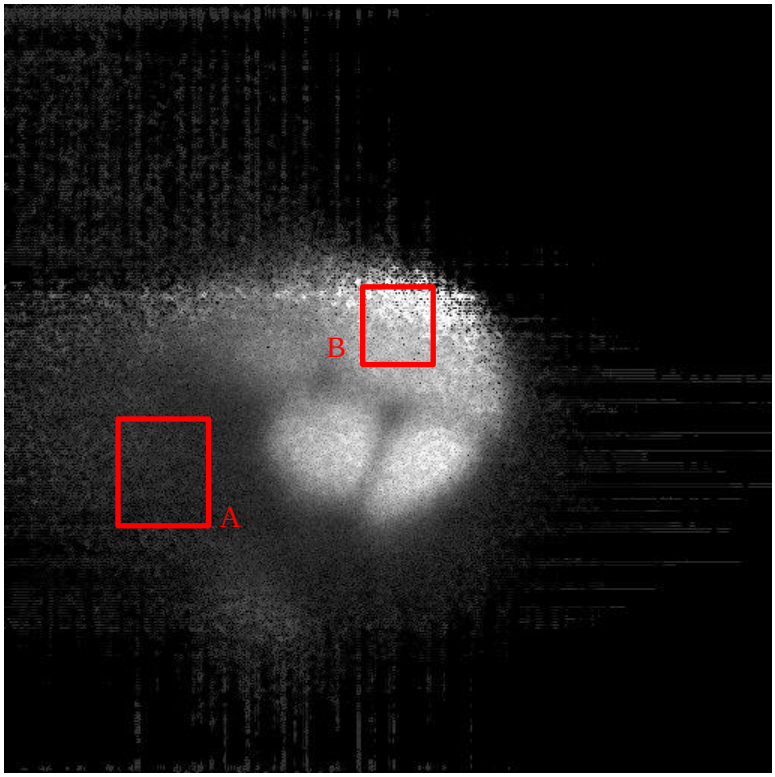

Figure **6-6**: Normalized image taken at 125 fps at HFIR.

The SNR and the CNR have been calculated for Figures 6-5 and 6-6 and can be found in Table 6-2. The SNR and the CNR are both higher for selected images of the 125 fps than for the 60 fps. This change could be created by the motion artifacts reducing the signal in the images or the impact of different sized bubbles. As stated before, the void fraction begins to play a larger role in image quality with smaller bubble sizes. The SNR for the raw and normalized image stacks can be found in Figures 6-7 and 6-8.

Table **6-2**: Comparison of SNR and CNR from images taken at 125 fps at HFIR.

	Figure 6-5 (raw)		Figure 6-6 (normalized)	
	Mean	StdDev	Mean	StdDev
Full Image	119.826	73.719	1.121	0.561
Region A	27.455	12.816	0.408	0.098
Region B	241.602	26.793	1.767	0.189
SNR	1.625		1.998	
CNR	7.210		6.383	

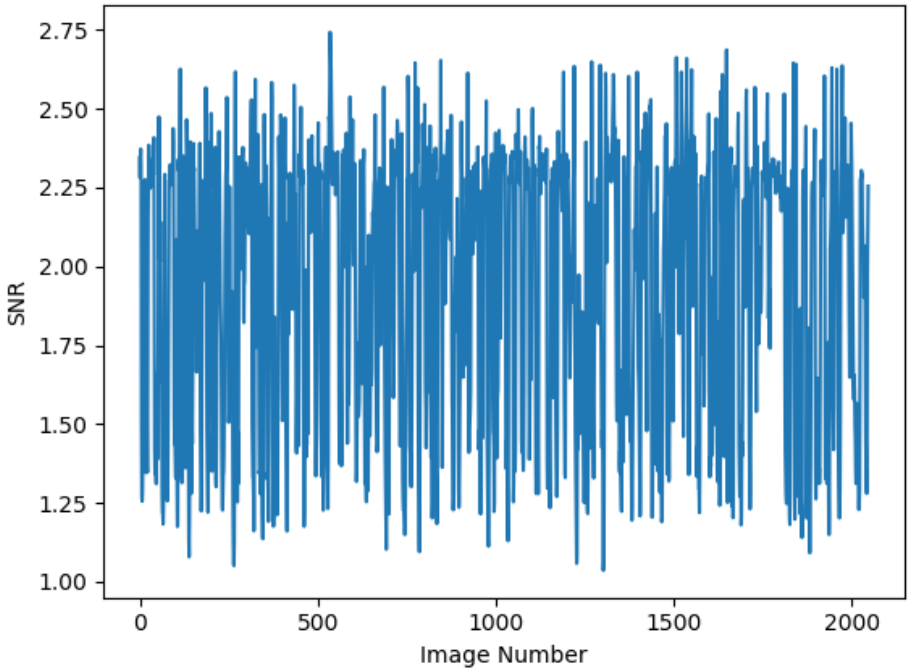

Figure **6-7**: SNR of raw 125 fps image stack from HFIR.

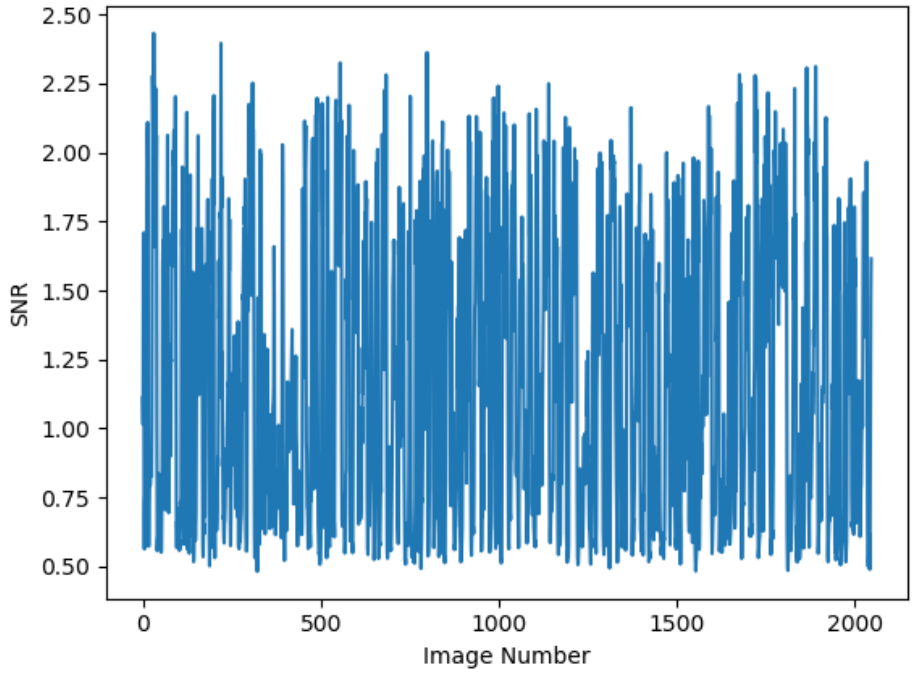

Figure **6-8**: SNR of normalized 125 fps image stack from HFIR.

Overall the 125 fps SNR values are lower than those of the 60 fps values. These lower values are expected because there is less light per image leading to a lower signal. However, the SNR values can be higher at 125 fps than 60 fps depending on the images that are being compared. In addition to the lower values, it can be seen that the value spread has also decreased with an increase in the frame rate. This reduction in range is brought about by the reduction of signal causing lower extreme values to occur.

250 fps Results

Figures 6-9 and 6-10 show a bubble taken at a frame rate of 250 fps. Looking at the tail of the bubble, it is now difficult to determine whether there are any motion artifacts present. The

blurring of the bubbles that this frame rate is more due to the resolution of the imaging system rather than motion artifacts.

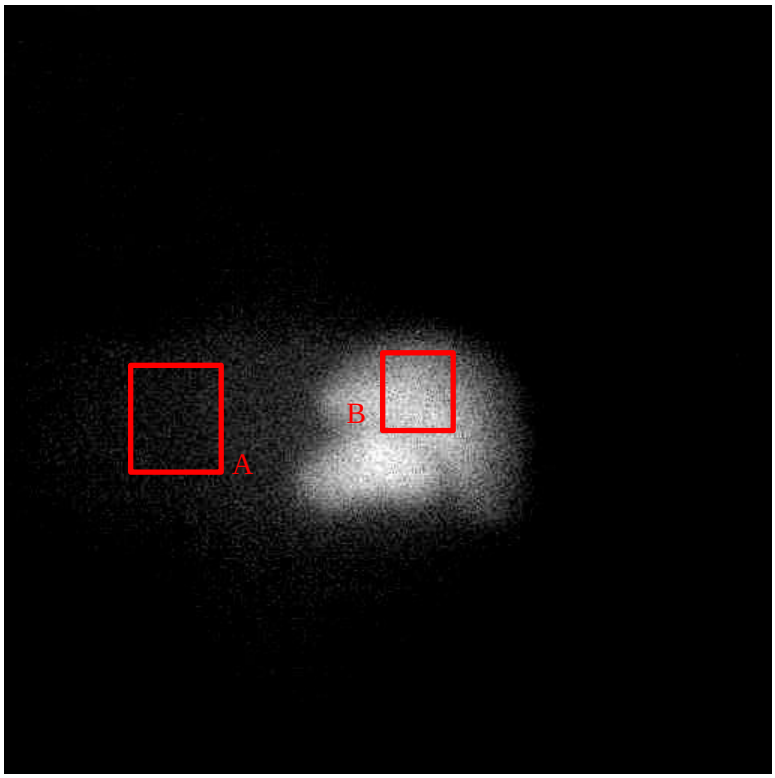

Figure **6-9**: Raw image taken at 250 fps at HFIR.

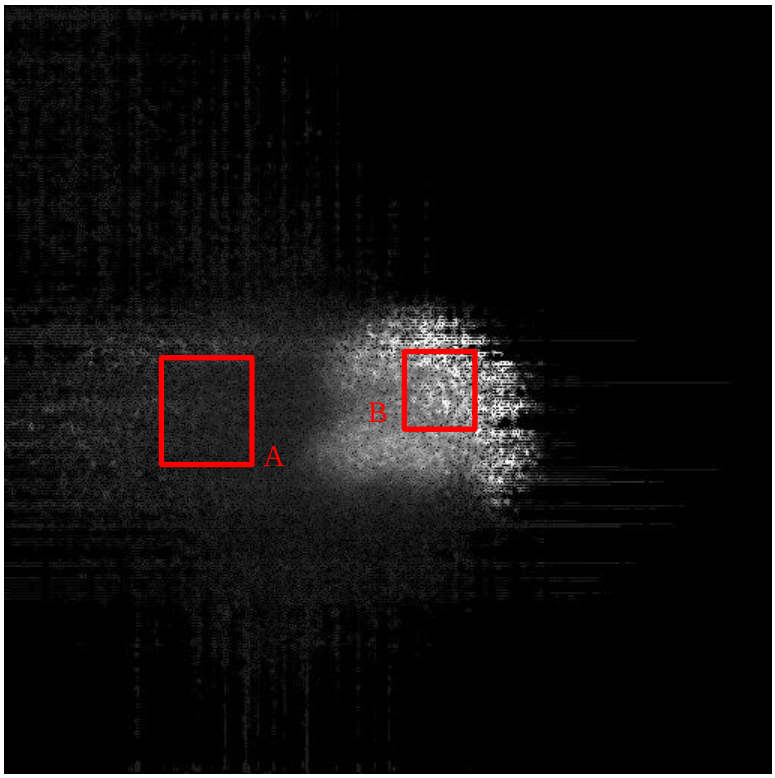

Figure **6-10**: Normalized image taken at 250 fps at HFIR.

The CNR and SNR have been determined for Figures 6-9 and 6-10 and the results are present in Table 6-3. Compared to the images taken at 125 fps, the SNR and the CNR of the 250 fps are both lower since the light per image is lower. The SNR for the entire image stack taken at 250 for both the raw and the normalized images are presented in Figures 6-11 and 6-12.

Table **6-3**: Comparison of SNR and CNR from images taken at 250 fps from HFIR.

	Figure 6-9 (raw)		Figure 6-10 (normalized)	
	Mean	StdDev	Mean	StdDev
Full Image	54.145	36.278	1.152	0.767
Region A	22.416	11.109	0.519	0.170
Region B	102.732	17.578	2.017	0.682
SNR	1.493		1.502	
CNR	3.862		2.131	

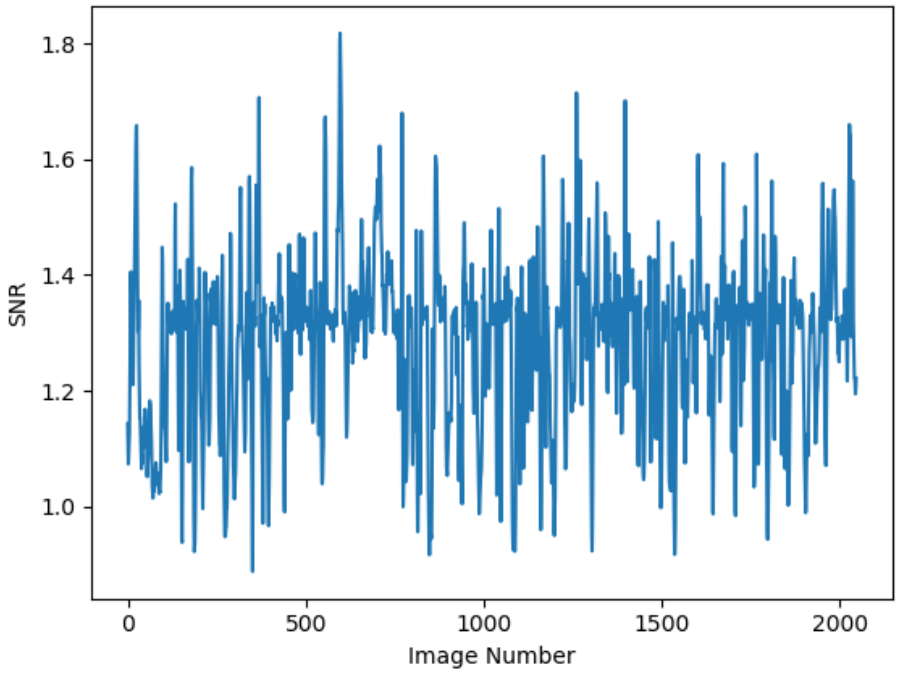

Figure **6-11**: SNR of normalized 250 fps image stack from HFIR.

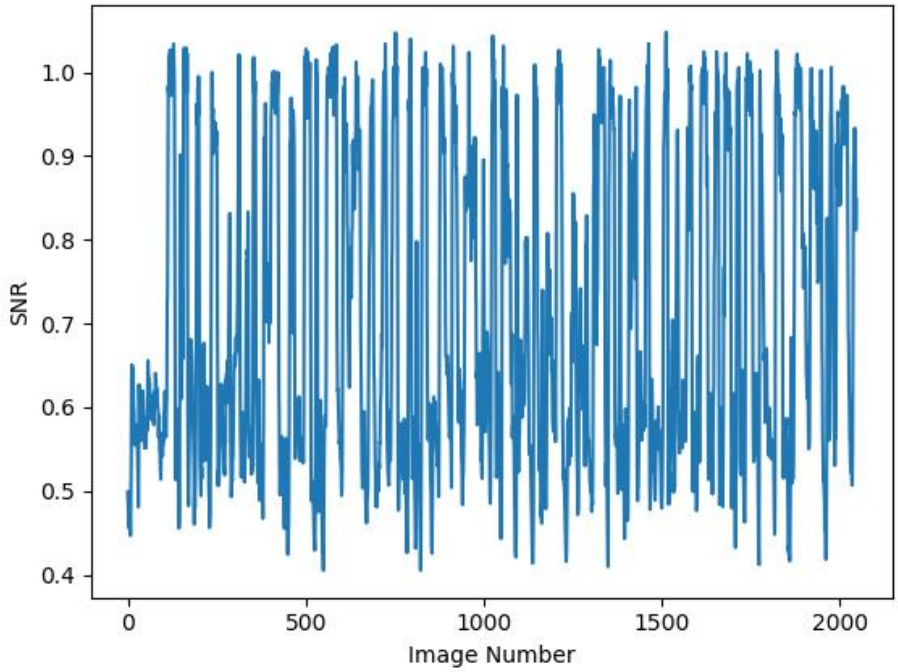

Figure **6-12**: SNR of normalized 250 fps image stack from HFIR.

It can be seen that the decreasing SNR range trend continues and that the average SNR of the images is also reducing with an increase in the frame rate due to the reduction in signal and extremes.

500 fps Results

Imaging at 500 fps with the continuous neutron source of HFIR was conducted, which corresponds to the lowest frame rate obtained at the PSBR. Figures 6-13 and 6-14 show a raw and normalized image taken at 500 fps. At this frame rate there are no visible motion artifacts caused by the speed at which the bubbles are moving since the artifacts sizes are less than the resolution of the imaging system.

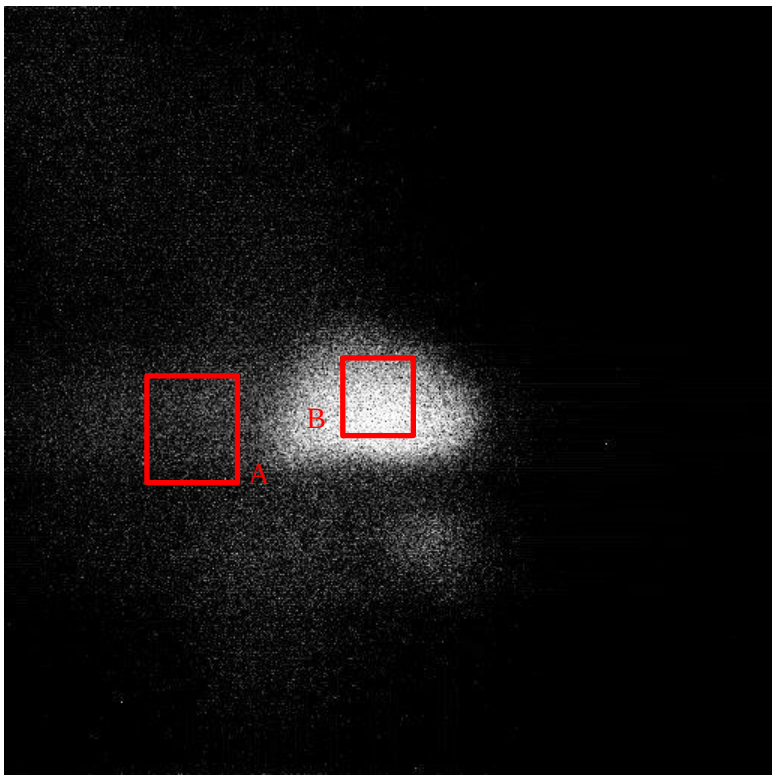

Figure **6-13**: Raw image taken at 500 fps at HFIR.

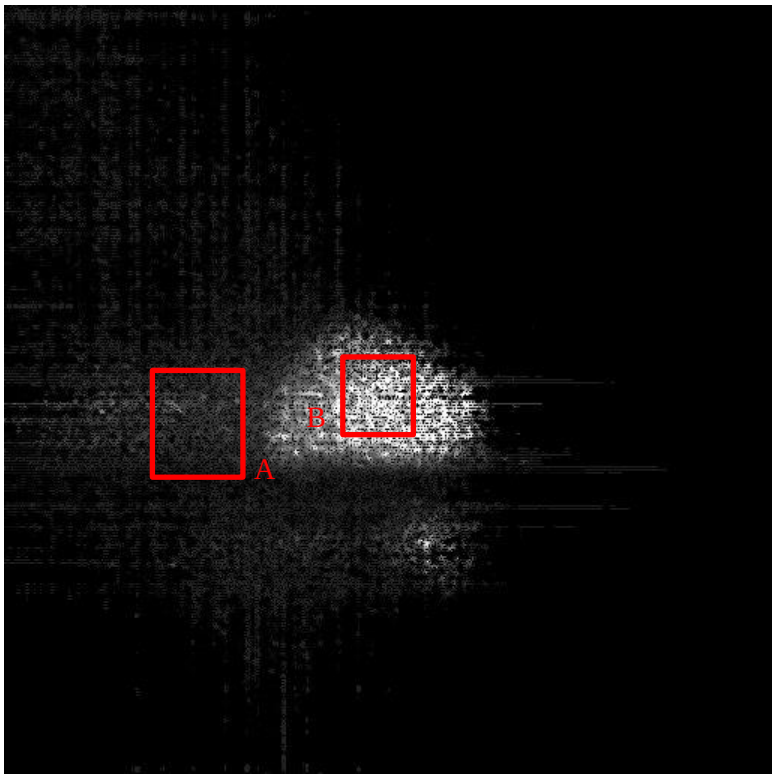

Figure **6-14**: Normalized image taken at 500 fps at HFIR.

Table 6-4 shows the SNR and CNR calculated for Figures 6-13 and 6-14. The SNR is calculated for the entire raw and normalized image stacks and are featured in Figures 6-15 and 6-16. In Figure 6-15 and 6-16, it can be seen that the peaks and valleys of the SNR values are becoming wider. This widening of the peaks is due to the faster frame rate capturing more frames of the bubbles as they pass the FOV.

Table **6-4**: Comparison of SNR and CNR from images taken at 500 fps from HFIR.

	Figure 6-13 (raw)		Figure 6-14 (normalized)	
	Mean	StdDev	Mean	StdDev
Full Image	22.664	19.940	0.940	1.185
Region A	7.665	7.841	0.323	0.242
Region B	52.201	14.808	2.503	1.770
SNR	1.137		0.793	
CNR	2.658		1.220	

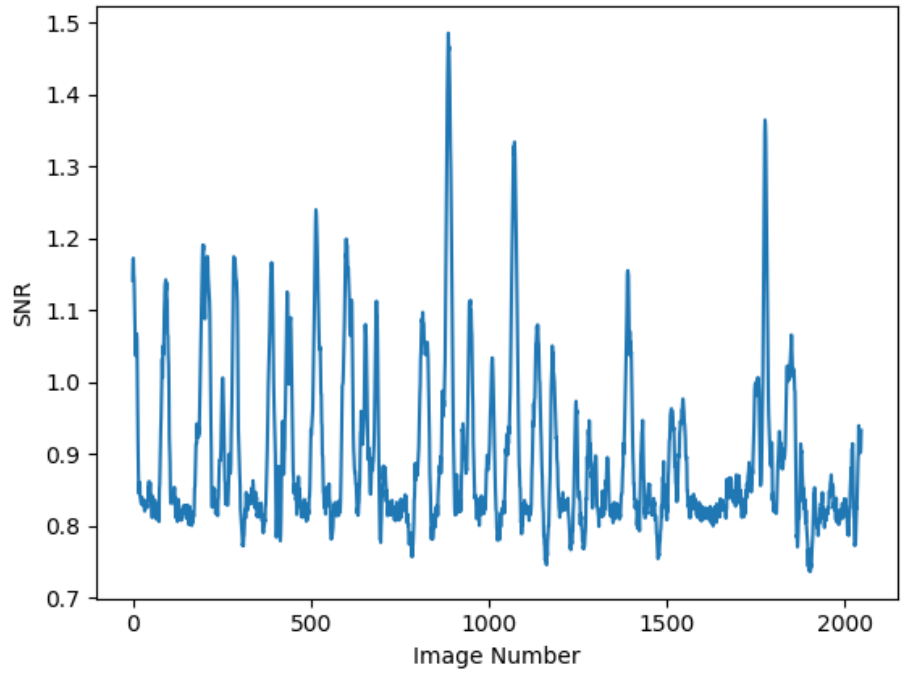

Figure **6-15**: SNR of raw 500 fps image stack from HFIR.

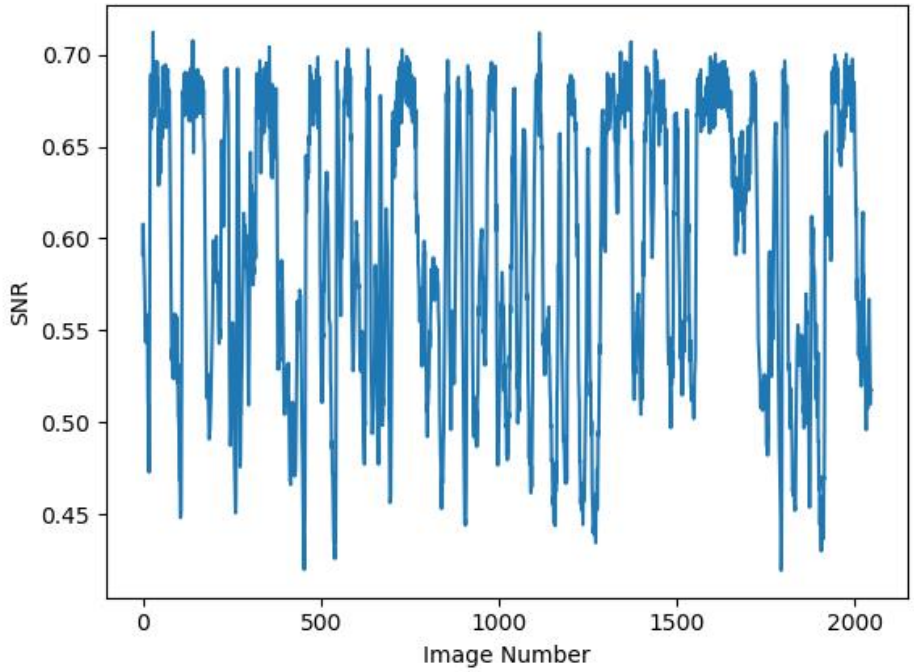

Figure **6-16**: SNR of normalized 500 fps image stack at HFIR.

1,000 fps Results

The highest frame rate that was achieved from the HFIR beamline was 1,000 fps. Figures 6-17 and 6-18 show images captured at 1,000 fps both raw and normalized, respectively. It can be seen by the images that the intensity of the images is reduced. With the lower intensity, the SNR and CNR are reduced as shown by their calculated values in Table 6-5.

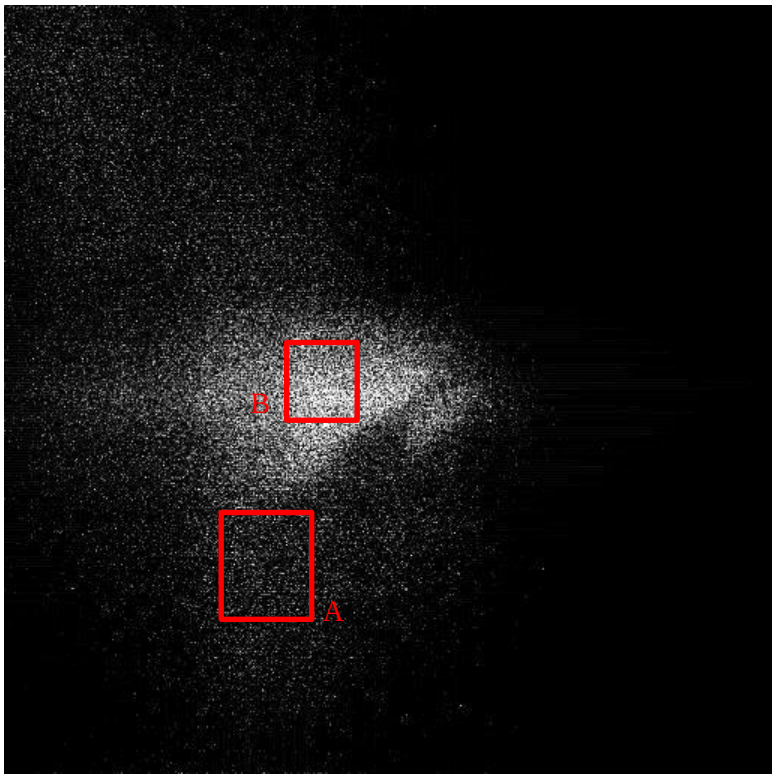

Figure **6-17**: Raw image taken at 1,000 fps at HFIR.

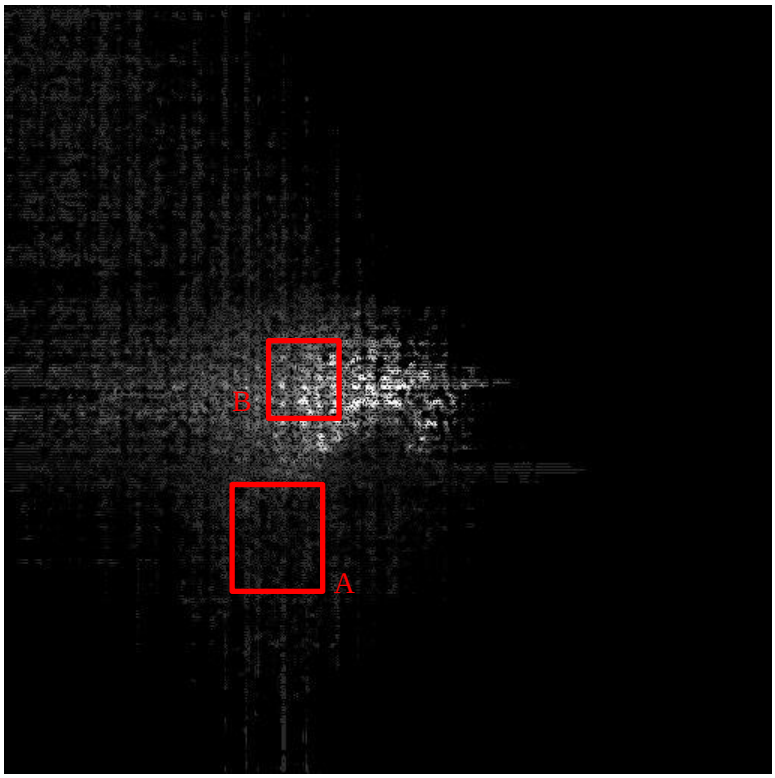

Figure **6-18**: Normalized image taken at 1,000 fps at HFIR.

At this frame rate the CNR and the SNR values reach the lowest of any of the frame rates. With an SNR less than one, more noise is being measured than the desired signal and that the image can vary greatly depending on the background noise. This lack of signal is caused by too few neutrons per frame detected and the corresponding little light per frame. This variation is shown in the SNR graphs of the total image stacks in Figures 6-19 and 6-20. Similar to SNR values of the 500 fps HFIR experiment, the SNR values of the 1,000 fps experiment also exhibit the widening of the peaks due to the increase in frame rate.

Table **6-5**: Comparison of SNR and CNR from images taken at 1,000 fps from HFIR.

	Figure 6-17 (raw)		Figure 6-18 (normalized)	
	Mean	StdDev	Mean	StdDev
Full Image	9.690	10.349	0.653	0.819
Region A	3.674	5.451	0.200	0.230
Region B	19.958	10.901	1.323	1.208
SNR	0.936		0.797	
CNR	1.336		0.913	

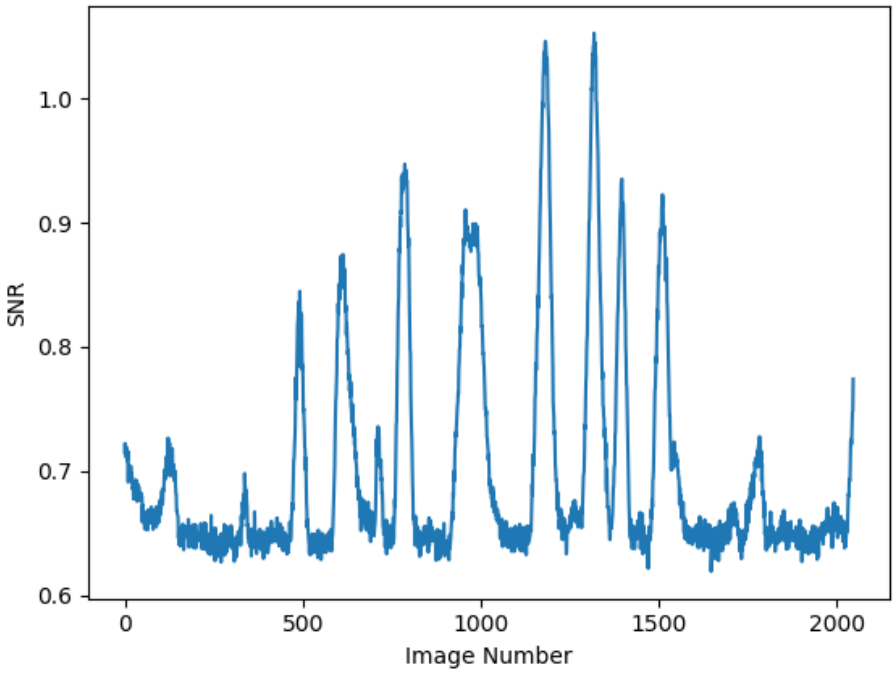

Figure **6-19**: SNR of raw 1,000 fps image stack from HFIR.

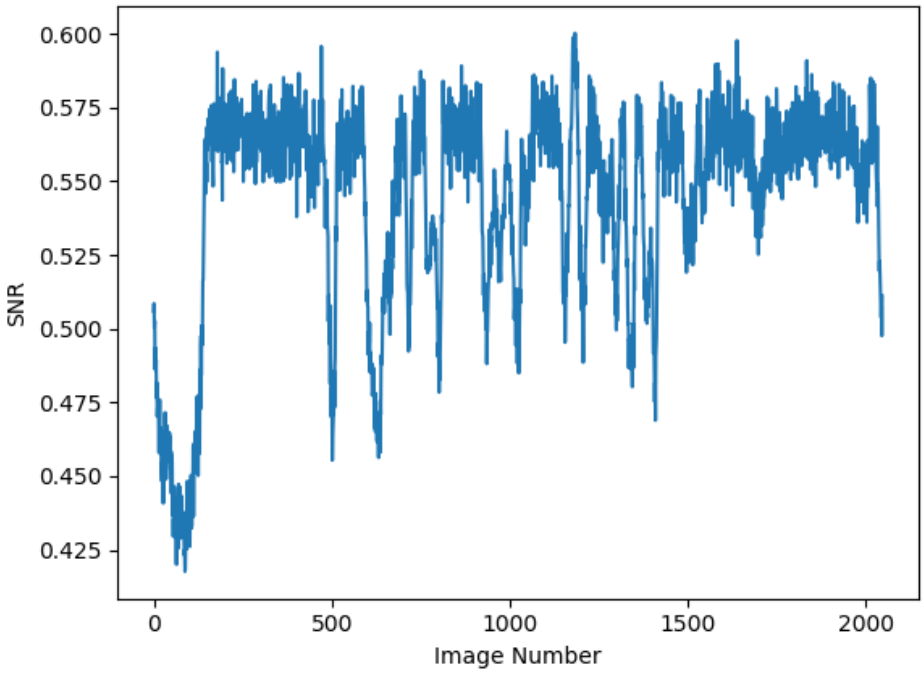

Figure **6-20**: SNR of normalized 1,000 fps image stack from HFIR.

Continuous Source Analysis

From the multiple experiments at HFIR there are multiple trends can be characterized from the acquired data. Since HFIR is a continuous neutron source, the SNR values of the images remain relatively constant for a single frame rate. Changes that occur in the SNR values are caused by the motion and size of bubbles passing the FOV and the rise and fall of the biases. When increasing frame rates, the SNR decreased due to the reduction of the amount of light captured by the camera's chip for each frame.

Table 6-6 highlights the SNR and CNR values of the chosen image featured in Chapter 6 which were obtained at HFIR. Examining the SNR of each of the frame rates, the value does not decrease with increase frame rate as expected. Instead of decreasing, for the frame rates of 60 fps

to 250 fps, the SNR of the raw images remains fairly constant. This phenomenon is due to the dependency bubble size in the FOV. By area, the 60 fps image has the smallest bubbles while the size in increased in the 125 fps and 250 fps images.

Table **6-6**: SNR and CNR values from randomly chosen raw and normalized images obtained at HFIR.

Frame Rate	60 fps		125 fps		250 fps		500 fps		1,000 fps	
Readout	512 x 512		512 x 512		512 x 512		512 x 512		512 x 512	
Image Type	Raw	Norm	Raw	Norm	Raw	Norm	Raw	Norm	Raw	Norm
SNR	1.488	1.863	1.625	1.998	1.493	1.502	1.137	0.793	0.936	0.797
CNR	3.379	6.357	7.210	6.383	3.862	2.131	2.658	1.220	1.336	0.913

Since the HIFR data spans measurement times as long as 34 seconds, many bubbles move into and out of the FOV of the imaging setup. As the bubble exits the FOV the biases become more impactful to the quality of the image. Figures 6-21 and 6-22 illustrate the change in the statistical error as both void fraction and frame rate change. With a bubble absent from the FOV and a void fraction of approximately zero, the statistical error can reach as high as 30% at 1,000 fps. Figures 6-21 and 6-22 also show that with a decrease in the frame rate the statistical error decreases.

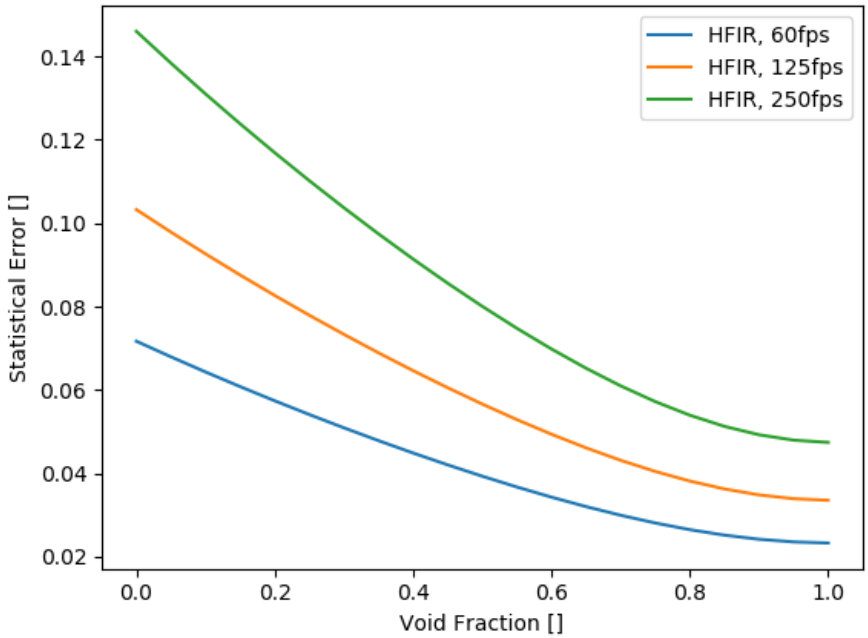

Figure **6-21**: Statistical error as a function of void fraction for HFIR at 60, 125, and 250 fps.

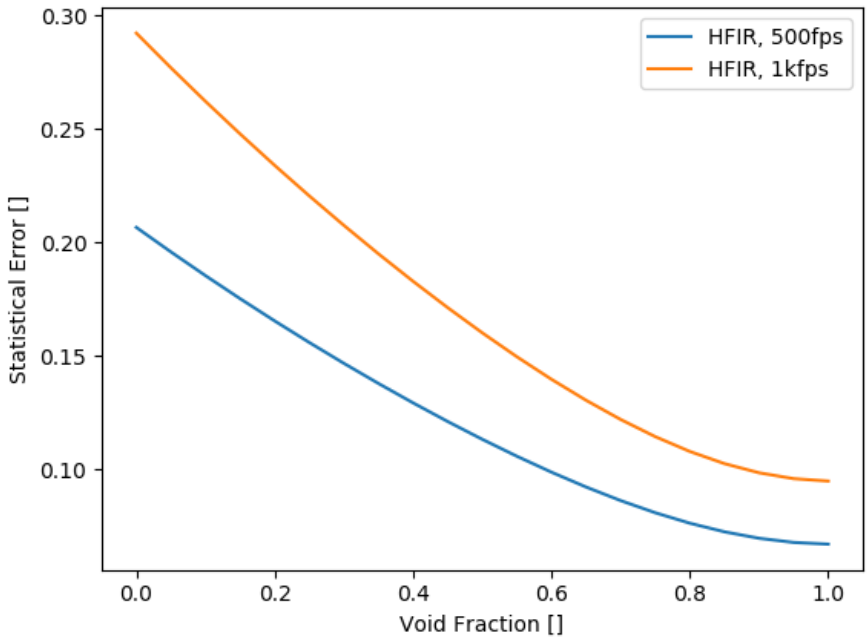

Figure **6-22**: Statistical error as a function of void fraction for HFIR at 500 and 1,000 fps.

The low count bias exhibits a similar behavior to the statistical error. Figures 6-23 and 6-24 show how the void fraction affects the low count bias values at different frame rates. A low void fractions the low count bias starts of high and decreases exponentially, approaching zero after a void fraction of 0.4. The low count bias also decreases as the frame rate is reduced due to the greater signal obtained per frame. Note that random numbers were used to generate the low count bias values and that Figures 6-23 and 6-24 are the results of multiple iterations.

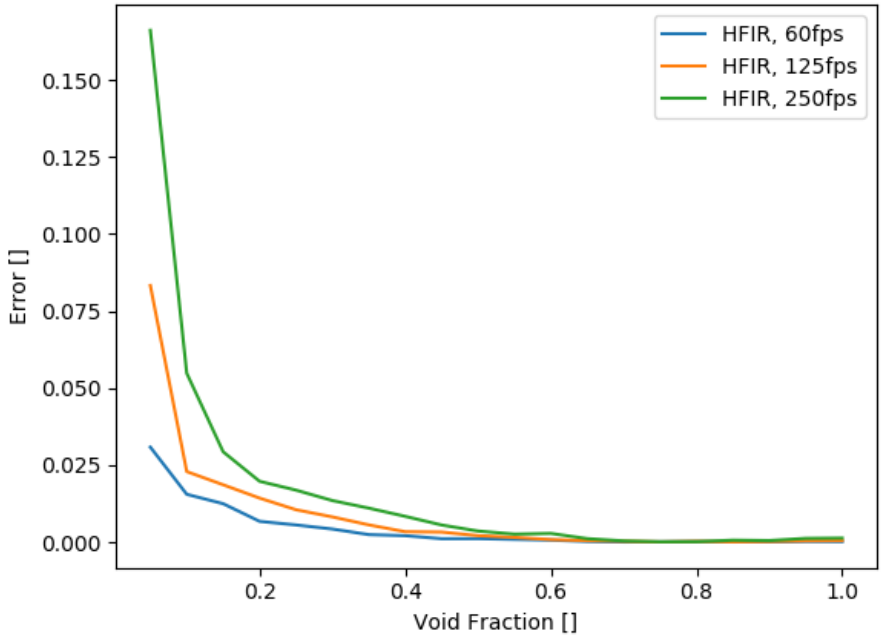

Figure **6-23**: Low count bias as a function of void fraction for HFIR at 60, 125 and 250 fps.

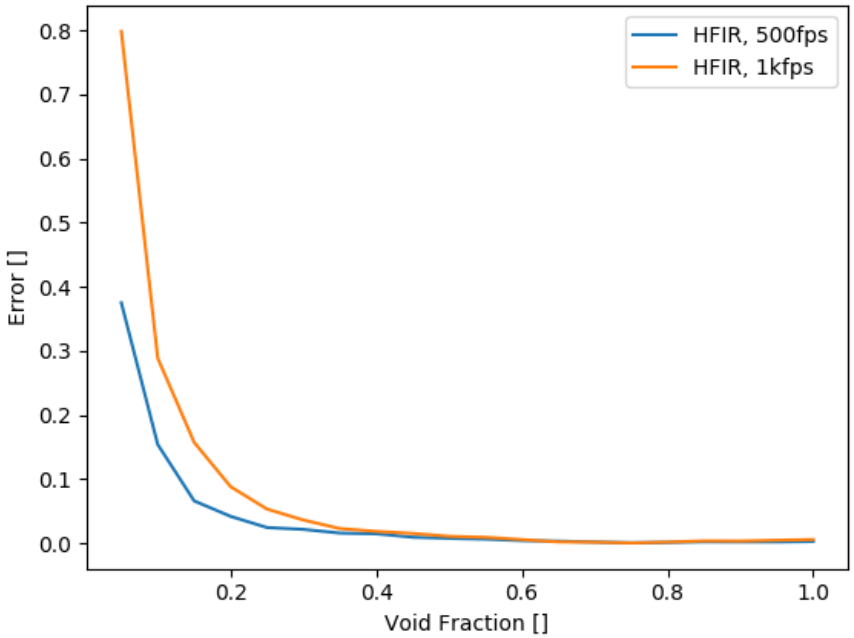

Figure **6-24**: Low count bias as a function of void fraction for HFIR at 500 and 1,000 fps.

Chapter 7

Comparison, Conclusions, and Future Recommendation

The intent of this work was to evaluate a new neutron imaging system utilizing pulses of a TRIGA reactor and would operate at frame rates in the kfps range and to determine whether bright flash imaging was a viable option. The experiments that were performed utilized the imaging beamline at the Penn State Breazeale Reactor as the bright flash source and as a reference the CG-1D beamline of the High Flux Isotope Reactor of Oak Ridge National Laboratory as a continuous source. Images of air-water two-phase flow in a simple bubbler were obtained from both sites using the same camera setup and acquisition speeds from 60 fps to 1,000 fps at HFIR and 500 fps to 4,000 fps as the Breazeale reactor. The images were evaluated both visually and quantitatively based on calculating the signal-to-noise ratio and the contrast-to-noise ratio to determine their quality.

Bright Flash and Continuous Source Comparison

Before looking at the finer details of both the SNR and the CNR it is clear that bright flash neutron imaging has an edge even on the strongest current continuous neutron sources and as a consequence higher frame rates were capable of being obtained. When looking into the specific quality of the images, comparing the images of 500 fps and 1,000 fps for both data from PSBR and HFIR, the PSBR images have a SNR and a CNR value multiple times than those of the HFIR images. Tables 7-1 and 7-2 show a comparison of the 500 fps and the 1,000 fps raw and normalized images obtained from both neutron sources. These results are due to the higher neutron flux that is created during the neutron pulse of the PSBR. The higher flux leads to a greater light output of the scintillator which in turn results in a greater number of photons absorbed on average by each pixel

of the camera's chip. This additional intensity can be seen in Figure 7-1 which compares the raw images from the PSRB and the HFIR experiments taken at 1,000 fps.

Table **7-1**: Comparison of SNR and CNR values from raw images obtained at the PSBR and HFIR.

	500 fps		1,000 fps	
	PSBR	HFIR	PSBR	HFIR
SNR	5.692	1.137	3.847	0.936
CNR	6.222	2.658	2.095	1.336

Table **7-2**: Comparison of SNR and CNR values from normalized images obtained at the PSBR and HFIR.

	500 fps		1,000 fps	
	PSBR	HFIR	PSBR	HFIR
SNR	5.526	0.793	5.047	0.797
CNR	14.031	1.220	24.562	0.913

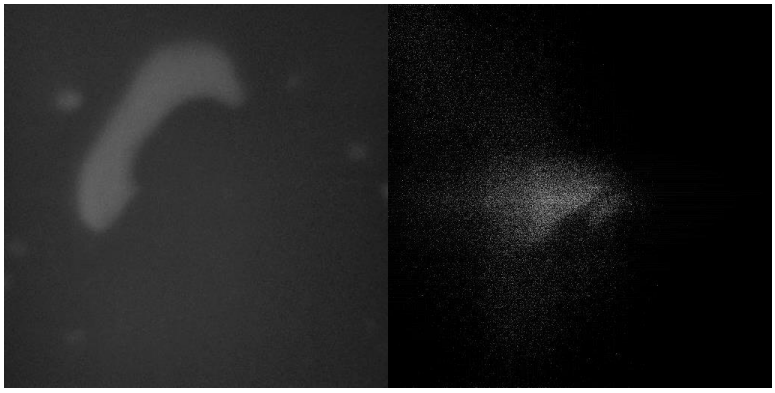

Figure **7-1**: (Left) Raw image taken at the PSBR at 1,000 fps. (Right) Raw image taken at HFIR at 1,000 fps. Both images are presented with the same grey scale values.

Looking at Figure 7-1, it appears that the image that was taken with the bright flash of the PSBR is less grainy than that of HFIR. This quality is caused by the number of neutrons detected per image pixels and the corresponding levels of Poisson noise present in each of the systems. With lower number of neutrons detected, bias effects become also more significant in affecting the quality of the image. Figure 7-2 and 7-3 illustrate the statistical error and the low count bias for each of the frame rates that the two neutron sources share, 500 fps and 1,000 fps. Due to the decrease in the neutron flux, the images taken at HFIR have a much higher statistical error and low

count bias. These biases reduce the quality of the image and lead to the grainy image that can be seen on the right of Figure 7-1.

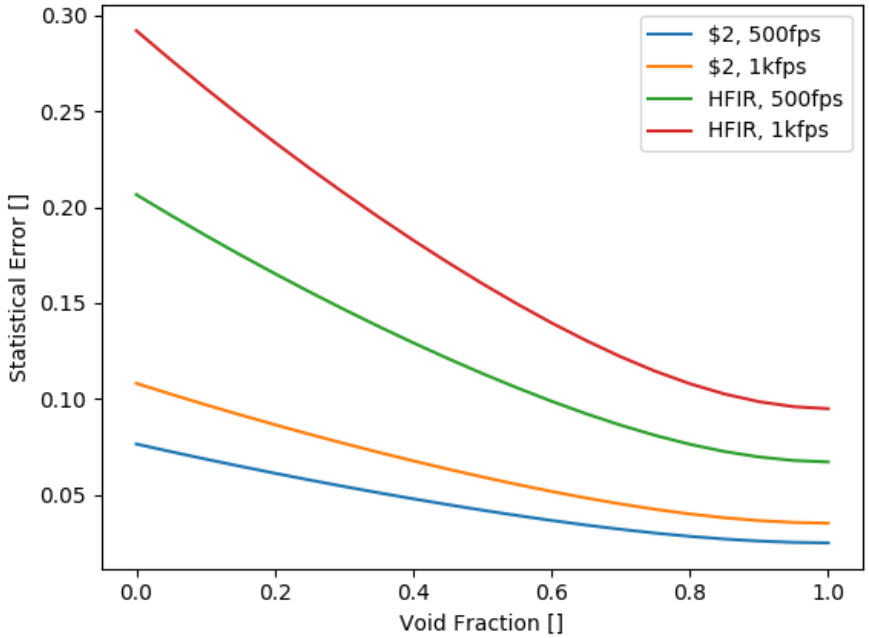

Figure **7-2**: Statistical error as a function of void fraction for PSBR and HFIR at 500 and 1,000 fps.

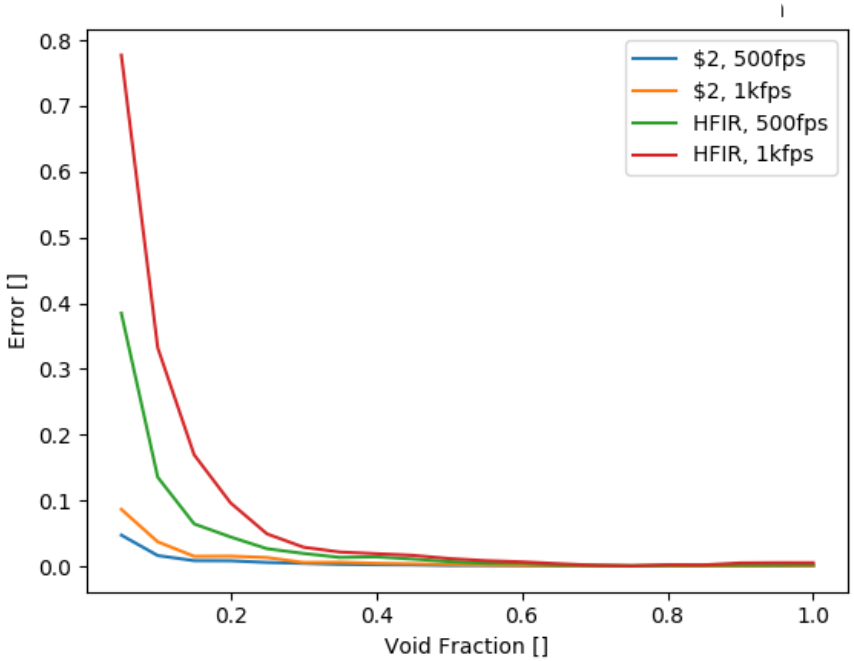

Figure **7-3**: Low count bias as a function of void fraction for PSBR and HFIR at 500 and 1,000 fps.

Bright flash imaging has some drawbacks that are not present with a continuous neutron source. One disadvantage is that the images are limited to several tens of milliseconds and does not allow capturing transients that last longer than that range. This pulse time scale is short and can lead to problems with system synchronization depending on the process of interest. Unlike the continuous source, the bright flash pulse cannot be repeated nonstop, only one pulse can be utilized every 15 – 20 minutes. In addition to the slow repeatability of the reactor pulse, the initial conditions are typically slightly varying from one neutron pulse to another. For slow and longer transients and for fluctuating processes that need to be sampled long for good statistics to be imaged, a continuous source can be more beneficial than a bright flash. However, once a process reaches increasing speeds, bright flash imaging would be required to limit the amount of motion artifacts.

Conclusions

From the experimental results of this thesis it is evident that the neutron camera setup paired with bright flash neutron imaging is superior compared to continuous neutron sources for imaging fast, noncyclic dynamic processes. The quality of the raw images from the bright flash imaging method were on average 4-5 times better than the raw images taken at the continuous source with the same frame rate. Due to the higher neutron flux of the bright flash, more light is generated in the scintillator which allows higher frame rates to be utilized with less biases. A frame rate of 4,000 fps was capable of being utilized at the PSBR while the maximum frame rate at HFIR was only 1,000 fps. These higher frame rates result in the ability to capture more rapid processes without the need for a cyclical nature as used in phase lock imaging. An additional benefit from increased frame rates is the reduction in motion artifacts. Increased frame rates do increase the biases created from statistical error and the normalization process. This thesis shows that the

neutron imaging camera setup developed at Penn State works and can be utilized for further studies into two-phase flows and other rapid processes.

Future Work

The experimental imaging method described in this thesis can be further improved. The CMOS camera that was utilized created a number of small errors that compounded as the experiments were performed. The observed problems included the limited memory, the interfacial software, biases of the chip, and the dead pixels. All of these problems could be mitigated or solved by the utilization of a newer camera and lens. It is also recommended to install a digital control system synchronizing and triggering the camera with the neutron pulse allowing for more precise capture of data and better utilization of the pulse. The control system would also enable images to be acquired at even higher imaging rates. Additional pulses higher than 2.5 $, if allowed by reactor licensing, could also contribute to achieving higher frame rates or better signal quality.

The technique, once optimized as explained above, should be deployed and tests on processes where one can leverage on the penetrating power of neutrons. These tests could include many thermohydraulic processes such as two-phase flows, flows and transition in supercritical fluids, cavitating flows, water hammers in piping, steam explosions, wavy film flows, or impact and explosion processes just to mention a few. As the logical next step, the technique could be further developed trying to utilize the fast neutrons available at the imaging beamline at Breazeale using appropriate converters/scintillator screens with the high-speed camera. This would open up the way for high speed imaging of more robust and attenuating process settings.

Static regions of interest caused problems when calculating the CNR for many of the images. A program could be made to follow and measure the size and shapes of bubbles. The program could be utilized to increase the accuracy of the void fraction, signal to noise ratio, and

the contrast to noise ratios. Additional values that could be obtained from this program would be the surface area of the bubble and its velocity.

As stated in the introduction, neutron imaging is still a growing field. The possibilities of the uses of bright flash imaging have only just begun and the advantages of high frame rate imaging are numerous. So much more could be understood if these processes were more readily available and better understood.

References

[1] Chadwick J. Possible Existence of a Neutron. *Nature* 1932; **129:**312

[2] Kallmann H. Neutron Radiography. *Research* 1948; 1:254-260

[3] Brenizer J.S., A review of significant advances in neutron imaging from conception to present. *Physics Procedia* doi.org/10.1016/j.phpro.2013.03.002

[4] Cho K.T., Turhan A., Lee J.H., Brenizer J.S., Heller A.K., Chi L., Mench M.M. Probing Water Transport in Polymer Electrolyte Fuel Cells with Neutron Radiography. *Nuclear Instruments and Methods in Physics Research Section A* 2008 doi.org/10.1016/j.nima.2009.01.144

[5] Robinson A.H., Barton J.P. High-Speed Motion Neutron Radiography. *Transactions of the American Nuclear Society* 1972; **15:**140

[6] Bossi R.H., Robinson A.H., Barton J.P. High-Speed Motion Neutron Radiography. *Nuclear Technology* 1982; 59, 363

[7] Wang Shou-Kong. High Speed Motion Neutron Radiography of Two-Phase Flow. Ph.D Dissertation Oregon State University 1981

[8] Nakamura H., Sibamoto Y., Anoda Y., Kukita Y., Mishima K., Hibiki T. Visualization of Simulated Molten-Fuel Behavior in a Pressure Vessel Lower Head Using High-Frame-Rate Neutron Radiography. *Nuclear Technology* 1999; 125:2, 213-224, DOI: 10.13182/NT99-A2943

[9] Sibamoto Y., Kukita Y., Nakamura H. Visualization and Measurement of Subcooled Water Jet Injection into High-Temperature Melt by Using High-Frame-Rate Neutron Radiography. *Nuclear Technology* 2002; 139:3, 205-220, DOI: 10.13182/NT02-A3314

[10] Tremsin A.S., Lerche M., Schillinger B., Feller W.B. Bright flash neutron radiography capability of the research reactor at McClellan Nuclear Research Center. *Nuclear Instruments and Methods in Physics Research A* 2014; 748, 46-53 doi.org/10.1016/j.nima.2014.02.034

[11] Lerche M., Tremsin A.S., Schillinger B. Bright Flash Neutron Radiography at the McClellan Nuclear Research Reactor. *Physics Procedia* 2015; 69, 299-303 doi.org/10.1016/j.phpro.2015.07.042

[12] Zboray R, Trtik P. 800 fps neutron radiography of air-water two-phase flow. *MethodsX* 2018; 5, 96-102 doi.org/10.1016/j.mex.2018.01.008

[13] Ott K.O., Neuhold R.J. Introductory Nuclear Reactor Dynamics. *American Nuclear Society* 1985

[14] Hetrick D. L. Dynamics of Nuclear Reactors, University of Chicago Press, 1971

[15] Binney S.E., Reese S.R., Pratt D.S. *University Research Reactors: Contributing to the National Scientific and Engineering Infrastructure from 1953 to 2000 and Beyond*. *National Organization of Test, Research and Training Reactors* 2000

[16] Uçar D., Ünlü K., Heidrich B.J., Ivanov K.N., Avramova M.N. Neutronic Design and Analyses of a New Core-Moderator Assembly and Neutron Beam Ports for the Penn State Breazeale Reactor. Ph.D. Thesis The Pennsylvania State University, PA,

[17] Cimbala J.M., Brenizer J.S., Po-Ya Chuang A., Hanna S., Thomas Conroy C., El-Ganayni A., *et al.* Study of a loop heat pipe using neutron radiography. *Applied Radiation and Isotopes* 2004; 61, 701-705, doi.org/10.1016/j.apradiso.2004.03.104

[18] Brenizer J., Mench M.M., Ünlü K., Heller K., Turhan A., Shi L., Kowal J.J., Neutron Imaging System Improvements. available at http://www.rsec.psu.edu (accessed July 10th, 2018)

[19] scintacor, Neutron Screens, available at scintacor.com/wp-content/uploads/2015/09/Datasheet-Neutron-Screens-High-Res.pdf, (accessed on July 20th, 2018)

[20] Photron. FASTCAM-ultima 512 Harware Manual Rev. 1.2. 2015

[21] Baechler S., Kardjilov N., Dierick M., Jolie J., Kühne G., Lehmann E., Materna T. New features in cold neutron radiography and tomography Part I: thinner scintillators and a neutron

velocity selector to improve the spatial resolution, *Nuclear Instruments and Methods in Physics Research A* 2002; 491, 481-491 doi.org/10.1016/S0168-9002(02)01238-X

[22] Gnezdilov I.I., Dedenko G.L., Ibragimov R.F., Idalov V.A., Kadilin V.V., Kaplun A.A., Klemetiev A.V., Mukhin V.I., Taraskin A.A, Turin E.M., Zaripov R.N. Optimization of the neuron detector design based on the ^{6}LiF/ZnS(Ag) scintillation screens for the GAMMA-400 space observatory. *Physics Procedia* 2015; 74, 199 – 205 doi.org/10.1016/j.phpro.2015.09.192

[23] Photron. Photron FASTCAM Viewer for High Speed Digital Imaging User's Manual Ver. 3 Rev. 2.10E Configuration section.

[24] Photron. Photron FASTCAM Viewer for High Speed Digital Imaging User's Manual Ver. 3 Rev. 2.11E Operation section.

[25] Abramoff, M.D., Magalhães P.J., Ram S.J. Image processing with imagej. *Biophotonics International* 2004; 11, 36-42.

[26] Munch B., Trtik P., Marone F., Stampanoni M. Stripe and ring artifact removal with combined wavelet Fourier filtering. *Optics Express* May 2009; 17 (10), 8567-8591. doi.org/10.1364/OE.17.008567

[27] Liang M., Kang K., Zhang Z., Lou X. Low count bias in gamma ray thickness detection and its correction *Nuclear Instruments and Methods in Physics Research A.* 2006; **568** 912–4 doi.org/10.1016/j.nima.2006.08.097

[28] Jayakumar P., Munshi P. A comprehensive study of measurement uncertainty in tomographic reconstruction of void profiles in a mercury-nitrogen flow. *Exp. Fluids* 1999; 26, 535, dx.doi.org/10.1007/s003480050320.

[29] *History of the Hight Flux Isotope Reactor*. Oak Ridge National Laboratory. web page. (accessed on March 1st, 2019)

[30] Santodonato L, Bilheux H, Bailey B, Bilheux J, Nguyen P, Tremsin A, Selby D, Walker L The CG-1D neutron imaging beamline at the Oak Ridge National Laboratory High Flux Isotope Reactor. *Physics Procedia* 2015; 69, 104-108, doi.org/10.1016/j.phpro.2015.07.015

[31] Lani, C., Zboray, R. Development of a high frame rate neutron imaging method for two-phase flows. *Nuclear Instruments and Methods in Physics research Section A*. 2018 In Press. doi.org/10.1016/j.nima.2018.12.022

[32] Nakamura M., Sugimoto K., Asano H., Murakawa H., Takenaka N., Mochiki N. Visualization of oil behavior in a small 4-cycle engine with electrical motoring by neutron radiography, *Nuclear Instruments and Methods in Physics research Section A*. 2009; 605, 204–207. doi.org/10.1016/j.nima.2009.01.155

Appendix A

Python Code for Image Normalization

```
import os
import sys
import numpy as np
import glob
import scipy.ndimage
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from matplotlib import gridspec
#matplotlib notebook

root_folder = os.path.dirname(os.getcwd())
sys.path.append(root_folder)
import NeuNorm as neunorm
from NeuNorm.normalization import Normalization
from NeuNorm.roi import ROI


'''imports each of the data sets'''
path_im                                                                    =
glob.glob('C:/Users/Chad/Desktop/Cleaned_Pulses/H60fps/60fps_bubbles/*.tif')
path_ob                                                                    =
glob.glob('C:/Users/Chad/Desktop/Cleaned_Pulses/H60fps/60fps_flat_filled/*.tif'
)
#path_dark                                                                 =
glob.glob('C:/Users/Chad/Desktop/Pulses/4000_gain4_dark_2halfdollar_tiff/*.tif'
)
path_emp                                                                   =
glob.glob('C:/Users/Chad/Desktop/Cleaned_Pulses/H60fps/60fps_flat_empty/*.tif')

'''Determines the maximum index for each of the data sets'''
o_norm = Normalization()

o_norm.load(file=path_im,          notebook=False,          gamma_filter=False,
gamma_threshold=.5, dead_filter=False,
            dead_threshold=.5, gamma_rad=2, dead_rad=2)
o_norm.load(file=path_ob,  data_type='ob',  notebook=False,  gamma_filter=False,
gamma_threshold=.5, dead_filter=False,
            dead_threshold=.5, gamma_rad=2, dead_rad=2)
o_norm.load(file=path_emp, data_type='emp', notebook=False, gamma_filter=False,
gamma_threshold=.5, dead_filter=False,
            dead_threshold=.5, gamma_rad=2, dead_rad=2)

sample = o_norm.get_sample_data()
ob = o_norm.get_ob_data()
emp = o_norm.get_emp_data()
avg_samp = np.empty(len(sample))
avg_ob = np.empty(len(ob))
avg_emp = np.empty(len(emp))

i = 0
while i < len(sample):
```

```
    avg_samp[i] = np.average(sample[i])
    i = i+1
j = 0
while j < len(ob):
    avg_ob[j] = np.average(ob[j])
    j = j+1
k = 0
while k < len(emp):
    avg_emp[k] = np.average(emp[k])
    k = k +1

index_samp = np.argmax(avg_samp)
index_ob = np.argmax(avg_ob)
index_emp = np.argmax(avg_emp)

'''
#Find the largest arrays and create 3 arrays of that size
'''
if len(path_im) >= len(path_emp) and len(path_ob) >= len(path_emp):
    total_size = len(path_im)+len(path_ob)
elif len(path_ob) >= len(path_im) and len(path_emp) >= len(path_im):
    total_size = len(path_emp)+len(path_ob)
else:
    total_size = len(path_emp)+len(path_im)

if total_size % 2 == 0:
    mid_pt = total_size/2
else:
    total_size = total_size + 1
    mid_pt = total_size/2

ray_im = np.empty(total_size, dtype='object')
im_diff = mid_pt - index_samp
ray_ob = np.empty(total_size, dtype='object')
ob_diff = mid_pt - index_ob
#ray_dark = np.empty(total_size, dtype='object')
#dark_diff = mid_pt - (len(path_dark)//2)
ray_emp = np.empty(total_size, dtype='object')
emp_diff = mid_pt - index_emp
bool_ray = np.empty(total_size, dtype= np.bool)

i = 0
while i < len(path_im):
    ray_im[i + int(im_diff)] = path_im[i]
    i = i+1
j = 0
while j < len(path_ob):
    ray_ob[j + int(ob_diff)] = path_ob[j]
    j = j+1
k = 0
#while k < len(path_dark):
#    ray_dark[k + int(dark_diff)] = path_dark[k]
#    k = k+1
z = 0
while z < len(path_emp):
    ray_emp[z + int(emp_diff)] = path_emp[z]
    z = z+1

i = 0
count = 0
while i < total_size:
```

```
    if ray_ob[i] is None or ray_im[i] is None or ray_emp[i] is None:
        bool_ray[i] = False
    else:
        bool_ray[i] = True
        count = count + 1
    i = i+1

final_samp = np.empty(count, dtype='object')
final_ob = np.empty(count, dtype='object')
#final_dark = np.empty(count, dtype='object')
final_emp = np.empty(count, dtype='object')

i = 0
j = 0
while i < total_size:
    if bool_ray[i]:
        final_samp[j] = ray_im[i]
        final_ob[j] = ray_ob[i]
        #final_dark[j] = ray_dark[i]
        final_emp[j] = ray_emp[i]
        j = j+1
    i = i+1

o_norm = Normalization()
i = 0
while i < count:
    o_norm.load(file=final_samp[i],      notebook=False,      gamma_filter=False,
gamma_threshold=.5, dead_filter=False,
                dead_threshold=.5, gamma_rad=2, dead_rad=2)
    o_norm.load(file=final_ob[i],         data_type='ob',         notebook=False,
gamma_filter=False, gamma_threshold=.5,
                dead_filter=False, dead_threshold=.5, gamma_rad=2, dead_rad=2)
    #o_norm.load(file=final_dark[i],       data_type='df',       notebook=False,
gamma_filter=False, gamma_threshold=.5,
    #            dead_filter=False, dead_threshold=.5, gamma_rad=2, dead_rad=2)
    o_norm.load(file=final_emp[i],        data_type='emp',        notebook=False,
gamma_filter=False, gamma_threshold=.5,
                dead_filter=False, dead_threshold=.5, gamma_rad=2, dead_rad=2)
    i = i+1
x0 = 0
y0 = 0
width = 511
height = 511

norm_roi = ROI(x0=x0, y0=y0, width=width, height=height)
o_norm.normalization(roi=norm_roi)

#normalized_data = o_norm.data['normalized']

#np.shape(normalized_data)

#roi_to_keep = ROI(x0=0, y0=0, width=2, height=2)
#o_norm.crop(roi=roi_to_keep)

#norm_crop = o_norm.data['normalized']
#np.shape(norm_crop)

o_norm.export(folder='C:/Users/Chad/Desktop/Cleaned_Pulses/H6000000fps/normaliz
ed', file_type='tif')
```

Appendix B

Supporting Python Function

```
from pathlib import Path
#import tomopy as tp
import numpy as np
import os
import copy
import time
import math
from scipy.ndimage import convolve
from scipy.ndimage import median_filter
from scipy.ndimage import center_of_mass


from NeuNorm.loader import load_hdf, load_tiff, load_fits
from NeuNorm.exporter import make_fits, make_tif
from NeuNorm.roi import ROI
from NeuNorm._utilities import get_sorted_list_images, average_df


class Normalization(object):


    def __init__(self):
        self.shape = {'width': np.NaN,
                      'height': np.NaN}
        self.dict_image = { 'data': [],
                            'oscilation': [],
                            'file_name': [],
                            'metadata': [],
                            'shape': self.shape.copy()}
        self.dict_ob = {'data': [],
                        'oscilation': [],
                        'metadata': [],
                        'file_name': [],
                        'data_mean': [],
                        'shape': self.shape.copy()}
        self.dict_emp = {'data': [],
                        'oscilation': [],
                        'metadata': [],
                        'file_name': [],
                        'data_mean': [],
                        'shape': self.shape.copy()}
        self.dict_df = {'data': [],
                        'metadata': [],
                        'data_average': [],
                        'file_name': [],
                        'shape': self.shape.copy()}

        __roi_dict = {'x0': np.NaN,
                      'x1': np.NaN,
                      'y0': np.NaN,
                      'y1': np.NaN}
        self.roi = {'normalization': __roi_dict.copy(),
                    'crop': __roi_dict.copy()}
```

```
        self.__exec_process_status = {'df_correction': False,
                                      'normalization': False,
                                      'crop': False,
                                      'oscillation': False,
                                      'bin': False}

        self.data = {}
        self.data['sample'] = self.dict_image
        self.data['ob'] = self.dict_ob
        self.data['emp'] = self.dict_emp
        self.data['df'] = self.dict_df
        self.data['normalized'] = []
        self.export_file_name = []

    def load(self, file='', folder='', data=[], data_type='sample',
             gamma_filter=[], notebook=[], gamma_threshold=0.1, gamma_rad=1,
             dead_filter=[], dead_threshold=0.1, dead_rad=1):
        '''
        Function to read individual files or entire files from folder specify for
the given
        data type

        Parameters:
           file: full path to file
           folder: full path to folder containing files to load
           data: numpy array
           data_type: ['sample', 'ob', 'df' , 'emp']
           gamma_filter: boolean (default True) apply or not gamma filtering to
the data loaded
           notebooks: boolean (default False) turn on this option if you run the
library from a
             notebook to have a progress bar displayed showing you the progress
of the loading
           dead_filter: boolean (default True) apply or not dead pixel filtering
to the data loaded
           gamma/dead_rad: Pixel radius for either the gamma or dead_filter

        Algorithm won't be allowed to run if any of the main algorithm have been
run already, such as
        oscillation, crop, binning, df_correction.

        '''

        list_exec_flag = [_flag for _flag in self.__exec_process_status.values()]
        box1 = None
        if True in list_exec_flag:
            raise IOError("Operation not allowed as you already worked on this
data set!")

        if notebook:
            from ipywidgets import widgets
            from IPython.core.display import display

        if not file == '':
            if isinstance(file, str):
                self.load_file(file=file,                    data_type=data_type,
gamma_threshold=gamma_threshold,                    gamma_rad=gamma_rad,
dead_threshold=dead_threshold, dead_rad=dead_rad)
            elif isinstance(file, list):
                if notebook:
                    # turn on progress bar
```

```
                    _message = "Loading {}".format(data_type)
                    box1 = widgets.HBox([widgets.Label(_message,

layout=widgets.Layout(width='20%')),
                                         widgets.IntProgress(max=len(file)),
                                         widgets.Label("Time remaining:",

layout=widgets.Layout(width='10%')),
                                         widgets.Label(" >> calculating << ")])
                    display(box1)
                    w1 = box1.children[1]
                    time_remaining_ui = box1.children[-1]

                start_time = time.time()
                for _index, _file in enumerate(file):
                    self.load_file(file=_file, data_type=data_type,
                                   gamma_filter=gamma_filter,
                                   gamma_threshold=gamma_threshold,
                                   gamma_rad=gamma_rad,
                                   dead_filter=dead_filter,
                                   dead_threshold=dead_threshold,
                                   dead_rad=dead_rad)
                    if notebook:
                        w1.value = _index+1
                        end_time = time.time()
                        takes_its_going_to_take                                    =
self.calculate_how_long_its_going_to_take(index_we_are=_index + 1,

time_it_took_so_far=end_time - start_time,

total_number_of_loop=len(file))
                        time_remaining_ui.value                                    =
"{}".format(takes_its_going_to_take)

                if notebook:
                    box1.close()

        elif not folder == '':
            # load all files from folder
            list_images = get_sorted_list_images(folder=folder)
            if notebook:
                # turn on progress bar
                _message = "Loading {}".format(data_type)
                box1 = widgets.HBox([widgets.Label(_message,

layout=widgets.Layout(width='20%')),
                                     widgets.IntProgress(max=len(list_images)),
                                     widgets.Label("Time remaining:",

layout=widgets.Layout(width='10%')),
                                     widgets.Label(" >> calculating << ")])
                display(box1)
                w1 = box1.children[1]
                time_remaining_ui = box1.children[-1]

            start_time = time.time()
            for _index, _image in enumerate(list_images):
                full_path_image = os.path.join(folder, _image)
                self.load_file(file=full_path_image, data_type=data_type,
                               gamma_filter=gamma_filter,
                               gamma_threshold=gamma_threshold,
```

```
                                gamma_rad=gamma_rad,
                                dead_filter=dead_filter,
                                dead_threshold=dead_threshold,
                                dead_rad=dead_rad)
                if notebook:
                    # update progress bar
                    w1.value = _index+1
                    end_time = time.time()
                    takes_its_going_to_take                                    =
self.calculate_how_long_its_going_to_take(index_we_are=_index+1,

time_it_took_so_far=end_time-start_time,

total_number_of_loop=len(list_images))
                    time_remaining_ui.value                                    =
"{}".format(takes_its_going_to_take)

            if notebook:
                box1.close()

        elif not data == []:
            self.load_data(data=data, data_type=data_type)

    def       calculate_how_long_its_going_to_take(self,       index_we_are=-1,
time_it_took_so_far=0, total_number_of_loop=1):
        time_per_loop = time_it_took_so_far / index_we_are
        total_time_it_will_take = time_per_loop * total_number_of_loop
        time_left = total_time_it_will_take - time_per_loop * index_we_are

        # convert to nice format h mn and seconds
        m, s = divmod(time_left, 60)
        h, m = divmod(m, 60)

        if h == 0:
            if m == 0:
                return "%02ds" %(s)
            else:
                return "%02dmn %02ds" %(m, s)
        else:
            return "%dh %02dmn %02ds" % (h, m, s)

    def load_data(self, data=[], data_type='sample', notebook=False):
        '''Function to save the data already loaded as arrays

        Paramters:
        ==========
        data: np array 2D or 3D
        data_type: string ('sample')
        notebook: boolean (default False) turn on this option if you run the
library from a
             notebook to have a progress bar displayed showing you the progress
of the loading
        '''
        if notebook:
            from ipywidgets import widgets
            from IPython.core.display import display

        if len(np.shape(data)) > 2:
            if notebook:
                _message = "Loading {}".format(data_type)
                box1 = widgets.HBox([widgets.Label(_message,
```

```
layout=widgets.Layout(width='20%')),

widgets.IntProgress(max=len(list_images))])
                display(box1)
                w1 = box1.children[1]

            for _index, _data in enumerate(data):
                _data = _data.astype(float)
                self.__load_individual_data(data=_data, data_type=data_type)
                if notebook:
                    # update progress bar
                    w1.value = _index+1

            if notebook:
                box1.close()

        else:
            data = data.astype(float)
            self.__load_individual_data(data=data, data_type=data_type)

    def __load_individual_data(self, data=[], data_type='sample'):
        self.data[data_type]['data'].append(data)
        index = len(self.data[data_type]['data'])
        self.data[data_type]['file_name'].append("image_{:04}".format(index))
        self.data[data_type]['metadata'].append('')
        self.save_or_check_shape(data=data, data_type=data_type)

    def   load_file(self,   file='',   data_type='sample',   gamma_filter=True,
gamma_threshold=0.1,    gamma_rad=1,    dead_filter=True,    dead_threshold=0.1,
dead_rad=1):
        """
        Function to read data from the specified path, it can read FITS, TIFF and
HDF.

        Parameters
        ----------
        file : string_like
            Path of the input file with his extention.
        data_type: ['sample', 'df']
        gamma_filter: Boolean (default True) apply or not gamma filtering

        Notes
        -----
        In case of corrupted header it skips the header and reads the raw data.
        For the HDF format you need to specify the hierarchy.
        """

        my_file = Path(file)
        if my_file.is_file():
            metadata = {}
            if file.lower().endswith('.fits'):
                data = np.array(load_fits(my_file), dtype=np.float)
            elif file.lower().endswith(('.tiff','.tif')) :
                [data, metadata] = load_tiff(my_file)
                data = np.array(data, dtype=np.float)
            elif
file.lower().endswith(('.hdf','.h4','.hdf4','.he2','h5','.hdf5','.he5')):
                data = np.array(load_hdf(my_file), dtype=np.float)
            else:
```

```
                raise OSError('file extension not yet implemented....Do it your
own way!')

            if gamma_filter:
                data             =             self._gamma_filtering(data=data,
gamma_threshold=gamma_threshold, gamma_rad=gamma_rad)
            if dead_filter:
                data             =             self._dead_filtereing(data=data,
dead_threshold=dead_threshold, dead_rad=dead_rad)

            data = np.squeeze(data)

            self.data[data_type]['data'].append(data)
            self.data[data_type]['metadata'].append(metadata)
            self.data[data_type]['file_name'].append(file)
            self.save_or_check_shape(data=data, data_type=data_type)

        else:
            raise OSError("The file name does not exist")

    def _gamma_filtering(self, data=[], gamma_threshold=0.1, gamma_rad=1):
        '''perform gamma filtering on the data

        Algorithm looks for all the very hight counts

        if self.gamma_filter_threshold * pixels[x,y] > mean_counts(data) then
this pixel counts
        is replaced by the average value of the 8 pixels surrounding him

        Parameters:
        ===========
        data: numpy 2D array

        Returns:
        =======
        numpy 2D array
        '''
        if data == []:
            raise ValueError("Data array is empty!")

        data_gamma_filtered = np.copy(data)
        mean_counts = np.mean(data_gamma_filtered)
        gamma_indexes   =   np.where(data_gamma_filtered   *   gamma_threshold   >
mean_counts)

        mean_kernel  =  np.ones([gamma_rad  *  2  +  1,  gamma_rad  *  2  +  1])  /
(((gamma_rad*2+1)**2)-1)
        mean_kernel[gamma_rad, gamma_rad] = 0

        convolved_data     =     convolve(data_gamma_filtered,     mean_kernel,
mode='constant')

        data_gamma_filtered[gamma_indexes] = convolved_data[gamma_indexes]

        #      data_gamma_filtered      =      median_filter(data_gamma_filtered,
size=gamma_rad)
        return data_gamma_filtered

        # data_gamma_filtered = np.copy(data)
        #
        # # find mean counts
```

```
        # mean_counts = np.mean(data_gamma_filtered)
        #
        # thresolded_data_gamma_filtered = data_gamma_filtered * threshold
        #
        # # get pixels where value is above threshold
        # position = []
        # [height, width] = np.shape(data_gamma_filtered)
        # for _x in np.arange(width):
        #     for _y in np.arange(height):
        #         if thresolded_data_gamma_filtered[_y, _x] > mean_counts:
        #             position.append([_y, _x])
        #
        # convolve entire image using 3x3 kerne
        # mean_kernel = np.array([[1,1,1], [1,0,1], [1,1,1]]) / 8.0
        #    convolved_data    =    convolve(data_gamma_filtered,    mean_kernel,
mode='constant')
        #
        # # replace only pixel above threshold by convolved data
        # for _coordinates in position:
        #     [_y, _x] = _coordinates
        #     data_gamma_filtered[_y, _x] = convolved_data[_y, _x]
        #
        # return data_gamma_filtered

    def _dead_filtereing(self, data=[], dead_threshold=0.1, dead_rad=1):
        '''perform dead pixel filtering on the data

                Algorithm looks for all the very low counts

                if self.dead_filter_threshold * pixels[x,y] > mean_counts(data)
then this pixel counts
                is replaced by the average value of the 8 pixels surrounding him

                Parameters:
                ===========
                data: numpy 2D array

                Returns:
                =======
                numpy 2D array
                '''
        if data == []:
            raise ValueError("Data array is empty!")

        data_dead_filtered = np.copy(data)
        mean_counts = np.mean(data_dead_filtered)
        dead_indexes   =   np.where(data_dead_filtered   *   dead_threshold   <
mean_counts)

        mean_kernel  =  np.ones([dead_rad  *  2  +  1,  dead_rad  *  2  +  1])  /
(((dead_rad*2+1)**2)-1)
        mean_kernel[dead_rad, dead_rad] = 0

        convolved_data     =     convolve(data_dead_filtered,     mean_kernel,
mode='constant')

        data_dead_filtered[dead_indexes] = convolved_data[dead_indexes]
        #data_dead_filtered = median_filter(data_dead_filtered, size=dead_rad)
        return data_dead_filtered

    def save_or_check_shape(self, data=[], data_type='sample'):
```

```
        '''save the shape for the first data loaded (of each type) otherwise
        check the size match

        Raises:
        IOError if size do not match
        '''
        [height, width] = np.shape(data)
        if np.isnan(self.data[data_type]['shape']['height']):
            _shape = self.shape.copy()
            _shape['height'] = height
            _shape['width'] = width
            self.data[data_type]['shape'] = _shape
        else:
            _prev_width = self.data[data_type]['shape']['width']
            _prev_height = self.data[data_type]['shape']['height']

            if (not (_prev_width == width)) or (not (_prev_height == height)):
                raise IOError("Shape of {} do not match previous loaded data
set!".format(data_type))

    def normalization(self, roi=None, force=False, force_mean_ob=False,
notebook=False):
        '''normalization of the data

        Parameters:
        ===========
        roi: ROI object or list of ROI objects that defines the region of the
sample and OB that have to match
        in intensity
        force: boolean (default False) that if True will force the normalization
to occur, even if it had been
        run before with the same data set
        notebook: boolean (default False) turn on this option if you run the
library from a
             notebook to have a progress bar displayed showing you the progress
of the loading

        Raises:
        =======
        IOError: if no sample loaded
        IOError: if no OB loaded
        IOError: if size of sample and OB do not match

        '''
        if not force:
            # does nothing if normalization has already been run
            if self.__exec_process_status['normalization']:
                return
        self.__exec_process_status['normalization'] = True

        # make sure we loaded some sample data
        if self.data['sample']['data'] == []:
            raise IOError("No normalization available as no data have been
loaded")

        # make sure we loaded some ob data
        if self.data['ob']['data'] == []:
            raise IOError("No normalization available as no OB have been loaded")

        # make sure the data loaded have the same size
        if not self.data_loaded_have_matching_shape():
```

```
            raise ValueError("Data loaded do not have the same shape!")

        # make sure, if provided, roi has the right type and fits into the images
        b_list_roi = False
        if roi:
            if type(roi) is list:
                for _roi in roi:
                    if not type(_roi) == ROI:
                        raise ValueError("roi must be a ROI object!")
                    if not self.__roi_fit_into_sample(roi=_roi):
                        raise ValueError("roi does not fit into sample image!")
                b_list_roi = True

            elif not type(roi) == ROI:
                raise ValueError("roi must be a ROI object!")
            else:
                if not self.__roi_fit_into_sample(roi=roi):
                    raise ValueError("roi does not fit into sample image!")

        if notebook:
            from ipywidgets.widgets import interact
            from ipywidgets import widgets
            from IPython.core.display import display, HTML

        # heat normalization algorithm
        _sample_corrected_normalized = []
        _ob_corrected_normalized = []

        if roi:

            if b_list_roi:

                _sample_corrected_normalized = []
                for _sample in self.data['sample']['data']:
                    sample_mean = []
                    for _roi in roi:
                        _x0 = _roi.x0
                        _y0 = _roi.y0
                        _x1 = _roi.x1
                        _y1 = _roi.y1
                        sample_mean.append(np.mean(_sample[_y0:_y1 + 1, _x0:_x1
+ 1]))

                    # print(sample_mean)
                    full_sample_mean = np.mean(sample_mean)

_sample_corrected_normalized.append(_sample/full_sample_mean)

                _ob_corrected_normalized = []
                for _ob in self.data['ob']['data']:
                    ob_mean = []
                    for _roi in roi:
                        _x0 = _roi.x0
                        _y0 = _roi.y0
                        _x1 = _roi.x1
                        _y1 = _roi.y1
                        ob_mean.append(np.mean(_ob[_y0:_y1 + 1, _x0:_x1 + 1]))

                    full_ob_mean = np.mean(ob_mean)
                    _ob_corrected_normalized.append(_ob / full_sample_mean)
```

```
            else:
                _x0 = roi.x0
                _y0 = roi.y0
                _x1 = roi.x1
                _y1 = roi.y1

                _sample_corrected_normalized = [_sample / np.mean(_sample[_y0:_y1+1, _x0:_x1+1]) for _sample in self.data['sample']['data']]
                _ob_corrected_normalized = [_ob / np.mean(_ob[_y0:_y1+1, _x0:_x1+1]) for _ob in self.data['ob']['data']]

        else:
            _sample_corrected_normalized = copy.copy(self.data['sample']['data'])
            _ob_corrected_normalized = copy.copy(self.data['ob']['data'])

        self.data['sample']['data'] = _sample_corrected_normalized
        self.data['ob']['data'] = _ob_corrected_normalized

        # if the number of sample and ob do not match, use mean of obs
        nbr_sample = len(self.data['sample']['file_name'])
        nbr_ob = len(self.data['ob']['file_name'])
        if (nbr_sample != nbr_ob) or force_mean_ob: # work with mean ob
            _ob_corrected_normalized = np.mean(_ob_corrected_normalized, axis=0)
            self.data['ob']['data_mean'] = _ob_corrected_normalized
            _working_ob = _ob_corrected_normalized.copy()
            _working_ob[_working_ob == 0] = np.NaN

            if notebook:
                # turn on progress bar
                _message = "Normalization"
                box1 = widgets.HBox([widgets.Label(_message,
                                                   layout=widgets.Layout(width='20%')),
                                     widgets.IntProgress(max=len(self.data['sample']['data']))])
                display(box1)
                w1 = box1.children[1]

            normalized_data = []
            for _index, _sample in enumerate(self.data['sample']['data']):
                _norm = np.divide(_sample, _working_ob)
                _norm[np.isnan(_norm)] = 0
                _norm[np.isinf(_norm)] = 0
                normalized_data.append(_norm)

                if notebook:
                    w1.value = _index+1

        else: # 1 ob for each sample
            # produce normalized data
            sample_ob = zip(self.data['sample']['data'], self.data['ob']['data'])

            if notebook:
                # turn on progress bar
                _message = "Normalization"
                box1 = widgets.HBox([widgets.Label(_message,
```

```
layout=widgets.Layout(width='20%')),

widgets.IntProgress(max=len(self.data['sample']['data']))])
                display(box1)
                w1 = box1.children[1]

            normalized_data = []
            for _index, [_sample, _ob] in enumerate(sample_ob):
                _working_ob = _ob.copy()
                _working_ob[_working_ob == 0] = np.NaN
                _norm = np.divide(_sample, _working_ob)
                _norm[np.isnan(_norm)] = 0
                _norm[np.isinf(_norm)] = 0
                normalized_data.append(_norm)
                # print(_index, center_of_mass(_norm))
                if notebook:
                    w1.value = _index+1

        self.data['normalized'] = normalized_data

        return True

    def data_loaded_have_matching_shape(self):
        '''check that data loaded have the same shape

        Returns:
        =======
        bool: result of the check
        '''
        _shape_sample = self.data['sample']['shape']
        _shape_ob = self.data['ob']['shape']

        if not (_shape_sample == _shape_ob):
            return False

        _shape_df = self.data['df']['shape']
        if not np.isnan(_shape_df['height']):
            if not (_shape_sample == _shape_df):
                return False

        return True

    def __roi_fit_into_sample(self, roi=[]):
        '''check if roi is within the dimension of the image

        Returns:
        ========
        bool: True if roi is within the image dimension

        '''
        [sample_height, sample_width] = np.shape(self.data['sample']['data'][0])

        [_x0, _y0, _x1, _y1] = [roi.x0, roi.y0, roi.x1, roi.y1]
        if (_x0 < 0) or (_x1 >= sample_width):
            return False

        if (_y0 < 0) or (_y1 >= sample_height):
            return False

        return True
```

```
    def df_correction(self, force=False):
        '''dark field correction of sample and ob

        Parameters
        ==========
        force: boolean (default False) that if True will force the df correction
to occur, even if it had been
        run before with the same data set

        sample_df_corrected = sample - DF
        ob_df_corrected = OB - DF

        '''
        if not force:
            if self.__exec_process_status['df_correction']:
                return
        self.__exec_process_status['df_correction'] = True

        if not self.data['sample']['data'] == []:
            self.__df_correction(data_type='sample')

        if not self.data['ob']['data'] == []:
            self.__df_correction(data_type='ob')

    def __df_correction(self, data_type='sample'):
        '''dark field correction

        Parameters:
           data_type: string ['sample','ob]
        '''
        if not data_type in ['sample', 'ob']:
            raise KeyError("Wrong data type passed. Must be either 'sample' or
'ob'!")

        if self.data['df']['data'] == []:
            return

        if self.data['df']['data_average'] == []:
            _df = self.data['df']['data']
            if len(_df) > 1:
                _df = average_df(df=_df)
            self.data['df']['data_average'] = np.squeeze(_df)

        else:
            _df = np.squeeze(self.data['df']['data_average'])

        if             np.shape(self.data[data_type]['data'][0])             !=
np.shape(self.data['df']['data'][0]):
            raise   IOError("{}   and   df   data   must   have   the   same
shape!".format(data_type))

        _data_df_corrected    =    [_data    -    _df    for    _data    in
self.data[data_type]['data']]
        _data_df_corrected = [np.squeeze(_data) for _data in _data_df_corrected]
        self.data[data_type]['data'] = _data_df_corrected

    def crop(self, roi=None, force=False):
        ''' Cropping the sample and ob normalized data

        Parameters:
```

```
        ===========
        roi: ROI object that defines the region to crop
        force: Boolean (default False) that force or not the algorithm to be run
more than once
        with the same data set

        Raises:
        =======
        ValueError if sample and ob data have not been normalized yet
        '''
        if (self.data['sample']['data'] == []) or \
           (self.data['ob']['data'] == []):
            raise IOError("We need sample and ob Data !")

        if not type(roi) == ROI:
            raise ValueError("roi must be of type ROI")

        if not force:
            if self.__exec_process_status['crop']:
                return
        self.__exec_process_status['crop'] = True

        _x0 = roi.x0
        _y0 = roi.y0
        _x1 = roi.x1
        _y1 = roi.y1

        new_sample = [_data[_y0:_y1+1, _x0:_x1+1] for
                      _data in self.data['sample']['data']]
        self.data['sample']['data'] = new_sample

        new_ob = [_data[_y0:_y1+1, _x0:_x1+1] for
                  _data in self.data['ob']['data']]
        self.data['ob']['data'] = new_ob

        if not (self.data['df']['data'] == []):
            new_df = [_data[_y0:_y1+1, _x0:_x1+1] for
                      _data in self.data['df']['data']]
            self.data['df']['data'] = new_df

        if not (self.data['normalized'] == []):
            new_normalized = [_data[_y0:_y1+1, _x0:_x1+1] for
                              _data in self.data['normalized']]
            self.data['normalized'] = new_normalized

        return True

    def export(self, folder: object = './', data_type: object = 'normalized',
file_type: object = 'tif') -> object:
        '''export all the data from the type specified into a folder

        Parameters:
        ===========
        folder: String (default is './') where to create all the images. Folder
must exist otherwise an error is raised
        data_type: String (default is 'normalized'). Must be one of the following
'sample','ob','df','normalized'
        file_type: String (default is 'tif') format in which to export the data.
Must be either 'tif' or 'fits'

        Raises:
```

```
        =======
        IOError if the folder does not exist
        KeyError   if   data_type   can   not   be   found   in   the   list
['normalized','sample','ob','df']

        '''
        if not os.path.exists(folder):
            raise IOError("Folder '{}' does not exist!".format(folder))

        if not data_type in ['normalized','sample','ob','df']:
            raise KeyError("data_type '{}' is wrong".format(data_type))

        prefix = ''
        if data_type == 'normalized':
            data = self.get_normalized_data()
            prefix = 'normalized'
            data_type = 'sample'
        else:
            data = self.data[data_type]['data']

        if data ==[]:
            return False

        metadata = self.data[data_type]['metadata']

        list_file_name_raw = self.data[data_type]['file_name']
        self.__create_list_file_names(initial_list=list_file_name_raw,
                                      output_folder = folder,
                                      prefix=prefix,
                                      suffix=file_type)

        self.__export_data(data=data,
                           metadata=metadata,
                           output_file_names = self._export_file_name,
                           suffix=file_type)


    def    __export_data(self,    data=[],    metadata=[],    output_file_names=[],
suffix='tif'):
        '''save the list of files with the data specified

        Parameters:
        ===========
        data: numpy array that contains the array of data to save
        output_file_names: numpy array of string of full file names
        suffix: String (default is 'tif') format in which the file will be created
        '''
        name_data_metadata_array = zip(output_file_names, data, metadata)
        for _file_name, _data, _metadata in name_data_metadata_array:
            if suffix == 'tif':
                make_tif(data=_data, metadata=_metadata, file_name=_file_name)
            elif suffix == 'fits':
                make_fits(data=_data, file_name=_file_name)

    def    __create_list_file_names(self,    initial_list=[],    output_folder='',
prefix='', suffix=''):
        '''create a list of the new file name used to export the images

        Parameters:
        ==========
        initial_list: array of full file name
```

```
            ex:
['/users/me/image001.tif',/users/me/image002.tif',/users/me/image003.tif']
        output_folder: String (default is ./ as specified by calling function)
where we want to create the data
        prefix: String. what to add to the output file name in front of base name
            ex: 'normalized' will produce 'normalized_image001.tif'
        suffix: String. extenstion to file. 'tif' for TIFF and 'fits' for FITS
        '''
        _base_name = [os.path.basename(_file) for _file in initial_list]
        _raw_name = [os.path.splitext(_file)[0] for _file in _base_name]
        _prefix = ''
        if prefix:
            _prefix = prefix + '_'
        full_file_names = [os.path.join(output_folder, _prefix + _file + '.' +
suffix) for _file in _raw_name]
        self._export_file_name = full_file_names

    def get_normalized_data(self):
        '''return the normalized data'''
        return self.data['normalized']

    def get_sample_data(self):
        '''return the sample data'''
        return self.data['sample']['data']

    def get_ob_data(self):
        '''return the ob data'''
        return self.data['ob']['data']

    def get_df_data(self):
        '''return the df data'''
        return self.data['df']['data']

    def get_emp_data(self):
        '''return the emp data'''
        return self.data['emp']['data']

    def  pulse_normalization(self,  roi=None,  force=False,  force_mean_ob=False,
notebook=False):
        '''normalization of the data

        Parameters:
        ===========
        roi: ROI object or list of ROI objects that defines the region of the
sample and OB that have to match
        in intensity
        force: boolean (default False) that if True will force the normalization
to occur, even if it had been
        run before with the same data set
        notebook: boolean (default False) turn on this option if you run the
library from a
             notebook to have a progress bar displayed showing you the progress
of the loading

        Raises:
        =======
        IOError: if no sample loaded
        IOError: if no OB loaded
        IOError: if size of sample and OB do not match

        '''
```

```
        if not force:
            # does nothing if normalization has already been run
            if self.__exec_process_status['normalization']:
                return
        self.__exec_process_status['normalization'] = True

        # make sure we loaded some sample data
        if self.data['sample']['data'] == []:
            raise IOError("No normalization available as no data have been
loaded")

        # make sure we loaded some ob data
        if self.data['ob']['data'] == []:
            raise IOError("No normalization available as no OB have been loaded")

        # make sure we loaded some ob data
        if self.data['emp']['data'] == []:
            raise IOError("No normalization available as no empty beam have been
loaded")

        # make sure the data loaded have the same size
        if not self.data_loaded_have_matching_shape():
            raise ValueError("Data loaded do not have the same shape!")

        # make sure, if provided, roi has the right type and fits into the images
        b_list_roi = False
        if roi:
            if type(roi) is list:
                for _roi in roi:
                    if not type(_roi) == ROI:
                        raise ValueError("roi must be a ROI object!")
                    if not self.__roi_fit_into_sample(roi=_roi):
                        raise ValueError("roi does not fit into sample image!")
                b_list_roi = True

            elif not type(roi) == ROI:
                raise ValueError("roi must be a ROI object!")
            else:
                if not self.__roi_fit_into_sample(roi=roi):
                    raise ValueError("roi does not fit into sample image!")

        if notebook:
            from ipywidgets.widgets import interact
            from ipywidgets import widgets
            from IPython.core.display import display, HTML

            # heat normalization algorithm
        _sample_corrected_normalized = []
        _ob_corrected_normalized = []

        if roi:
            print(b_list_roi)
            if b_list_roi:

                _sample_corrected_normalized = []
                for _sample in self.data['sample']['data']:
                    sample_mean = []
                    for _roi in roi:
                        _x0 = _roi.x0
                        _y0 = _roi.y0
                        _x1 = _roi.x1
```

```
                        _y1 = _roi.y1
                        sample_mean.append(np.mean(_sample[_y0:_y1 + 1, _x0:_x1
+ 1]))

                    full_sample_mean = np.mean(sample_mean)
                    _sample_corrected_normalized.append(_sample                /
full_sample_mean)

                _ob_corrected_normalized = []
                for _ob in self.data['ob']['data']:
                    ob_mean = []
                    for _roi in roi:
                        _x0 = _roi.x0
                        _y0 = _roi.y0
                        _x1 = _roi.x1
                        _y1 = _roi.y1
                        ob_mean.append(np.mean(_ob[_y0:_y1 + 1, _x0:_x1 + 1]))

                    full_ob_mean = np.mean(ob_mean)
                    _ob_corrected_normalized.append(_ob / full_sample_mean)

                _emp_corrected_normalized = []
                for _emp in self.data['emp']['data']:
                    emp_mean = []
                    for _roi in roi:
                        _x0 = _roi.x0
                        _y0 = _roi.yo
                        _x1 = _roi.x1
                        _y1 = _roi.y1
                        emp_mean.append(np.mean(_emp[_y0:_y1 + 1, _x0:_x1 + 1]))

                    full_emp_mean = np.mean(emp_mean)
                    _emp_corrected_normalized.append(_emp / full_sample_mean)
            else:
                _x0 = roi.x0
                _y0 = roi.y0
                _x1 = roi.x1
                _y1 = roi.y1

                '''_sample_corrected_normalized       =       [_sample       /
np.mean(_sample[_y0:_y1 + 1, _x0:_x1 + 1])
                                                  for          _sample          in
self.data['sample']['data']]
                _ob_corrected_normalized  =  [_ob  /  np.mean(_ob[_y0:_y1  +  1,
_x0:_x1 + 1])
                                                for _ob in self.data['ob']['data']]
                _emp_corrected_normalized =  [_emp  /  np.mean(_emp[_y0:_y1  +  1,
_x0:_x1 + 1])
                                                 for             _emp             in
self.data['emp']['data']]'''

        else:
            _sample_corrected_normalized                                          =
copy.copy(self.data['sample']['data'])
            _ob_corrected_normalized = copy.copy(self.data['ob']['data'])
            _emp_corrected_normalized = copy.copy(self.data['emp']['data'])

        '''self.data['sample']['data'] = _sample_corrected_normalized
        self.data['ob']['data'] = _ob_corrected_normalized
        self.data['emp']['data'] = _emp_corrected_normalized'''
```

```
        # if the number of sample and ob do not match, use mean of obs
        nbr_sample = len(self.data['sample']['file_name'])
        nbr_ob = len(self.data['ob']['file_name'])
        nbr_emp = len(self.data['emp']['file_name'])

        if (nbr_sample != nbr_ob) or force_mean_ob or (nbr_sample != nbr_emp):  #
work with mean ob
            _ob_corrected_normalized = np.mean(_ob_corrected_normalized, axis=0)
            self.data['ob']['data_mean'] = _ob_corrected_normalized
            _working_ob = _ob_corrected_normalized.copy()
            _working_ob[_working_ob == 0] = np.NaN

            _emp_corrected_normalized    =    np.mean(_emp_corrected_normalized,
axis=0)
            self.data['emp']['data_mean'] = _emp_corrected_normalized
            _working_emp = _emp_corrected_normalized.copy()
            _working_emp[_working_emp == 0] = np.NaN

            if notebook:
                # turn on progress bar
                _message = "Normalization"
                box1 = widgets.HBox([widgets.Label(_message,

layout=widgets.Layout(width='20%')),

widgets.IntProgress(max=len(self.data['sample']['data']))])
                display(box1)
                w1 = box1.children[1]

            normalized_data = []
            for _index, _sample in enumerate(self.data['sample']['data']):
                _norm_ob = np.divide(-np.log(np.divide(_sample, _working_ob)),
                                     -np.log(np.divide(_working_emp,
_working_ob)))
                _norm_ob[np.isnan(_norm_ob)] = 0
                _norm_ob[np.isinf(_norm_ob)] = 0
                normalized_data.append(_norm_ob)

                if notebook:
                    w1.value = _index + 1

        else:  # 1 ob for each sample
            # produce normalized data
            sample_ob_emp           =           zip(self.data['sample']['data'],
self.data['ob']['data'], self.data['emp']['data'])

            if notebook:
                # turn on progress bar
                _message = "Normalization"
                box1 = widgets.HBox([widgets.Label(_message,

layout=widgets.Layout(width='20%')),

widgets.IntProgress(max=len(self.data['sample']['data']))])
                display(box1)
                w1 = box1.children[1]

            normalized_data = []
            for _index, [_sample, _ob, _emp] in enumerate(sample_ob_emp):
                _working_ob = _ob.astype('float')
                # print(type(_working_ob))
```

```
                _working_emp = _emp.astype('float')

                _working_sample = _sample.astype('float')


                _sub_ob=_working_ob[_x0:_x1, _y0:_y1]
                _sub_sample=_working_sample[_x0:_x1, _y0:_y1]
                # print(np.shape(_sub_ob))
                _Dw=np.average(_sub_ob)
                _Di=np.average(_sub_sample)

                _working_ob[_working_ob < 0] = 0
                _working_emp[_working_emp < 0] = 0
                _working_sample[_working_sample < 0] = 0
                #print('image {}  flat {}', [_Di, _Dw])

                _num=(np.divide(_working_sample, _working_ob))
                _num2=np.divide(_Dw, _Di)
                # print(_num2)
                _denom=(np.divide(_working_emp, _working_ob))
                _norm_ob      =      np.divide(np.log(np.multiply(_num,      1)),
np.log(_denom))
                error=(1/np.log(np.divide(_working_emp,
_working_ob)))*np.sqrt((1/(_working_sample*.00025))+(((_norm_ob-1)*(_norm_ob-
1))/(_working_ob*.00025))+(_norm_ob*_norm_ob)/(_working_emp*.00025))
                print(error)
                ''' x = np.shape(_working_ob)[0]
                y = np.shape(_working_ob)[1]
                # print(type(x))
                i = 0
                j = 0
                _norm_ob = np.zeros([x, y])
                while i < x:
                    while j < y:
                        num = float(_sample[i, j]) / float(_working_ob[i,j])
                        denom = float(_working_emp[i, j]) / float(_working_ob[i,
j])
                        print(num, denom)
                        if num <= 0:
                            num = np.NaN
                        _norm_ob[i,  j]  =  -(np.log10(num)/np.log10(np.e))  /  -
(np.log10(denom)/np.log10(np.e))
                        j += 1
                    j = 0
                    i += 1'''

                _norm_ob[np.isnan(_norm_ob)] = 0
                _norm_ob[np.isinf(_norm_ob)] = 0
                normalized_data.append(_norm_ob)

                if notebook:
                    w1.value = _index + 1

        self.data['normalized'] = normalized_data

        return True
```

Appendix C

Python Code for Calculating SNR and CNR

```
import os
import sys
import numpy as np
import glob
import scipy.ndimage
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from matplotlib import gridspec
#matplotlib notebook
import math
root_folder = os.path.dirname(os.getcwd())
sys.path.append(root_folder)
import NeuNorm as neunorm
from NeuNorm.normalization import Normalization
from NeuNorm.roi import ROI


'''imports each of the data sets'''
path_im                                                                    =
glob.glob('C:/Users/Chad/Desktop/Cleaned_Pulses/H60fps/60fps_bubbles/*.tif')

'''Determines the maximum index for each of the data sets'''
o_norm = Normalization()

o_norm.load(file=path_im,          notebook=False,          gamma_filter=False,
gamma_threshold=.5, dead_filter=False,
            dead_threshold=.5, gamma_rad=2, dead_rad=2)


sample = o_norm.get_sample_data()
sample = np.array(sample, float)
print(np.shape(sample))
avg_samp = np.empty(len(sample))
std_dev_samp = np.empty(len(sample))
SNR = np.empty((len(sample)))

ROI_SNR = [128, 217, 343, 376]
SNR_box  =  np.zeros([np.shape(sample)[0],  (ROI_SNR[3]-ROI_SNR[1]),  (ROI_SNR[2]-
ROI_SNR[0])])
i = 0
j = 0
k = 0
while i < np.shape(sample)[0]:
    while j < (ROI_SNR[3] - ROI_SNR[1]):
        while k < (ROI_SNR[2] - ROI_SNR[0]):
            SNR_box[i, j, k] = sample[i, ROI_SNR[1]+j, ROI_SNR[0]+k]
            k = k + 1
        k = 0
        j = j + 1
    j = 0
    i = i + 1

i = 0
while i < len(sample):
```

```
    avg_samp[i]                                                                    =
np.sum(SNR_box[i])/(np.shape(SNR_box[i])[0]*np.shape(SNR_box[i])[1])
    std_dev_samp[i] = np.std(SNR_box[i])
    SNR[i] = avg_samp[i] / std_dev_samp[i]
    i = i+1

#print(avg_samp[52],std_dev_samp[52])
ROI_light = [466, 343, 486, 362]
ROI_dark = [51, 243, 129, 310]
dark_box     =     np.zeros([np.shape(sample)[0],     (ROI_dark[3]-ROI_dark[1]),
(ROI_dark[2]-ROI_dark[0])])
light_box    =    np.zeros([np.shape(sample)[0],    (ROI_light[3]-ROI_light[1]),
(ROI_light[2]-ROI_light[0])])
i = 0
j = 0
k = 0
while i < np.shape(sample)[0]:
    while j < (ROI_light[3] - ROI_light[1]):
        while k < (ROI_light[2] - ROI_light[0]):
            light_box[i, j, k] = sample[i, ROI_light[1]+j, ROI_light[0]+k]
            k = k + 1
        k = 0
        j = j + 1
    j = 0
    i = i + 1

i = 0
j = 0
k = 0
while i < np.shape(sample)[0]:
    while j < (ROI_dark[3] - ROI_dark[1]):
        while k < (ROI_dark[2] - ROI_dark[0]):
            dark_box[i, j, k] = sample[i, ROI_dark[1]+j, ROI_dark[0]+k]
            k = k + 1
        k = 0
        j = j + 1
    j = 0
    i = i + 1

avg_light = np.empty(len(sample))
avg_dark = np.empty(len(sample))
std_dev_light = np.empty(len(sample))
std_dev_dark = np.empty(len(sample))
CNR = np.empty(len(sample))
i = 0
while i < len(sample):
    avg_light[i] = np.average(light_box[i])
    avg_dark[i] = np.average(dark_box[i])
    std_dev_light[i] = np.std(light_box[i])
    std_dev_dark[i] = np.std(dark_box[i])
    CNR[i] = (avg_light[i] - avg_dark[i]) / math.sqrt(math.pow(std_dev_light[i],
2) + math.pow(std_dev_dark[i], 2))
    i = i+1

print(np.shape(light_box))

plt.plot(SNR)
plt.ylabel(('SNR'))
plt.xlabel('Image Number')
plt.show()
```

```
plt.plot(CNR)
plt.ylabel(('CNR'))
plt.xlabel('Image Number')
plt.show()
```

Appendix D

Supplementary Images at 2,000 fps

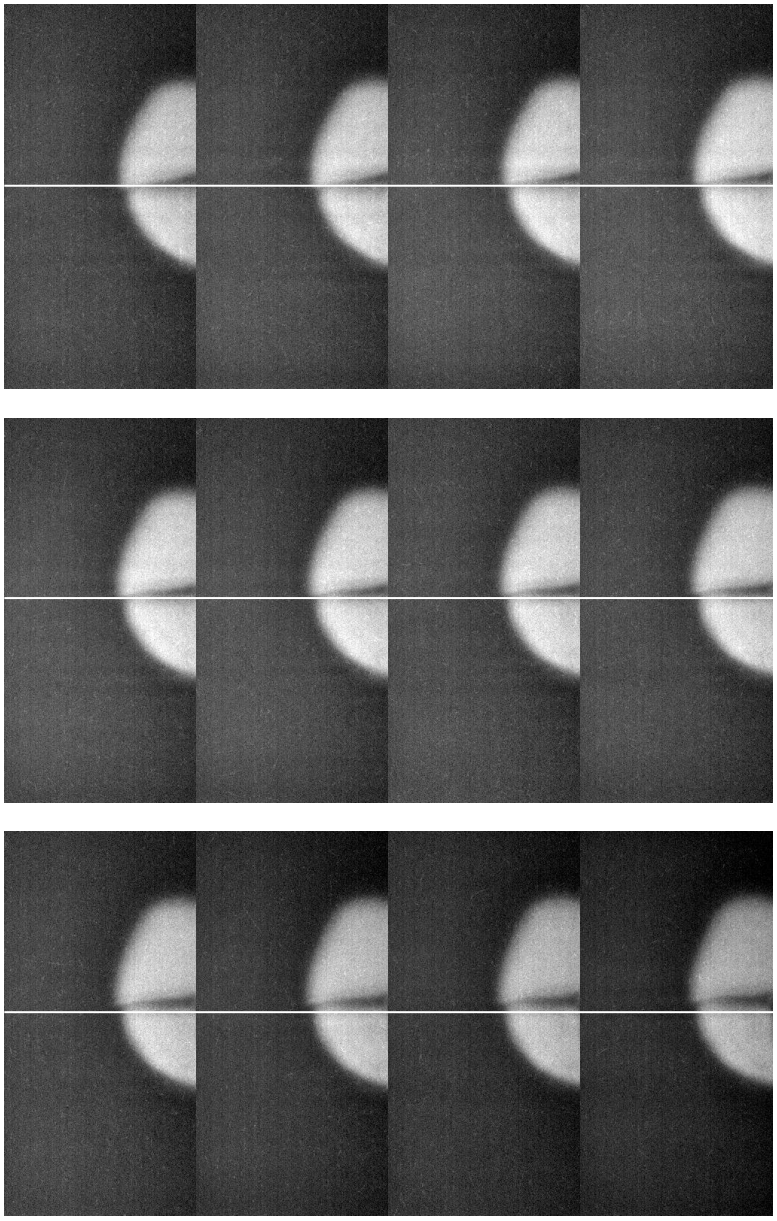

Figure **D-1**: Raw two-phase flow images taken at 2,000 fps, the white line is for movement reference. Note that the image gray scale is arbitrary.

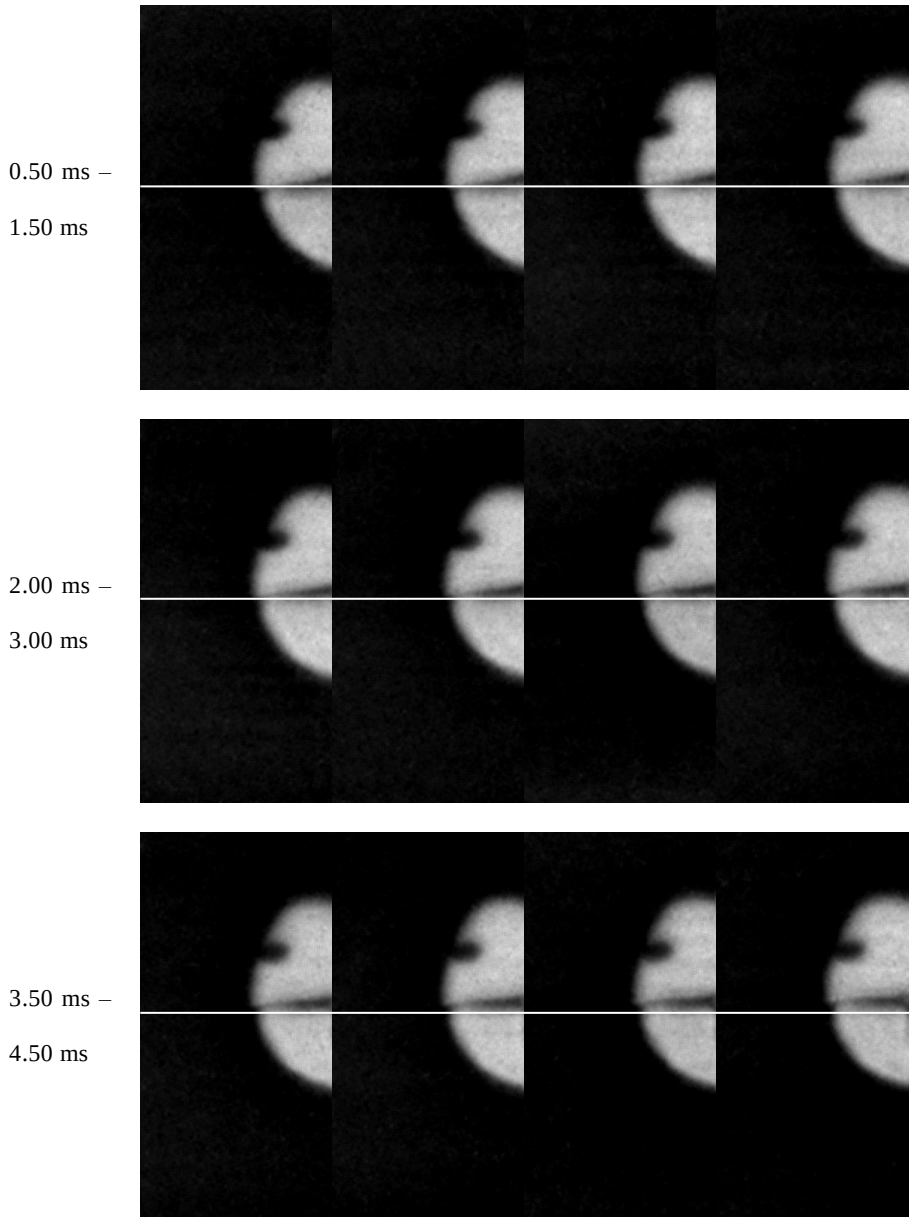

Figure **D-2**: Normalized two-phase flow images taken at 2,000 fps, the white line is for movement reference.

Appendix E

Supplementary Images at 4,000 fps

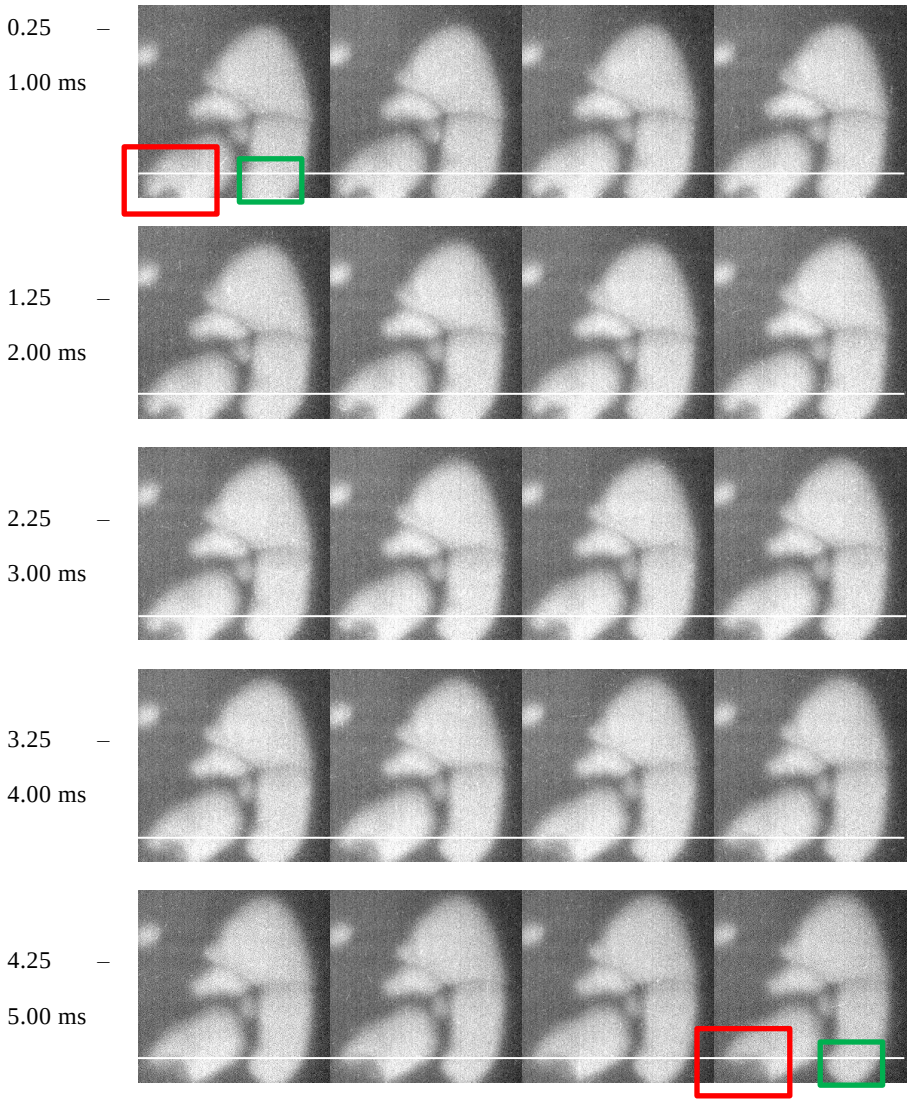

Figure **E-1**: Twenty raw frames of two-phase flow taken at 4,000 fps, the white line is for easy reference of movement just as the regions highlighted by the red and green rectangles in the first and last images of the sequence shown. Note that the FOV is only half of that in Figure D-1, due to partial readout.

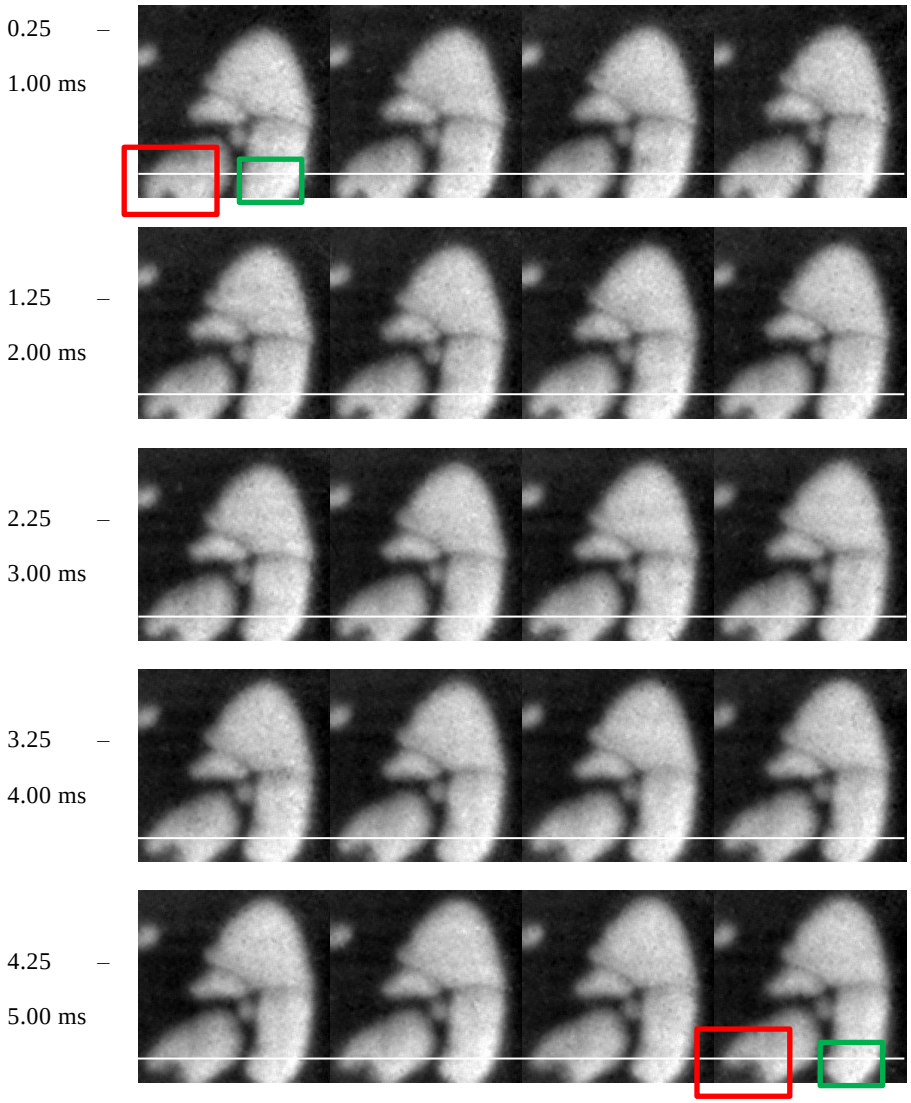

Figure **E-2**: Twenty normalized frames of two-phase flow taken at 4,000 fps, the white line is for easy reference of movement just as the regions highlighted by the red and green rectangles in the first and last images of the sequence shown.