DESIGN OF A ROBOTIC ARM MANIPULATOR CAMERA UNIT FOR MINI UNDERWATER REMOTELY OPERATED VEHICLES

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Electrical Engineering

by

Michael John Poretti

December 2013

COMMITTEE MEMBERSHIP

TITLE:	Design of a Robotic Arm Manipulator Camera Unit for Mini Underwater Remotely Operated Vehicles
AUTHOR:	Michael Poretti
DATE SUBMITTED:	December 2013
COMMITTEE CHAIR:	Bridget Benson, PhD Assistant Professor of Electrical Engineering
COMMITTEE MEMBER:	John Oliver, PhD Associate Professor, Director of CPE Program
COMMITTEE MEMBER:	Lynne Slivovsky, PhD Professor of Electrical Engineering

ABSTRACT

Design of a Robotic Arm Manipulator Camera Unit for Mini Underwater Remotely Operated Vehicles

Michael John Poretti

Underwater remotely operated vehicles are essential tools for marine researchers and workers. Their robust nature allows them to serve a wide range of purposes. For example, they can be used for remote visual inspection of pipelines and can manipulate tools such as screwdrivers and claws. Cameras are the main method for providing operator feedback to the surface as they enable an operator to accurately maneuver or handle objects from thousands of feet away. Although large ROVs have cameras attached to their robotic arms for closer inspection of objects, mini ROVs do not because no camera tool has been specifically designed to support the low-cost, lightweight design of a mini ROV. This thesis discusses the design considerations, component selection, and system prototype (including the use of image processing to improve the underwater image for the operator's viewing) of an affordable camera unit for mini ROV robotic manipulators.

ACKNOWLEDGMENTS

I wish to extend a special thanks to Chris Rauch at Rauch Engineering LLC for his expertise and support in design, construction, and financial aspects of this project.

I would also like to thank Seabotix for their technical support early in the project, and Tom Molyan at the Cal Poly Pier for getting access to testing resources.

TABLE OF CONTENTS

PAGE

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 – INTRODUCTION

Industries around the world employ underwater remotely operated vehicles (ROVs) for a wide range of research, inspection, and work. Such activities include investigating sea life, collecting data, and inspecting structures. ROVs offer different advantages depending on their size; they can be used as underwater workhorses or utilized for their quick and easy deployment [1]. Smaller ROVs are commonly employed for their relative cheapness and ease of operation compared to larger options [1-2].

Cameras are essential for remote operation of these vehicles since they act as eyes to the surface. They serve two main purposes: remotely driving the ROV and visual inspection of underwater objects. Almost all ROVs come with a camera fixed to the main body for remote operation. Often times the camera will be on a pan/tilt mechanism. Some of the larger ROVs have maneuverable cameras attached to robotic arms, as seen in Figure 1. These cameras are used for close inspection because turbulent underwater conditions make macro shots with the main camera near impossible [1]. For a size reference, an ROV that can use the arm shown in Figure 1 would be the Schilling Robotics UHD ROV, which weighs over 10,000 lbs.

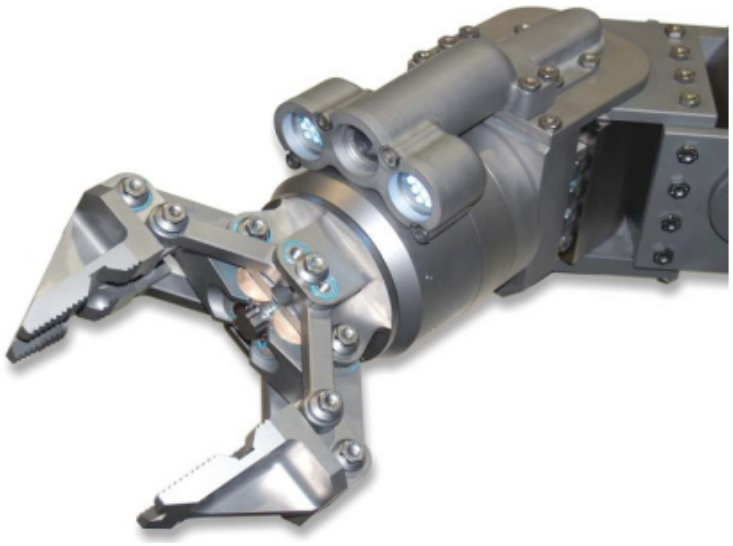


Figure 1: Schilling TITAN 4 manipulator with wrist camera option

A mini ROV like the SeaBotix LBV300-5 shown in Figure 2, weighing in at only 29 pounds, cannot use a several hundred pound arm like the Schilling TITAN 4. Current mini ROV robotic manipulators are lacking in functionality, typically only having clasping abilities [3]. The motivation for this project stems from an idea to mimic work-class manipulator abilities on a mini ROV. The overall design will include a robotic arm with multiple degrees of freedom. The end attachment of the arm can be changed on the fly with a tool kit attached to the ROV. The tools may include a screwdriver, claw, scrubber, etc. One of these tools will be a camera unit for close inspection tasks.

Figure 2: SeaBotix LBV300-5 mini ROV with simple robotic manipulator

This paper describes the design of an affordable underwater camera unit for mini ROV robotic manipulators. The added functionality of a maneuverable camera to the mini class will make them much more versatile inspection tools, and therefore a viable option for a broader range of missions. This paper details the design considerations, component selection, and system verification of the modular camera tool. The minimum requirements for the prototype camera tool are a minimum light-sensitivty of 1 lux, 100ft depth rating and at least standard resolution for under $200. In addition, the video output of the camera is enhanced using a standard computer by correcting optical distortion and filtering Gaussian noise. An existing Gaussian noise removal algorithm is improved for better performance and speed.

CHAPTER 2 – BACKGROUND INFORMATION

2.1 INTRODUCTION

The following chapter provides detailed background information essential for understanding concepts addressed in this paper, including ROV classification, the history of ROVs, basic optics and light definitions, camera lenses, camera technology, image processing related to optical correction, and noise removal.

2.2 ROV CLASSIFICATION

Underwater vehicles can be broken up into several categories, shown in Figure 3 [7]. ROVs are separated from autonomous underwater vehicles (AUV) because they require constant operator attention and use a tether or cable for power, video, and controls. The two main subcategories within the ROV section are the work and observation class ROVs. The project motivation is to bridge the gap between these classes to allow mini ROVs to have more versatility.

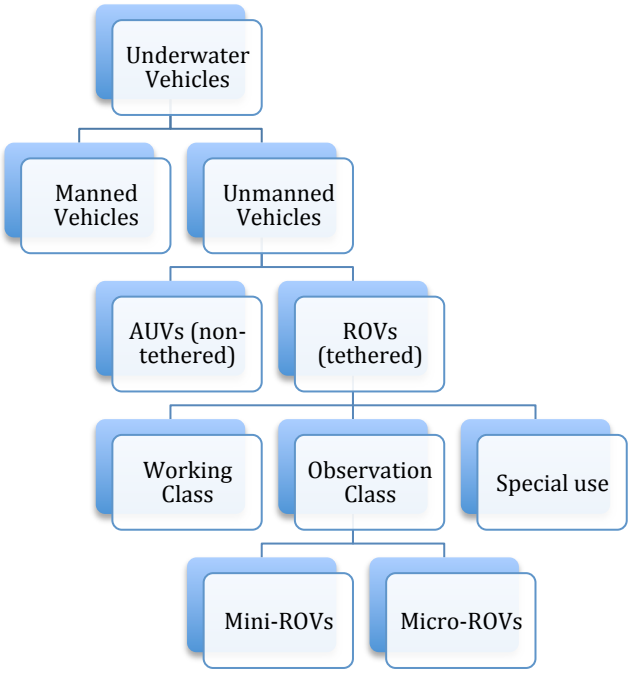


Figure 3: Underwater vehicle taxonomy

The designation of “mini” or “micro” is at the discretion of the manufacturer since there is no official definition. Even the “mini” ROV class comes in several variations. Typically the “micro” designation is reserved for ROVs under 20 lbs. Some mini ROV class variations from leading ROV manufacturer SeaBotix can be seen in Figure 4. Notice how each mini ROV has a differing number of thrusters and different attachments. The wheels and tracks on two of the models demonstrate the versatility of mini ROVs. For ship hull inspection, an ROV can flip itself over and use its thrusters to essentially suction itself to the hull so it can drive around. This enables the ROV to remain steady relative to the ship making inspection much easier.

Figure 4: Top Left – SeaBotix Little Benthic Crawler. Top Right – vLBV-10. Bottom Left – SeaBotix LBV150-4. Bottom Right – SeaBotix vLBC (Source www.seabotix.com).

An image of a working class ROV is provided in Figure 5 for comparison to the mini class ROVs in Figure 4. Note the working class is much larger and heavier but it is more suited for heavy payloads, and operation in open-water conditions. Significant cost increases are incurred with working class vehicles because of their size; they require a mechanism for launching and retrieving, and

may require a larger crew to operate. One person can carry a mini class ROV, which greatly reduces operation difficulties.

Figure 5: Launching of a Work Class ROV

Any standard ROV model contains electronics enclosed in a waterproof pressure housing; the electronics control the thrusters for maneuvering and send a video signal back to the surface. The communication link with the ROV is achieved with a special tether that provides DC power and control lines and relays video or other sensor feedback from the ROV. Many vehicles are also outfitted with an arm for further versatility.

2.3 ROV HISTORY

Dimitri Rebikoff is credited with designing and building the first ROV, called the POODLE in 1953 [7]. However, the technology was originally developed to serve the needs of the US Navy. The Cable-Controlled Underwater

Research Vehicle, or CURV, was developed for the US Navy by VARE Industries to retrieve lost torpedoes [7]. The project proved to be a massive success and led to the development of a work-class-style ROV, called the Pontoon Implacement Vehicle (PIV) shown in Figure 6, and the first mini class style ROV called SNOOPY shown in Figure 7. SNOOPY was eventually outfitted with sonar and other sensors, marking the beginnings of small ROVs.

Figure 6: US Navy PIV ROV (Source [7])

Figure 7: US Navy SNOOPY ROV (Source [7])

Most of the vehicles built through 1974 were government-funded, but from 1974 to 1982 there was an explosion of activity from the private sector funding almost all of the newly-built ROVs [7]. The private industry started finding more uses for ROV technology, driving its development to its present-day state.

Observation-class ROVs use a video camera as their main method of water-to-surface feedback. Knowledge of camera and light-related terms is necessary to understand the modular camera tool discussed in this paper. A brief overview of camera technology is provided, followed by light definitions and equations.

2.4 ROV Camera Systems

A camera system consists of a light source to illuminate a subject, lens to focus incoming light, image sensor, and electronics to manipulate the raw file into a more useable format. Figure 8 shows the process flow for a standard camera system.

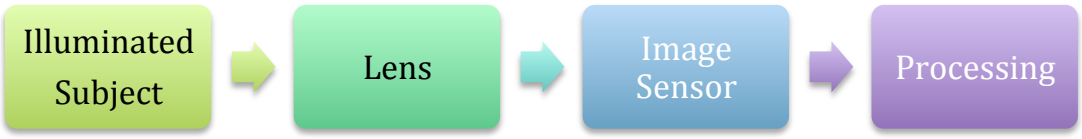


Figure 8: Camera system flow diagram

Lighting sources can be natural, such as sunlight, or artificial, such as light bulbs or LEDs. They emit light that will bounce or reflect off of the subject towards the camera lens.

Aperture and focal length are the two main characteristics of a lens that determine its ability to focus light onto an image sensor. Aperture refers to the opening in the lens through which light travels. This controls how much light can enter, therefore determining how much light exposure the image sensor will receive. Another effect of aperture is depth-of-field, or the range of distances from the camera that appear in focus. A large aperture setting will have a small depth-of-field. Within the depth-of-field, a point source of light can hit the pixel as a single point or a circle of light. As long as the point source of light is contained completely within the pixel, the image will be focused. A larger light-collecting area can achieve a brighter image, but will have a shallower depth of field. The aperture control on digital cameras is known as f-stop, calculated as the ratio between the focal length and aperture diameter.

Focal length refers to how much the lens can bend light rays to bring them into focus; a shorter focal length has a wider angle of view, as shown in Figure 9. The longer focal length cannot bend light rays with a large incidence angle enough for them to focus on the image sensor. Wide-angle lenses also have a close minimum focusing distance and may cause more emphasis on closer subjects.

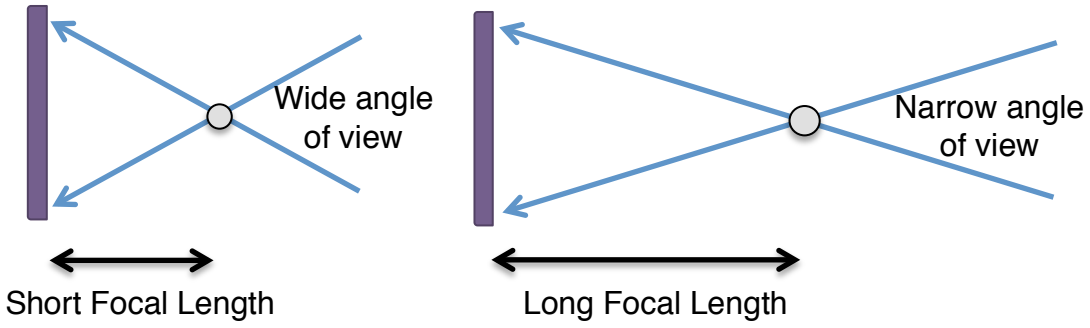


Figure 9: Left - Short focal allows for wider viewing angle, Right - Long focal length has narrow viewing angle.

The key aspects of an image sensor are physical size, number of pixels, color detection, light sensitivity, and sensor architecture. The physical size of the sensor and number of pixels are strongly correlated. A larger physical size means the pixels are larger and will be able to gather more light in one exposure, improving signal-to-noise ratio and dynamic range. The tradeoff is large pixels are much more expensive.

The number of pixels determines the finest resolution that can be achieved although this is not the only factor that determines this characteristic. Marketing hype only states the number of pixels, usually quoted in Megapixels, but this is not sufficient to determine the overall image quality. A large sensor will produce a much cleaner image compared to a small sensor with the same number of pixels. Other factors that affect the image quality will be discussed later.

A pixel by itself can only indicate the amount of light received, with no indication of color. There are two popular methods to achieve color images. One

method places a color filter array above the sensor so each pixel only captures one color; another method has three separate sensors at each pixel to capture different colors. In the latter method, the color sensors at each pixel are combined to produce a properly-colored image. This setup is typically much more expensive, so the method using color filter arrays is much more common.

The most popular color filter array is called the Bayer filter, named after Bryce Bayer who invented the filter array pattern while working at Kodak in 1974. A Bayer filter allows each pixel to capture only one color, and the colored image is produced after light collection using a demoisacing algorithm to interpolate colors at every pixel. The filter is shown in Figure 10. Notice there are twice as many green pixels as there are red or blue. Human eyes are more sensitive to light wavelengths around green, so having twice as many green pixels makes the image appear brighter [8].

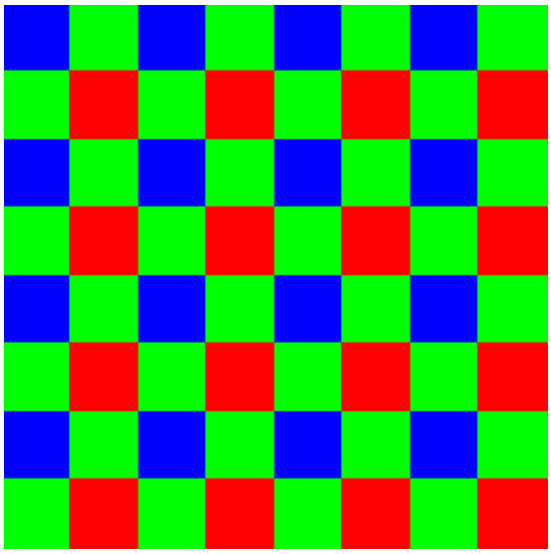


Figure 10: Bayer filter array

One of the most important camera characteristics for underwater filming is light sensitivity. The setting on a digital camera to control light exposure is specified by the International Standards Organization number, or ISO. Digital camera users may have noticed that a camera can adjust for differences in lighting between scenes, but darker scenes always appear noisier or grainier. Noise is a negative side effect from increasing ISO. Put another way, increasing the light sensitivity of the image sensor increases the artificial noise.

2.5 Camera Technology

The most commonly found cameras use either charge-coupled device (CCD) or complementary metal-oxide semiconductor (CMOS) technologies. CCD and CMOS technologies were invented in the 1960's, but CMOS cameras did not see wide public use until 20 years after CCD cameras [9]. CMOS camera technology has improved with other CMOS technologies because they share the same process control and lithography in fabrication. CMOS cameras have become more prevalent due to their small size and power requirements [9] [10]. The basic operation of both technologies is the same: convert light into electricity using the photoelectric effect. CCD pixels accomplish this operation by using MOS capacitors to capture and store charge, then moving the charge one pixel at time until it reaches an amplifier to convert the charge into voltage [9]. The process is analogous to a bucket brigade, passing a bucket of charge from pixel to pixel sequentially until it reaches the amplifier. CMOS pixels differ significantly

from CCD in that they convert the charge into a voltage immediately at the pixel [9]. The voltages are then read with column and row selectors. Figure 11 is an illustration of the charge to voltage conversion for both technologies.

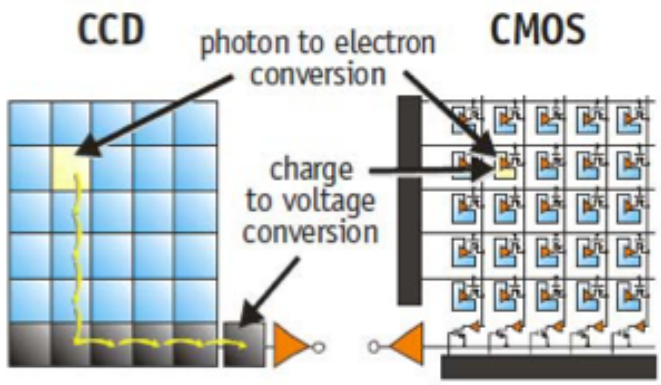


Figure 11: CCD and CMOS charge to voltage conversion (Source: [9])

Camera manufacturers have employed several light sensitivity-enhancing technologies in the past few decades. These use electronic image intensifiers for extreme low light levels. Three such technologies are Silicon Intensified Target (SIT), Intensified SIT (ISIT), and Intensified CCD (ICCD). These intensifiers can achieve a light sensitivity several orders of magnitude better than CCD or CMOS [4]. The cost, however, can also be several orders of magnitude higher and require complex supporting circuitry [4]. Image sensors capable of functioning in low light, whether or not they use an intensifier, are commonly employed in ROVs, surveillance systems (CCTV), and astronomy.

Image sensor technology is only one part of the imaging chain. As mentioned before, the number and size of pixels will limit the resolution of the

final image. The next largest contribution to image resolution is the lens used to focus the light rays. The camera optics must be able to focus a point source of light to a single pixel, otherwise the point source will appear smeared across several pixels, reducing the resolution. A commonly-used specification for resolution is television lines (TVL), which defines the maximum number of resolvable vertical lines [11]. This specification takes into account both the lens and image sensor to give a better idea of the true resolution.

There are other factors that can affect image quality besides resolution. The camera sensor adds intrinsic noise to the pixel values [10], and encoding for transmission may result in a loss of information. Video outputs from cameras are encoded according to the National Television System Committee (NTSC) for North America, and Phase Alternating Line (PAL) for Europe and Asian countries. NTSC is composed of 480 horizontal lines sent at a rate of about 30 frames per second [12]. Digital displays show the video as 640 by 480 pixels. The output is seen in households as the yellow composite video in RCA connections.

2.6 LIGHT

Light plays a critical role in any camera system, and underwater camera systems are no exception with additional difficulties from light refraction and attenuation. Two terms quantify the amount of light in a system: Luminous flux and illuminance [13]. Luminous Flux, denoted in lumens, is a measure of the total

power emitted from a light source. A related and more common term is illuminance, denoted in lux, which measures the amount of light on a surface ($1\ lux = 1\ lm/m^2$). A camera's low-light ability is usually specified using lux, or in other words, the amount of light incident on the image sensor.

Light attenuation in water occurs from interactions between photons and water molecules [13]. Attenuation is wavelength dependent. The ends of the visible spectrum are most affected, and blue-green light is least affected; this causes water's blue-green appearance [13] [14].

Most cameras capture objects in the same medium as the camera, avoiding refraction at a medium boundary. Refraction occurs when a light ray enters a different medium, causing the light ray to bend at the boundary [15]. Snell's law of refraction explains the relationship between the incident light ray and the refracted light ray as given by the equation in Figure 12 and illustration in Figure 13 [15]. A light ray entering a faster medium (lower index of refraction) will refract to a greater angle compared to the incident light ray.

$$n_i \sin(\theta_i) = n_r \sin(\theta_r)$$

Figure 12: Snell's law of refraction

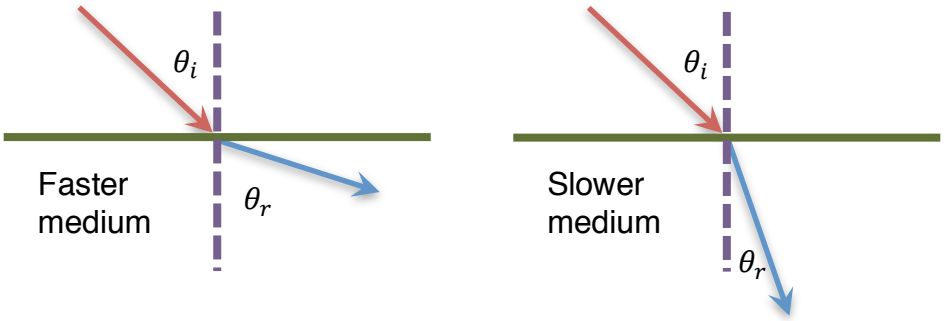


Figure 13: Refraction illustration for two different mediums

Snell's law uses the index of refraction defined in Figure 14 [15].

$$index\ of\ refraction, n = \frac{speed\ of\ light\ in\ vacuum}{speed\ of\ light\ in\ medium} = \frac{c}{v}$$

Figure 14: Index of refraction definition

The index of refraction for several common mediums is listed in Table 1.

Table 1: Index of refraction for common materials

Substance	Index of Refraction
Air	1 (1.000293)
Water	1.33
PMMA (Plexiglass)	1.492

An effect called total internal reflection can occur when a light ray is directed at a medium having a smaller index of refraction. A critical angle defines the largest incident angle before the refracted angle becomes parallel to the

boundary, illustrated in Figure 15 and defined in Figure 16. Notice the largest-angle incident light ray is completely reflected by the air.

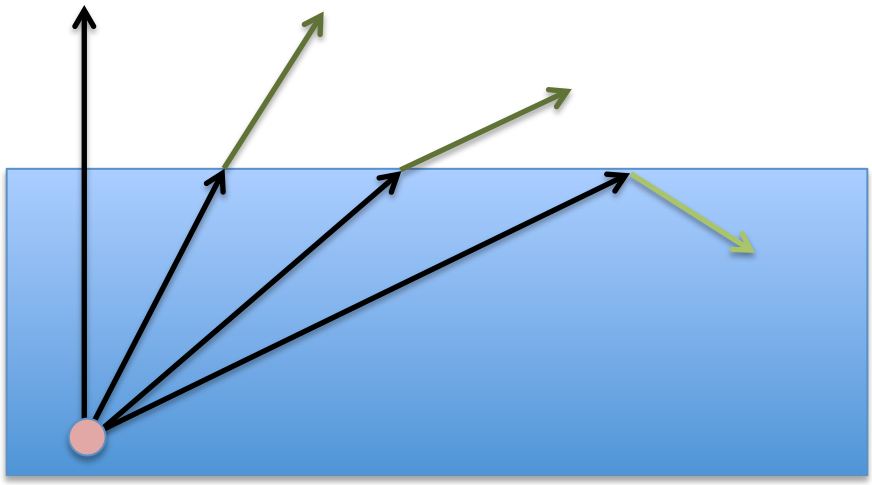


Figure 15: Increasing incident light angle until total internal reflection

$$\sin(\theta_c) = \frac{n_2}{n_1} \quad for\ n_1 > n_2$$

Figure 16: Critical angle equation

Using the values from Table 1, the critical angle for a water-to-air interface is 48.8º.

2.7.1 Image Processing: Spatial Distortion

Digital spatial transformations are commonly applied in applications of remote sensing, medical imaging, computer vision, and computer graphics [16]. An image can be warped to shear, scale, rotate, change perspective, or fix distortions. A spatial transformation defines a geometric relationship between the

original image and the desired output image. As illustrated in Figure 17, transformations map integers (pixels) into real numbers (between pixels) causing irregular output distributions [16]. For example, pixel C is mapped near the top of D' (real number) instead of in the center (integer). It is then possible that holes or overlaps will appear in the output image (C' and D' in Figure 17). Inverse mapping helps avoid these issues by mapping output pixels to input values.

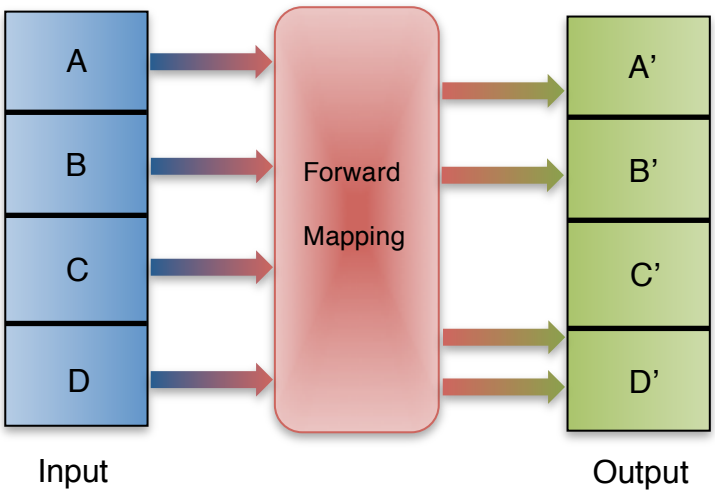


Figure 17: Forward mapping diagram

A general 3x3 matrix describes transformations for scaling, shearing, rotation, reflection, translation, and perspective as shown in Figure 18. Depending on the chosen transformation coefficients, the transformation matrix will warp each pixel value to a new value.

$$[x', y', w'] = [u, v, w] \cdot T$$

Where T maps from the uv plane to the x'y' plane.

$$T = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Figure 18: Transformation matrix for simple warping

More complex geometric transformations for distortion require polynomial transformations of the form listed in Figure 19 [16]. Again, the transformation is applied to every pixel (x,y) to get the corresponding output pixel (u,v).

Mapping functions U and V:

$$u = \sum_{i=0}^{N} \sum_{j=0}^{N-i} a_{ij} x^i y^j$$

$$v = \sum_{i=0}^{N} \sum_{j=0}^{N-i} b_{ij} x^i y^j$$

where a_{ij} and b_{ij} are constant polynomial coefficients

Figure 19: Polynomial Transformation

Reference [16] demonstrates several methods for solving the polynomial coefficients, typically with N = 2. All of the methods use known control points to determine the coefficients. The transformation will map the distorted control points to the undistorted control points. A polynomial order of one is a special

case of Figure 19 and reduces to the transformation matrix given in Figure 18. The polynomial transformation is a low-order global mapping function, meaning that local, high frequency deformations will not be corrected [16]. However, camera distortions are adequately described and corrected with the transformation.

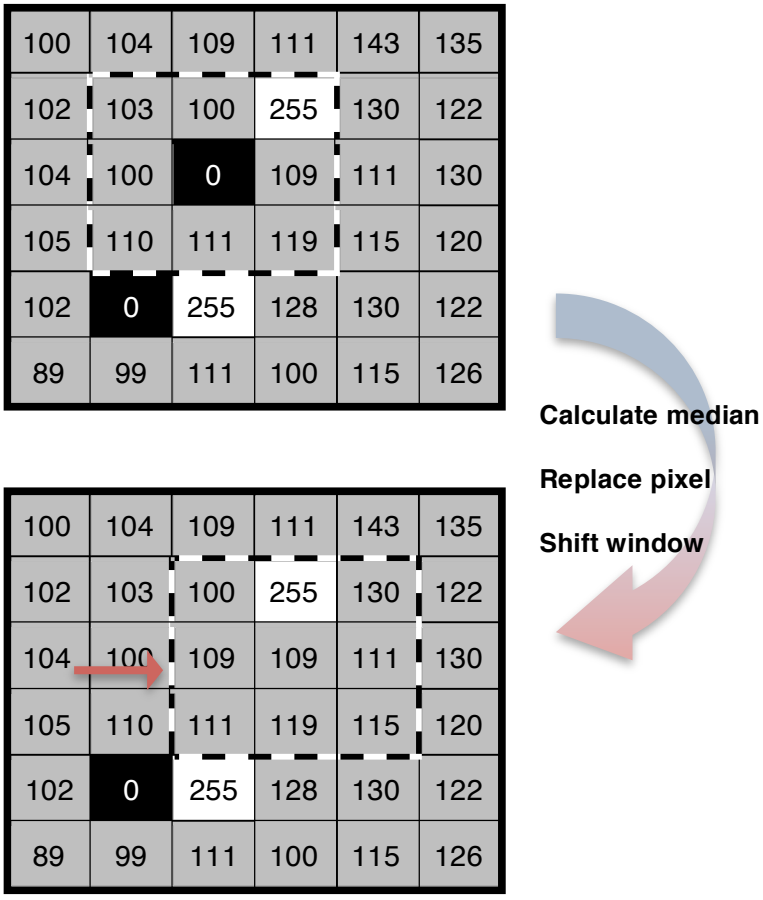
2.7.2 Image Processing: Noise Filtering

A real-world camera system will always have some level of noise. Image noise can be generated by the image sensor, during A/D conversion of the pixel values, or even during transmission of the image [17]. Filtering the noise leads to a perceptually more appealing image and is more efficient during encoding [18]. Two common noise types are salt & pepper and Gaussian-distributed noise. Salt & pepper noise derives its name from the white and black pixels it causes on an image. On an 8-bit image the salt & pepper noise can replace pixel values with either 0 or 255. Gaussian noise is typically caused by electronics [19] and is approximated by an additive white Gaussian noise (AWGN) model [20]. Figure 20 shows both types of noise applied to the same image.

Figure 20: From left to right - Original noiseless image, Added salt & pepper noise, Added Gaussian noise

Noise filtering commonly employs spatial filters like mean and median filters to remove noise. These can be decision-based filters that are only applied to a pixel if it is noisy, or ignore noisy pixels within the filter window (reducing the influence of noise on clean pixels). Regardless, the spatial filters operate in the same way, as demonstrated in Figure 21 for a median filter. A window is applied to groups of pixels, and based off some criteria the middle pixel is replaced.

A median filter can have varying dimensions, but this example will use a 3x3 window, where the middle cell is the pixel being replaced.



The median filter moves through every pixel, replacing the middle cell with the median of the values within the window.

Figure 21: Spatial filtering using median filter

Notice Figure 21 did not evaluate pixels along the edge, where the median filter would extend beyond the image. There are several ways to deal with an

edge condition, such as setting non-existent pixels to zero or mirroring the image into non-existent pixels. Gaussian noise filtering is discussed in Chapter 6.

Noise algorithms are typically compared using mean squared error (MSE) and peak signal-to-noise ratio (PSNR). The comparison is between the clean image and the filtered image. Ideally, the two images would match causing the MSE to be zero and therefore PSNR to be infinite. MSE and PSNR are defined in Figure 22.

Let I be the noiseless image and K be the filtered image with dimensions m x *n, then MSE is defined as:*

$$MSE = \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}[I(i,j) - K(i,j)]^2$$

PSNR is defined in decibels, where MAX_I is the maximum possible pixel value of the image (MAX is 255 for an 8-bit image).

$$PSNR = 10\log_{10}\left(\frac{MAX_I^2}{MSE}\right)$$

Figure 22: Definition for MSE and PSNR

CHAPTER 3 – DESIGN CONSIDERATIONS

3.1 INTRODUCTION

The basic design considerations for an affordable underwater camera tool for mini ROV robotic manipulators are cost, light sensitivity, resolution, and pressure housing. Each section includes information on the minimum specifications for the camera tool.

3.2 COST

More focus was put towards lowering cost so the end product is in line with the cost structure for mini ROV customers. The cost of the overall modular camera system, including the new robotic manipulator, should be modest compared to the $40k price range of mini ROVs. Another market to consider is marine research at universities, who have a more limited budget. The target cost for the camera including any additional processing is $200. The pressure housing is excluded from the included cost because the prototype will use more material than the product version.

3.3 LIGHT SENSITIVITY

The most important camera specification to consider for high performance is light-sensitivity [4]. A more light-sensitive image sensor within a camera reduces the need for a large amount of artificial lighting produced by the ROV. Less forward-directed light will minimize the turbid water masking effect caused by

reflection from suspended particles [4]. As a result, the image can have a greater depth of field [4] and appear clearer.

Different technologies have been developed over the years to increase light-sensitivity of an image sensor, such as SIT and ICCD aforementioned in the camera technology section. Cost and light-sensitivity are a function of the image sensor technology, so the image sensor technology does not need to be considered directly. It suffices to compare the minimum illumination and cost of different options. A minimum of 1 lux (lower number is better) is required for sufficient underwater use.

3.4 Resolution

Resolution, cost, and light-sensitivity have a close relationship. It is not possible to have all three at an ideal level. Great resolution and light-sensitivity come at higher cost, while great resolution and low cost come with poorer light-sensitivity and so forth. Resolution is the flexible camera specification since low cost and high light-sensitivity are of upmost importance for this application. However, the product will lose appeal if it below standard resolution. Therefore the minimum resolution must meet the standard resolution of 640x480.

Originally a high-definition camera was considered, but the monetary difference between a high-definition (HD) and standard-definition (SD) camera made the former option unfeasible. The technology involved in producing an HD camera that works in low lighting is currently very expensive. A suitable SD

camera costs less than $100 while an equivalently light-sensitive HD camera costs over $500. It was decided that a standard-definition camera would be more appropriate for the needs of the mini ROV community.

3.5.1 Pressure Housing

The pressure pod has several design considerations that can affect its depth rating and the image produced by the camera. Using a stronger material such as titanium or high-grade steel rather than aluminum can increase depth rating. Another option to improve depth rating is to fill the pressure housing with an optically transparent fluid instead of air, allowing the housing material to be thinner [5]. For the purposes of verifying the optical quality of the camera, a depth rating of 100ft will suffice. The final product can be upgraded to withstand greater pressure.

3.5.2 Pressure Housing Optics

The pressure pod can also be outfitted with either a hemispherical or planar lens. Hemispherical lenses allow for a larger field of view [6], but they are significantly more expensive and difficult to seal properly [7]. A planar lens may produce more image distortion due to greater refraction [6], but are less costly in time and money to produce.

Refraction is particularly important for understanding underwater cameras because it may cause several negative effects: unequal magnification and reduced viewing angle. A typical underwater camera sits in an air-filled pressure

housing and captures light rays through a viewing window. This interaction involves three mediums, shown in Figure 23 (which assumes a flat or planar viewing window). The light ray undergoes refraction at the water-to-plexiglass interface, and again at the plexiglass-to-air interface.

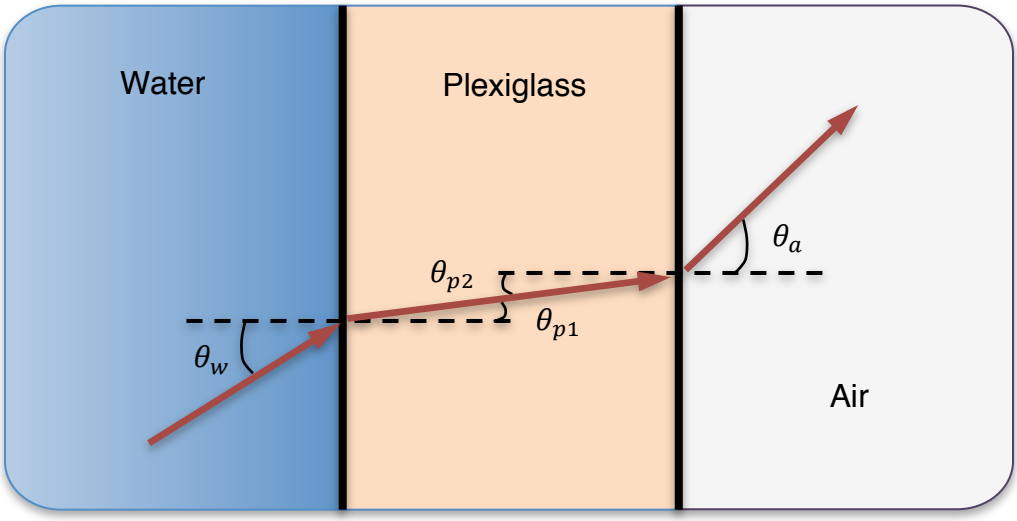


Figure 23: Underwater camera system light ray interactions

Since the plexiglass index of refraction is greater than that of water, there is no critical angle at the water-to-plexiglass interface. There is however, a critical angle at the plexiglass-to-air interface. Using the equation in Figure 16, the maximum incident angle from water is found to be 30.3º in Figure 24.

Critical angle for plexiglass-to-air interface:

$$\sin(\theta_c) = \frac{n_a}{n_p} \quad where\ n_a = 1, n_p = 1.492$$

$$\theta_c = \sin^{-1}\left(\frac{1}{1.492}\right) = \boxed{42.1°}$$

Incident angle from water-to-plexiglass interface for critical angle:

$$n_w \sin(\theta_w) = n_p \sin(\theta_p)$$

$$\theta_w = \sin^{-1}\left(\frac{n_p}{n_w}\sin(\theta_p)\right) = \sin^{-1}\left(\frac{1}{1.33}\sin(42.1°)\right) = \boxed{30.3°}$$

Figure 24: Derivation of critical angle for underwater camera using planar viewing port

Any light ray incident on the plexiglass viewing port will totally reflect and not be captured by the camera if the angle is beyond 30.3º. Snell's law can also be used to predict how the camera will observe incoming light. Figure 25 shows the derivation for observed light ray angles by the camera. Note that the angles are independent of the viewing port material (assuming flat port).

Light ray originates in water:

$$n_w \sin(\theta_w) = n_{p1} \sin(\theta_{p1})$$

The light ray enters the plexiglass at an angle of:

$$\theta_{p1} = \sin^{-1}\left(\frac{n_w}{n_p}\sin(\theta_w)\right)$$

The light ray will enter the air at an angle of:

$$\theta_a = \sin^{-1}\left(\frac{n_p}{n_a}\sin(\theta_{p2})\right)$$

Noting that θ_{p1} and θ_{p2} are the same and substituting:

$$\theta_a = \sin^{-1}\left(\frac{n_p}{n_a}\sin\left(sin^{-1}\left(\frac{n_w}{n_p}sin(\theta_w)\right)\right)\right)$$

$$\boxed{\theta_a = \sin^{-1}\left(\frac{n_w}{n_a}\sin(\theta_w)\right)}$$

Figure 25: Derivation for observed light ray angle

Plotting the water incident angle (true angle) versus apparent angle in air (Figure 26) shows that larger angles are exaggerated more than smaller angles. Apparent angle describes the perceived light ray angle in air. The only angle that is unaltered from its true position is a light ray perpendicular (0º) to the viewing port. The non-linearity in angle change gives rise to a radial distortion known as pincushion distortion, which is an increase in magnification towards the edges of an image.

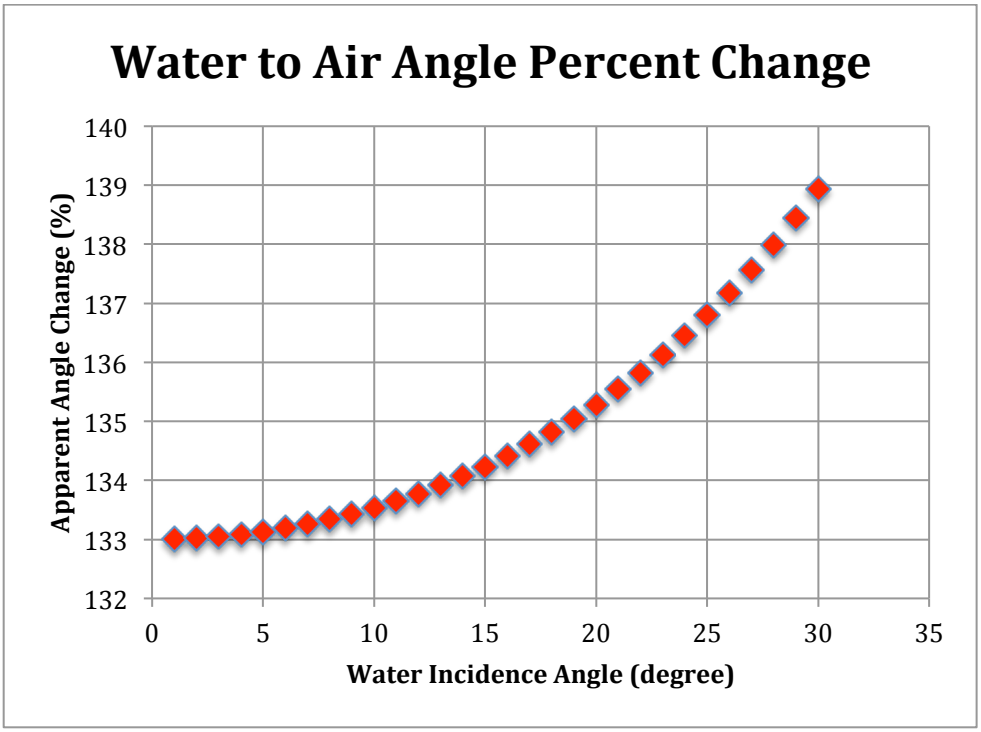


Figure 26: Light ray angle in air as a percentage of the water incident angle

The increase in magnification due to the water-air interface is derived in Figure 28 using Figure 27. Figure 29 applies the derivation to show magnification versus water incident angle.

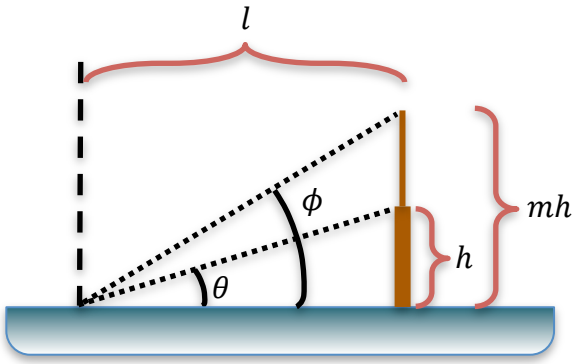


Figure 27: Illustration for magnification derivation

The image in Figure 27 *represents the actual image height* (h)*, and the perceived height* $(mh$*, where* m *is the magnification of* $h)$*.*

$$\tan(\theta) = \frac{h}{l}$$

$$\tan(\phi) = \frac{mh}{l} \rightarrow \frac{\tan(\phi)}{m} = \frac{h}{l}$$

Setting the equations equal to each other:

$$\frac{\tan(\phi)}{m} = \tan(\theta)$$

$$\boxed{m = \frac{\tan(\phi)}{\tan(\theta)}}$$

Figure 28: Derivation for magnification increase caused by water-air interface

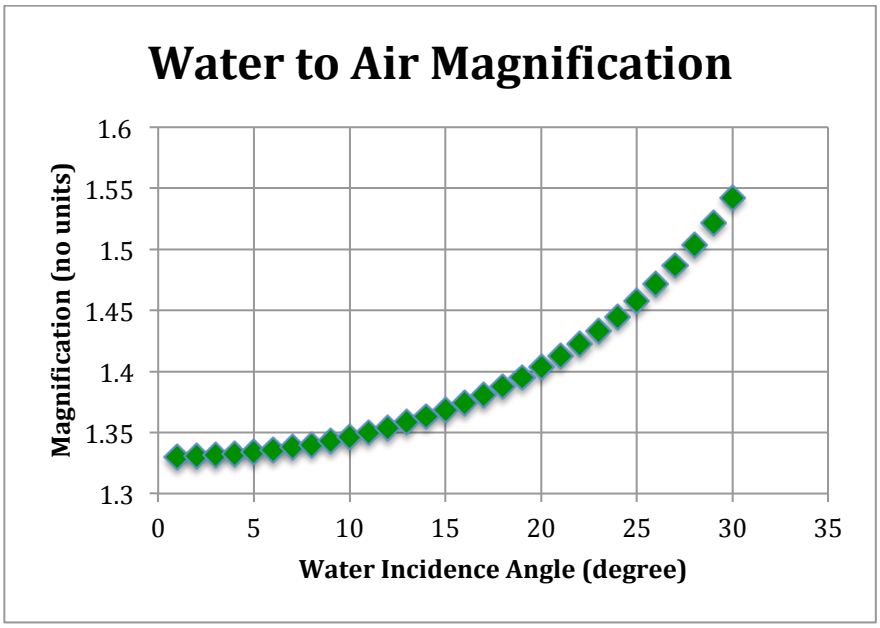


Figure 29: Magnification ratio for water-air interface. The magnification represents the object size as perceived by the camera relative to the actual object size.

As mentioned earlier, the increase in magnification towards the edges results in pincushion distortion, a type of radial distortion. Another type of radial distortion is caused by decreasing magnification towards the edges of an image, and is known as barrel distortion. Both types of distortion are illustrated in Figure 30.

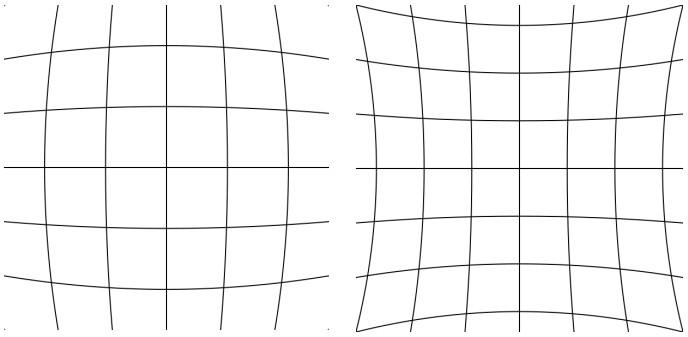
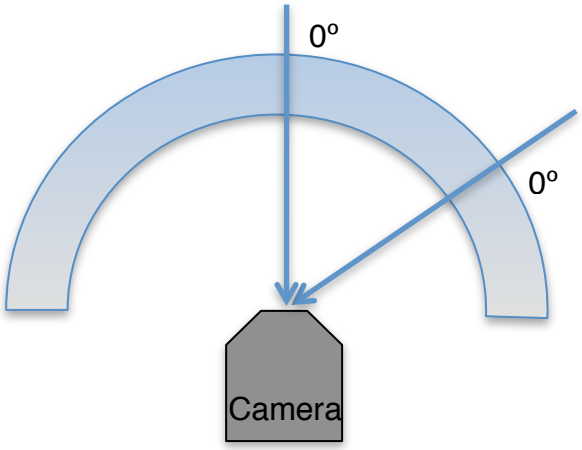


Figure 30: Radial distortions. Left - Barrel, Right - Pincushion

The planar port analysis has shown it will introduce pincushion distortion into the system. The light rays must enter the port unaffected to avoid any distortion. Using Figure 25, the only unaffected angle for an incident light ray is zero. The hemispherical port lens takes advantage of this property as shown in Figure 31. Other light rays are refracted away from the camera.



Camera must centered in hemispherical port

Figure 31: Hemispherical port only allows perpendicular light rays to enter the camera

CHAPTER 4 – COMPONENT SELECTION

4.1 INTRODUCTION

Using the design considerations and minimum project requirements, components can be chosen. The camera is picked first so the pressure housing can be custom designed to fit the form-factor of the camera.

4.2 CAMERA

A number of cameras were considered based on the qualities mentioned in Chapter 3. Table 2 shows the cameras under consideration with additional notes specific to each camera. The design requirements are best met with the Mintron MTB CM3160 camera shown in Figure 32. It specified for a light sensitivity of 0.2 lux at F2.0 for color and 0.03 lux at F1.2 for black and white, with a resolution of 520 Television Lines (TVL) and 600 TVL respectively with a cost of only $79. This camera exceeded the project requirement for minimum light-sensitivity and met the minimum resolution. Other cameras with a suitable price and light-sensitivity come inside a housing meant for other applications, such as security CCTV systems. The MTB CM3160 does not require any modification to be used inside a pressure housing, reducing production time. The infrared filter, the black box left of the lens seen in Figure 32, is removed when placed in the pressure housing. The removal of the filter gives the camera a square form-factor making it more suitable for a pressure enclosure.

Table 2: Camera comparison

Camera	Company	Minimum Illumination	Resolution	Notes
54C0/54C5 $75/$59	Mintron	1 lux (5600K color temp)	400TVL	Same camera but 54C5 layout may fit better in a tube.
MN2S 720p HD $545	Mintron	0.05 lux	800TVL	Has many image control options.
MN1P 1080p HD $595	Mintron	0.05 lux	1000TVL	Higher resolution version of the MN2S.
MTB CM3160 $79	Mintron	0.2 lux/F2.0 (color), 0.03 lux/F1.2 (B&W)	520TVL (color)/ 600TVL (B&W)	Has IR cut filter. Mintron's recommendation.
HDR-CX190/B $280	Sony	3 lux	TVL not given. 1080p	Would require the most modification to get uncompressed video streaming.
KCE-120	KCE	0.1 lux	700TVL	Box camera so is larger than other options. May require modification to reduce size.
HDT470	Speco Tech	0.1 lux	1000TVL	Box camera. Has many image control options.
Zues Plus, Atlas, Aurora, Sorpio Plus, Nova, Titan	Insite Pacific Inc	N/A	N/A	Only some information on the cameras listed is available online. Won't respond.

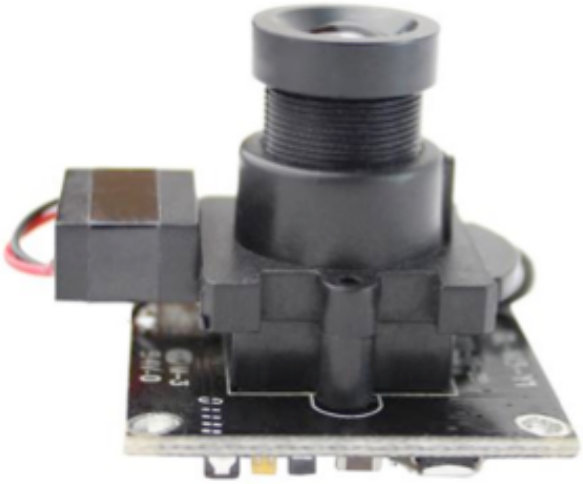


Figure 32: Mintron MTB CM3160 standard definition camera

The included large aperture, fixed-focal length lens serves the product's purpose well. It puts more focus on closer objects because its depth of field is shallower and located closer to the camera. Another effect of the lens is a wider field of view and a closer minimum focusing distance, which is necessary for inspection-style shots.

4.3 Pressure Housing

A prototype pressure housing currently allows for some preliminary testing of image integrity of the camera. The housing is machined from aluminum to save prototyping costs as seen in Figure 33. The prototype differs from the final pressure enclosure because it holds batteries to power the camera. This adds mass and weight that is unnecessary in the end product, since the camera will be

powered by the ROV. The prototype allows for underwater testing without an ROV for power.

Figure 33: Pressure housing showing planar lens with camera

Using an optically transparent fluid can save weight for the pressure housing, but the weight benefits are offset by additional difficulties. A corrective lens is needed to prevent visual distortion [5]. However the refractive index changes with increasing pressure, which reduces image quality [5]. Therefore an air-filled housing is employed because it is much easier to work with and lowers project cost.

Hemispherical lenses have an advantage in accurate image representation when using rectilinear lenses, but they are difficult to work with in practice [7]. A planar lens is a more prudent option for this project because it has a lower cost and is much easier to seal with a pressure housing. Additionally, a planar lens has pincushion distortion as described in Chapter 3, 3.5.1 Pressure Housing. The pincushion distortion will help neutralize the barrel distortion introduced by cheaper lenses (prevalent in small, low-cost cameras). Hence, a planar lens is used in the prototype and will be used in the final product.

CHAPTER 5 – SYSTEM VERIFICATION

5.1 INTRODUCTION

Visual testing of the camera was first performed by itself in air to verify low-light performance. After proving its low light capabilities, the camera was tested inside the pressure housing while underwater. The results prove the camera meets the minimum specifications and performs well underwater.

5.2 MINTRON CAMERA

The camera's close range ability, color light-sensitivity, and black/white light-sensitivity was tested in air. The close range focusing ability of the camera does not have a sharp transition from clear to blurry, but rather depends on the user's definition of acceptable. The minimum focusing distance without any noticeable blurring measures at 4.5 inches from the camera, as shown in Figure 34. This demonstrates the ability for close-up shots, the main function of the camera tool.

Figure 34: Minimum focusing distance is 4.5in (beyond camera view). Closest distance before black to white transition is no longer sharp.

A testing apparatus was created to verify the camera's performance in low light as shown in Figure 35. It consisted of a variable ambient light source created using a strip of LEDs and a white fabric to disperse the light.

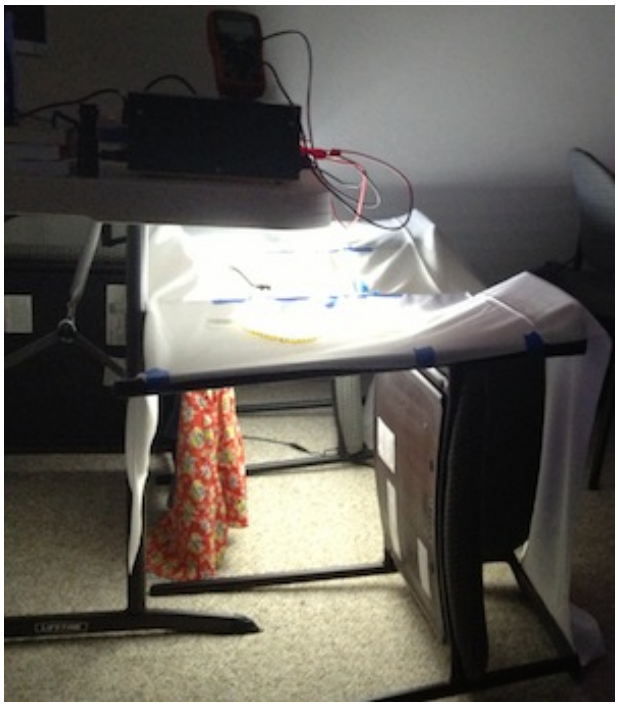


Figure 35: Light sensitivity testing setup

Current through the LEDs is related to brightness, which was measured in lux by a light meter. The circuit diagram seen in Figure 36 shows the testing circuitry. Varying the bias voltage changes the current into the base of the BJT, which changes the current through the LEDs and consequently determines their brightness.

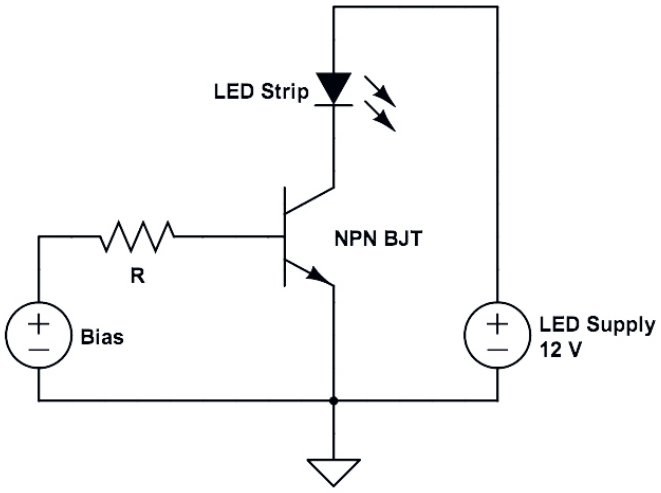


Figure 36: LED brightness control circuit

The light meter used was a Dr. Meter LX1010B, able to measure light down to 1 lux in increments of 1 lux. Light levels below 1 lux were calculated using extrapolation from the current measurement. Table 3 shows the measured brightness for different currents.

Table 3: Brightness versus current data

Current (μA)	Brightness (Lux)
3631	36
3464	35
3363	34
3184	32
2938	30
2759	28
2533	26
2333	24
2177	22
1953	20
1799	18
1597	16
1423	14
1232	12
1026	10
854	8
662	6
443	4
356	3
284	2
218	1

Figure 37 shows the linear relationship between brightness and current with a linear trendline of $y = 0.0101x$. The coefficient of determination is 0.99849, proving the accuracy of the linear fit and that the data can be extrapolated.

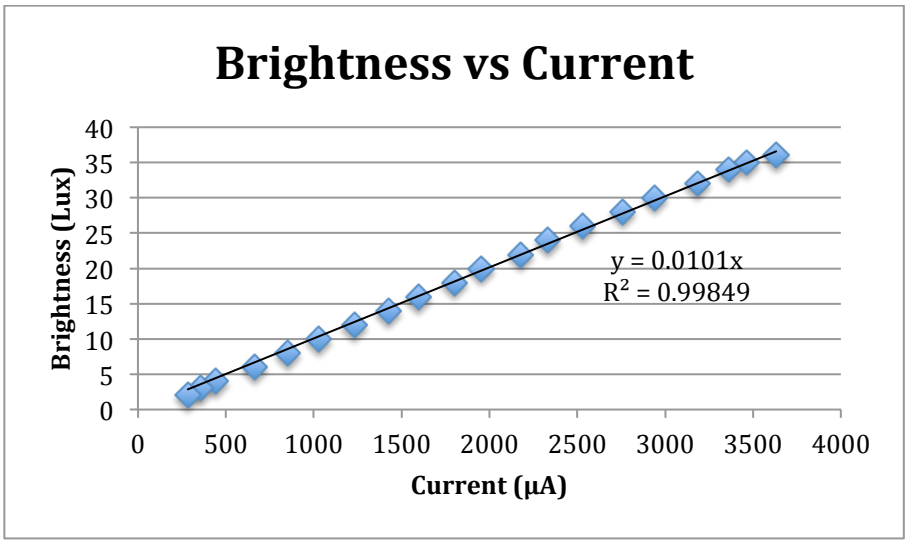


Figure 37: Brightness versus current plot with linear trendline

Figure 38 through Figure 41 shows the camera's color performance at several light levels. Light levels beyond full brightness (36.2 lux) appear the same in brightness and color. Decreasing light levels shows overall brightness becoming lower and colors become less saturated.

Figure 38: Color mode performance - Full brightness at 36.2 Lux

Figure 39: Color mode performance - Fading color at 11.5 Lux

Figure 40: Color mode performance - Notable increase in noise at 1.15 Lux

Figure 41: Color mode performance – Further increase in noise at 0.27 Lux

The light performance does not change between color and black/white modes when holding the aperture setting constant, as expected when using a Bayer filter. The camera works well in low light levels and meets the project design criteria. For comparison, an iPhone 4S can only operate down to 3.7 Lux before the image looks black. A comparison between the Mintron camera and iPhone 4S is shown in Figure 42 using a single candle 10ft away as the room's light source.

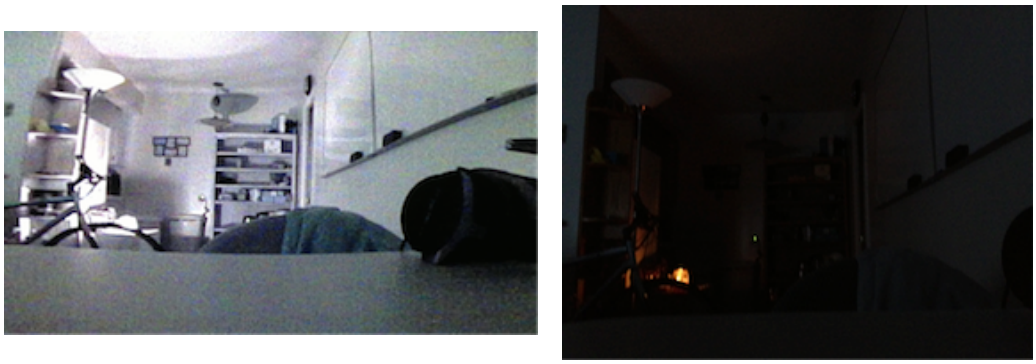


Figure 42: Single candle light source comparison. Left - Mintron camera, Right - iPhone 4S camera

There are some visual distortions, known as barrel distortion, around the outer edge of the image. This is apparent when looking at the left wall in Figure 42. This can be attributed to the wide-angle lens on the camera, a trade off for greater viewing angles.

5.3 Pressure Housing

A prototype pressure housing was built to allow for visual characterization, but differs from the final design in weight and size. Camera power comes from batteries contained in the housing, so the design can be tested without an ROV present. Figure 43 shows a cross-sectional view of the pressure housing. Note the extra length of the housing due to the batteries.

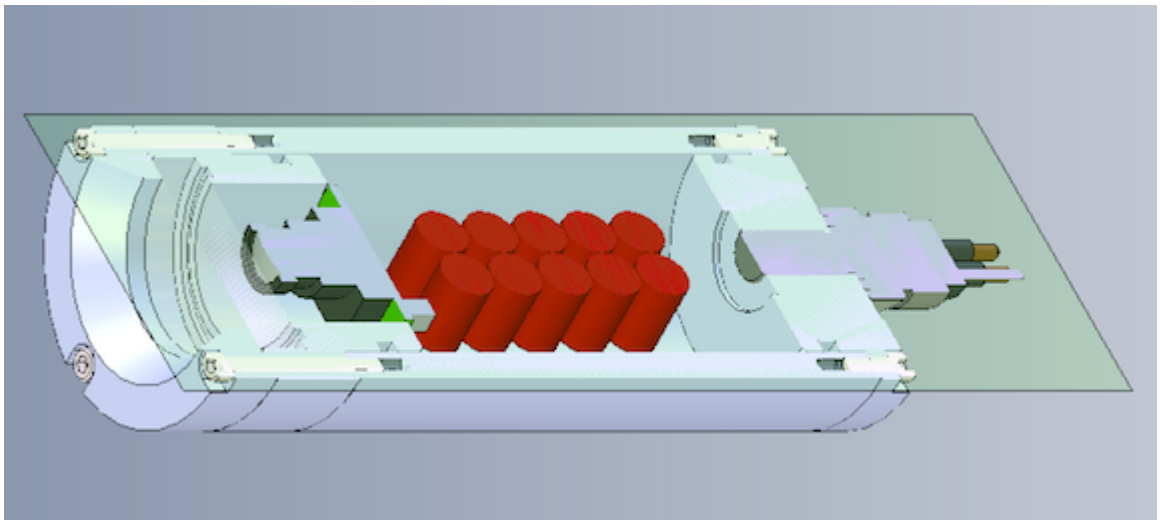


Figure 43: Cross-section view of pressure housing

Directly comparing an air view to an underwater view shows two differences: subjects appear closer in underwater shots, and barrel distortion is decreased towards the edges. However, the minimum focusing distance is unchanged from air to water.

Figure 44 is provided to demonstrate the increased barrel effect towards the edges of an image in air and water. The barrel distortion decreased in water as expected. The angle of view also decreased from 76º in air without the housing, to about 60º underwater. The underwater image in Figure 44 gets darker towards the edges, known as vignetting, due to approaching the critical angle.

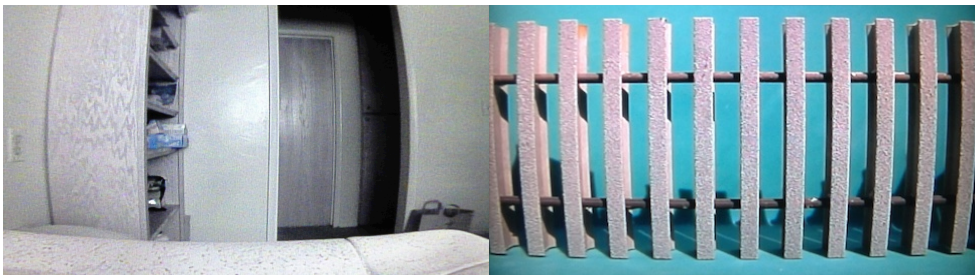


Figure 44: Increasing barrel distortion from center to edges in air (left) and water (right).

An obvious way to avoid vignetting is to use different lenses on the camera. Using a longer focal length will decrease the field of view, but as a result would eliminate vignetting and decrease barrel distortion. The view would be a zoomed version of wide-angle lens. Figure 45 shows a comparison between several focal lengths. Using Figure 45, it can be seen that the factory lens included with the Mintron camera is about 3.6 mm.

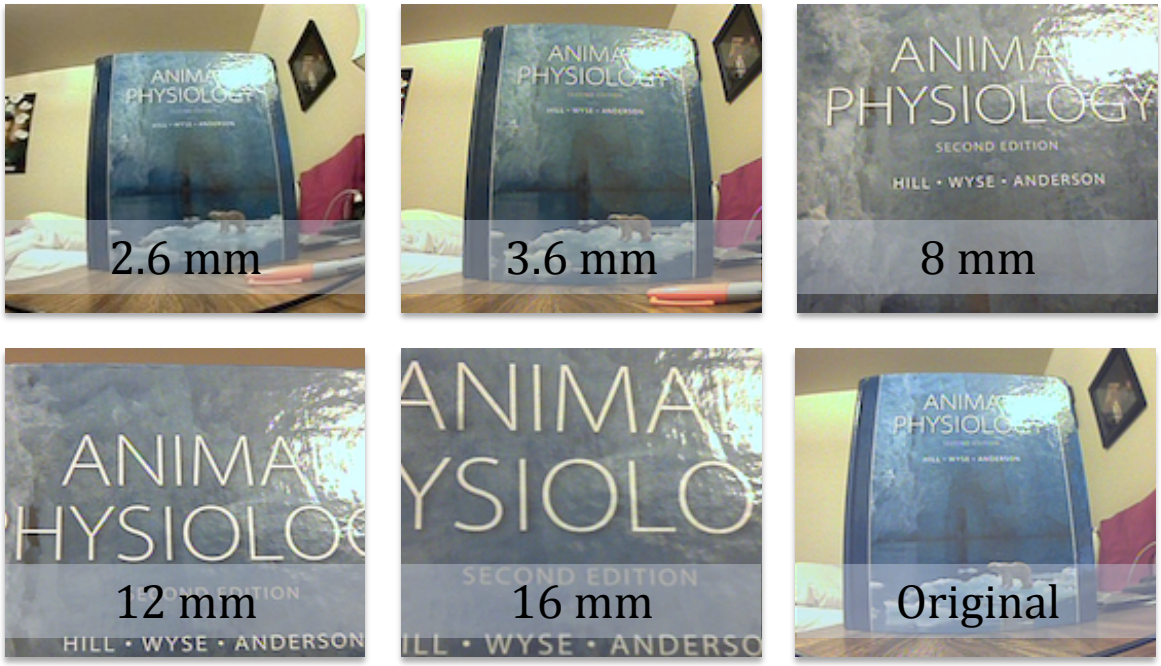


Figure 45: Comparison of focal lengths. Original lens matches 3.6 mm lens

The intended application of the camera is for close up inspection, not landscape shots. While it is up to the user, the most prudent lens options are between 3.6 mm to 8 mm. The wide field of view and close focusing distance allow these to get close to objects and remain stable from vibrations relative to other options. Longer focal lengths can achieve the same image from a further distance, but will be more susceptible to ROV movement.

CHAPTER 6 – VIDEO PROCESSING

6.1 INTRODUCTION

Processing the output of Minton camera is used to remove barrel distortion and enhance the video the quality. This chapter first considers video processing hardware, and then removes barrel distortion and enhances the video using Gaussian noise removal.

6.2 VIDEO PROCESSING HARDWARE

Processing of the video feed from the Mintron can happen anywhere between the camera and the display. There are advantages and disadvantages for each option. Processing the video feed immediately after it is outputted from the camera would require building a custom-printed circuit board (PCB) containing the items shown in Figure 46.

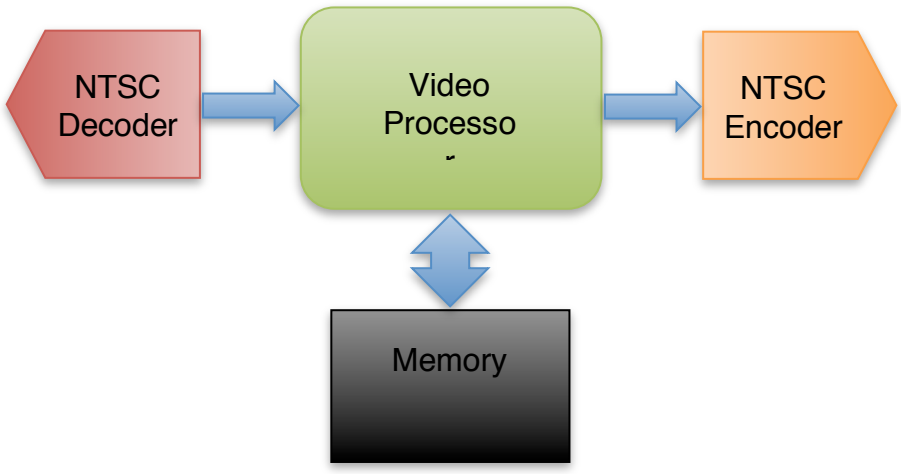


Figure 46: PCB block diagram for video processing on-board pressure housing

The PCB must fit behind the camera to fit in the pressure housing, limiting its length and width dimensions to 32x32mm. A two-sided board would make these limitations less of an issue. The biggest problem with using PCB is the cost per unit, assuming low quantity production. Using ExpressPCB prices for building PCBs, a single PCB would cost about $3. A quote for PCB assembly (loading components) from Protoexpress would cost at least $100 per board, exceeding the cost of the camera by $21. The total price for a video processing PCB including estimated part cost is shown in Table 4.

Table 4: Estimated costs for video processing in pressure housing

Item	Estimated Price	Notes
Building PCB	$3	6 boards in one request for $51 before shipping.
PCB Assembly	$100	For 10 boards with slowest assembly time.
Video Processor	$20	Considered FPGAs, SoCs, and dedicated video DSPs for average price. Range is ~$10-35.
NTSC Encoder	$7	May require external oscillator.
NTSC Decoder	$7	May require external oscillator.
Memory	$10	8MBIT SRAM
Total	$147	

The estimated total cost of $147 is a minimum, and the final price could be $200+ once supporting circuitry is added. For example the NTSC encoder and

decoder would most likely require an external oscillator, and more memory may be needed depending on the needs of the video processing algorithm. The estimated cost also ignores initial costs for prototypes and revisions. One of the primary goals for this project is to be cost effective, therefore the tripling of cost for video processing inside the pressure housing is impractical. In addition, this option must be conducted in real time and does not allow for recording and later processing of the video feed.

A more practical alternative for video processing is to use a computer on the surface side. Laptops offer significant computing power, and they are ubiquitous enough that most, if not all, ROV operations will have access to them. The only hardware required to process video from the camera would be a composite video to USB converter, like the “ION Video 2 PC” video converter available for ~$35 as of September 2013. Another advantage of using a computer is the ability to use existing image processing libraries, allowing for faster production of video processing software. Additional processing needs can be added with a simple software update, and the processing power will increase with Moore’s law. On the other hand, a PCB video processing method would most likely need to be redesigned in order to increase its processing power.

For the reasons listed above such as cost, scalability, and expandability, this project will implement all video processing needs on a computer using an existing computer vision library, OpenCV. Video input will be obtained using the

"ION Video 2 PC" video converter. This project uses Visual Studios 2010 on a Windows 7 virtual machine.

6.3 Barrel Distortion Correction

The previous chapter discussed two undesirable characteristics produced by the camera: barrel distortion and additive white Gaussian noise. Conveniently, the OpenCV library comes with a camera calibration sequence to correct the barrel distortion. The algorithm follows the basics outlined in the background information. It uses several images of a black and white chessboard at different orientations to identify control points. The control point locations relative to each other in each frame are compared against the known distances (size of the board, number of squares). It then generates a camera matrix to fix the distortion. The sample code contained within the OpenCV can be used directly to calibrate the camera. The included XML file simply needs to be set to use a camera. The algorithm is able to work in real-time with an average time of 15.13 milliseconds per frame averaged over 1 second of video (30 frames). Before and after calibration photos are shown in Figure 47 and Figure 48. Notice how the before picture shows the chessboard with barrel distortion, and the after image shows the chessboard with straight lines. The colored circles on checkerboard are added by the program to show the identification of control points.

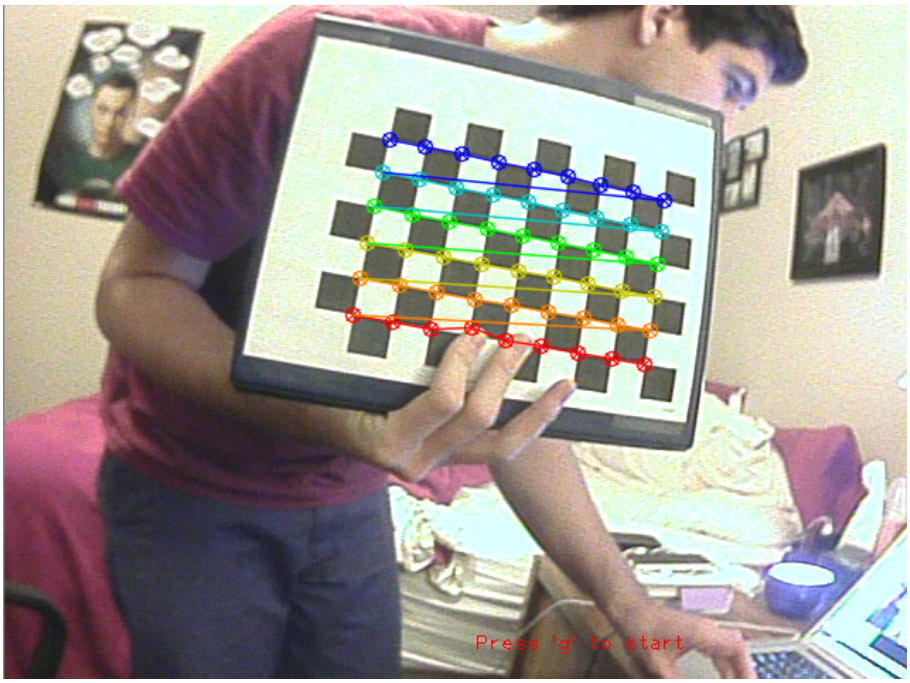


Figure 47: Screenshot of calibration program using a checkerboard

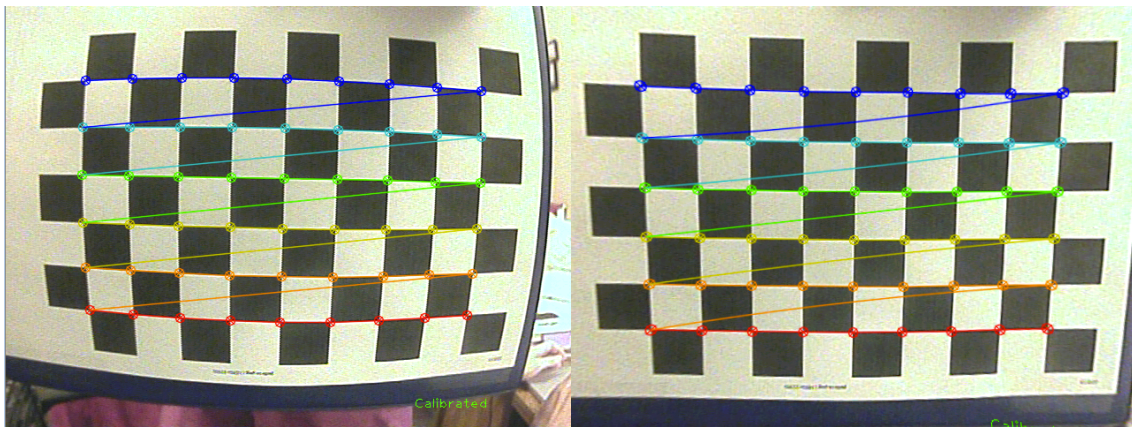


Figure 48: Left - Before calibration. Right - After calibration

6.4.1 Image Enhancement

There are dozens of algorithms to enhance video and images, but often the most effective methods come at the cost of computation time. For a real-time

video application, an enhancement algorithm must be completed in less than one frame period, $^{30frames}/_{sec} = 0.033sec$. Enhancement can include sharpening, increasing contrast, or a form of noise removal.

6.4.2 Image Enhancement: Noise Removal

The Mintron camera's noise was not experimentally characterized, but video noise is commonly modeled using an additive white Gaussian distribution (AWGN) [18] [20-22]. Many published algorithms for Gaussian noise removal use the wavelet domain, but the computational complexity is too high for real-time applications [18]. Simpler spatial domain methods are more suitable for real-time operation, and thus will be explored. The method is first implemented in Matlab and then extended into C++ using OpenCV.

An adaptive fuzzy filtering technique is employed to remove Gaussian noise for image enhancement. The algorithm first estimates the noise level, then detects similar pixels around every pixel, and finally filters the noise. The five steps in the algorithm are shown in Figure 49 and explained in detail below.

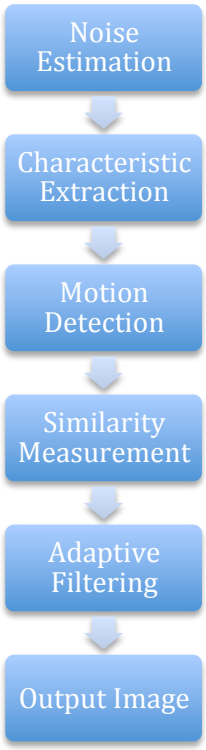


Figure 49: Noise removal algorithm flow diagram

The first step is estimating the standard deviation of the noise using the matrix in Figure 50, derived from two direction Laplacian operators. The operator reduces the influence of object edges and picks out noise. The standard deviation calculation is shown in Figure 50.

The difference dL operator derived from two direction Laplacian operators:

$$dL = \begin{bmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{bmatrix}$$

The noise information for a given pixel can be found using:

$$n(x,y) = \sum_{j=-1}^{1} \sum_{i=-1}^{1} dL(x+i, y+j) v(x+1, y+j)$$

Where $v(x,y)$ is the noisy frame. The standard deviation can then be calculated using:

$$\sigma_n = \sqrt{\frac{\sum_{y=1}^{H-2} \sum_{x=1}^{W-2} n(x,y)^2}{36(W-2)(H-2)}}$$

Figure 50: Standard deviation of noise calculation

The validity of the standard deviation calculation can be proved using a test image. Zero-mean Gaussian noise is added to a uniform gray image with a gray level of 128 (out of 255). Any deviation from a value of 128 is caused by noise. This allows the use of a standard deviation function in Matlab to compare against the proposed method. The method proved very accurate as can be seen by in Table 5.

Table 5: Standard Deviation comparison. Actual is using Matlab function, Calculated uses the method in [18]

Actual	Calculated	% Error
8.0849	8.0198	0.81
17.9801	17.8700	0.61
25.5378	25.3108	0.89
55.8491	55.9888	0.25
72.7281	72.4302	0.41

The next stage is characteristic extraction to determine between object edges, noise, and motion. A simple 3x3 averaging filter (shown in Figure 51) will obtain the local average intensity, which is most resistant to noise [18].

$$\bar{v}(x,y) = \left(\sum_{j=-1}^{1} \sum_{i=-1}^{1} v(x+i, y+j) \right) \Big/ 9$$

Figure 51: Average filtering equation

Stage three detects motion between frames. The motion information is used in a later stage to exploit temporal neighboring pixels if they belong to the same object. Motion is detected if the value of a pixel between two frames is greater than three standard deviations. Only 0.3% of noise would be beyond three standard deviations, therefore it is likely the pixel now belongs to a different object. The motion calculation is shown in Figure 52.

$$m(x,y,t) = \begin{cases} 1, |\bar{v}(x,y,t) - \bar{v}(x,y,t-1)| > 3\sigma_n \\ 0, |\bar{v}(x,y,t) - \bar{v}(x,y,t-1)| \leq 3\sigma_n \end{cases}$$

Figure 52: Motion detection equation

Stage four determines if pixels within a 3D window (N_s x N_s x N_t) are similar to the center pixel. The window spatial size is chosen based on the noise level as shown in Table 6. The window temporal size can be two if no noise is detected at a given location. A larger window will perform the stronger smoothing necessary for higher noise levels.

Table 6: Spatial window size for a given noise level

Noise Standard Deviation (σ_n)	Spatial Window Size (N_s)
≤ 3	3
5	5
7	7
≥ 9	9

The similarity measurement counts a neighboring pixel as similar if it is within a certain percentage of the current pixel. Alpha is typically set to 0.1 for the best results [18]. For example, if the current pixel is less in value compared to a neighbor pixel, they must be within 10% to be considered similar. The similarity equation is given in Figure 53.

$$s(x', y', t') = \begin{cases} 1, & \left| 1 - \dfrac{\bar{v}(x, y, t)}{\bar{v}(x', y', t')} \right| < \alpha \\ 0, & \left| 1 - \dfrac{\bar{v}(x, y, t)}{\bar{v}(x', y', t')} \right| \geq \alpha \end{cases}$$

Figure 53: Similarity measurement. Prime pixel values represent neighboring pixels.

In the final step, the adaptive fuzzy filter is applied to remove the noise. The filtered output pixel is a weighted average of its similar neighboring pixels (including temporally). The calculation is given in Figure 54.

A filtered output pixel is given by:

$$v'(x,y,t) = \frac{\sum v(x',y',t')\mu\big(v(x',y',t')\big)}{\sum \mu\big(v(x',y',t')\big)}$$

Where the summation includes all similar pixels within the filtering window (N_s x N_s x N_t) and $\mu(x)$ is a Gaussian distance function to describe the relationship between pixels by their distance.

$$\mu\big(v(x,y,t)\big) = e^{-\frac{\left(v(x',y',t')-v(x,y,t)\right)^2}{2\sigma_f^2}}$$

Typically $\sigma_f = \sigma_n$.

Figure 54: Adaptive fuzzy filtering equation for cleaned pixel

A simple example is shown in Figure 55 to prove the algorithm is functional. The example adds zero-mean Gaussian noise with a standard deviation of 8, and was able to improve the noisy image from a PSNR of 29.99 dB to a PSNR of 35.65 dB.

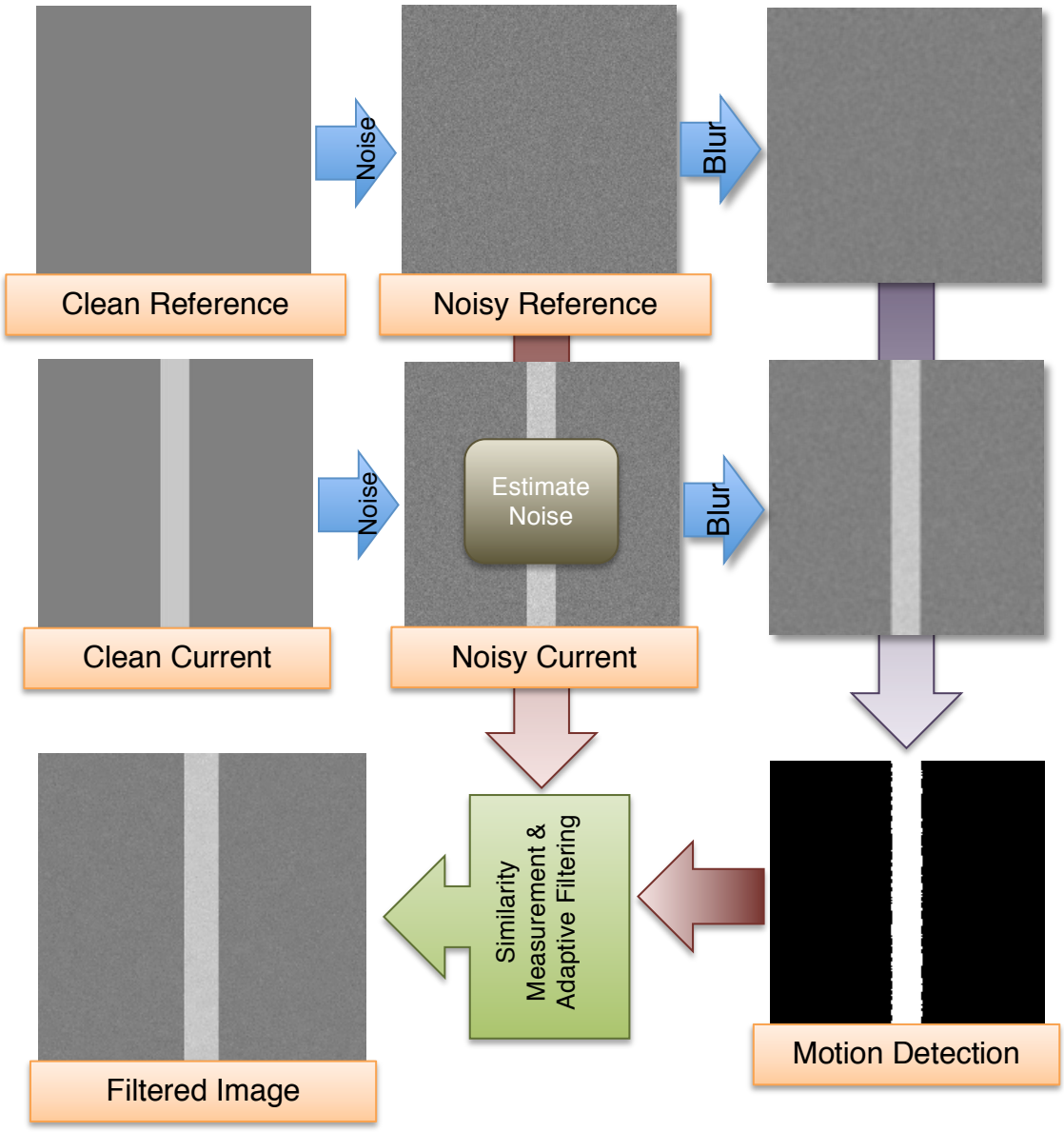


Figure 55: Gaussian noise removal algorithm flow diagram

Several more examples are shown in Figure 56 for a gray image and a color image. Each color channel is treated separately. Although these examples are using the same image, the noise is generated separately for the reference image and current image. This is analogous to still motion between video frames.

Figure 56: Top to bottom - Noiseless image, noisy image, filtered image.

Noise removal is applied to each color channel individually

The algorithm can be sped up by calculating the noise standard deviation once, and reusing the value for all future frames. Gaussian noise is intrinsic to the camera and does not change from frame to frame. To evaluate the Mintron camera's noise level, a uniform white surface was filmed under varying light conditions to maximize the noise characterization. It was found that in every case the noise had a standard deviation of less than three pixel values, meaning a spatial filter size of 3x3 will be used for future filtering. The average standard deviation of noise across all three channels was 1.5, with a maximum of 1.7. Dark frames exhibited less noise since black pixels cannot go below zero. A cleaned frame from a Mintron camera video feed is shown in Figure 57. The results show the image is smoothed without blurring details, which enhances the quality of the image.

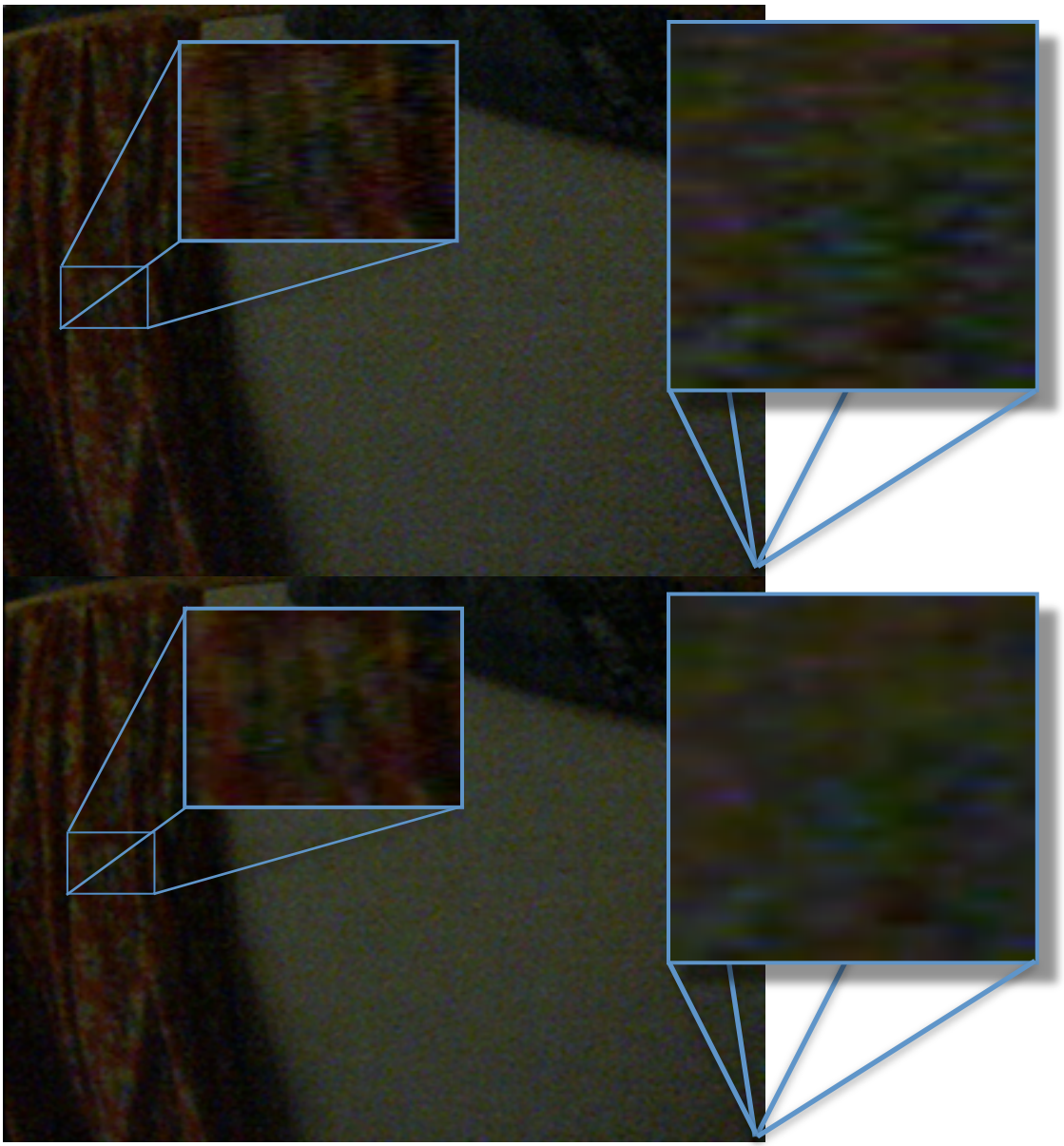


Figure 57: Before and after noise removal using images from Mintron camera

6.4.3 Image Enhancement: Algorithm Improvements

The method proposed by [18] for calculating similarity is not symmetric. This means that a neighbor pixel can be at most 25.5 levels greater than the

current pixel, however it can only be 23.2 levels less than the current pixel. There is no reason for the non-symmetric quality of the similarity measurement when applied to zero-mean noise. A modified method for calculating similarity is proposed in Figure 58. A comparison of the two methods in Table 7 shows the modified similarity measurement gives, on average, results at least 0.03 dB better. The data was taken for 11 noise levels and averaged over 10 runs.

The similarity measurement method in [18] is:

$$s(x', y', t') = \begin{cases} 1, \left| 1 - \dfrac{\bar{v}(x, y, t)}{\bar{v}(x', y', t')} \right| < \alpha \\ 0, \left| 1 - \dfrac{\bar{v}(x, y, t)}{\bar{v}(x', y', t')} \right| \geq \alpha \end{cases}$$

Let $v' = \bar{v}(x', y', t')$ (neighbor pixel) and $v = \bar{v}(x, y, t)$ (current pixel).

Then if $v' > v$ the similarity equation becomes: $1 - \frac{v}{v'} < 0.1 \Rightarrow 0.9 < \frac{v}{v'}$

Using the maximum case of $v' = 255$,

$$v > v' \cdot 0.9 = 229.5$$

$$v' - v = 255 - 229.5 = \boxed{25.5}$$

The maximum difference between the pixels for $v' > v$ is 25.5.

Now if $v > v'$ the similarity equation becomes: $\frac{v}{v'} - 1 < 0.1 \Rightarrow \frac{v}{v'} < 1.1$

Using the maximum case of $v = 255$,

$$v' > \frac{v}{1.1} = 231.8$$

$$v' - v = 255 - 231.8 = \boxed{23.2}$$

The maximum difference between the pixels for $v > v'$ is 23.2.

Therefore the similarity measurement is non-symmetric and will be modified to create symmetry.

For the case $v > v'$, the similarity measurement should be:

$$1 - \frac{v'}{v} < 0.1$$

Thus the maximum difference would be:

$$1 - \frac{v'}{255} < 0.1 \Rightarrow 0.9 < \frac{v'}{255} \Rightarrow v' > 229.5$$

$$v - v' = 255 - 229.5 = \boxed{25.5}$$

The similarity measurement is now symmetrical.

Figure 58: Modified similarity measurement derivation

Table 7: PSNR (dB) comparison of similarity measurement calculations

	Gray Image			Lenna Image		
Noise (σ_n)	**Original method**	**Modified method**	**Difference**	**Original method**	**Modified method**	**Difference**
2.6	4.17	4.28	0.12	1.13	1.19	0.06
5.5	5.46	5.50	0.04	3.58	3.59	0.01
8.4	5.63	5.67	0.04	3.87	3.91	0.04
11.5	5.64	5.66	0.02	4.08	4.10	0.02
14.1	5.56	5.61	0.05	4.17	4.17	0.01
16.4	5.55	5.55	0.00	4.21	4.23	0.01
18.1	5.45	5.50	0.04	4.23	4.24	0.01
19.9	5.38	5.43	0.04	4.22	4.26	0.04
21.3	5.34	5.37	0.03	4.22	4.27	0.05
23.0	5.28	5.33	0.05	4.23	4.25	0.02
24.2	5.22	5.27	0.05	4.22	4.28	0.05
		Average	0.043 dB		Average	0.031 dB

An improvement can be made to the algorithm by simplifying the final step. Replacing the adaptive fuzzy filter with an adaptive averaging filter improves both PSNR and speed. The proposed adaptive averaging filter is described in Figure 59.

The averaging filter is given by:

$$f_a = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 0 \end{bmatrix} for\ \sigma_n \leq 10$$

$$f_a = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} for\ \sigma_n > 10$$

Only similar pixels one Manhattan distance away are averaged. A weighted averaging filter is used to reduce excessive blurring from too many similar pixels with low noise levels.

A filtered output pixel then becomes:

$$v'(x,y,t) = \frac{\sum v(x',y',t')\mu\big(v(x',y',t')\big)}{\sum \mu\big(v(x',y',t')\big)}$$

Where the summation includes all similar pixels within the filtering window (N_s x N_s x N_t). Notice the spatial window size is always three and does not adjust with noise as it did in [18].

Figure 59: Proposed adaptive averaging filter

The proposed adaptive averaging filter always has a spatial window size of three and does not change with noise. The reduced window size at higher noise levels allows it run much faster. A comparison of PSNR, CPU time and noise level was computed in Matlab using a gray, two-edge image and the cameraman image. The comparison results are shown in Table 8, and the test images are shown in Figure 60.

Table 8: Comparison of PSNR, CPU time, and noise level for fuzzy and averaging filters

Image	Measured Noise (σ_n)	Algorithm	PSNR (dB)	CPU time (s)
Gray	2.5	Fuzzy Filter	4.14	0.2
		Proposed Filter	8.78	0.8
	8.0	Fuzzy Filter	5.61	2.0
		Proposed Filter	9.25	0.8
	25.2	Fuzzy Filter	5.17	1.8
		Proposed Filter	7.98	0.8
Camera Man	2.7	Fuzzy Filter	2.25	1.2
		Proposed Filter	3.02	5.2
	8.0	Fuzzy Filter	3.83	12.0
		Proposed Filter	6.59	5.0
	24.2	Fuzzy Filter	4.51	12.0
		Proposed Filter	6.45	5.0

Figure 60: Gray test image and cameraman test image

It was observed during testing that both algorithms are sensitive to image size and quality of the original “noiseless” image.

6.4.4 Image Enhancement: OpenCV Implementation

The noise removal algorithm was extended into OpenCV. The PSNR results for different noise levels are the same of course, but the computation time changed. The noise estimation stage is left out to speed up the algorithm. Instead, the pre-computed noise standard deviation value is used. The CPU time for the original algorithm and proposed modifications are listed in Table 9. The time for individual stages varied considerably so the results were averaged over the entire algorithm for 10 runs. Initialize, characteristic extraction, motion detection, and similarity measurement stages are identical between the original and modified algorithm. The difference in time lies in the filtering step and the size of the filtering window. The gray image has a noise standard deviation of 8 requiring a 9x9 spatial filtering window in the original algorithm. The modified algorithm only requires a 3x3 spatial window allowing it to achieve speeds over 6 times faster compared to the original algorithm. The color image sees a less dramatic increase in speed because both filters use a spatial window size of 3x3 for a noise standard deviation of 1.5. The modified algorithm is still 37% faster than the original algorithm because it only uses neighbors one Manhattan distance away.

Table 9: CPU time comparison of Original and Modified algorithm using OpenCV

Image	Original Algorithm	Modified Algorithm
Gray image (200x200) $\sigma_n = 8$	322 ms	52 ms
Color image (640x480) $\sigma_n = 1.5$	429 ms	312 ms

Even with the speed improvement, the algorithm is too slow to be implemented in real time. It needs to be about 10x faster to be completed within the 33ms time window between frames.

The noise level on the Mintron camera is low enough (~1.5 standard deviation) that it will not impede real-time operation. Video footage can be recorded and then cleaned using the algorithm when speed is less of a concern. Figure 61 shows an underwater image before and after the filtering image enhancement. The enhancement for such low noise standard deviations is minimal. Figure 62 shows a zoomed in portion of the before and after images to show the smoothing effect from the image enhancement.

Figure 61: Left - Before image enhancement, Right - After image enhancement

Figure 62: Left - Zoomed in portion of before image, Right - Zoomed in portion of after image

CHAPTER 7 – CONCLUSION AND FUTURE WORK

This paper considered the design of an underwater camera tool while balancing cost with performance. To meet the design criteria, which are summarized in Table 10, a highly light sensitive, standard-definition camera was housed in an air-filled pressure enclosure with a planar lens. Testing demonstrated the camera tool is highly light sensitive and effective for close underwater inspection while staying affordable. Barrel distortion was corrected and Gaussian noise was removed using OpenCV, but only the barrel distortion proved suitable for real-time. However, the image enhancement can be applied to recorded video footage where real-time is not a concern.

Table 10: Summary of design requirements

Design Requirement	Minimum	Achieved
Cost	$200	$114
Light-Sensitivity	1 lux	0.2 lux
Resolution	Standard Resolution	Standard Resolution
Depth Rating	100 ft	> 100 ft

Future work will include testing in deeper ocean waters, and affixing the camera tool to a robotic manipulator arm of a mini ROV. Noise removal can be improved to remove more noise either from improvements to the proposed algorithm, or using a more accurate noise model. The requirements for real-time operation should also be investigated more closely.

REFERENCES

[1] Costa, M.J.; Goncalves, P.; Martins, A.; Silva, E., "Vision-based assisted teleoperation for inspection tasks with a small ROV," *Oceans, 2012* , vol., no., pp.1,8, 14-19 Oct. 2012

[2] Sakagami, N.; Shibata, M.; Hashizume, H.; Hagiwara, Y.; Ishimaru, K.; Ueda, T.; Saitou, T.; Fujita, K.; Kawamura, S.; Inoue, T.; Onishi, H.; Murakami, S., "Development of a human-sized ROV with dual-arm," *OCEANS 2010 IEEE - Sydney* , vol., no., pp.1,6, 24-27 May 2010

[3] Jee-Hwan Ryu; Dong-Soo Kwon; Pan-Mook Lee, "Control of underwater manipulators mounted on an ROV using base force information," *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on* , vol.4, no., pp.3238,3243 vol.4, 2001

[4] Stewart, D. J., “Conventional” underwater camera systems-an update," *OCEANS '94. 'Oceans Engineering for Today's Technology and Tomorrow's Preservation.' Proceedings* , vol.1, no., pp.I/176,I/180 vol.1, 13-16 Sep 1994

[5] Gelze, J.; Lehr, H., "On the way to a pressure-tolerant imaging system," *OCEANS 2011*, vol., no., pp.1,4, 19-22 Sept. 2011

[6] Kunz, C.; Singh, H., "Hemispherical refraction and camera calibration in underwater vision," *OCEANS 2008* , vol., no., pp.1,7, 15-18 Sept. 2008

[7] R. D. Christ and R. L. Wernli Sr, “A bit of history” in *THE ROV MANUAL A User Guide for Observation Class Remotely Operated Vehicles,* 1st ed. Jordan Hill, Oxford: Elsevier Ltd., 2007, pp. 1-10.

[8] R. C. Gonazales and R. E. Woods, “Color image processing,” in *Digital Image Processing*, 3rd ed. Upper Saddle River, NJ: Pearson Education Inc., 2008, p. 418.

[9] Lustica, A., "CCD and CMOS image sensors in new HD cameras," *ELMAR, 2011 Proceedings* , vol., no., pp.133,136, 14-16 Sept. 2011

[10] Martin-Gonthier, P.; Molina, R.; Cervantes, P.; Magnan, P., "Analysis and optimization of noise response for low-noise CMOS image sensors," *New Circuits and Systems Conference (NEWCAS), 2012 IEEE 10th International* , vol., no., pp.513,516, 17-20 June 2012

[11] IEEE Standard on Video Techniques: Measurement of Resolution of Camera Systems, 1993 Techniques," *IEEE Std 208-1995* , vol., no., pp.0_1,, 1995

[12] G. Wolberg, “Preliminaries,” in *Digital Image Warping*, 1st ed. Lost Alamitos, CA: IEEE Computer Society Press, 1990, pp. 37-38.

[13] A. Steiner, “Understanding the Basics of Underwater Lighting,” *ON&T*, vol. 19, no. 4, pp.10-12, May 2013.

[14] Stewart, D. J., ""Conventional" underwater camera systems-an update," *OCEANS '94. 'Oceans Engineering for Today's Technology and Tomorrow's Preservation.' Proceedings* , vol.1, no., pp.I/176,I/180 vol.1, 13-16 Sep 1994

[15] R. A. Serway and J. W. Jewett, "The nature of light and the laws of geometric optics" in *Physics for Scientists and Engineers with Modern Physics*, 7th ed. Belmont, CA: Thompson Higher Education, 2008, pp. 985-987.

[16] G. Wolberg, "Spatial transformations," in *Digital Image Warping*, 1st ed. Los Alamitos, CA: IEEE Computer Society Press, 1990, pp. 41-94.

[17] Balasubramanian, S.; Kalishwaran, S.; Muthuraj, R.; Ebenezer, D.; Jayaraj, V., "An efficient non-linear cascade filtering algorithm for removal of high density salt and pepper noise in image and video sequence," *Control, Automation, Communication and Energy Conservation, 2009. INCACEC 2009. 2009 International Conference on* , vol., no., pp.1,6, 4-6 June 2009

[18] Jing Wu; Xin Du; Yun-fang Zhu; Gu Wei-kang, "Adaptive fuzzy filter algorithm for real-time video denoising," *Signal Processing, 2008. ICSP 2008. 9th International Conference on* , vol., no., pp.1287,1291, 26-29 Oct. 2008

[19] Yong Huang; Hui, L., "An adaptive spatial filter for additive Gaussian and impulse noise reduction in video signals," *Information, Communications*

and Signal Processing, 2003 and Fourth Pacific Rim Conference on Multimedia. Proceedings of the 2003 Joint Conference of the Fourth International Conference on , vol.1, no., pp.523,526 Vol.1, 15-18 Dec. 2003

[20] Gupta, N.; Plotkin, E.I.; Swamy, M.N.S., "Temporally-Adaptive MAP Estimation for Video Denoising in the Wavelet Domain," *Image Processing, 2006 IEEE International Conference on* , vol., no., pp.1449,1452, 8-11 Oct. 2006

[21] Acharya, A.; Meher, S., "Robust video denoising for better subjective evaluation," *Image Information Processing (ICIIP), 2011 International Conference on* , vol., no., pp.1,5, 3-5 Nov. 2011

[22] Florian Luisier, Thierry Blu, and Michael Unser, “SURE-LET for Orthonormal Wavelet Domain Video Denoising,” *IEEE Trans. Circuits Syst. Video Tech.*, vol. 20, no. 6,pp 913-919, June 2010

APPENDIX A: MATLAB CODE

Matlab code to implement noise removal algorithm from [18]:

```
function difference = fuzzyMeClean(referenceImage, inputImage, noiseEnable,
noiseValue, figureNum)
% function fuzzyMeClean(input, addNoise, noiseValue, figureNum)
%
% This function implements the video noise removal from:
% "Adaptive Fuzzy Filter Algorithm for Real-time Video Denoising"
%
% The process goes: Noise Estimation -> Characteristic Extraction -> Motion
% Detection -> Similarity Measurement -> Adaptive Filtering
%
% Inputs:
%   referenceImage - Filename for reference image, frame(t-1)
%
%   inputImage - Filename for input image to denoise, frame(t)
%
%   noiseEnable - Enable for adding Gaussian noise to image
%
%   noiseValue - Defines variance of gaussian noise to be added to input
%
%
% Outputs:
%   Figure showing the noiseless image, noisy image, and corrected image

%% Initialize
refImage = imread(referenceImage);
origImage = imread(inputImage);

if noiseEnable
   image = imnoise(origImage, 'gaussian', 0, noiseValue); % Add zero mean
Gaussian noise to image
   refImage = imnoise(refImage, 'gaussian', 0, noiseValue);
else
   image = origImage;
end


[Y X Z] = size(origImage);
```

```
Ns = 1; % floor(halfSpatialWindow). Experimentally determined spatial window
size of 3
Nt = 2; % Temporal window size

%% Noise Estimation
fprintf('\nNoise Estimation: ');
tic;
% Define Laplacian smooth operator
laplacian = [1 -2 1; -2 4 -2; 1 -2 1]; % Smoothes image

% Preallocate noise information matrix
noiseInfo = int64(zeros(Y,X,Z));


% Apply laplacian
for y = 1:Y
    for x = 1:X
        for z = 1:Z

            leftReflect = 0;
            rightReflect = 0;
            topReflect = 0;
            bottomReflect = 0;

            % Adjust neighbor calc for edges. If edge, limit window size
            if ( x == 1 ) % Left edge
                leftReflect = 1;
            end

            if ( x == X ) % Right edge
                rightReflect = -1;
            end

            if ( y == 1 ) % Top edge
                topReflect = 1;
            end

            if ( y == Y ) % Bottom edge
                bottomReflect = -1;
            end

            % Add weighted pixels in Laplacian window; Set pixels outside
            % of image to zero by limiting window size at edges
```

```
            for yy = -1+topReflect:1+bottomReflect
                for xx = -1+leftReflect:1+rightReflect
                    noiseInfo(y,x,z) = noiseInfo(y,x,z) +
int64(laplacian(yy+2,xx+2))*int64(image(y+yy,x+xx,z));
                end
            end

        end
    end
end

% Calculate standard deviation of noise
noiseInfoTotal = int64(zeros(1,3));

for z = 1:Z
    for y = 2:Y-2
        for x = 2:X-2
            noiseInfoTotal(z) = noiseInfoTotal(z) + noiseInfo(y,x,z)*noiseInfo(y,x,z);
        end
    end
end

for z = 1:Z
    noiseSTD(z) = sqrt( double(noiseInfoTotal(z)) / ( 36*(X-2)*(Y-2) ) );
end

fprintf([num2str(noiseSTD) ' stdev\n']);
toc;
% noiseSTD = [1.5 1.5 1.5];% TESTING

if (noiseSTD(1) <= 3)
    Ns = floor(3/2);
elseif (noiseSTD(1) <= 5)
    Ns = floor(5/2);
elseif (noiseSTD(1) <= 7)
    Ns = floor(7/2);
else
    Ns = floor(9/2);
end

%% Characteristic Extraction
% Blur both input images for use in Motion Detection Step. This step is
% modified compared to real-time program because the reference blur image
```

```
% would already exist.
% Paper uses blur window of size 3x3.
fprintf('\nBlur time:\n');
tic;
blurImage = zeros(Y,X,Z);
blurImagePrev = zeros(Y,X,Z);

% Calculate blur
for y = 1:Y
   for x = 1:X
      for z = 1:Z

         blurWinTotal = uint16(0); % Must be large enough to hold window
summation
         blurWinTotalPrev = uint16(0);

         leftReflect = 0;
         rightReflect = 0;
         topReflect = 0;
         bottomReflect = 0;

         % Adjust neighbor calc for edges. If edge, limit window size
         if ( x <= Ns ) % Left edge
            leftReflect = 1;
         end

         if ( x > X-Ns ) % Right edge
            rightReflect = -1;
         end

         if ( y <= Ns ) % Top edge
            topReflect = 1;
         end

         if ( y > Y-Ns ) % Bottom edge
            bottomReflect = -1;
         end

         denom = 0;

         % Calulate total pixel values in window
         for yy = -1+topReflect:1+bottomReflect
            for xx = -1+leftReflect:1+rightReflect
```

```
                blurWinTotal = blurWinTotal + uint16(image(y+yy,x+xx,z));
                blurWinTotalPrev = blurWinTotalPrev +
uint16(refImage(y+yy,x+xx,z));
                denom = denom + 1;
            end
        end

        % Calculate average. Slight inaccuracy at edges from dividing
        % by number greater than number of summed pixels
        blurImage(y,x,z) = blurWinTotal/denom;
        blurImagePrev(y,x,z) = blurWinTotalPrev/denom;

    end
  end
end

blurImage = uint8(blurImage); % Convert from uint16 to uint8 for imshow()
blurImagePrev = uint8(blurImagePrev);
toc;

%% Motion Detection
% Motion detection threshold noise relationship determined experimentally by
paper
fprintf('\nMotion Detection time:\n');
tic;

motionThreshold = 3.*noiseSTD; % Can probably set noiseSTD to 1.5 for Mintron
motion = zeros(Y,X,Z); % Holds the detected motion between two frames

for y = 1:Y
  for x = 1:X
    for z = 1:Z

        if ( abs( int16(blurImage(y,x,z)) - int16(blurImagePrev(y,x,z)) ) >
motionThreshold(z))
            motion(y,x,z) = 255; % Set to 255 instead of one to show difference
        else
            motion(y,x,z) = 0;
        end

    end
  end
end
```

```
toc;

%% Similarity Measurement & Adaptive Fuzzy Filtering
% alpha is set to 0.1 from paper

fprintf('\nSimilarity Measurement & Adaptive Fuzzy Filtering time:\n');
tic;

alpha = 0.1;
sigmaf = noiseSTD; % Spread parameter. Larger -> more smoothing

cleanedFrame = zeros(Y,X,Z); % Initialize filtered frame

offset = Ns + 1; % Offset used for indexing

for z = 1:Z
   for y = 1:Y
      for x = 1:X

         pixel = blurImage(y,x,z);

         leftReflect = 0;
         rightReflect = 0;
         topReflect = 0;
         bottomReflect = 0;

         % Adjust neighbor calc for edges. If edge, limit window size
         if ( x <= Ns ) % Left edge
            leftReflect = Ns-x+1;
         end

         if ( x > X-Ns ) % Right edge
            rightReflect = X-x-Ns;
         end

         if ( y <= Ns ) % Top edge
            topReflect = Ns-y+1;
         end

         if ( y > Y-Ns ) % Bottom edge
            bottomReflect = Y-y-Ns;
         end
```

```
        %%% Check similarity of surrounding pixels %%%

        similar = zeros(3,3,2); % Initialize filter window similarity check

        for yy = -Ns+topReflect:Ns+bottomReflect
           for xx = -Ns+leftReflect:Ns+rightReflect

              temp = blurImagePrev(y+yy,x+xx,z);%%TESTING

              % Check previous frame first if no motion
              if (motion(y,x,z) == 0)
                 pixDiv = double(pixel)/double(blurImagePrev(y+yy,x+xx,z));

                 if (pixDiv > 1)
                    result = pixDiv - 1;
                 else
                    result = 1 - pixDiv;
                 end

                 if (result < alpha)
                    similar(yy+offset,xx+offset,1) = 1; % Average gray value is
similar to pixel
                 else
                    similar(yy+offset,xx+offset,1) = 0; % Neighbor pixel belongs to
different object
                 end
              end

              % Check current frame
              pixDiv = double(pixel)/double(blurImage(y+yy,x+xx,z));

              if (pixDiv > 1)
                 result = pixDiv - 1;
              else
                 result = 1 - pixDiv;
              end

              if (result < alpha)
                 similar(yy+offset,xx+offset,2) = 1; % Average gray value is similar
to pixel
              else
```

```
                similar(yy+offset,xx+offset,2) = 0; % Neighbor pixel belongs to
different object
            end

        end
    end

    numerator = double(0);
    denom = double(0);

    %%% Apply fuzzy filtering %%%
    for yy = -Ns+topReflect:Ns+bottomReflect
        for xx = -Ns+leftReflect:Ns+rightReflect

            if (similar(yy+offset,xx+offset,1)) % Neighbor pixel only contributes if
similar
                gaussDist = exp( -( (double(refImage(y+yy,x+xx,z)) -
double(image(y,x,z)))^2 ) / (2*sigmaf(z)*sigmaf(z)) );

                numerator = numerator +
double(refImage(y+yy,x+xx,z))*gaussDist;

                denom = denom + gaussDist;
            end

            if (similar(yy+offset,xx+offset,2)) % Neighbor pixel only contributes if
similar
                gaussDist = exp( -( (double(image(y+yy,x+xx,z)) -
double(image(y,x,z)))^2 ) / (2*sigmaf(z)*sigmaf(z)) );

                numerator = numerator + double(image(y+yy,x+xx,z))*gaussDist;

                denom = denom + gaussDist;
            end

        end
    end

    cleanedFrame(y,x,z) = uint8(numerator/denom);

  end
 end
end
```

```
toc;

%% Show Results
% Calculate PSR results
PSNRval1 = PSNR(imread(inputImage),image);
fprintf(['\n\nNoisy Image PSNR = ' num2str(PSNRval1) 'dB']);
PSNRval2 = PSNR(imread(inputImage),cleanedFrame);
fprintf(['\n\nFiltered Image PSNR = ' num2str(PSNRval2) 'dB\n\n']);

difference = PSNRval2-PSNRval1 % Print difference

% Show reference frame, current frame, and filtered frame
figure(figureNum)
set(gcf, 'Name', 'FuzzyMeClean Results');

subplot(1,3,1)
imshow(imread(inputImage));
title('Clean Frame');

subplot(1,3,2)
imshow(image);
title('Current Frame');

subplot(1,3,3);
imshow(uint8(cleanedFrame));
title('Cleaned Frame');
```

Matlab code to implement a modified version of [18] using a symmetric similarity calculation and adaptive average filtering:

```
function difference = fuzzyMeCleanIntMean(referenceImage, inputImage,
noiseEnable, noiseValue, figureNum)
% function fuzzyMeCleanIntMean(referenceImage, inputImage, noiseEnable,
noiseValue, figureNum)
%
% This function implements the video noise removal from:
% "Adaptive Fuzzy Filter Algorithm for Real-time Video Denoising"
%
% The process goes: Noise Estimation -> Characteristic Extraction -> Motion
% Detection -> Similarity Measurement -> Adaptive Mean Filtering
%
% The algorithm is altered to use a symmetrical similarity measurement and
% uses an adaptive averaging filter instead of the fuzzy filter
%
% Inputs:
%   referenceImage - Filename for reference image, frame(t-1)
%
%   inputImage - Filename for input image to denoise, frame(t)
%
%   noiseEnable - Enable for adding Gaussian noise to image
%
%   noiseValue - Defines variance of gaussian noise to be added to input
%
%
% Outputs:
%   Figure showing the noiseless image, noisy image, and corrected image

%% Initialize
refImage = imread(referenceImage);
origImage = imread(inputImage);

if noiseEnable
   image = imnoise(origImage, 'gaussian', 0, noiseValue); % Add zero mean
Gaussian noise to image
   refImage = imnoise(refImage, 'gaussian', 0, noiseValue);
else
   image = origImage;
end
```

```
[Y X Z] = size(origImage);

Ns = 1; % floor(halfSpatialWindow). Experimentally determined spatial window
size of 3
Nt = 2; % Temporal window size


%% Noise Estimation
fprintf('\nNoise Estimation: ');
tic;
% Define Laplacian smooth operator
laplacian = [1 -2 1; -2 4 -2; 1 -2 1]; % Smoothes image

% Preallocate noise information matrix
noiseInfo = int64(zeros(Y,X,Z));


% Apply laplacian
for y = 1:Y
   for x = 1:X
      for z = 1:Z

         leftReflect = 0;
         rightReflect = 0;
         topReflect = 0;
         bottomReflect = 0;

         % Adjust neighbor calc for edges. If edge, limit window size
         if ( x == 1 ) % Left edge
            leftReflect = 1;
         end

         if ( x == X ) % Right edge
            rightReflect = -1;
         end

         if ( y == 1 ) % Top edge
            topReflect = 1;
         end

         if ( y == Y ) % Bottom edge
            bottomReflect = -1;
         end
```

```
            % Add weighted pixels in Laplacian window; Set pixels outside
            % of image to zero by limiting window size at edges
            for yy = -1+topReflect:1+bottomReflect
                for xx = -1+leftReflect:1+rightReflect
                    noiseInfo(y,x,z) = noiseInfo(y,x,z) +
int64(laplacian(yy+2,xx+2))*int64(image(y+yy,x+xx,z));
                end
            end

        end
    end
end


% Calculate standard deviation of noise
noiseInfoTotal = int64(zeros(1,3));

for z = 1:Z
    for y = 2:Y-2
        for x = 2:X-2
            noiseInfoTotal(z) = noiseInfoTotal(z) + noiseInfo(y,x,z)*noiseInfo(y,x,z);
        end
    end
end

for z = 1:Z
    noiseSTD(z) = sqrt( double(noiseInfoTotal(z)) / ( 36*(X-2)*(Y-2) ) );
end

fprintf([num2str(noiseSTD) ' stdev\n']);
toc;

Ns=1; % Set spatial window to 3x3, but use noiseSTD for other steps


%% Characteristic Extraction
% Blur both input images for use in Motion Detection Step. This step is
% modified compared to real-time program because the reference blur image
% would already exist.
% Paper uses blur window of size 3x3.
fprintf('\nBlur time:\n');
tic;
```

```
blurImage = zeros(Y,X,Z);
blurImagePrev = zeros(Y,X,Z);

% Calculate blur
for y = 1:Y
   for x = 1:X
      for z = 1:Z

         blurWinTotal = uint16(0); % Must be large enough to hold window
summation
         blurWinTotalPrev = uint16(0);

         leftReflect = 0;
         rightReflect = 0;
         topReflect = 0;
         bottomReflect = 0;

         % Adjust neighbor calc for edges. If edge, limit window size
         if ( x <= Ns ) % Left edge
            leftReflect = 1;
         end

         if ( x > X-Ns ) % Right edge
            rightReflect = -1;
         end

         if ( y <= Ns ) % Top edge
            topReflect = 1;
         end

         if ( y > Y-Ns ) % Bottom edge
            bottomReflect = -1;
         end

         % Calulate total pixel values in window
         for yy = -1+topReflect:1+bottomReflect
            for xx = -1+leftReflect:1+rightReflect
               blurWinTotal = blurWinTotal + uint16(image(y+yy,x+xx,z));
               blurWinTotalPrev = blurWinTotalPrev +
uint16(refImage(y+yy,x+xx,z));
            end
         end
```

```
            % Calculate average. Slight inaccuracy at edges from dividing
            % by number greater than number of summed pixels
            blurImage(y,x,z) = blurWinTotal/9;
            blurImagePrev(y,x,z) = blurWinTotalPrev/9;

        end
    end
end

blurImage = uint8(blurImage); % Convert from uint16 to uint8 for imshow()
blurImagePrev = uint8(blurImagePrev);
toc;

%% Motion Detection
% Motion detection threshold noise relationship determined experimentally by
paper
fprintf('\nMotion Detection time:\n');
tic;

motionThreshold = 3.*noiseSTD; % Can probably set noiseSTD to 1.5 for Mintron
motion = zeros(Y,X,Z); % Holds the detected motion between two frames

for y = 1:Y
    for x = 1:X
        for z = 1:Z

            if ( abs( int16(blurImage(y,x,z)) - int16(blurImagePrev(y,x,z)) ) >
motionThreshold(z))
                motion(y,x,z) = 1;
            else
                motion(y,x,z) = 0;
            end

        end
    end
end
toc;

%% Similarity Measurement & Adaptive Average Filtering
% alpha is set to 0.1 from paper

fprintf('\nSimilarity Measurement & Adaptive Average Filtering time:\n');
tic;
```

```
alpha = 26; %int32(0.1*256); % Usually set to 0.1, I'm scaling by 256 (2^8)
sigmaf = noiseSTD; % Spread parameter. Larger -> more smoothing

cleanedFrame = zeros(Y,X,Z); % Initialize filtered frame

offset = Ns + 1; % Offset used for indexing

% Set Average filter mask based on noise level. Less aggressive for less
% noise. A value of 10 was chosen from testing
if (noiseSTD(1) < 10)
   mask = uint32([0 1 0; 1 2 1; 0 1 0]);
else
   mask = uint32([0 1 0; 1 1 1; 0 1 0]);
end


for z = 1:Z
   for y = 1:Y
      for x = 1:X

         pixel = blurImage(y,x,z);

         leftReflect = 0;
         rightReflect = 0;
         topReflect = 0;
         bottomReflect = 0;

         % Adjust neighbor calc for edges. If edge, limit window size
         if ( x <= Ns ) % Left edge
            leftReflect = Ns-x+1;
         end

         if ( x > X-Ns ) % Right edge
            rightReflect = X-x-Ns;
         end

         if ( y <= Ns ) % Top edge
            topReflect = Ns-y+1;
         end

         if ( y > Y-Ns ) % Bottom edge
            bottomReflect = Y-y-Ns;
```

```
        end


        %%% Check similarity of surrounding pixels %%%

        similar = zeros(3,3,2); % Initialize filter window similarity check

        for yy = -Ns+topReflect:Ns+bottomReflect
           for xx = -Ns+leftReflect:Ns+rightReflect

              % Check previous frame first if no motion
              if (motion(y,x,z) == 0)

                 if ( blurImagePrev(y+yy,x+xx,z) > pixel )
                    pixDiv =
bitshift(uint32(pixel),8)/uint32(blurImagePrev(y+yy,x+xx,z));
                 else
                    pixDiv =
bitshift(uint32(blurImagePrev(y+yy,x+xx,z)),8)/uint32(pixel);
                 end

                 result = 256 - pixDiv;

                 if (result < alpha)
                    similar(yy+offset,xx+offset,1) = 1; % Average gray value is
similar to pixel
                 else
                    similar(yy+offset,xx+offset,1) = 0; % Neighbor pixel belongs to
different object
                 end
              end

              % Check current frame
              if ( blurImage(y+yy,x+xx,z) > pixel )
                 pixDiv = bitshift(uint32(pixel),8)/uint32(blurImage(y+yy,x+xx,z));
              else
                 pixDiv = bitshift(uint32(blurImage(y+yy,x+xx,z)),8)/uint32(pixel);
              end

              result = 256 - pixDiv;


              if (result < alpha)
```

```
                similar(yy+offset,xx+offset,2) = 1; % Average gray value is similar
to pixel
            else
                similar(yy+offset,xx+offset,2) = 0; % Neighbor pixel belongs to
different object
            end

        end
    end

    numerator = uint32(0);
    denom = uint32(0);

    %%% Apply average filtering %%%
    for yy = -Ns+topReflect:Ns+bottomReflect
        for xx = -Ns+leftReflect:Ns+rightReflect

            if (similar(yy+offset,xx+offset,1)) % Neighbor pixel only contributes if
similar
                numerator = numerator +
uint32(refImage(y+yy,x+xx,z))*mask(yy+offset,xx+offset);
                denom = denom + mask(yy+offset,xx+offset);
            end

            if (similar(yy+offset,xx+offset,2)) % Neighbor pixel only contributes if
similar
                numerator = numerator +
uint32(image(y+yy,x+xx,z))*mask(yy+offset,xx+offset);
                denom = denom + mask(yy+offset,xx+offset);
            end

        end
    end

    cleanedFrame(y,x,z) = uint8(numerator/denom);

  end
 end
end
toc;


%% Show Results
```

```
% Calculate PSR results
PSNRval1 = PSNR(imread(inputImage), image);
fprintf(['\n\nNoisy Image PSNR = ' num2str(PSNRval1) 'dB']);
PSNRval2 = PSNR(imread(inputImage), cleanedFrame);
fprintf(['\n\nFiltered Image PSNR = ' num2str(PSNRval2) 'dB\n\n']);

difference = PSNRval2-PSNRval1 % Print difference

% Show reference frame, current frame, and filtered frame
figure(figureNum)
set(gcf, 'Name', 'FuzzyMeCleanIntMean Results');

subplot(1,3,1)
imshow(imread(inputImage));
title('Clean Frame');

subplot(1,3,2)
imshow(image);
title('Current Frame');

subplot(1,3,3);
imshow(uint8(cleanedFrame));
title('Cleaned Frame');
```

Matlab code to calculate Peak signal-to-noise ratio in decibels:

```
function dBval = PSNR(noiseFreeImage, noisyImage)
% function dBval = PSNR(noiseFreeImage, noisyImage)
% Inputs - two images to compare (noisy versus noiseFree)
% Output - Peak signal to noise ratio in decibels
%
% Assumes 8-bit Black/White or Color image

[M N D] = size(noiseFreeImage);

% Calculate Mean Square Error
diff = (double(noiseFreeImage) - double(noisyImage)).^2;
MSE = sum(sum(sum(diff)))/(M*N);

% Calculate PSNR in dB, assuming max value in image is 255
dBval = 20*log10(255) - 10*log10(MSE);
```

APPENDIX B: OPEN CV

OpenCV implementation of the Matlab function “fuzzyMeCleanIntMean”:

```
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv\cv.h>
#include <iostream>
#include <math.h>

using namespace cv;
using namespace std;

#define SPATIAL_WINDOW_SIZE 3
#define Ns SPATIAL_WINDOW_SIZE/2
#define NOISE_STDEV 1.5
#define ALPHA 26 // 0.1 times 256
#define SIGMAF NOISE_STDEV

int main( int argc, char** argv )
{
 double t;

/********** Initialize **********/
        cout << "Noise Standard Deviation = " << NOISE_STDEV << endl;
        cout << endl << "Initialize..." << endl << endl;
        t = (double)getTickCount();


        char* ref = argv[1];  // Previous Frame
        char* orig = argv[2];          // Current Frame

        Mat refImage = imread( ref, 1 );
        Mat image = imread( orig, 1 );

        if( argc != 3 || !image.data || !refImage.data )
        {
                cout << " No image data" << endl;
                return -1;
        }

        int channels = image.channels();          // gray = 1, color = 3
```

```
	int width = image.cols * channels;	// Total columns. One image
column has BGR sub-columns
	int height = image.rows;


	// Precalculate all gaussian distances for Fuzzy Filtering
	unsigned int gaussDistLUT[256];

	for (int g = 0; g < 256; g++) {
		gaussDistLUT[g] = unsigned(exp( -1*( pow( float(g) ,2 )) /
(2*SIGMAF*SIGMAF) ) * 256);
	}


	t = ((double)getTickCount() - t)/getTickFrequency();
	cout << "Time passed in seconds: " << t << endl << endl;


/** Noise Estimation step completed in Matlab for speed **/

/********** Characteristic Extraction **********/
	cout << "Characteristic Extraction..." << endl << endl;
	t = (double)getTickCount();

	Mat blurImagePrev;
	blur(refImage, blurImagePrev, Size(3,3), Point(-1,-1),
BORDER_REFLECT_101);

	Mat blurImage;
	blur(image, blurImage, Size(3,3), Point(-1,-1), BORDER_REFLECT_101);

	t = ((double)getTickCount() - t)/getTickFrequency();
	cout << "Time passed in seconds: " << t << endl << endl;


/********** Motion Detection **********/
	cout << "Motion Detection..." << endl << endl;
	t = (double)getTickCount();

	double motionThreshold = 3*NOISE_STDEV;
	Mat motion;
```

```
		//Mat diff = abs( blurImage.convertTo(blurImage,CV_32F) - blurImagePrev.convertTo(blurImagePrev,CV_32F) );

		Mat diff;
		absdiff(blurImage, blurImagePrev, diff);

		compare(diff, motionThreshold, motion, CMP_GT);

		t = ((double)getTickCount() - t)/getTickFrequency();
		cout << "Time passed in seconds: " << t << endl << endl;


/********** Similarity Measurement & Adaptive Fuzzy Filtering **********/
		cout << "Similarity Measurement & Adaptive Fuzzy Filtering..." << endl << endl;
		t = (double)getTickCount();

		int leftReflect, rightReflect, topReflect, bottomReflect;
		unsigned int pixDiv, result, numerator, denom, gaussDist, blurPrevNeighbor, blurNeighbor, blurPixel;
		int offset = Ns; // Offset for indexing

		int avgMask[3][3] = { {0,1,0} , {1,2,1} , {0,1,0} }; // Less blurring for lower noise

		if (NOISE_STDEV > 10) {
			avgMask[1][1] = 1;
		}

		Mat similar = Mat(3, 3, CV_8UC3, Scalar(0)); // Initialize simlilar matrix (Can hold up to three frames, only use two though)

		int sz2[] = {image.rows, image.cols, image.channels()};
		Mat cleanedFrame = Mat(image.rows, image.cols, image.type(), Scalar(0)); // Initialize cleaned frame matrix

		//Mat gaussDistMAT = Mat(3, 3, CV_8UC3, Scalar(0)); // Initialize gaussian distance matrix

		int simRows = similar.rows;
		int simCols = similar.cols;

		for (int z = 0; z < channels; z++) {
```

```
                for (int r = 0; r < image.rows; r++) {
                        for (int c = 0; c < image.cols; c++) {

                                topReflect = 0;
                                bottomReflect = 0;
                                leftReflect = 0;
                                rightReflect = 0;

                                // Adjust neighbor calc for edges. If edge, limit window
size
                                if ( r == 0) { topReflect = 1; }
        // Top edge
                                if ( r == image.rows-1) { bottomReflect = -1; } //
Bottom edge
                                if ( c == 0) { leftReflect = 1; }
        // Left edge
                                if ( c == image.cols-1 ) { rightReflect = -1; }       //
Right edge


                                /***** Check similarity of surrounding pixels *****/
                                similar = Mat::zeros(3, 3, CV_8UC3);

                                for (int yy = -Ns+topReflect; yy <= Ns+bottomReflect;
yy++) {
                                        for (int xx = -Ns+leftReflect; xx <=
Ns+rightReflect; xx++) {

                                                // Check previous frame if no motion
                                                if ( motion.at<Vec3b>(r,c)[z] == 0) {

                                                        blurPixel =
unsigned(blurImage.at<Vec3b>(r,c)[z]);
                                                        blurPrevNeighbor =
unsigned(blurImagePrev.at<Vec3b>(r+yy,c+xx)[z]);

                                                        if (blurPrevNeighbor > blurPixel) {
                                                                pixDiv = (blurPixel << 8
)/blurPrevNeighbor;
                                                        }
                                                        else {
                                                                pixDiv = (blurPrevNeighbor
<< 8 )/blurPixel;
```

```
                                                }

                                                result = 256 - pixDiv;

                                                if (result < ALPHA) {

        similar.at<Vec3b>(yy+offset,xx+offset)[1] = 1; // Average gray value us
similar to pixel
                                                }
                                                else {

        similar.at<Vec3b>(yy+offset,xx+offset)[1] = 0; // Neighbor pixel belongs to
different object
                                                }

                                        }
                                        else {

        similar.at<Vec3b>(yy+offset,xx+offset)[1] = 0;
                                        }

                                        // Check current frame
                                                blurNeighbor =
unsigned(blurImage.at<Vec3b>(r+yy,c+xx)[z]);
                                                blurPixel =
unsigned(blurImage.at<Vec3b>(r,c)[z]);

                                                if (blurNeighbor > blurPixel) {
                                                        pixDiv = (blurPixel << 8
)/blurNeighbor;
                                                }
                                                else {
                                                        pixDiv = (blurNeighbor <<
8 )/blurPixel;
                                                }

                                                result = 256 - pixDiv;

                                                if (result < ALPHA) {

        similar.at<Vec3b>(yy+offset,xx+offset)[2] = 1; // Average gray value us
similar to pixel
                                                }
```

```
                                                    else {

        similar.at<Vec3b>(yy+offset,xx+offset)[2] = 0; // Neighbor pixel belongs to
different object
                                                    }
                                            }// for xx
                                    }// for yy


                                    /***** Apply Adaptive Average Filtering *****/
                                    numerator = 0;
                                    denom = 0;

                                    for (int yy = -Ns+topReflect; yy <= Ns+bottomReflect;
yy++) {
                                            for (int xx = -Ns+leftReflect; xx <=
Ns+rightReflect; xx++) {

                                                    if (
similar.at<Vec3b>(yy+offset,xx+offset)[1] ) { // Neighbor pixels only contributes if
similar
                                                    //gaussDist = exp( -1*( pow(
float(refImage.at<Vec3b>(r+yy,c+xx)[z]) - float(image.at<Vec3b>(r,c)[z]) ,2 )) /
(2*SIGMAF*SIGMAF) );
                                                    //gaussDist = gaussDistLUT[
abs(refImage.at<Vec3b>(r+yy,c+xx)[z] - image.at<Vec3b>(r,c)[z]) ];

                                                    numerator = numerator +
unsigned(refImage.at<Vec3b>(r+yy,c+xx)[z])*avgMask[yy+offset][xx+offset];

                                                    denom = denom +
avgMask[yy+offset][xx+offset];
                                                    }

                                                    if (
similar.at<Vec3b>(yy+offset,xx+offset)[2] ) { // Neighbor pixels only contributes if
similar
                                                    //gaussDist = exp( -1*( pow(
float(image.at<Vec3b>(r+yy,c+xx)[z]) - float(image.at<Vec3b>(r,c)[z]) ,2 )) /
(2*SIGMAF*SIGMAF) );
                                                    //gaussDist = gaussDistLUT[
abs(image.at<Vec3b>(r+yy,c+xx)[z] - image.at<Vec3b>(r,c)[z]) ];
```

```
                                                    numerator = numerator +
unsigned(image.at<Vec3b>(r+yy,c+xx)[z])*avgMask[yy+offset][xx+offset];

                                                    denom = denom +
avgMask[yy+offset][xx+offset];
                                                }

                                            }// for xx
                                        }// for yy

                                        cleanedFrame.at<Vec3b>(r,c)[z] =
uchar(numerator/denom);

                                    }// forc
                                }// for r
                            }// for z



                            t = ((double)getTickCount() - t)/getTickFrequency();
                            cout << "Time passed in seconds: " << t << endl << endl;


                            /********** Show Results **********/
                            cout << "Show Results..." << endl << endl;
                            t = (double)getTickCount();

                            namedWindow( "Orig Image", CV_WINDOW_AUTOSIZE );
                            imshow("Orig Image", image);

                            namedWindow( "Cleaned Frame", CV_WINDOW_AUTOSIZE );
                            imshow("Cleaned Frame", cleanedFrame);

                            namedWindow( "Motion Frame", CV_WINDOW_AUTOSIZE );
                            imshow("Motion Frame", motion);

                            //imwrite("origImage.jpg", image);
                            //imwrite("blurredOrig.jpg", blurImage);
                            //imwrite("motion.jpg", motion);
                            //imwrite("cleanedFrame.jpg", cleanedFrame);

                            t = ((double)getTickCount() - t)/getTickFrequency();
                            cout << "Time passed in seconds: " << t << endl << endl;
```

```
        waitKey(0);

        return 0;
}
```

OpenCV camera calibration code to be used with provided XML file:

```
#include <iostream>
#include <sstream>
#include <time.h>
#include <stdio.h>

#include <opencv2/core/core.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/calib3d/calib3d.hpp>
#include <opencv2/highgui/highgui.hpp>

using namespace cv;
using namespace std;

static void help()
{
   cout <<  "This is a camera calibration sample." << endl
        <<  "Usage: calibration configurationFile"  << endl
        <<  "Near the sample file you'll find the configuration file, which has detailed
help of "
            "how to edit it.  It may be any OpenCV supported file format XML/YAML."
<< endl;
}
class Settings
{
public:
   Settings() : goodInput(false) {}
   enum Pattern { NOT_EXISTING, CHESSBOARD, CIRCLES_GRID,
ASYMMETRIC_CIRCLES_GRID };
   enum InputType {INVALID, CAMERA, VIDEO_FILE, IMAGE_LIST};

   void write(FileStorage& fs) const                        //Write serialization for this
class
   {
      fs << "{" << "BoardSize_Width"  << boardSize.width
               << "BoardSize_Height" << boardSize.height
               << "Square_Size"         << squareSize
               << "Calibrate_Pattern" << patternToUse
               << "Calibrate_NrOfFrameToUse" << nrFrames
               << "Calibrate_FixAspectRatio" << aspectRatio
               << "Calibrate_AssumeZeroTangentialDistortion" <<
calibZeroTangentDist
```

```
            << "Calibrate_FixPrincipalPointAtTheCenter" << calibFixPrincipalPoint

            << "Write_DetectedFeaturePoints" << bwritePoints
            << "Write_extrinsicParameters"   << bwriteExtrinsics
            << "Write_outputFileName"  << outputFileName

            << "Show_UndistortedImage" << showUndistorsed

            << "Input_FlipAroundHorizontalAxis" << flipVertical
            << "Input_Delay" << delay
            << "Input" << input
        << "}";
    }
    void read(const FileNode& node)                          //Read serialization for this
class
    {
        node["BoardSize_Width" ] >> boardSize.width;
        node["BoardSize_Height"] >> boardSize.height;
        node["Calibrate_Pattern"] >> patternToUse;
        node["Square_Size"]  >> squareSize;
        node["Calibrate_NrOfFrameToUse"] >> nrFrames;
        node["Calibrate_FixAspectRatio"] >> aspectRatio;
        node["Write_DetectedFeaturePoints"] >> bwritePoints;
        node["Write_extrinsicParameters"] >> bwriteExtrinsics;
        node["Write_outputFileName"] >> outputFileName;
        node["Calibrate_AssumeZeroTangentialDistortion"] >>
calibZeroTangentDist;
        node["Calibrate_FixPrincipalPointAtTheCenter"] >> calibFixPrincipalPoint;
        node["Input_FlipAroundHorizontalAxis"] >> flipVertical;
        node["Show_UndistortedImage"] >> showUndistorsed;
        node["Input"] >> input;
        node["Input_Delay"] >> delay;
        interprate();
    }
    void interprate()
    {
        goodInput = true;
        if (boardSize.width <= 0 || boardSize.height <= 0)
        {
            cerr << "Invalid Board size: " << boardSize.width << " " <<
boardSize.height << endl;
            goodInput = false;
        }
```

```
        if (squareSize <= 10e-6)
        {
            cerr << "Invalid square size " << squareSize << endl;
            goodInput = false;
        }
        if (nrFrames <= 0)
        {
            cerr << "Invalid number of frames " << nrFrames << endl;
            goodInput = false;
        }

        if (input.empty())      // Check for valid input
                inputType = INVALID;
        else
        {
            if (input[0] >= '0' && input[0] <= '9')
            {
                stringstream ss(input);
                ss >> cameraID;
                inputType = CAMERA;
            }
            else
            {
                if (readStringList(input, imageList))
                    {
                        inputType = IMAGE_LIST;
                        nrFrames = (nrFrames < (int)imageList.size()) ? nrFrames :
(int)imageList.size();
                    }
                else
                    inputType = VIDEO_FILE;
            }
            if (inputType == CAMERA)
                inputCapture.open(cameraID);
            if (inputType == VIDEO_FILE)
                inputCapture.open(input);
            if (inputType != IMAGE_LIST && !inputCapture.isOpened())
                    inputType = INVALID;
        }
        if (inputType == INVALID)
        {
            cerr << " Inexistent input: " << input;
            goodInput = false;
```

```
        }

        flag = 0;
        if(calibFixPrincipalPoint) flag |= CV_CALIB_FIX_PRINCIPAL_POINT;
        if(calibZeroTangentDist)   flag |= CV_CALIB_ZERO_TANGENT_DIST;
        if(aspectRatio)            flag |= CV_CALIB_FIX_ASPECT_RATIO;


        calibrationPattern = NOT_EXISTING;
        if (!patternToUse.compare("CHESSBOARD")) calibrationPattern =
CHESSBOARD;
        if (!patternToUse.compare("CIRCLES_GRID")) calibrationPattern =
CIRCLES_GRID;
        if (!patternToUse.compare("ASYMMETRIC_CIRCLES_GRID"))
calibrationPattern = ASYMMETRIC_CIRCLES_GRID;
        if (calibrationPattern == NOT_EXISTING)
           {
              cerr << " Inexistent camera calibration mode: " << patternToUse <<
endl;
              goodInput = false;
           }
        atImageList = 0;

    }
    Mat nextImage()
    {
        Mat result;
        if( inputCapture.isOpened() )
        {
            Mat view0;
            inputCapture >> view0;
            view0.copyTo(result);
        }
        else if( atImageList < (int)imageList.size() )
            result = imread(imageList[atImageList++], CV_LOAD_IMAGE_COLOR);

        return result;
    }

    static bool readStringList( const string& filename, vector<string>& l )
    {
        l.clear();
        FileStorage fs(filename, FileStorage::READ);
```

```
        if( !fs.isOpened() )
            return false;
        FileNode n = fs.getFirstTopLevelNode();
        if( n.type() != FileNode::SEQ )
            return false;
        FileNodeIterator it = n.begin(), it_end = n.end();
        for( ; it != it_end; ++it )
            l.push_back((string)*it);
        return true;
    }
public:
    Size boardSize;            // The size of the board -> Number of items by width
and height
    Pattern calibrationPattern;// One of the Chessboard, circles, or asymmetric
circle pattern
    float squareSize;          // The size of a square in your defined unit (point,
millimeter,etc).
    int nrFrames;              // The number of frames to use from the input for
calibration
    float aspectRatio;         // The aspect ratio
    int delay;                 // In case of a video input
    bool bwritePoints;         //  Write detected feature points
    bool bwriteExtrinsics;     // Write extrinsic parameters
    bool calibZeroTangentDist; // Assume zero tangential distortion
    bool calibFixPrincipalPoint;// Fix the principal point at the center
    bool flipVertical;         // Flip the captured images around the horizontal axis
    string outputFileName;     // The name of the file where to write
    bool showUndistorsed;      // Show undistorted images after calibration
    string input;              // The input ->



    int cameraID;
    vector<string> imageList;
    int atImageList;
    VideoCapture inputCapture;
    InputType inputType;
    bool goodInput;
    int flag;

private:
    string patternToUse;
```

```
};

static void read(const FileNode& node, Settings& x, const Settings&
default_value = Settings())
{
    if(node.empty())
        x = default_value;
    else
        x.read(node);
}

enum { DETECTION = 0, CAPTURING = 1, CALIBRATED = 2 };

bool runCalibrationAndSave(Settings& s, Size imageSize, Mat&  cameraMatrix,
Mat& distCoeffs,
                           vector<vector<Point2f> > imagePoints );

int main(int argc, char* argv[])
{
    help();

        //---------------------- Read input XML file -------------------------
    Settings s;
    const string inputSettingsFile = argc > 1 ? argv[1] : "default.xml";
    FileStorage fs(inputSettingsFile, FileStorage::READ); // Read the settings
    if (!fs.isOpened())
    {
        cout << "Could not open the configuration file: \"" << inputSettingsFile << "\""
<< endl;
        return -1;
    }
    fs["Settings"] >> s;
    fs.release();                                         // close Settings file

    if (!s.goodInput)
    {
        cout << "Invalid input detected. Application stopping. " << endl;
        return -1;
    }

        //---------------------- Initalize Variables --------------------------
    vector<vector<Point2f> > imagePoints;
```

```
    Mat cameraMatrix, distCoeffs;
    Size imageSize;
    int mode = s.inputType == Settings::IMAGE_LIST ? CAPTURING :
DETECTION; //if already have images then capture, otherwise detect
    clock_t prevTimestamp = 0;
    const Scalar RED(0,0,255), GREEN(0,255,0);
    const char ESC_KEY = 27;

          // ----------------------  ----------------------------
    for(int i = 0;;++i)
    {
      Mat view;
      bool blinkOutput = false;

      view = s.nextImage(); // Grab next image in list or grab current frame from
video (file or camera)

      //-----  If no more image, or got enough, then stop calibration and show result
-------------
      if( mode == CAPTURING && imagePoints.size() >= (unsigned)s.nrFrames )
      {
         if( runCalibrationAndSave(s, imageSize,  cameraMatrix, distCoeffs,
imagePoints))
            mode = CALIBRATED;
         else
            mode = DETECTION;
      }
      if(view.empty())          // If no more images then run calibration, save and stop
loop.
      {
           if( imagePoints.size() > 0 )
              runCalibrationAndSave(s, imageSize,  cameraMatrix, distCoeffs,
imagePoints);
           break;
      }


        imageSize = view.size();  // Format input image.
        if( s.flipVertical )    flip( view, view, 0 ); // Specified in input XML file

        vector<Point2f> pointBuf;

        bool found;
```

```
        switch( s.calibrationPattern ) // Find feature points on the input format
        {
        case Settings::CHESSBOARD:
            found = findChessboardCorners( view, s.boardSize, pointBuf,
                CV_CALIB_CB_ADAPTIVE_THRESH |
CV_CALIB_CB_FAST_CHECK | CV_CALIB_CB_NORMALIZE_IMAGE);
            break;
        case Settings::CIRCLES_GRID:
            found = findCirclesGrid( view, s.boardSize, pointBuf );
            break;
        case Settings::ASYMMETRIC_CIRCLES_GRID:
            found = findCirclesGrid( view, s.boardSize, pointBuf,
CALIB_CB_ASYMMETRIC_GRID );
            break;
        default:
            found = false;
            break;
        }

        if ( found)                // If done with success,
        {
              // improve the found corners' coordinate accuracy for chessboard
              if( s.calibrationPattern == Settings::CHESSBOARD)
              {
                  Mat viewGray;
                  cvtColor(view, viewGray, CV_BGR2GRAY);
                  cornerSubPix( viewGray, pointBuf, Size(11,11),
                      Size(-1,-1), TermCriteria(
CV_TERMCRIT_EPS+CV_TERMCRIT_ITER, 30, 0.1 ));
              }

              if( mode == CAPTURING &&  // For camera only take new samples
after delay time
                  (!s.inputCapture.isOpened() || clock() - prevTimestamp > s.delay*1e-
3*CLOCKS_PER_SEC) )
              {
                  imagePoints.push_back(pointBuf);
                  prevTimestamp = clock();
                  blinkOutput = s.inputCapture.isOpened();
              }

              // Draw the corners.
              drawChessboardCorners( view, s.boardSize, Mat(pointBuf), found );
```

```
        }

        //---------------------------- Output Text ------------------------------------------------
        string msg = (mode == CAPTURING) ? "100/100" :
                     mode == CALIBRATED ? "Calibrated" : "Press 'g' to start";
        int baseLine = 0;
        Size textSize = getTextSize(msg, 1, 1, 1, &baseLine);
        Point textOrigin(view.cols - 2*textSize.width - 10, view.rows - 2*baseLine -
10); //place text in bottom right corner

        if( mode == CAPTURING )
        {
            if(s.showUndistorsed) // show undistorted images if true in input XML file
                msg = format( "%d/%d Undist", (int)imagePoints.size(), s.nrFrames );
            else
                msg = format( "%d/%d", (int)imagePoints.size(), s.nrFrames );
        }

        putText( view, msg, textOrigin, 1, 1, mode == CALIBRATED ?  GREEN :
RED);

        if( blinkOutput )
            bitwise_not(view, view);

        //------------------------- Video capture  output  undistorted ----------------------------
--
        if( mode == CALIBRATED && s.showUndistorsed )
        {
            Mat temp = view.clone();
            undistort(temp, view, cameraMatrix, distCoeffs);
        }

        //------------------------------ Show image and check for input commands -------------------
---------
        imshow("Image View", view);
        char key = (char)waitKey(s.inputCapture.isOpened() ? 50 : s.delay); //33ms
for 30 frame/s, delay is for showing undistorted pictures

        if( key  == ESC_KEY )
            break;

        if( key == 'u' && mode == CALIBRATED )
           s.showUndistorsed = !s.showUndistorsed;
```

```
        if( s.inputCapture.isOpened() && key == 'g' )
        {
            mode = CAPTURING;
            imagePoints.clear();
        }
    }

    // ----------------------Show the undistorted image for the image list ------------------
------
    if( s.inputType == Settings::IMAGE_LIST && s.showUndistorsed )
    {
        Mat view, rview, map1, map2;
        initUndistortRectifyMap(cameraMatrix, distCoeffs, Mat(),
            getOptimalNewCameraMatrix(cameraMatrix, distCoeffs, imageSize, 1,
imageSize, 0),
            imageSize, CV_16SC2, map1, map2);

        for(int i = 0; i < (int)s.imageList.size(); i++ )
        {
            view = imread(s.imageList[i], 1);
            if(view.empty())
                continue;
            remap(view, rview, map1, map2, INTER_LINEAR);
            imshow("Image View", rview);
            char c = (char)waitKey();
            if( c  == ESC_KEY || c == 'q' || c == 'Q' )
                break;
        }
    }


    return 0;
}

static double computeReprojectionErrors( const vector<vector<Point3f> >&
objectPoints,
                                         const vector<vector<Point2f> >& imagePoints,
                                         const vector<Mat>& rvecs, const vector<Mat>& tvecs,
                                         const Mat& cameraMatrix , const Mat& distCoeffs,
                                         vector<float>& perViewErrors)
{
    vector<Point2f> imagePoints2;
```

```
    int i, totalPoints = 0;
    double totalErr = 0, err;
    perViewErrors.resize(objectPoints.size());

    for( i = 0; i < (int)objectPoints.size(); ++i )
    {
        projectPoints( Mat(objectPoints[i]), rvecs[i], tvecs[i], cameraMatrix,
                       distCoeffs, imagePoints2);
        err = norm(Mat(imagePoints[i]), Mat(imagePoints2), CV_L2);

        int n = (int)objectPoints[i].size();
        perViewErrors[i] = (float) std::sqrt(err*err/n);
        totalErr        += err*err;
        totalPoints     += n;
    }

    return std::sqrt(totalErr/totalPoints);
}

static void calcBoardCornerPositions(Size boardSize, float squareSize,
vector<Point3f>& corners,
                                     Settings::Pattern patternType /*=
Settings::CHESSBOARD*/)
{
    corners.clear();

    switch(patternType)
    {
    case Settings::CHESSBOARD:
    case Settings::CIRCLES_GRID:
        for( int i = 0; i < boardSize.height; ++i )
            for( int j = 0; j < boardSize.width; ++j )
                corners.push_back(Point3f(float( j*squareSize ), float( i*squareSize ),
0));
        break;

    case Settings::ASYMMETRIC_CIRCLES_GRID:
        for( int i = 0; i < boardSize.height; i++ )
            for( int j = 0; j < boardSize.width; j++ )
                corners.push_back(Point3f(float((2*j + i % 2)*squareSize),
float(i*squareSize), 0));
        break;
    default:
```

```
        break;
    }
}

static bool runCalibration( Settings& s, Size& imageSize, Mat& cameraMatrix,
Mat& distCoeffs,
                            vector<vector<Point2f> > imagePoints, vector<Mat>& rvecs,
vector<Mat>& tvecs,
                            vector<float>& reprojErrs,  double& totalAvgErr)
{

    cameraMatrix = Mat::eye(3, 3, CV_64F);
    if( s.flag & CV_CALIB_FIX_ASPECT_RATIO )
        cameraMatrix.at<double>(0,0) = 1.0;

    distCoeffs = Mat::zeros(8, 1, CV_64F);

    vector<vector<Point3f> > objectPoints(1);
    calcBoardCornerPositions(s.boardSize, s.squareSize, objectPoints[0],
s.calibrationPattern);

    objectPoints.resize(imagePoints.size(),objectPoints[0]);

    //Find intrinsic and extrinsic camera parameters
    double rms = calibrateCamera(objectPoints, imagePoints, imageSize,
cameraMatrix,
                                 distCoeffs, rvecs, tvecs,
s.flag|CV_CALIB_FIX_K4|CV_CALIB_FIX_K5);

    cout << "Re-projection error reported by calibrateCamera: "<< rms << endl;

    bool ok = checkRange(cameraMatrix) && checkRange(distCoeffs);

    totalAvgErr = computeReprojectionErrors(objectPoints, imagePoints,
                                            rvecs, tvecs, cameraMatrix, distCoeffs, reprojErrs);

    return ok;
}

// Print camera parameters to the output file
static void saveCameraParams( Settings& s, Size& imageSize, Mat&
cameraMatrix, Mat& distCoeffs,
                              const vector<Mat>& rvecs, const vector<Mat>& tvecs,
```

```
                    const vector<float>& reprojErrs, const vector<vector<Point2f>
>& imagePoints,
                    double totalAvgErr )
{
    FileStorage fs( s.outputFileName, FileStorage::WRITE );

    time_t tm;
    time( &tm );
    struct tm *t2 = localtime( &tm );
    char buf[1024];
    strftime( buf, sizeof(buf)-1, "%c", t2 );

    fs << "calibration_Time" << buf;

    if( !rvecs.empty() || !reprojErrs.empty() )
        fs << "nrOfFrames" << (int)std::max(rvecs.size(), reprojErrs.size());
    fs << "image_Width" << imageSize.width;
    fs << "image_Height" << imageSize.height;
    fs << "board_Width" << s.boardSize.width;
    fs << "board_Height" << s.boardSize.height;
    fs << "square_Size" << s.squareSize;

    if( s.flag & CV_CALIB_FIX_ASPECT_RATIO )
        fs << "FixAspectRatio" << s.aspectRatio;

    if( s.flag )
    {
        sprintf( buf, "flags: %s%s%s%s",
            s.flag & CV_CALIB_USE_INTRINSIC_GUESS ? " +use_intrinsic_guess" :
"",
            s.flag & CV_CALIB_FIX_ASPECT_RATIO ? " +fix_aspectRatio" : "",
            s.flag & CV_CALIB_FIX_PRINCIPAL_POINT ? " +fix_principal_point" : "",
            s.flag & CV_CALIB_ZERO_TANGENT_DIST ? " +zero_tangent_dist" : ""
);
        cvWriteComment( *fs, buf, 0 );

    }

    fs << "flagValue" << s.flag;

    fs << "Camera_Matrix" << cameraMatrix;
    fs << "Distortion_Coefficients" << distCoeffs;
```

```
    fs << "Avg_Reprojection_Error" << totalAvgErr;
    if( !reprojErrs.empty() )
        fs << "Per_View_Reprojection_Errors" << Mat(reprojErrs);

    if( !rvecs.empty() && !tvecs.empty() )
    {
        CV_Assert(rvecs[0].type() == tvecs[0].type());
        Mat bigmat((int)rvecs.size(), 6, rvecs[0].type());
        for( int i = 0; i < (int)rvecs.size(); i++ )
        {
            Mat r = bigmat(Range(i, i+1), Range(0,3));
            Mat t = bigmat(Range(i, i+1), Range(3,6));

            CV_Assert(rvecs[i].rows == 3 && rvecs[i].cols == 1);
            CV_Assert(tvecs[i].rows == 3 && tvecs[i].cols == 1);
            //*.t() is MatExpr (not Mat) so we can use assignment operator
            r = rvecs[i].t();
            t = tvecs[i].t();
        }
        cvWriteComment( *fs, "a set of 6-tuples (rotation vector + translation vector)
for each view", 0 );
        fs << "Extrinsic_Parameters" << bigmat;
    }

    if( !imagePoints.empty() )
    {
        Mat imagePtMat((int)imagePoints.size(), (int)imagePoints[0].size(),
CV_32FC2);
        for( int i = 0; i < (int)imagePoints.size(); i++ )
        {
            Mat r = imagePtMat.row(i).reshape(2, imagePtMat.cols);
            Mat imgpti(imagePoints[i]);
            imgpti.copyTo(r);
        }
        fs << "Image_points" << imagePtMat;
    }
}

bool runCalibrationAndSave(Settings& s, Size imageSize, Mat&  cameraMatrix,
Mat& distCoeffs,vector<vector<Point2f> > imagePoints )
{
    vector<Mat> rvecs, tvecs;
    vector<float> reprojErrs;
```

```
    double totalAvgErr = 0;

    bool ok = runCalibration(s,imageSize, cameraMatrix, distCoeffs, imagePoints,
rvecs, tvecs,
                             reprojErrs, totalAvgErr);
    cout << (ok ? "Calibration succeeded" : "Calibration failed")
         << ". avg re projection error = "  << totalAvgErr ;

    if( ok )
        saveCameraParams( s, imageSize, cameraMatrix, distCoeffs, rvecs ,tvecs,
reprojErrs,
                          imagePoints, totalAvgErr);
    return ok;
}
```

XML file for use with OpenCV camera calibration:

```
<?xml version="1.0"?>
<opencv_storage>
<Settings>
  <!-- Number of inner corners per a item row and column. (square, circle) -->
  <BoardSize_Width> 9</BoardSize_Width>
  <BoardSize_Height>6</BoardSize_Height>

  <!-- The size of a square in some user defined metric system (pixel, millimeter)--
>
  <Square_Size>50</Square_Size>

  <!-- The type of input used for camera calibration. One of: CHESSBOARD
CIRCLES_GRID ASYMMETRIC_CIRCLES_GRID -->
  <Calibrate_Pattern>"CHESSBOARD"</Calibrate_Pattern>

  <!-- The input to use for calibration.
                To use an input camera -> give the ID of the camera, like "1"
                To use an input video  -> give the path of the input video, like
"/tmp/x.avi"
                To use an image list   -> give the path to the XML or YAML file
containing the list of the images, like "/tmp/circles_list.xml"
                -->
  <Input>"1"</Input>
  <!--  If true (non-zero) we flip the input images around the horizontal axis.-->
  <Input_FlipAroundHorizontalAxis>0</Input_FlipAroundHorizontalAxis>

  <!-- Time delay between frames in case of camera. -->
  <Input_Delay>100</Input_Delay>

  <!-- How many frames to use, for calibration. -->
  <Calibrate_NrOfFrameToUse>25</Calibrate_NrOfFrameToUse>
  <!-- Consider only fy as a free parameter, the ratio fx/fy stays the same as in the
input cameraMatrix.
            Use or not setting. 0 - False Non-Zero - True-->
  <Calibrate_FixAspectRatio> 1 </Calibrate_FixAspectRatio>
  <!-- If true (non-zero) tangential distortion coefficients  are set to zeros and stay
zero.-->

<Calibrate_AssumeZeroTangentialDistortion>1</Calibrate_AssumeZeroTangenti
alDistortion>
```

```
  <!-- If true (non-zero) the principal point is not changed during the global
optimization.-->
  <Calibrate_FixPrincipalPointAtTheCenter> 1
</Calibrate_FixPrincipalPointAtTheCenter>

  <!-- The name of the output log file. -->
  <Write_outputFileName>"out_camera_data.xml"</Write_outputFileName>
  <!-- If true (non-zero) we write to the output file the feature points.-->
  <Write_DetectedFeaturePoints>1</Write_DetectedFeaturePoints>
  <!-- If true (non-zero) we write to the output file the extrinsic camera
parameters.-->
  <Write_extrinsicParameters>1</Write_extrinsicParameters>
  <!-- If true (non-zero) we show after calibration the undistorted images.-->
  <Show_UndistortedImage>1</Show_UndistortedImage>

</Settings>
</opencv_storage>
```