"The secret to successful analysis, Cardas: whenever possible, reduce matters to a single variable." - Mitth'raw'nuruodo

Outbound Flight
Tymothy Zahn

University of Alberta

An Underwater Six-Camera Array for Monitoring and Position Measurements in SNO+

by

Zachary Petriw

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of

Master of Science

Department of Physics

Fall 2012
Edmonton, Alberta

Abstract

This thesis deals with the creation, calibration and performance of the camera system to be installed in the SNO+ experiment. A system of six cameras in underwater enclosures will be used to monitor the position of the acrylic vessel and its hold-down rope net during the course of the experiment. The system will also be used to triangulate the positions of calibration sources lowered into the detector to an expected accuracy of ± 1.5 cm at a distance of 9 meters. This is an improvement to the previous ± 5 cm accuracy given by the rope manipulator system used to lower calibration sources into the detector. This result will not be subject to the same position-dependent systematic effects that dominated the previous system. The procedure of testing and calibrating the cameras before their installation is explained in this thesis.

Acknowledgments

I'd like to start by thanking my supervisor, Aksel Hallin, for choosing me for this project and for all the assistance and direction he has provided along the way. Thanks also go to the SNO+ collaboration for inviting me into their research and allowing me to make a small contribution to neutrino research. I am glad to have been part of the collaboration and look forward to the success of the SNO+ experiment.

A large thank-you goes to the SNO+ researchers at the University of Alberta who worked alongside me on their own projects, especially Pierre Gorel. Pierre and I worked together on the camera system for nearly two years handling hardware and software, respectively. Much was learned from working alongside him, and he continued to be available to answer questions after having moved on to focus on a different experiment. His help throughout was invaluable.

Thank you to the Ukrainian Shumka Dancers which helped maintain some of my sanity, physical health, and social skills. If I wasn't working on my thesis or sleeping I was most likely at rehearsal with my friends at Shumka, preparing for our next performance. At a certain point in a dancer's life it becomes more difficult to commit enough hours of each week to dance full-time, but I was dedicated to being part of this wonderful group that supports my heritage and creates many friendships and travel opportunities for its members. Though the physics faculty has some nice people, sometimes I just had to get away from all the beards.

I'd also like to thank the members of the University of Alberta Powerlifting Club for expanding my heavy metal collection and for fighting the disease that is poor technique. If you don't squat to full depth you only cheat yourself. I feel this principle applies ubiquitously in life, from back squats to overhead squats.

I also acknowledge the assistance and life coaching offered to me by fellow SNO+ student, Logan Sibley. Thank you for reassuring me that I would probably not fail my thesis and for listening to my endless stories about my weekend.

Changing stride, I'd like to recognize a fellow graduate student, Rhys Chouinard, for making completion of my Master's thesis more difficult with reasons that include but are not limited to: wrapping my desk in packing material, rearranging my desk into a mirror image, vandalizing my desk, sending me terrible Dolan comics, locking me out

of the office, and especially his beard. Another notorious time-waster was the many weekends he subjected me to learning how to do suspension work and engine swaps on Hondas in order to make them go faster (yes, it's possible).

Finally, I'd like to thank my family for their unwavering moral and housing support. My parents' basement was always available no matter how late I came home from university. My parents and grandparents encouraged me throughout my project. I'd also like to specially recognize my grandpa whose life is an inspiration to me, and who showed interest in my work even at the age of 90. Многая літа!

This thesis is dedicated to science. It works.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The SNO+ Detector

The SNO+ detector will make use of most of the equipment from its predecessor, SNO (Sudbudy Neutrino Observatory), which has previously made accurate measurements of the ^{8}B solar neutrino flux [1]. The detector is located in Vale's Creighton mine near Sudbury, Ontario, Canada. It resides at a depth of 1783 m with a rock overburden of 5890 meters water equivalent (mwe) to shield it from cosmic rays and atmospheric muons. A sketch of the detector is shown in Fig. 1.1.

The previous experiment, SNO, consisted of a 12 m diameter acrylic vessel (AV) surrounded by a geodesic stainless steel support structure (PSUP) supporting 9456 highly sensitive, inward-looking photomultiplier tubes (PMTs). The PMTs detect the light produced when high energy particles such as neutrinos pass through the detector and interact with the kilotonne of heavy water contained in the AV. Heavy water, a compound where the hydrogen atom in a water molecule is replaced with a more massive deuteron, was used in SNO to detect the oscillation of solar electron neutrinos into muon and tau neutrinos. In SNO+ this medium is being replaced with liquid organic scintillator to lower the energy threshold of detectable interactions, explained further in Sec. 1.3.

The detector is located in one of the large halls of SNOLAB, an underground research facility in the Creighton mine housing SNO+ and other low background experiments. At its depth the detector has an average of 70 muon interactions per day.

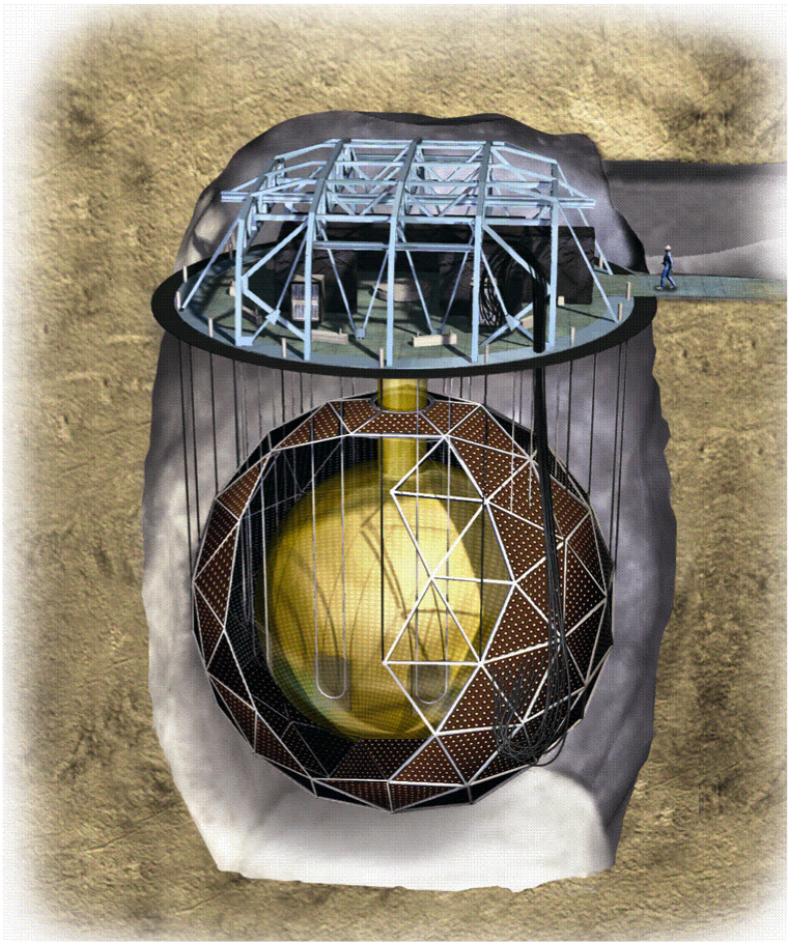


Figure 1.1: A cutaway diagram of the SNO experiment.

The detector medium in the AV is shielded from possible radiation from the PMTs by a 2 m thick buffer (1700 tonnes) of ultra pure water, with another 5300 tonnes of ultrapure water shielding the PSUP from radioactive decay in the walls of the cavity. Radon ingress into the cavity is minimized with thick layers of Urylon coating the inner walls of the cavity. The entire experiment is kept very clean to reduce radioactive backgrounds, and this has enabled SNO to make world-leading measurements of solar neutrino fluxes [1].

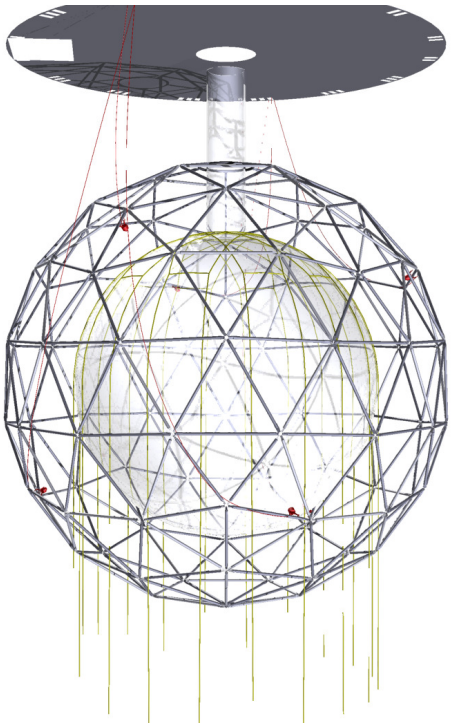


Figure 1.2: Generated image of SNO+ showing the camera locations and hoses in red and the hold-down rope net in yellow. The hold-up rope net is not represented in this image.

1.2 Upgrading to SNO+

To upgrade to the new SNO+ detector, several changes are being made to the existing detector. The heavy water of SNO is being replaced with liquid organic scintillator, lowering the energy threshold for detectable particles. A scintillator-filled detector will produce approximately 50 times more light than a water-filled Čerenkov detector, allowing observation of these lower-energy particle interactions [2]. Electrons passing through the detector easily excite the molecules of scintillator, causing light emission at a far greater rate than through Čerenkov light emission alone. The cleanroom rating and depth of the experiment results in an low background rate, allowing SNO+ to make a more competitive measurement of solar and geo-neutrinos than similar scintillation experiments, KamLAND and Borexino. The 6000 mwe of shielding from atmospheric particles reduces activation of ^{11}C in the scintillator from muon-spallation events. All

of these properties contribute to give SNO+ the ability to make a precise measurement of neutrinoless double beta decay.

The buoyancy of the liquid scintillator requires a means of holding down the AV inside the detector cavity. In SNO, a hold-up rope system was required to support the AV under the weight of the heavy water it contained. For SNO+, a rope net was added, this time to apply an even downward force on the AV (Fig. 1.2) to counter the buoyant force of the scintillator. The net is anchored at the bottom of the cavity and is monitored by load cells.

New techniques are being designed for calibrating the PMT timing and energy resolution, including an upgraded laserball and PSUP-mounted LED calibration system, ELLIE (Sec. 1.5). To reduce internal backgrounds in the AV, the top 2 μm of acrylic will be sanded from the inside of the vessel, removing radon daughters that have deposited on the vessel from exposure to mine air, which has high concentrations of radon.

1.3 Replacing D_2O with Scintillator

An organic scintillator for a large-scale particle detector is generally made up of two or three components: a solvent, which is an aromatic hydrocarbon that becomes efficiently excited by the passage of charged particles and forms the bulk of the scintillator; a fluor, typically 2,5-diphyniloxazole (PPO) at a level of a few g/L which collects the excitations via dipole interactions and emits them as photons; and occasionally a wavelength shifter at the level of a few mg/L which captures the light emitted by the fluor and re-emits it at a longer wavelength to reduce the chance of self-absorption by the scintillator [3].

The scintillator chosen for SNO+ is linear alkyl benzene (LAB) due to its relatively high light output and its compatibility with acrylic. LAB was compared to several other scintillators including diisopropylnaphthalene (DIN), but the latter was decided against due to observations of possible acrylic erosion. Degradation of acrylic was noticed in the Palo Verde experiment [4], where it was exposed to pseudocumene diluted to 20% in mineral oil. Visible changes were noticed in the acrylic, and these changes would not be acceptable in the high-stress application of the SNO+ acrylic vessel.

The testing of LAB's effect on acrylic was carried out by members of the SNO+ collaboration. Extensive work was done developing Monte Carlo models for light propagation from a neutrino event in the scintillator, with the results promising adequate light output with robust signal extraction [3].

One further adaptation required for using scintillator in the detector is the addition of a hold-down rope net. Due to its density, scintillator would cause the AV to float in the surrounding light water. A study of the stresses on the AV was done in SNO+ and a rope net designed was designed to hold down the AV and reduce these stresses [5]. A sketch of this rope net is shown in Fig. 1.2.

1.4 Research Aims of SNO+

There are several areas of physics research where SNO+ will be able to contribute with new measurements and data. Of main interest is the study of neutrinoless double beta decay, low energy solar reactions such as the CNO, ^{7}Be, and pep chains, geo-neutrinos produced from Uranium and Thorium decay in the Earth's crust and mantle, and reactor anti-neutrinos arriving from two nearby nuclear reactors.

1.4.1 Neutrinoless Double Beta Decay

One of the main aspects of SNO+ will be the search for neutrinoless double beta decay. A two neutrino double beta decay ($2\nu\beta\beta$) is a rare second order weak process which occurs when a single double beta decay from (Z, A) to $(Z+1, A)$ is energetically unfavorable, but a double beta decay is allowed.

$$(Z, A) \rightarrow (Z+2, A) + e_1^- + e_2^- + \bar{\nu}_{e1} + \bar{\nu}_{e2} \tag{1.1}$$

This effect has been previously observed in some even-even nuclei such as ^{48}Ca, ^{76}Ge, ^{96}Zr, ^{136}Xe, and ^{150}Nd [6, 7].

A potential different decay mode is the neutrinoless double beta decay ($0\nu\beta\beta$),

$$(Z, A) \rightarrow (Z+2, A) + e_1^- + e_2^- \tag{1.2}$$

where the two nucleons exchange a virtual neutrino rather than emitting two neutrinos. In order for this process to occur the neutrino has to be massive and invariant under charge conjugation, as this decay mode violates lepton number conservation. If this is the case, then the neutrino would be classified as a "Majorana particle", where the antiparticle does not differ from the particle. If neutrinoless double beta decay is not observed, then a strong case is made for the neutrino to be classified as a "Dirac particle", where a particle is distinct from its antiparticle.

Observing neutrinoless double beta decay involves observing a large number of decays while searching for a small mono-energetic neutrinoless double beta decay peak at the end of a larger two-neutrino double beta decay continuum [3]. The requirements for a double beta decay experiment are a sufficient detector resolution such that the $2\nu\beta\beta$ signal does not "smear out" the $0\nu\beta\beta$ signal, very low backgrounds in the $0\nu\beta\beta$ peak region, and a large number of decays in order to collect sufficient statistics in the $0\nu\beta\beta$ region.

The double beta decay experiment will be conducted in SNO+ by suspending neodymium in the scintillator. Natural neodymium contains 5.9% ^{150}Nd, which decays to ^{150}Sm with an endpoint energy of 3.37 MeV, and a half-life of $(9.2 \pm 0.8) \times 10^{18}$ a for the $2\nu\beta\beta$ decay mode [8].

Loading the detector with more ^{150}Nd will increase the observed decay rate, but comes at a price of reduced light transmittance in the scintillator. The neodymium will be loaded in the scintillator at 0.1% by weight, which optical studies have confirmed as being tolerable [3]. This will result in 48 kg of the ^{150}Nd isotope being suspended in 860 tonnes of liquid scintillator, a competitive amount compared to other $0\nu\beta\beta$ experiments. A second route is also being pursued by the SNO+ collaboration to obtain enriched neodymium at 50% ^{150}Nd. This would be result in a detector containing 480 kg of isotope, making it one of the largest double beta decay experiments currently envisioned [3].

1.4.2 Solar Neutrinos

With an energy threshold below 0.5 MeV, SNO+ will be sensitive to solar neutrinos that SNO could not previously detect, namely those from the *pep* and CNO

chains. Accurate measurements of these fluxes can help constrain helioseismological models and shed light on different aspects of neutrino physics [9].

Since the *pep* flux has an energy of 1.44 MeV, it falls between the low energy region where vacuum oscillations have dominant effect on ν_e survival rate (P_{ee}), and the high energy region where the MSW effect dominates. Measuring the flux at this region can make a more accurate measurement of Δm^2_{12}, and help constrain certain neutrino-matter interaction models [3].

SNO+ will also be able to measure the flux of neutrinos from the CNO process. Where the *pp* chain reactions produce about 98.5% of the Sun's energy, 1.5% comes from CNO cycle. The CNO chain depends greatly on the metalicity of the Sun, namely the content of N, O, and especially C [10]. Measuring the contribution of the CNO cycle would help resolve the discrepancies between spectroscopic measurements of the solar metalicity, and the metalicity required to reproduce helioseismological models.[1]. Reduced C^{11} backgrounds in SNO+ will also give it an advantage over the KamLAND detector, where the *pep* flux is obscured by C^{11} backgrounds generated by muon-spallation events in the detector [11].

1.4.3 Geo-neutrinos

Internal radioactivity of the Earth contributes to its overall heat flux, and is an area of active research [12]. The radioactive contribution from the crust is well constrained through geological measurements, but the contribution of the mantle is not yet well understood [13]. By measuring the flux of anti-neutrinos resulting from the decay of U and Th chains, SNO+ can help place constraints on the radioactivity of the mantle and its resulting contribution to the Earth's total energy loss [14].

Uranium and thorium chains have members that undergo beta decay, releasing anti-neutrinos which can be detected in SNO+. Anti-neutrinos interact with protons in the scintillator via the inverse beta decay reaction, producing a neutron and a positron:

$$p + \bar{\nu}_e \rightarrow n + e^+ \tag{1.3}$$

[1]Two types of models, one type based on spectroscopic measurements of the Sun and the other on helioseismology, reproduce differing abundances of heavy elements in the Sun [9] The predictions for ^{13}N and heavier elements differ by over 30%.

The positron annihilates with a nearby electron almost immediately, while the neutron travels on average 200 μs through the scintillator until it captures on a proton, releasing a monoenergetic 2.2 MeV γ-ray. This "delayed coincidence" signal provides a powerful method for extracting these interactions from the single-event backgrounds [3].

SNO+ is in a position to make an important contribution to the study of geo-neutrinos already started at KamLAND and Borexino. It is located in a well-studied region of continental crust that has higher concentrations of U and Th, whereas KamLAND is surrounded by both continental and oceanic crust (depleted in U and Th by comparison). The largest fraction (80%) of signal comes from the crustal rock, with most of it ($70 - 80\%$) originating from rock that is within 1200 km, allowing it to be measured through other geological methods [14]. This will deconvolve the crustal contribution from that of the mantle, giving insight into the heat production of the deep Earth.

There is also benefit from the lower reactor neutrino rate around SNO+. The closest nuclear reactor, the Bruce generating station, will produce a reactor neutrino rate of 179 events per 10^{32} proton-years, of which only 49 lie in the same spectral region as the geo-neutrinos [14]. The signal-to-background in SNO+ is 51/49, or roughly 1:1. KamLAND has 165 reactor background events per 10^{32} proton-years, giving it a signal-to-reactor background ratio of 1:4. Further detail including spectral plots is given in [14].

1.4.4 Reactor Anti-Neutrinos

Nuclear reactors emit anti-neutrinos generated from the beta decay of the reactor fuel's fission products. The energy spectrum of these neutrinos is usually known, and their flux is tightly constrained by the carefully measured power output of the reactor. Using this information along with the distance to a reactor, it is possible to perform long-baseline neutrino measurements to determine the effects vacuum oscillations have on the neutrinos.

Anti-neutrinos would be detected in SNO+ through the inverse beta decay reaction with delayed coincidence, the same process as geo-neutrino detection. SNO+ is located 240 km from the Bruce nuclear generating station and about 330 km from

reactors in Pickering and Darlington. Due to the baselines between the SNO+ and the reactors, it happens that the second oscillation minimum from the Bruce reactor coincides with the third oscillation minimum from the Pickering and Darlington reactors. This feature will make SNO+ very sensitive to Δm^2_{12} mass splitting [3], and it has even been suggested [15] that this favorable reactor positioning may allow SNO+ to make a more precise measurement of Δm^2_{12} than KamLAND. More detail of the mathematics behind making these measurements in SNO+ can be found in [3].

1.5 Calibration of the SNO+ Detector

It is important to distinguish here between calibration of the SNO+ detector and calibration of the camera system. Calibration of the detector involves placing an enclosed radioactive source or light emitter into the detector and observing the response of the PMTs. During a calibration run the position of each physics event is reconstructed, and an important check on the accuracy of the reconstruction is knowing the location of the source producing these events. Previously this was done using the rope manipulator system. This thesis outlines how the camera system will be used to triangulate the source location to greater precision.

In order to ensure that the camera system is calculating accurate source positions, it itself needs to be calibrated beforehand. This is a separate calibration from the detector calibration and will be done by photographing the PMTs with the installed cameras and mapping their positions onto the camera's pixel space. The mathematics behind the process is explained in Chapter 5 and the procedure is described in Chapter 7.

The general goals of the detector calibration are to understand the energy response of the detector in the range of $\sim$ 0.1–10 MeV, including focusing on the low energy threshold of 0.1–0.2 MeV, and the double beta endpoint energy of 3.37 MeV. Neutron detection efficiency must also be understood, as this process is highly dependent on event position when taking place near the edge of the AV. Gamma rays resulting from neutron capture near the edge of the detector can escape the scintillator volume resulting in less observed light, which limits the fiducial volume of the detector. The electronics and DAQ must be calibrated, and these calibrations will also be used to create useful Monte Carlo models of events in the SNO+ detector [16].

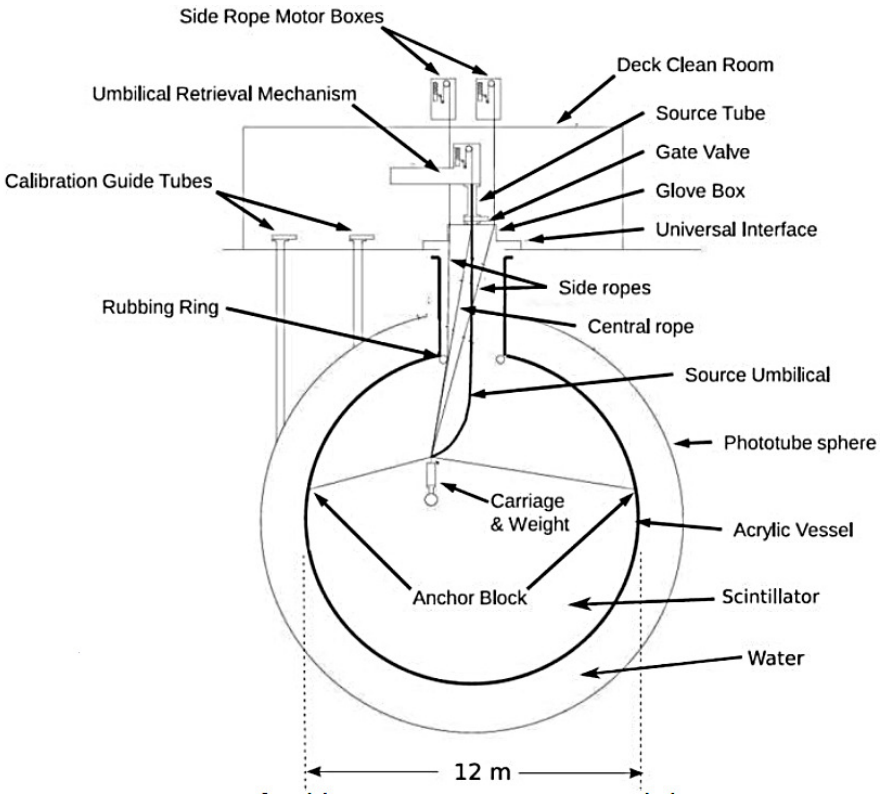


Figure 1.3: Schematic of the source (rope) manipulator system used in SNO and SNO+ to lower calibration sources such as the laserball into the detector. Adjustable ropes allow the source to be positioned across all three axes in the acrylic vessel.

1.5.1 Point Sources

To calibrate the energy response of the detector, various radioactive point sources will be used. Using the source manipulator system (Fig. 1.3, reproduced from [17]), these sources can be lowered into the detector to various positions where they will undergo decay and emit γ-rays into the detector. These γ-rays will Compton scatter off of the electrons in the scintillator and create scintillation light. By using different monoenergetic γ-ray sources, the dependence of light yield on γ-ray energy can be determined. Additional point sources, including a Čerenkov source and β source (^{8}Li), are still under study and would provide useful information when implemented. An Americium/Beryllium (AmBe) neutron source will be used to study the position dependence of neutron

capture light yield and how sharply it varies for events close to the edge of the detector.

1.5.2 Laser Ball

Optical calibrations of the detector are done using a "laserball" [18]. The laserball is a 109 mm diameter quartz flask filled with diffuser. Laser light generated on the SNO+ deck is transported into the quartz flask using optical fibers, where it emits isotropically. Lowered into the detector with the source manipulator system, it is used to calibrate the PMT response.

1.5.3 ELLIE

ELLIE, or Embedded Light Injection Entity, is a system of optical fibers mounted on the PSUP which transports light into the detector after it has been generated by LEDs or lasers on the deck surface [16]. This system is divided into several subsystems. Timing calibration is tested by TELLIE, which consists of 91 optical fibers emitting 432 nm LED light, covering all of the SNO+ PMTs [18]. Changes in absorption lengths in the scintillator are measured using AMELLIE, which can inject LED generated light in two different directions from four different source points [19]. Scattering measurements in the LAB are measured using SMELLIE, a system which emits light in three different directions using multiple wavelengths from lasers [20].

Since this calibration system is mounted on the PSUP, it has the advantage that it does not have to be redeployed each time it is used. This reduces the possibility of contaminating the scintillator with radon from the mine air. A second advantage of a stationary PSUP-mounted calibration system is that the light can be used as additional reference points for calibrating the camera system.

Chapter 2

Camera System

Vital to the calibration of PMTs is knowledge of the calibration source position within the detector. Previously this was calculated by measuring the rope lengths of the source manipulator system. This could determine the source position to within 5 cm just below the neck of the AV, but suffered higher uncertainty when the source was moved away from the center of the detector [21, 22].

A camera system was proposed to photograph the source and triangulate its location as a more accurate means of source localization. This will lead to better PMT calibrations and more accurate physics measurements from the SNO+ detector. The camera system also allows monitoring of the physical state of the detector, including the position of the rope net and the movement of the AV. The software written for the purpose of triangulation is detailed in the following chapters of this thesis.

The camera system hardware was principally designed by P. Gorel, with software work done by the author. Additional valuable input was given by A. Hallin and other members of the SNO+ collaboration. The design of the camera system was inspired in part by the camera system developed at Borexino [23]. This chapter of the thesis will focus on the hardware aspect of the project, including details of the camera enclosures and a summary of the camera system installation status.

2.1 Overview

The camera system consists of six Nikon D5000 cameras in watertight, stainless steel enclosures with clear acrylic dome-shaped viewports. The cameras are to be mounted on the PSUP looking inwards on the SNO+ AV in six symmetrically spaced locations.

Each camera will have a full view of the vessel owing to its fish eye lens, a Nikon AF-S DX 10.5mm F2.8G Lens. This lens has a 180° picture angle allowing for wide-angle shots such as those in Fig. 2.5. The entire detector is visible and will be in focus once the cavity is filled with water and AV is filled with scintillator. The water and scintillator fill will improve the camera focus as their indices of refraction closer match that of acrylic than does air.[1]

The symmetric spacing of the camera mounting shown in Fig. 2.1, with three on the bottom hemisphere of the PSUP and three on the top, ensures that a calibration source placed inside the detector will be visible from all directions. This symmetry in the cameras' arrangement also serves to reduce systematic errors when triangulating the source position.

Each camera is connected to the control deck of the SNO+ experiment through a 25 m long hose.[2] Each hose contains wires for power, RJ45 (ethernet) signal cable, and plastic hose for a N_2 gas flush system to remove moisture from the enclosure. The camera hoses penetrate the deck at two locations (top of Fig. 2.1) where they pass through a manifold that separates the deck surface from the detector underneath. After reaching the surface, each group of three hoses reaches a control box (Fig. 2.3) that manages the electrical connections and monitors gas flow.

From these boxes, the signal from each camera passes to an on-deck computer capable of remote near-simultaneous operation of the cameras. This computer will also provide an internet connection so that users can operate the camera system remotely and view photographs taken by it. This system is still in development.

As a precaution, the cameras also underwent small hardware modifications to avoid damaging the SNO+ PMTs while the camera system is in operation. The LEDs

[1] Water has an index of refraction $n_w = 1.33$, acrylic $n_{acr} = 1.49$, scintillator $n_s = 1.48$, while air has $n_{air} = 1.00$. Thus, the photographs in air appear more blurry than underwater photographs.

[2] Commonly termed the "umbilical".

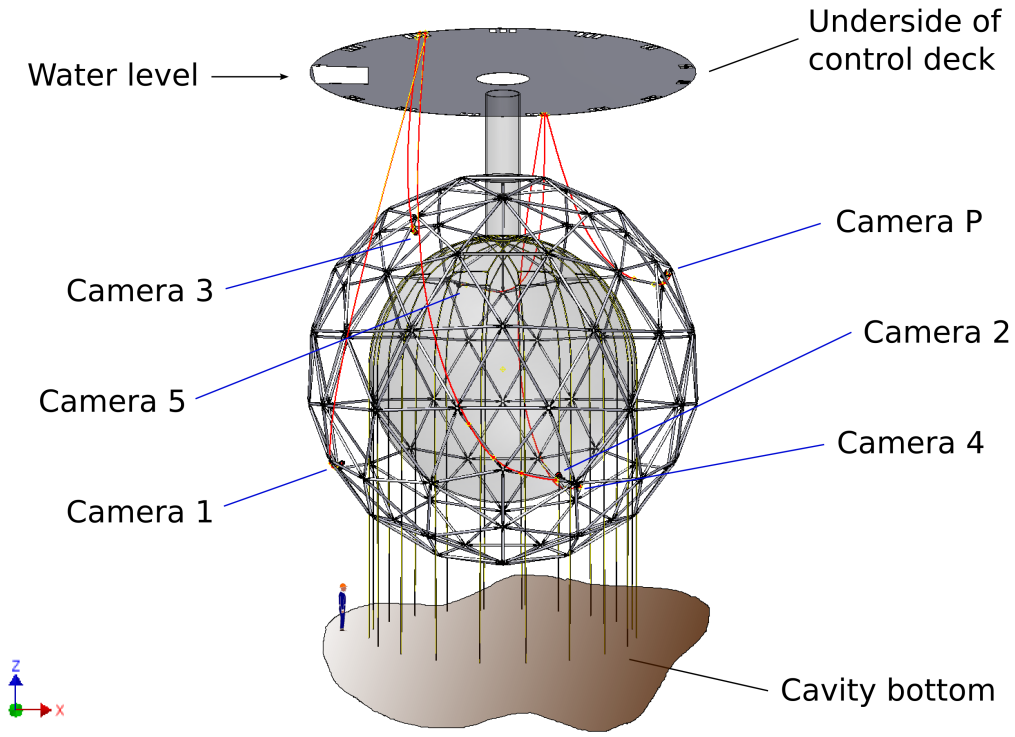


Figure 2.1: Camera positions relative to the PSUP. The six cameras are labeled P (prototype), followed by 1, . . . , 5. The camera cables are shown in red travelling to two locations underneath the deck surface above the cavity, and the hold-down rope net is shown in yellow. The hold-up rope net is not shown.

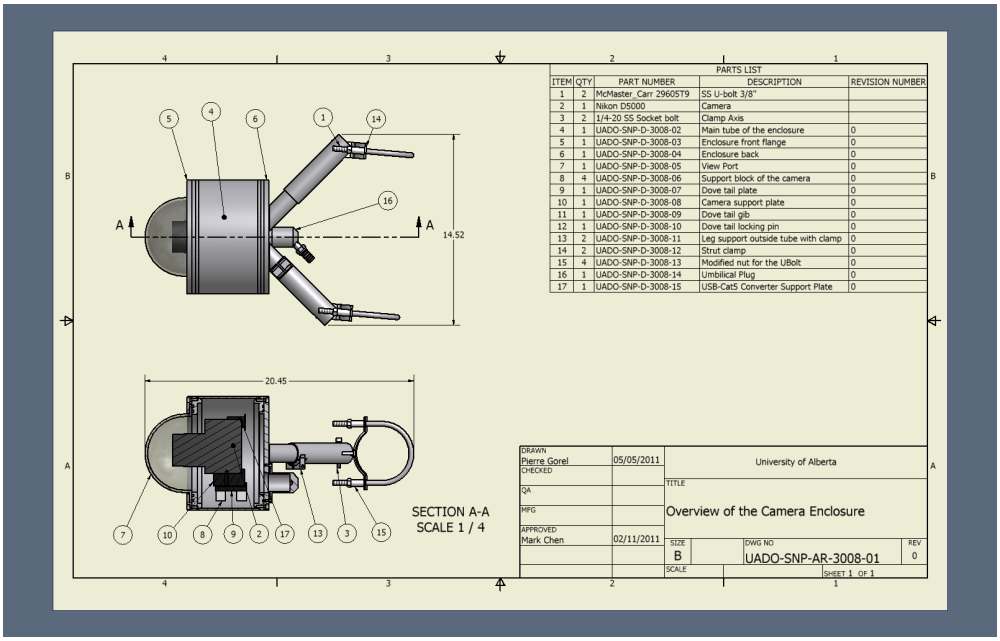


Figure 2.2: CAD drawing of camera enclosure showing the aluminum brackets inside for mounting the camera.

on the outside of the camera were disabled by carefully disassembling each camera and cutting the wires leading to the LEDs. The flash was also disabled by cutting the power wire leading to it. A flash accidentally going off during operation of the PMTs could damage them due to the amount of light being suddenly produced.

2.2 Camera Enclosures

Each camera enclosure (Fig. 2.2) has a clear acrylic dome, 144 mm in diameter and 5 mm thickness, sealed with a pair of O-rings. These domes[3] were chosen for their optical clarity and lack of visible distortions. A benefit of acrylic is that it contains less radioactivity than glass (which was used for the domes in the camera system at Borexino [23]), while making it somewhat more difficult to achieve a good optical interface. The O-rings are standard Buna-N nitrile O-rings, used commonly in SNO+ and deemed safe for the experiment in terms of Rn emanation. The back of each enclosure

[3]Manufactured by a local supplier, PG Plastics.

has mounting brackets for attachment to the PSUP frame, and a feed-through port for carrying cable and tubing to the SNO+ deck surface via the umbilical hose.

Each stainless steel enclosure contains a single camera secured inside on an aluminum mounting bracket. The camera was mounted inside such that the effective focal point of the camera would be at the center of the optical dome[4]. The cameras were also set for a near focus[5] of approximately 18 cm which would give the correct focus when the enclosures were submerged in water. This is the reason why photographs of the AV taken in air appear slightly blurry as in Fig. 2.5.

Different types of polyurethane hoses were tested at the University of Alberta Low Background Counting lab by A. Hallin of SNO+. All hoses tested were strong, flexible and suitable for the task of carrying wires underwater in SNO+. However, low backgrounds are critical to maintaining the low energy sensitivity threshold of the detector. The hose[6] with the lowest radioactivity was selected for the enclosures, to minimize radioactive contamination of the SNO+ detector. Cables were fed through the hose by attaching a thin, metal lead-wire to a small ball slightly smaller than the inner diameter of the hose (12.7 mm), and "sucking" it through with a vacuum cleaner placed on the other side. The lead-wire would then be used to pull a cable through the entire 23 m length of the hose.

The prototype (P) enclosure was sent to Queen's University where it underwent water pressure tests to test the strength of the acrylic domes. For each test, a dome was mounted on the P enclosure (the only existing enclosure at the time) and was subjected to gauge pressures of up to 90 PSI. The results are detailed in Table 2.1.

Machining for the prototype stainless steel enclosure as well as inner mounting brackets was done at the University of Alberta physics machine shop. The other five enclosures had their flanges and front and back pieces machined at a local machine shop,[7] with all welding done at the University of Alberta. Each enclosure was tested at the University of Alberta by pressurizing it while it was emersed in a bucket of water and looking for bubbles. Some welds were redone as a result of these tests.

[4]The effective focal point of the camera was measured to be in the center of the lens, just behind the painted gold ring on the lens.

[5]Calculations for this were done by P. Gorel.

[6]Blue polyurethane air hose, 1/2″ ID, 11/16″ OD, #54085K14 from McMaster-Carr.

[7]Precimax Mfg. Ltd.

Dome	Water (PSI)	Vacuum (PSI)	Gauge (PSI)	Passed
1	44	0	44	Yes
2	44	0	44	**No**
3	45	15	60	Yes
4	45	15	60	Yes
5	45	15	60	Yes
6	75	15	90	Yes
7	75	15	90	Yes

Table 2.1: Pressure test results, courtesy of [24]. Dome 2 suffered a crack in the acrylic during the tests. This was thought to be due to a defect in the dome.

As Table 2.1 shows, the second dome suffered cracks during the test. It was not used further, and was replaced with a spare dome. Since the highest pressure that a camera[8] will experience in SNO+ is 27 PSI of water pressure against the acrylic dome, these tests show that the camera enclosures should fare well once the cavity is filled with water.

In addition to wires, the umbilical hose also contained $1/8''$ plastic tubing for the N_2 gas flush system. This system was designed to operate at a low flow rate (a few L/h) and is meant to carry away moisture than will seep into the enclosure or condense out of the air over a period of months or years. The N_2 gas flows in through the tubing and flows out through the bulk of the umbilical hose. A N_2 cylinder on the SNO+ deck will be used to supply the gas once the system is in operation.

2.3 Cable Boxes

Two identical cable boxes were designed and constructed at the U of A to contain the power and signal wires, as well as the N_2 gas flush tubing and flow meters. Each box (Fig. 2.3) contains three smaller boxes, one for each camera hose. On the surface of each larger box are mounted six flow meters: three for input flow and three for output flow.

[8] Camera 4, the lowest camera in SNO+ coordinates, will be at a water depth of 18.8 m.

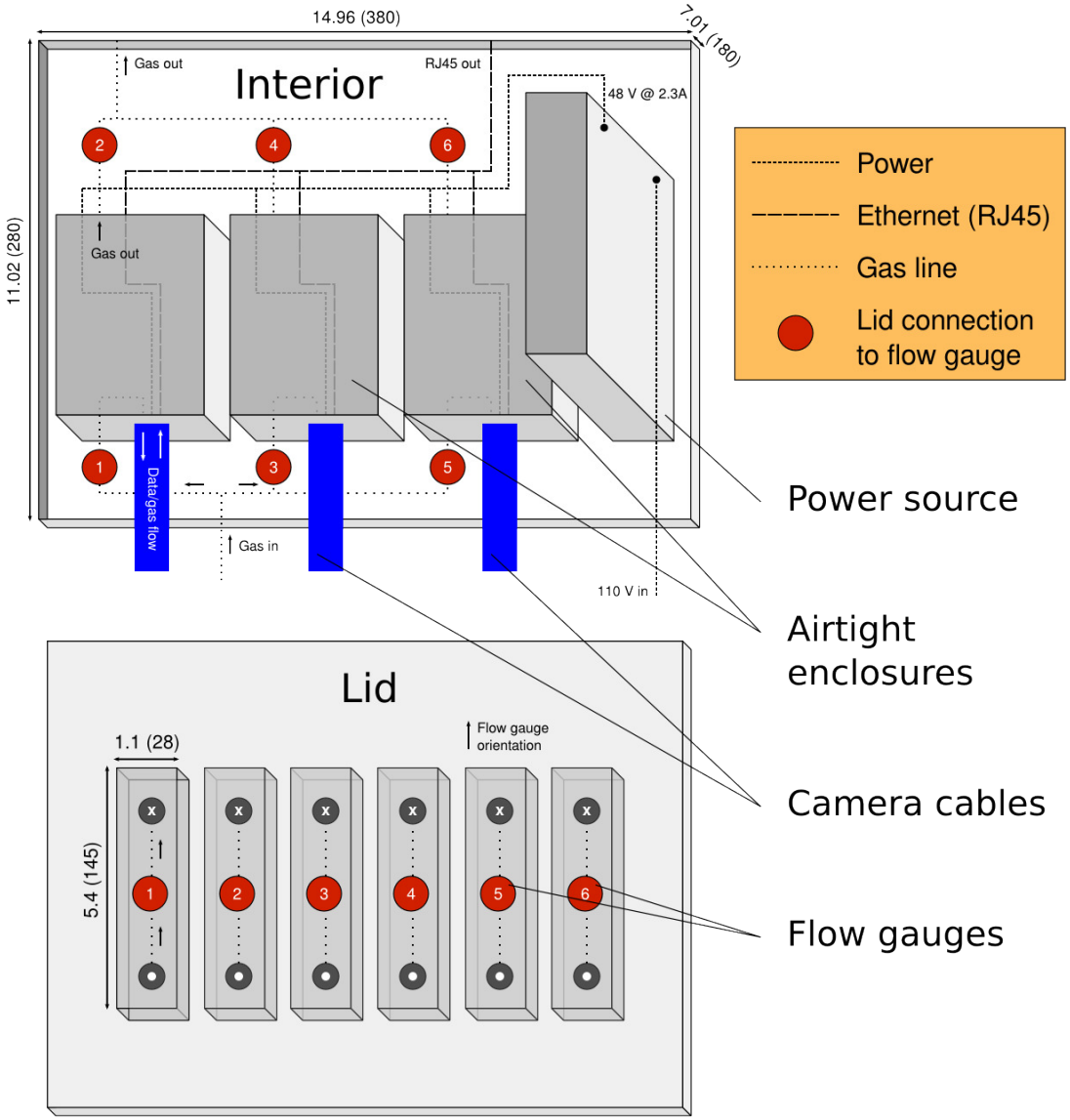


Figure 2.3: A schematic showing the general layout of a camera cable box. Two of these are used, one of each side of the SNO+ deck.

The smaller boxes were designed with IP67 rated[9] connectors. Since the return line of the N_2 gas is the interior of the umbilical hose, this gas must be trapped within the small box to which the hose connects and funneled out towards a single exhaust port. This gas port then sends the gas through a second flow meter, confirming that gas is circulating through the camera enclosures.

The camera boxes are designed to use a bottle of N_2 gas for its air supply. Power will come from a standard 110 V wall socket converted to 2.3 A at 48 V for each large box. All six ethernet cables will connect to a USB hub before reaching the control computer.

2.4 Installation of the Bottom Cameras

The three bottom cameras were installed in a week in October 2011. The cavity was empty at the time to allow replacement of damaged PMTs, replacement of the cavity liner, installation of the hold-down rope net anchors, and other cavity work. Using scaffolding and a Genie® lift, the bottom three cameras were installed in their respective positions on the PSUP. Table 2.2 shows the bottom three camera positions relative to the PSUP coordinate system.

Camera	X	Y	Z	Notes
P	7289.7	193.5	4246.6	Water-fill
1	-7256.2	-31.6	-4538.6	**Installed**
2	2230.82	-6941.3	-4291.6	**Installed**
3	-3666.4	-5050.2	5621.7	Water-fill
4	3669.5	5049.3	-5697.1	**Installed**
5	-2194.6	6952.5	4175.4	Water-fill

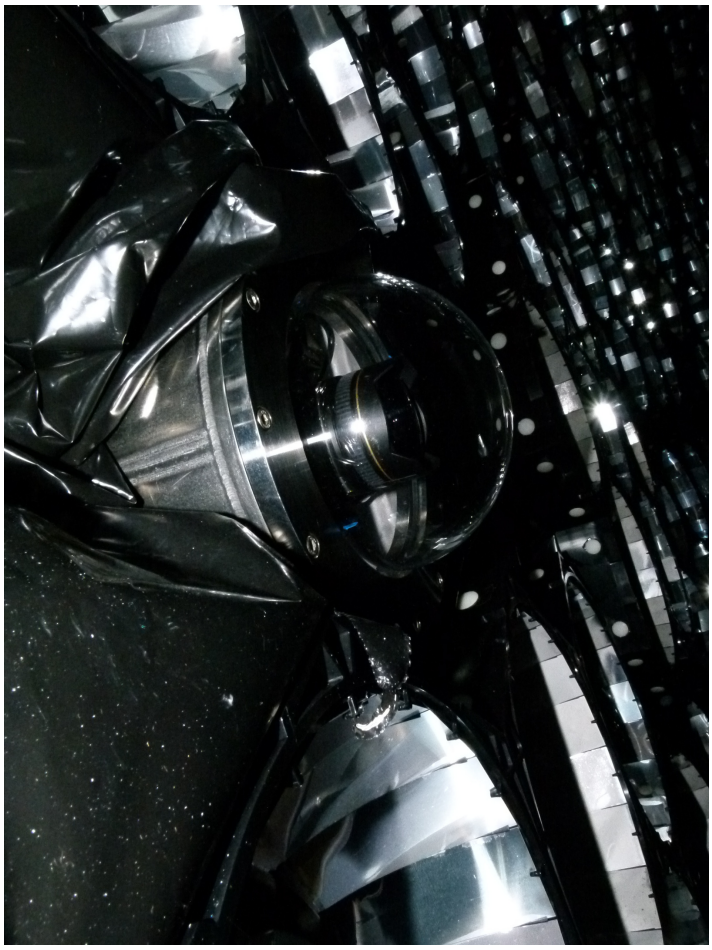
Table 2.2: Coordinates of the individual cameras relative to the PSUP coordinate system, given in units of mm. Three of the cameras are installed as of May 2012. The other three will be installed during the water-fill of the SNO+ detector cavity.

[9]The IP67 rating means that the device is dust-tight, and may be submerged in water for 30 minutes without harmful ingress of water.

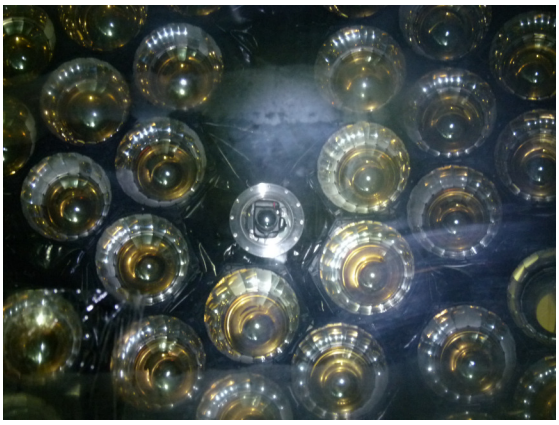
Installation of the cameras involved first lowering the cameras down into the cavity by basket. The hoses were then fed down into the cavity through two ports on opposite sides of the deck, being sure to pass them along the side of the PSUP to avoid having them get caught among the PMTs. All six hoses were lowered into the cavity. The end of each hose was sealed with a plastic cap, ensuring that unconnected hoses would remain safe and not take in water during the water-fill of the cavity. The three hoses for cameras 1, 2, and 4 had their caps removed and were connected to the installed cameras, ensuring a watertight seal.

Several pictures were taken with the cameras to ensure that they worked, shown in Fig. 2.5. These pictures appear slightly blurry due to the fact that the focus was set for submersion in water and not air.

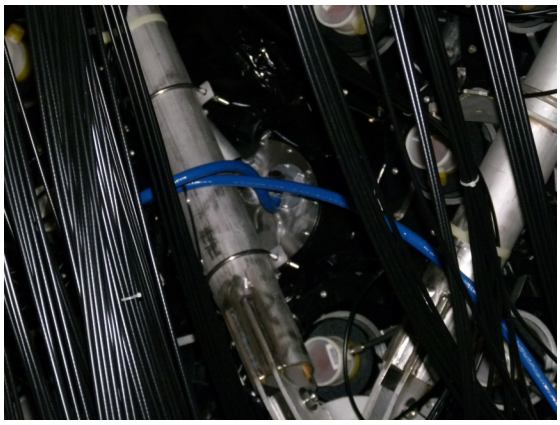
The remaining three camera locations are inaccesable from the cavity floor and will have to be accessed by boat during the water-fill phase of SNO+. This is planned for 2013.



(a) Camera 1 installed amongst the SNO+ PMTs. The black plastic visible creates a boundary between the water within the PSUP, and the water without.

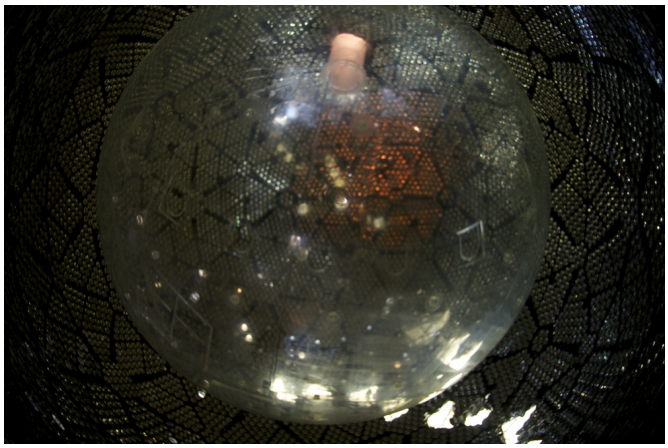


(b) Front of Camera 4, taken from within the AV.



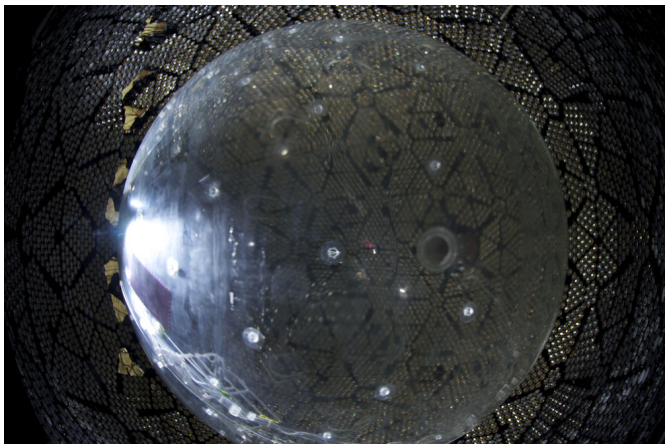
(c) Back of Camera 4, with mounting brackets and umbilical hose visible.

Figure 2.4: Pictures of the installed bottom cameras.



(a) Camera 1.

(b) Camera 2.



(c) Camera 4.

Figure 2.5: Pictures of the AV taken by lower three cameras after installation. The photographs are slightly out of focus as the cameras were set to be in focus only when submerged in water, and not in air.

Chapter 3

Camera Light Sensitivity

This section details the studies done on the light sensitivity of the cameras. This was done in order to ensure that an LED bright enough to be detected by the cameras in the SNO+ AV would not emit so much light that it would damage the surrounding PMTs. A PMT supplied with high voltage is sensitive to single photons, and even a single bright LED could produce enough light to overload and damage a PMT. The camera system must therefore be able to detect relatively dim LEDs when performing triangulation during a PMT calibration run when the PMTs are in use.

These tests were done to discover the minimum light sensitivity threshold of the cameras. Results show that when a camera would first detect an LED's light, the surrounding SNO+ PMTs would be detecting a flux of 40kHz of photons. This rate is safe for the PMTs, as shown in Sec. 3.3.

3.1 LED Output

The experiment was carried out in a dark room at the University of Alberta Physics Electronics Shop by first using a 38mm diameter Hamamatsu 9902B PMT to detect photons emitted from an LED. The light was channeled through an optical fiber towards the PMT. The goal was to determine the LED's light output curve before placing a camera in front of the LED.

The LED was set up to receive 20ns pulses from a digital delay/width generator. The signal from this generator also went through a gate generator, which would

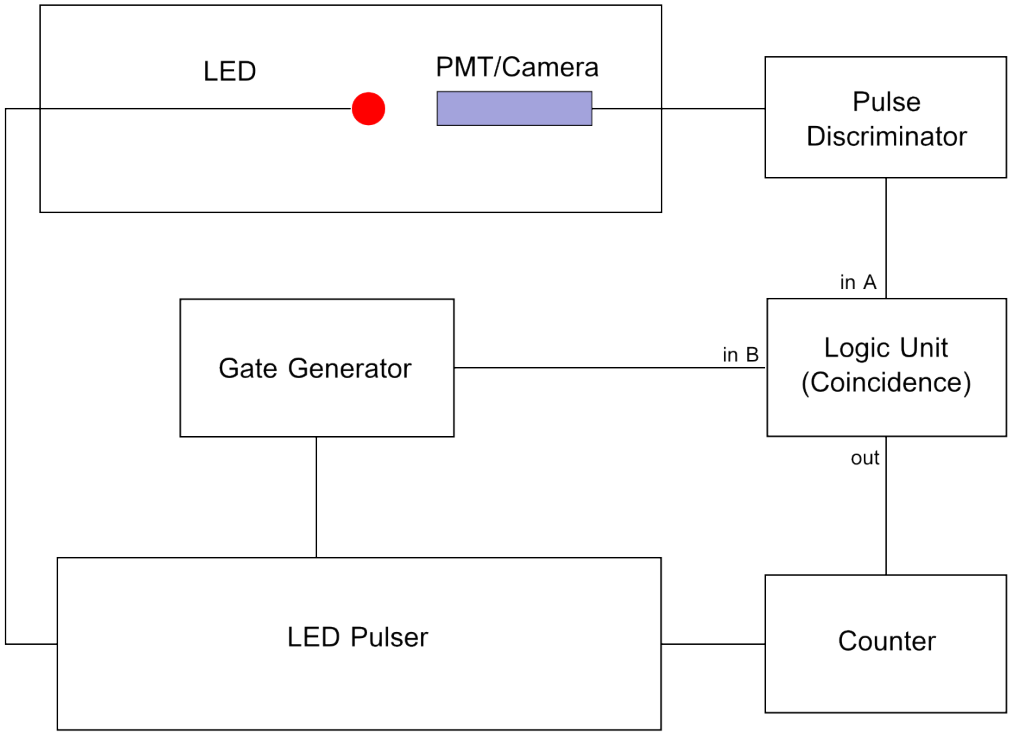


Figure 3.1: Schematic of the setup used to determine the emission response of the LED and the camera's sensitivity.

stretch the pulse length by a factor of 10 to create a coincidence window during which the PMT could detect a photon from the LED. This coincidence window helped to reduce noise from stray photons in the dark box. The setup is shown in Fig. 3.1.

The LED was initially supplied with a low frequency of 20 ns pulses ($< 10^4$ Hz), with the amplitude of the pulses tuned until the PMT was receiving an average of one hit for every 50 pulses delivered to the LED. Previous experiments determined that the LED delivered mainly single photons at this rate [25]. The data are shown in Fig. 3.2.

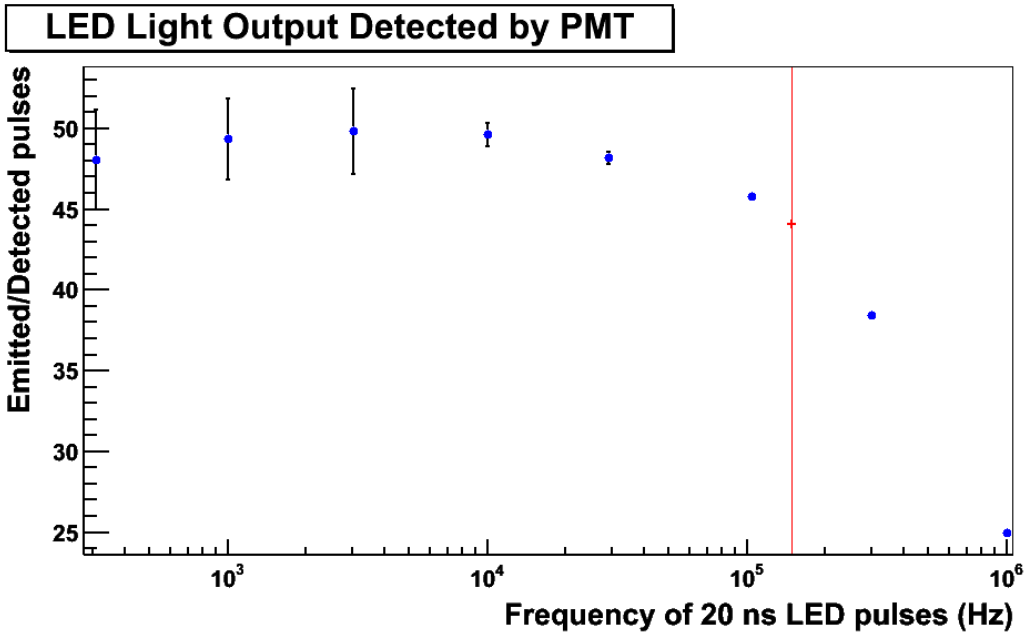


Figure 3.2: Rate of coincident PMT hits vs. frequency of single photons emitted by LED. Blue points are data, and the red cross shows the lower threshold where the camera was first able to resolve light from the LED.

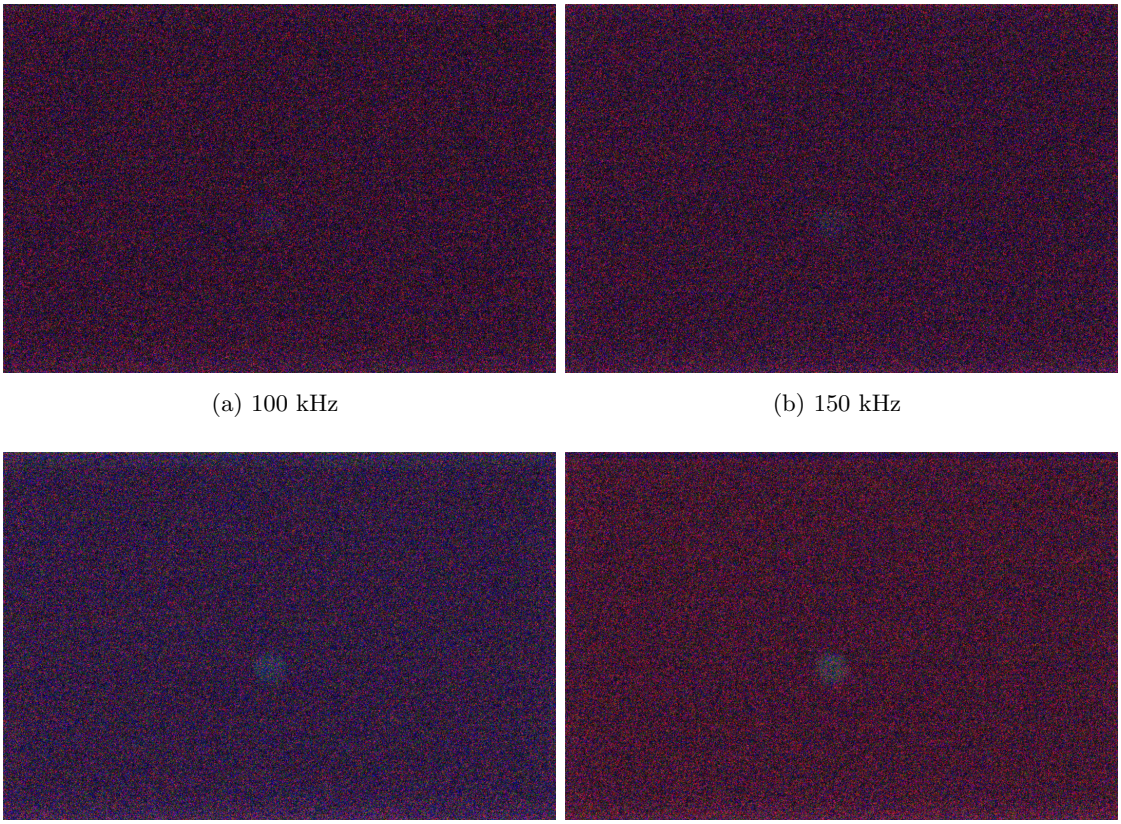
3.2 Camera Sensitivity

After determining the light output curve of the LED, the PMT was replaced with a camera and the test was repeated. The camera's threshold light sensitivity would be determined by increasing the light output of the LED until a visible spot was resolved in a 30 second exposure taken by the camera.

The camera had its ISO settings tuned to provide it with the greatest light exposure possible. The ISO setting plays two roles - the higher the number, the more light is allowed in the aperture per exposure. Raising the ISO setting also reduces the suppression of pixel noise which can result in a photograph with more "grain" [26]. The camera was set to the highest ISO setting for all exposures taken of the LED[1].

At an LED pulse rate of 150 kHz, indicated by the red cross in Fig. 3.2, the

[1]The camera's ISO setting and exposure time can be controlled remotely from a laptop. This means the cameras can alternate between taking regular photographs of the AV for calibration purposes and taking long exposures with high ISO for capturing an LED during a calibration run.



(a) 100 kHz (b) 150 kHz

(c) 200 kHz (d) 300 kHz

Figure 3.3: Photographs taken with a Nikon D5000 camera at various LED emission rates for 30 second exposures.

camera was able to resolve a spot of light. Pictures taken at and around this frequency are shown in Fig. 3.3.

3.3 Results

At the point where the camera first resolved a spot of light, the ratio of emitted to detected photons was 44 to 1. This equates to a flux of 3.4 kHz of photons detected by the PMT. To scale up to the light incident on a 20 cm diameter SNO+ PMT, the size of the camera lens needs to be compared to the SNO+ PMT size[2]. Scaling from a 60

[2]The size of the PMT in the dark box does not come into account because the fiber was placed against the PMT face.

mm fisheye lens to a 20 cm diameter SNO+ PMT gives a factor of 11 increase in area. Assuming the SNO+ PMTs have the same quantum efficiency as the Hamamatsu PMT used for the experiment equates to a 40 kHz detected flux of photoelectrons incident on the SNO+ PMTs. This excludes amplification of light from the light concentrators surrounding the PMTs, which may add an additional 50% to the light flux.

The PMTs in SNO+ will have a maximum allowed anode current of approximately 100 μA. A quick calculation shows the anode current that 40 kHz of photoelectrons would produce in a PMT:

$$(4 \cdot 10^4 \text{ e}^-/\text{s}) \times (1.60 \cdot 10^{-19}\text{J}/\text{e}^-) \times (\text{Gain of } 10^7) = 64 \text{ nA} \tag{3.1}$$

This is far below the bias current of 100 μA and suggests that the camera system may be safely used to triangulate an LED's position during a calibration run in the detector.

Chapter 4

Underwater Camera Tests

The cameras were tested at the University of Alberta's West Pool in June and September of 2011. These tests had several purposes. The functionality of the cameras and camera control software had to be confirmed. The stainless steel enclosures, hoses, and seals had to be checked to see if they were watertight during normal operation of the cameras. These tests were also necessary to acquire photographs of calibration targets from which a 3D $\rightarrow$ 2D pixel mapping could be designed and tested. Finally, a test was set up in order to perform triangulations once the cameras were calibrated.

4.1 June 2011 Pool Test

In June 2011, the first pool test confirmed the cameras' ability to take clear and focused pictures underwater without any leaks in the enclosures.

Cameras were set up on sawhorse ladders in two groups of three (Fig. 4.1a). The enclosures were assembled in the lab before bringing them to the pool. This included attaching the acrylic domes in place overtop of the cameras and feeding all cable connections into the hose through the back panel of the enclosures. The screws had to be torqued precisely to prevent any water leaks, and also with care to prevent any damage to the acrylic dome.

Each cable-carrying hose was wound onto a spool, which would be unwound once at the pool (Figs. 4.1a and 4.1b). The spools were kept out of the water on the pool deck. The CAT5 cables fed into a central hub that was connected via USB to a

laptop controlling the cameras, with a lab power supply providing current for all six cameras. All of the cables were run through the hoses to the underwater cameras.

A calibration target consisting of a 2.5m × 3m tarp covered in a grid of 720 circular, white reflectors was used for this experiment (Fig. 4.1d). The reflectors were 3 cm in diameter with adhesive on one side, and were spaced 10 cm apart from their centers on the tarp. The tarp was supported on a rectangular frame made up of one-inch diameter stainless steel piping. It was secured to the frame with bolts along one side and a bungee cord stretching the tarp along the other three sides, with 15 cm of space between tarp and frame on the bungee side. It was waterproof and easily dismantled, allowing it to be positioned in various ways in the pool.

The target was set up in both the deep end and the shallow end of the pool (Fig. 4.2). After measuring the positions of the sawhorses, which would later be used to calculate the positions of the cameras, sets of pictures were taken with each camera. Included were pictures taken with the pool lights turned off, and pictures with and without flash.[1] Seven sets of pictures were taken in total with some examples shown in Fig. 4.3.

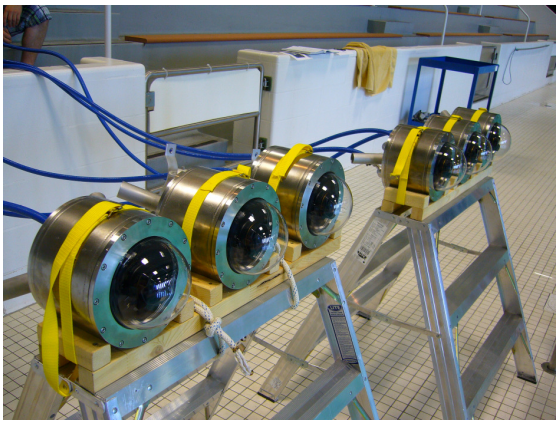
When the pictures were analyzed afterwards, it was found that the tarp had been placed at a distance further than needed to effectively calibrate the camera's entire field of view.[2] Pictures taken in the deep end covered approximately 15% of the field of view in both the x- and y-direction for a total of 2% total coverage. Pictures in the shallow end were taken closer to the target, resulting in 4% total coverage, but were unusable for calibration because the position of the tarp was poorly defined there. In all positions the tarp showed too much flexibility (bending and folding) to have the location of the reflectors accurately known.

4.2 September 2011 Pool Test

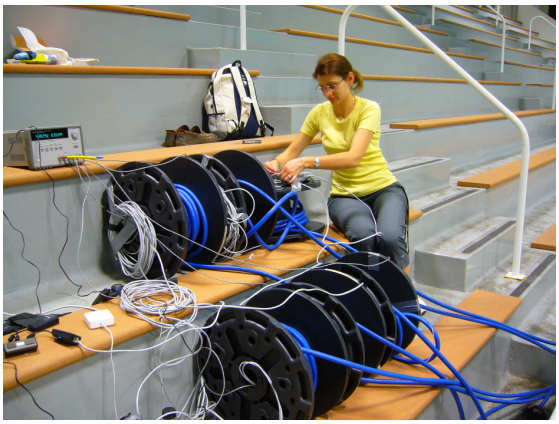
After examining pictures taken in the shallow end it was noticed that the pool tiles were resolvable in these pictures and very uniform across the camera's field of view.

[1] The flash was later disabled to avoid potentially damaging the PMTs when the cameras are in operation in SNO+. This modification is discussed is Chapter 2.

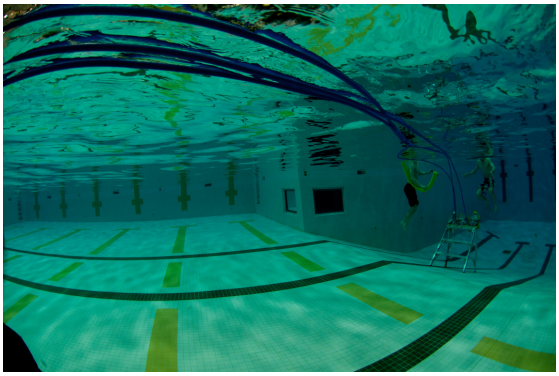
[2] The test did demonstrate, however, that the cameras could hold in focus a target as close as 2 m and as far away as 15 m.



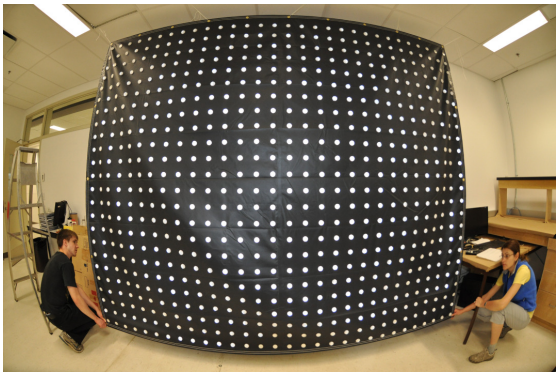
(a) Cameras set up on sawhorses.



(b) Alexandra Bialek, a postdoc with SNO+, helping manage the spools of CAT5 and power cables.



(c) Cameras being set in place for deep end photographs.



(d) Reflectors on tarp (without frame) being photographed before the pool test.

Figure 4.1: Setup pictures for the June 2011 pool test at the U of A West Pool.

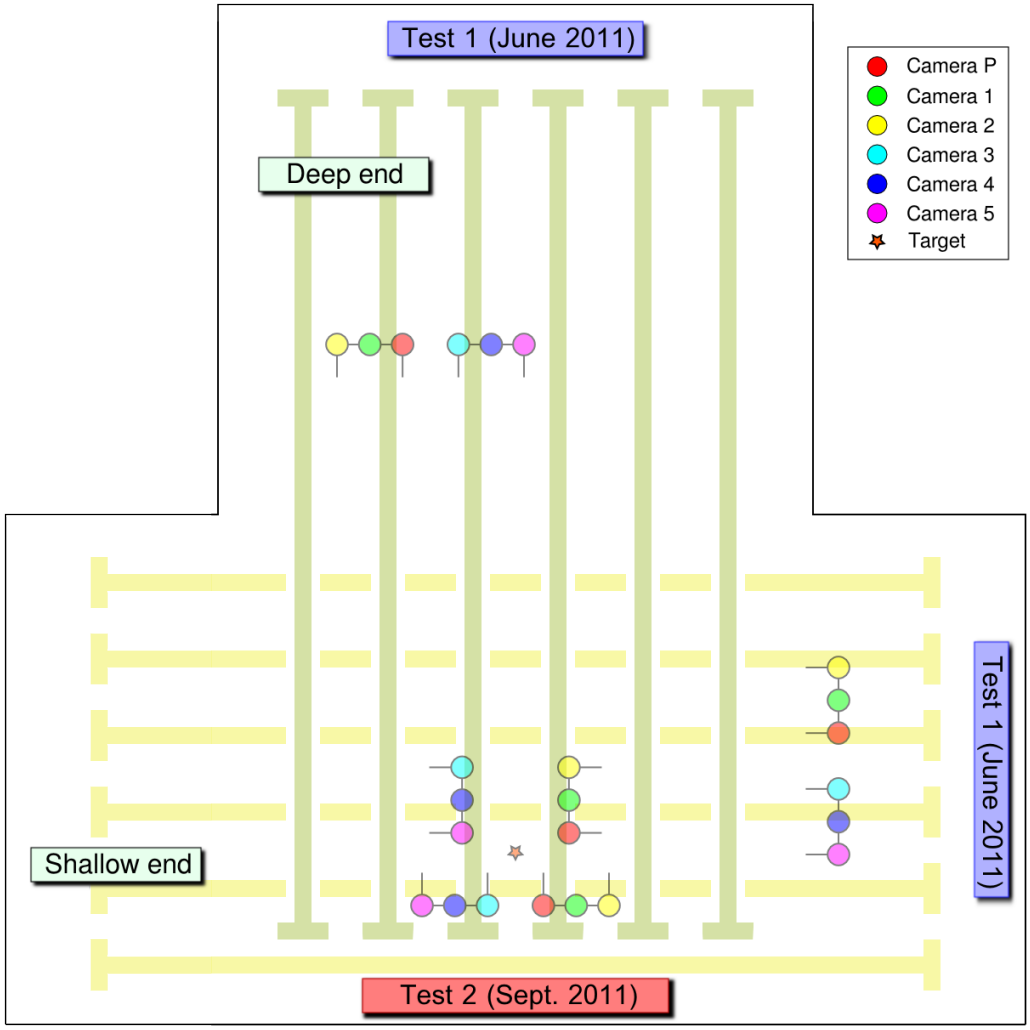
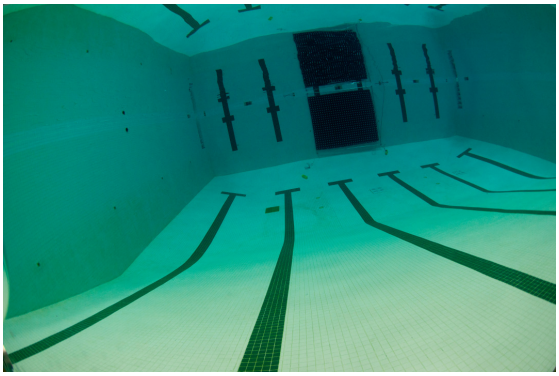
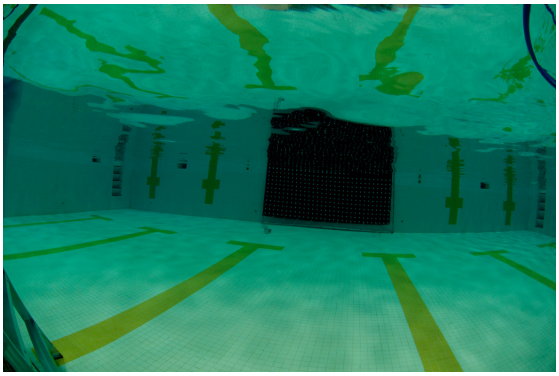


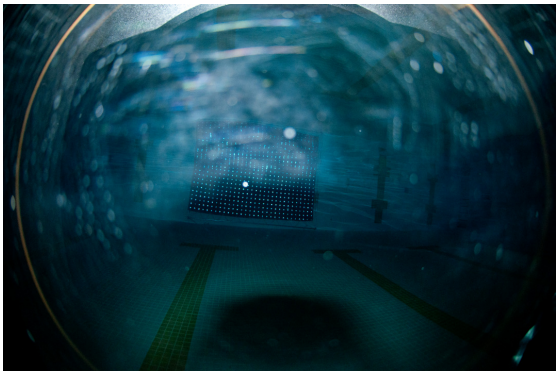
Figure 4.2: Schematic of where the pool tests were carried out. Test 1 was done using the black tarp with reflectors as a calibration target, and Test 2 was done using the pool wall tiles as a calibration target. The colored circles represent approximate positions of cameras during the different phases of the experiment. The star represents the target which was triangulated during Test 2.



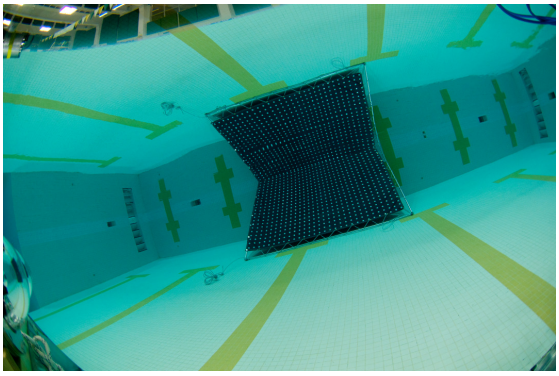
(a) Camera 2, deep end.



(b) Camera 5, shallow end.

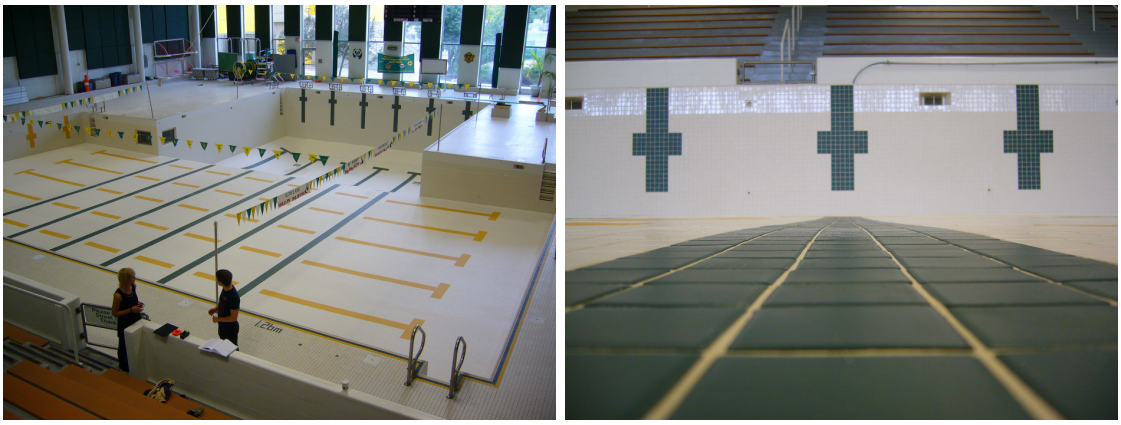


(c) Camera 1, shallow end; lights off and flash used. The reflection of the gold band around the lens is clearly visible in the acrylic dome.



(d) Camera 3, shallow end; neighboring cameras visible.

Figure 4.3: A sample of photographs from the first swimming pool test.



(a) A view of the entire pool.

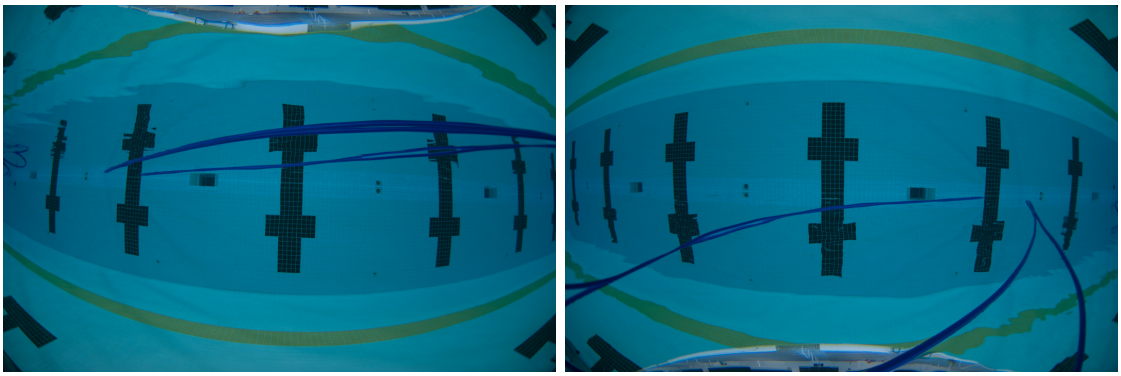
(b) Close-up showing uneven nature of pool floor on small scales.

Figure 4.4: U of A West Pool during cleaning in Sept. 2011.

This prompted the decision to perform a second swimming pool test, this time placing the cameras nearer to the underwater pool wall and using the tiles as targets for pixel mapping.

The timing of the second test included the advantage of being able to measure the dimensions of the West Pool while it was drained of water for its annual cleaning during the first week of September (Fig. 4.4a). This allowed the shallow end of the pool to be measured and mapped out a week before running the test with cameras. Measuring the pool features, such as the distance between yellow and green lines and the distance to the wall, was important for accurately interpolating the positions of pool tiles used in creating a pixel map for the cameras. Appendix A gives exact positions of the pool tiles and cameras used in the calibrations.

This test proceeded similarly to the one done in June, though all photos were taken in the shallow end. The cameras were first placed in a row about 3 meters away from the pool wall (Fig. 4.2). Here photos were taken of the pool wall, making use of the tiles as an already existing grid for calibration. These pictures were also used for the main triangulation test. The cameras were also rotated an angle $\psi = 180°$ around their z-axis so that they were upside down for a set of pictures in order to increase the



(a) Camera 1, no rotation (b) Camera 1, after 180° rotation around ψ axis.

Figure 4.5: Photographs from the second swimming pool test, used for camera calibration. The blue camera hoses are visible floating on the surface of the water. The limit of total internal reflection is also seen at the water's surface.

pixel space available for mapping.[3]

Once these pictures were taken, the cameras were moved to their new positions for a second triangulation test. They were placed facing each other, 3 meters apart (Fig. 4.2 and Fig. 4.6). A bright sticker on a metal pole was placed in view of all six cameras for one set of photos, and then moved for a second set. Photos were taken with a personal camera to record the positions of the sawhorses and triangulation target. This would allow their positions to be calculated later during analysis.

The target position is shown in Fig. 4.6. These pictures were analyzed and triangulation was performed, but it was not used in the final analysis. This was due to a lack of statistics from having only one target and systematic errors from having moved the cameras. Further uncertainties arose from deducing the position of the sticker on the metal pole. It contained uncertainties on the order of 1 cm since the pool tiles were not flat on the scale of the target base (Fig. 4.4b).

The pool tiles visible in Fig. 4.5a proved to be the most reliable targets for triangulation analysis. In addition to being regularly spaced and easily measured, they spanned a large portion of the cameras' field-of-view; 65% in the horizontally direction

[3]This extra information was not used in calibrations done in this work due to the difficulty of merging two photographs or performing two simultaneous fits. Focus instead went into understanding and improving the calibrations for the regularly aligned photographs.

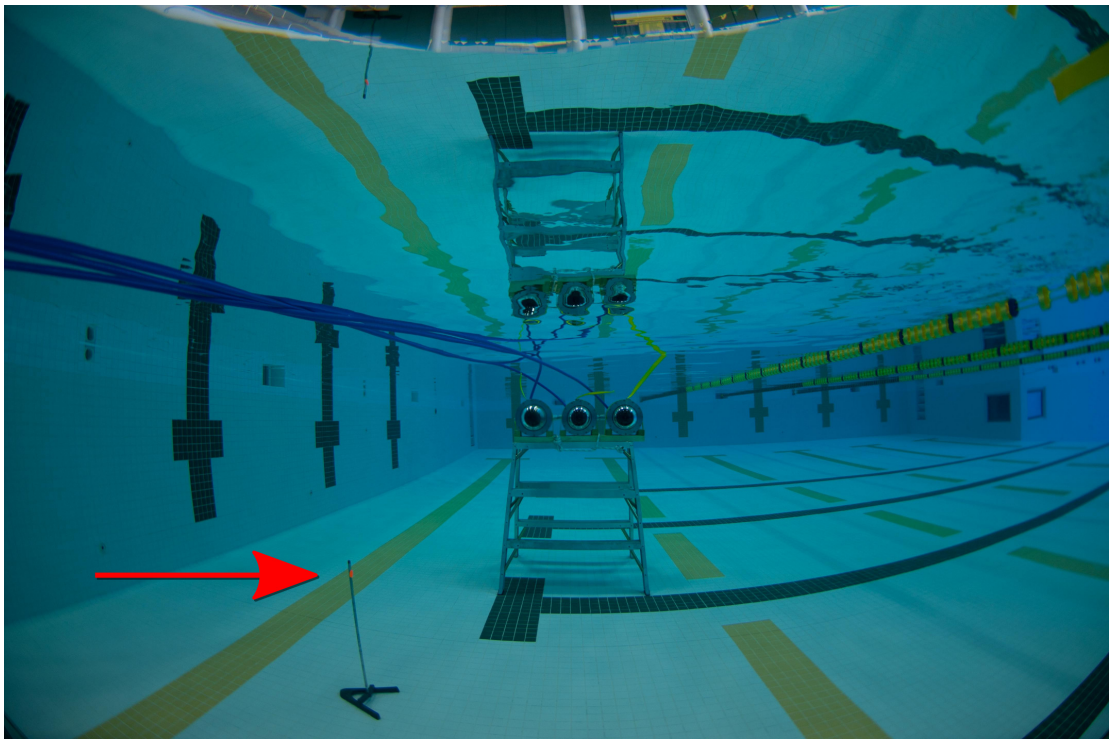


Figure 4.6: A view of the first triangulation target (red arrow) as seen by Camera 1.

and 25% in the vertical direction was mapped. This was an approximately 7-fold increase in covered area from the first pool test. The full triangulation analysis is detailed in Chapter 6.

4.3 From Swimming Pool to SNO+

To operate the cameras in SNO+ it will be necessary to translate the work done in the swimming pool at the University of Alberta to the detector cavity in SNO+. This will involve re-calibrating the cameras to map the larger field of view occupied by the AV (Fig. 4.7). The camera orientation[4] will have to be re-calculated for each camera, with the new positions given in Table 2.2. All other parameters will be recalculated for the new field of view, since triangulating outside of the mapped area leads to high

[4]Camera orientation relative to the global coordinate system is given by three angles: θ, ϕ, and ψ. Further explanation of these parameters and coordinates is given in Appendix A.

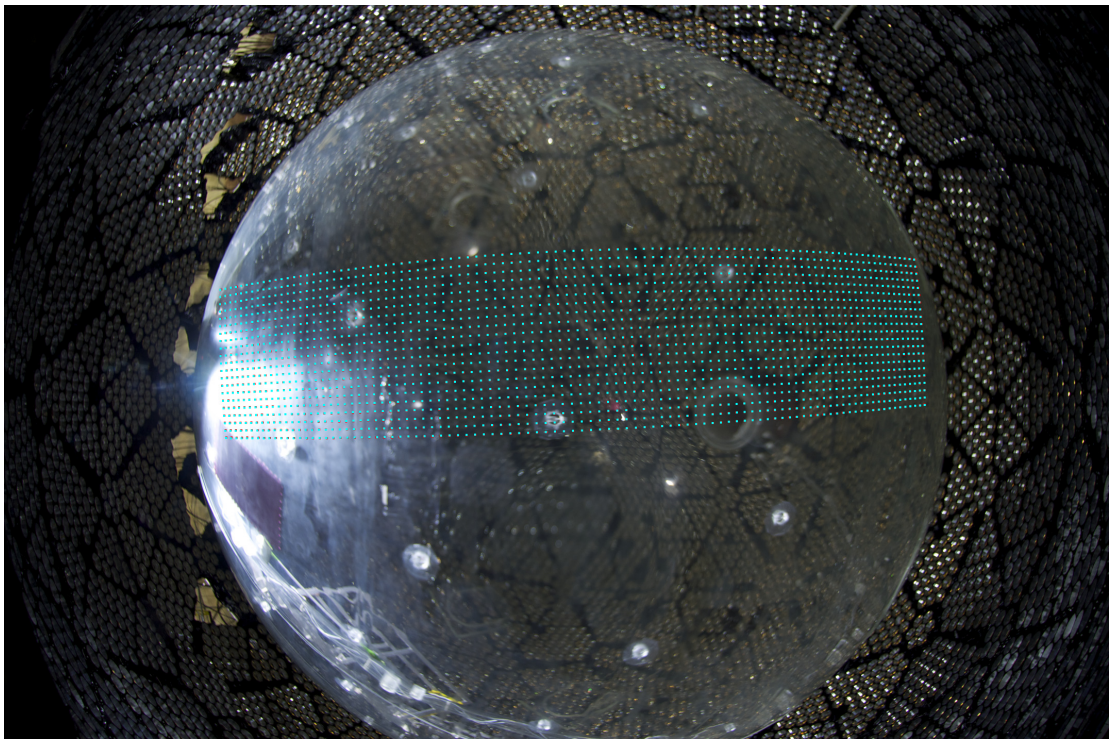


Figure 4.7: Calculated image points (cyan) drawn over actual image points (red, barely visible) as calculated using the best-fit 19-parameter model for Camera 4 are superimposed over a photograph of the SNO+ acrylic vessel. The cameras will have to be re-calibrated once installed to map the larger field of view.

inaccuracy (Fig. 6.2).

The most likely target candidate for the calibration in place of pool tiles will be the PMTs, as they are numerous, visible throughout the entire photograph, and have well-known locations. The procedure given in Chapter 7 will have to be adapted for the new environment and is still under development. One major adaptation will include changing the algorithm which sorts the target points (Sec. 7.3) to account for the triangular grouping of PMTs in contrast to the grid-like arrangement of the pool tiles.

Chapter 5

Calibration

To perform a triangulation with multiple cameras, it is first necessary to construct a mapping of object positions from a 3D global coordinate system to a 2D pixel space, or $f(X, Y, Z) \rightarrow (x_{pixel}, y_{pixel})$. This transformation is shown in Fig. 5.1.

This is done in two steps: first the object in the global reference frame is transformed into the camera's reference frame. In the camera frame, the camera sits at the origin. This transformation is done using Euler rotation matrices and is described in greater detail in Appendix A. Once the object is in the camera's reference frame, it can be transformed into the pixel coordinate system, which essentially projects the object onto the imaging plane of the camera. The geometry of this projection depends on the lens used by the camera, and is explained in Sec. 5.1.

A good calibration requires a large set of known points spanning the field-of-view of the camera. The width of this field-of-view depends on the application of the cameras, or in this case, the distance they will be from the AV in SNO+.

Once the calibration is complete, multiple cameras can be used to triangulate the position of an object. A minimum of two cameras is needed to perform a triangulation. However, accuracy increases with the number of cameras used, and a source placed anywhere in the AV will always be visible by all six cameras. The field of view mapped in the pool tests overlayed on the AV for reference can be seen on Fig. 4.7.

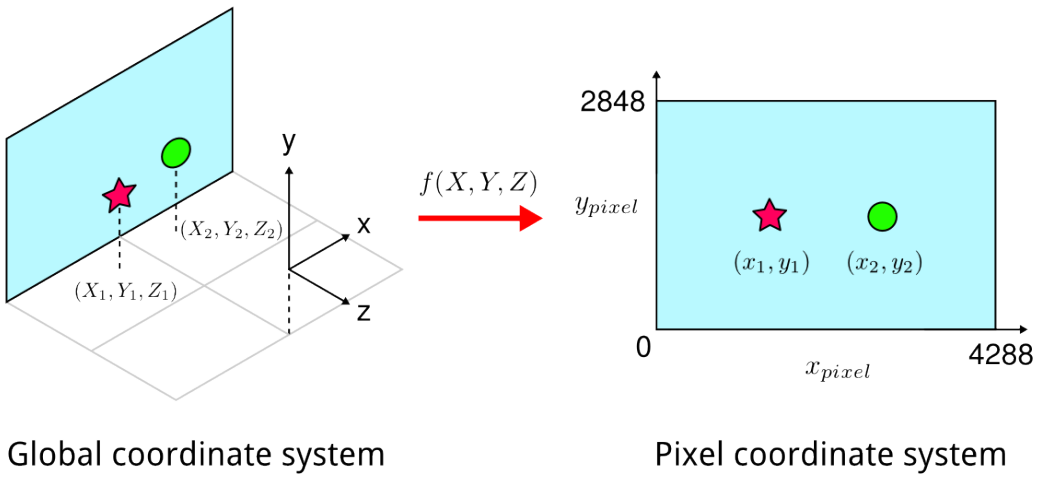


Figure 5.1: Calibrating the cameras requires predicting how an object's position in a global reference frame transforms to a coordinate in the pixel frame. These transformations are described in this chapter and in Appendix A.

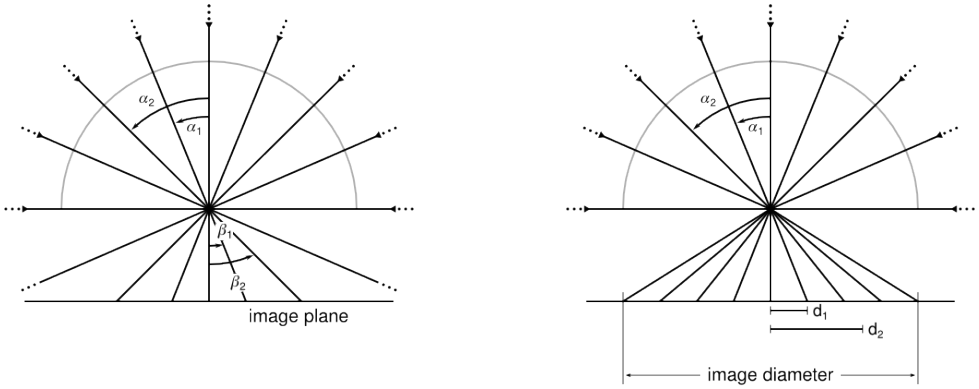
5.1 Projection Models

5.1.1 Central Perspective Projection Model

The central perspective projection model accurately describes a pinhole camera where the angle of projection of an incident light ray is proportional to its angle of incidence [27]. This geometry, shown in Fig. 5.2a, adequately models a camera with a standard lens. However, the fisheye lens used in this project has a field-of-view of 180° which, when using the central perspective model, would result in light rays incoming at 90° to be projected at infinity. Hence, the central perspective model cannot be used to mathematically describe the fisheye lens projection geometry.

5.1.2 Fisheye Projection Model

The basic geometry in the fisheye projection model is that the distance between the projection point and principle point (center point) is proportional to the angle of incidence of the incident light ray [27, 28]. As seen in Fig. 5.2b, an object with an angle of incidence of 90° would be mapped to a distance of R, or radius of the image.



$$\frac{\alpha_1}{\beta_1} = \frac{\alpha_2}{\beta_2}$$

(a) Central (Pinhole)

$$\frac{\alpha_1}{d_1} = \frac{\alpha_2}{d_2}$$

(b) Fisheye

Figure 5.2: Geometries of the central and fisheye projection models.

Similarly, an object with an angle of incidence of 45° would be mapped to a distance of $R/2$.

In order to map an object's 3D location to a camera's 2D pixel space, the object's location must first be transformed from the global coordinate system to the camera coordinate system. Here it will have a location of (X_c, Y_c, Z_c). This is done using an Euler rotation matrix:

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} X - X_0 \\ Y - Y_0 \\ Z - Z_0 \end{bmatrix} \tag{5.1}$$

where:

X_c, Y_c, Z_c = Object location in camera coordinate system

a_{ij} = Elements of Euler rotation matrix

X, Y, Z = Object location in global coordinate system

X_0, Y_0, Z_0 = Camera location in global coordinate system

A review of Euler rotation matrices as well as detailed explanation of the global coordinate system can be found in Appendix A. Once in the camera coordinate system, it can be shown [27] that a point $\vec{X_c} = (X_c, Y_c, Z_c)$ can be mapped to a pixel x' via the following relation:

$$x' = \frac{X_c}{|X_c|}\frac{2}{\pi}\frac{\tan^{-1}\left(\sqrt{\frac{X_c^2+Y_c^2}{-Z_c}}\right)}{\sqrt{\frac{Y_c^2}{X_c^2}+1}} \tag{5.2}$$

and similarly for y':

$$y' = \frac{Y_c}{|Y_c|}\frac{2}{\pi}\frac{\tan^{-1}\left(\sqrt{\frac{X_c^2+Y_c^2}{-Z_c}}\right)}{\sqrt{\frac{X_c^2}{Y_c^2}+1}} \tag{5.3}$$

The scaling factor of R present in [27] is removed here and added later as a separate scaling factor for x' and y'. Equations 5.2 and 5.3 are constructed to include the geometry of Fig. 5.2b, namely the relation between incidence angle and projection on the imaging plane,

$$\frac{\alpha_1}{d_1} = \frac{\alpha_2}{d_2} \tag{5.4}$$

After conversion to (x', y'), it is necessary to include the parameters $\vec{d}$ which will map the distortions present due to the refraction at the air-plastic interface of the dome, imperfections in the dome itself, and lens defects.

$$\begin{aligned} x_{calc} &= g_x(x', y'; \vec{d}) \\ y_{calc} &= g_y(x', y'; \vec{d}) \end{aligned} \tag{5.5}$$

These equations were incorporated into a `Minuit2` [29] fit to find the best parameter set $\vec{d}$. This set was defined as the one that would minimize the square deviation:

$$\chi^2 = \sum_{i=1}^{i=N} (x_{calc,i} - x_{pic,i})^2 + (y_{calc,i} - y_{pic,i})^2 \tag{5.6}$$

where χ^2 is calculated assuming errors of 1 for x and y and (x_{pic}, y_{pic}) are object locations in pixels as they appear on the picture. The fitter will generate coordinates (x_{calc}, y_{calc})

for each of N points used in the fit using the parameter set $\vec{d}$. The creation of the parameter set $\vec{d}$ is defined in Sec. 5.2.

5.2 Creation of the Distortion Model

Choosing a model with the best parameters required a systematic approach. The general procedure was to add parameters to $\vec{d}$ of Eq. 5.5 as a function of x' and y' and then observe the resulting residuals in x and y, in addition to the total square residual χ^2 from Eq. 5.6. The total residual represented the overall goodness of fit.

The shape of the residuals in x and y is what guided the choice of the next parameters to add. The residuals for all the cameras were compared after each fit. Fig. 5.3 shows plots of x- and y-residuals for Camera 4 as they vary with the number of parameters added to the fitting functions, Eqs. 5.5. These plots guided the choosing of parameters for the model. Fig. 5.3a shows the residuals of the data set modeled with:

$$\begin{aligned} x_{calc} &= x_0 + x_1 x' \\ y_{calc} &= y_0 + y_1 y' \end{aligned} \tag{5.7}$$

These parameters center and scale the object locations, respectively. As can be seen in Fig. 5.3b, this works well for the y-direction, but leaves an x-dependent residual in x. Expanding to a third-order polynomial gives:

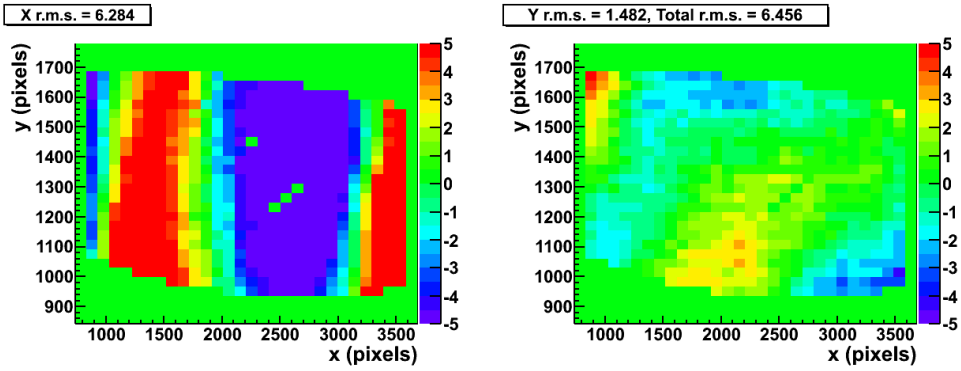
$$\begin{aligned} x_{calc} &= x_0 + x_1 x' + x_2 x'^2 + x_3 x'^3 \\ y_{calc} &= y_0 + y_1 y' + y_2 y'^2 + y_3 y'^3 \end{aligned} \tag{5.8}$$

This largely eliminates the x-dependent residual in x in Fig. 5.3a, leaving residuals x as a function of y and residuals in y as a function of x, shown in Fig. 5.3b. Neither of these residuals can be removed with further expansion of the polynomial model used thus far. To act on this, a set of radial distortion parameters are added:

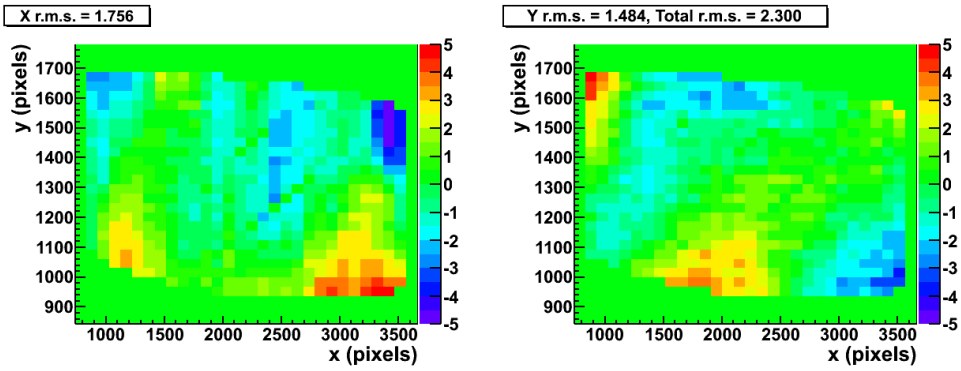
$$\begin{aligned} x_{calc} &= x_0 + x_1 x' + x_2 x'^2 + x_3 x'^3 + k_1 r^2 + k_2 r^4 + k_3 r^6 \\ y_{calc} &= y_0 + y_1 y' + y_2 y'^2 + y_3 y'^3 + k_1 r^2 + k_2 r^4 + k_3 r^6 \end{aligned} \tag{5.9}$$

where

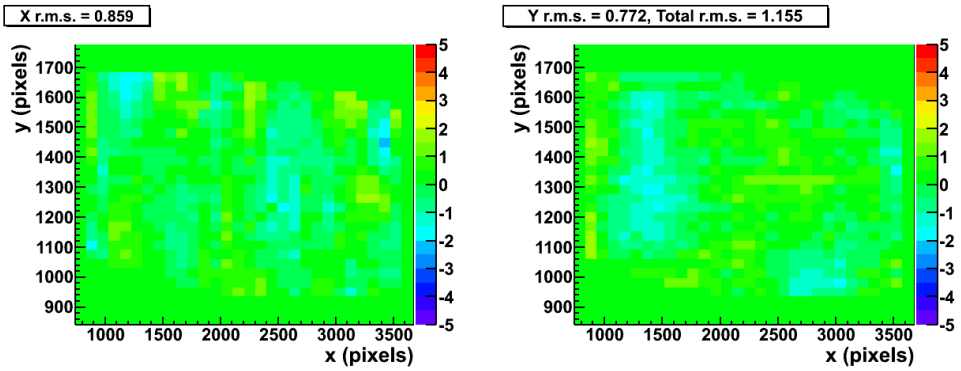
$$r^2 = (x_1 x' + x_0 - x_r)^2 + (y_1 y' + y_0 - y_r)^2$$



(a) 10 parameter model (Eq. 5.7). Note: the points at zero in the central negative region on the x-histogram are artifacts of binning.



(b) 14 parameter model (Eq. 5.8).



(c) 19 parameter model (Eq. 5.9).

Figure 5.3: The left column shows x-residuals while the right column shows y-residuals as a function of pixel location for Camera 4. The residual was calculated as $(x_{calc} - x_{pic})$ and $(y_{calc} - y_{pic})$ for x- and y-residual plots, respectively. Note: points beyond the z-axis range are drawn with the same color as the respective minimum or maximum.

The radial distortion parameters, being a function of both x' and y', manage to effectively reduce the remaining residuals when added to the polynomial model. The remaining residuals for Camera 4 are now on the order of a pixel, as seen in Fig. 5.3c.

5.3 Calibration Conclusion

Applying the 19 parameter model (Eq. 5.9) to all six cameras resulted in the residuals shown in Table 5.1. Each camera was fit with the same equation, with individual distortion parameters saved for later triangulations.

RMS Residuals (pixels)						
Camera	P	1	2	3	4	5
x	2.01	1.32	6.14	3.27	0.86	1.06
y	1.73	1.31	5.77	2.64	0.75	0.80
total	2.65	1.86	8.43	4.20	1.14	1.33

Table 5.1: Residuals from fitting with the 19 parameter model (Eq. 5.9).

Camera 4, which was used to produce the plots in Fig. 5.3, typically had the smallest residuals of all six cameras. Camera 5 was also well behaved. Cameras 2 and 3 still require additional parameters to minimize their residuals and bring them on par with Cameras 4 and 5. This works remains to be done at the time of writing of this thesis. It is believed that through careful addition of parameters the high residuals in Cameras 2 and 3 can be reduced through the same process shown in Fig. 5.3 with Camera 4. The challenge remains to conduct a thorough test of a wide range of parameters to see which can mimic the shape of the remaining residuals.

Chapter 6

Triangulation

6.1 Triangulation Setup

After the cameras have been calibrated, they can be used to triangulate the position of an object. This was tested using pictures and measurements taken during the second swimming pool test in September, 2011. The same pictures were used for the calibration and triangulation, with the intersections of pool tile grout being used as targets for both procedures.

The first advantage of this method was keeping variables such as camera position and orientation constant from one test to the next, thus reducing the systematic errors in the procedure. The second advantage was that the positions of the pool tiles were already well-defined from being measured when the pool was dry, allowing for precise calculation of residuals and better understanding of error.

For reference, a sketch of approximate camera locations in the global coordinate system is given in Fig. 6.1. Exact locations of cameras with a detailed explanation of how the global coordinate system was constructed is given in Sec. A.2 of the appendices.

Three separate triangulations were done, using 30 points within the calibrated region of the camera's pixel space, 30 points just outside of the calibrated region, and 6 points farther outside of this region (Fig. 6.2). These points were chosen such that they could be resolved by all six cameras, and thus appear in the central region between the two camera sawhorses.

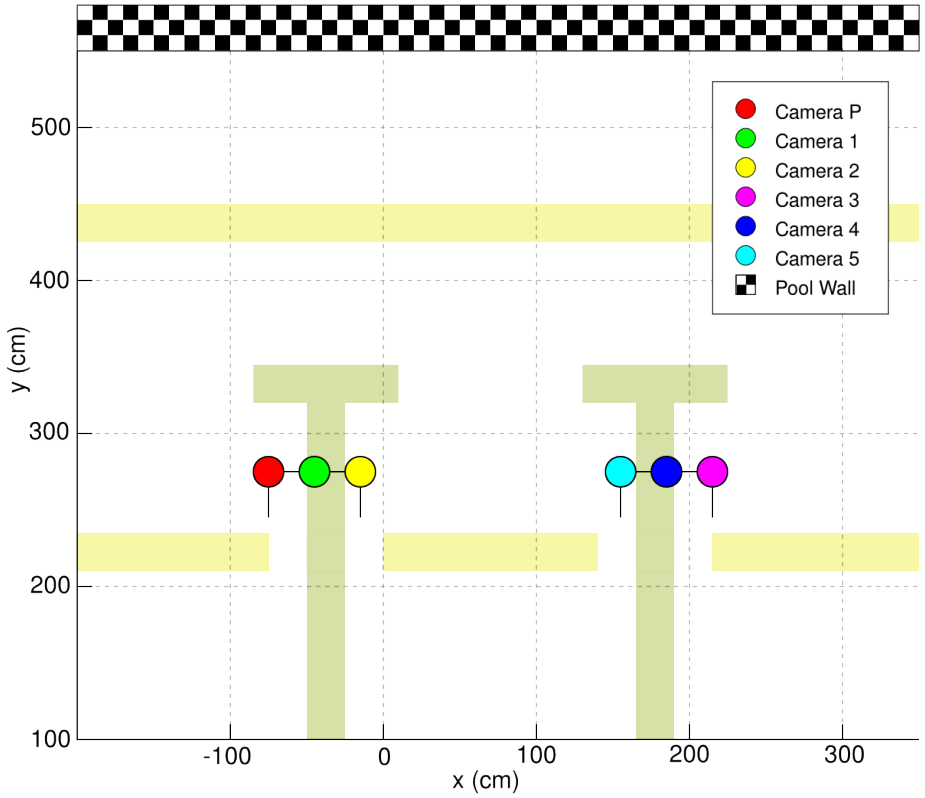


Figure 6.1: Bird's eye view showing the arrangement of cameras during the triangulation done in the second swimming pool test. The regularly spaced tiles of the pool wall were used as both calibration and triangulation targets. A detailed schematic with exact camera locations can be found in Sec. A.2.

6.2 Algorithm

The algorithm for triangulation is similar to that of calibration, though different parameters are held constant or allowed to float. To triangulate the location of a single point it is necessary to input its pixel location for each camera, and the set of distortion parameters $\vec{d}$ for each camera as described in Eq. 5.9. The result of the minimization gives the coordinates (X, Y, Z) which best translate into the pixel coordinates as seen by each camera. This is in contrast to the calibration procedure which takes

3D positions as fixed input and returns the distortion parameters which were allowed to float. The pixel position of each point is held constant in both procedures.

The quantity being minimized in this algorithm is the residual between each calculated (x_{calc}, y_{calc}) pixel position and the observed (x_{pic}, y_{pic}) position, summed over all cameras:

$$\chi^2 = \sum_{i=1}^{i=N_{cam}} (x_{calc,i} - x_{pic,i})^2 + (y_{calc,i} - y_{pic,i})^2 \tag{6.1}$$

This is similar to Eq. 5.6 except here the summation is over the number of cameras N_{cam} for a single point as opposed to the number of points N for a single camera. This procedure is repeated for each point triangulated in a photograph.

6.3 Residuals

Residuals for the first 30 point fit (1^{st} target set in Fig. 6.2) of the parameters in Eq. 5.9 are shown in Fig. 6.4. Due to the imperfections in its dome, excluding Camera 2 from the fit produces smaller variations in the residuals (Fig. 6.4b). The residuals are tabulated in Table 6.1. These results show that inside the calibrated region the triangulations are accurate to within 5 mm at a distance of 3 meters. Precision is dominated by statistical errors in this region. This extrapolates to an expected accuracy of 1.5 cm at 9 meters within the AV in SNO+.

Another set of 30 points (2^{nd} target set in Fig. 6.2) was chosen to lie just outside of the calibrated zone. Horizontally it was within the limits of the calibrated region, but vertically it was located at a distance of 110%–150% past the calibrated region's height. The residuals from this test are provided in Table 6.2. Here, it is seen that the results exhibit a systematic drift. It is most prominent in the x and z directions, corresponding to the horizontal and vertical, respectively. This is due to the fact that the fitting function will calculate pixel positions that diverge from the actual positions once past the calibrated region.

In the final test, points were taken far outside of the calibrated region in the vertical direction. Here, the residuals are dominated by systematic errors caused by extrapolation of the fitting function.

Residuals within calibrated region (cm)				
Residual	All six		Exclude Camera 2	
	μ (cm)	σ (cm)	μ (cm)	σ (cm)
x	0.0930	0.2783	0.0932	0.2507
y	-0.0489	0.3966	-0.1049	0.2654
z	-0.0325	0.2203	0.0147	0.1377
sum	0.0109	**0.5322**	0.1411	**0.3901**

Table 6.1: Average residual values with standard deviation for 30 points within the 1st target set (calibrated region).

Residuals just outside calibrated region (cm)				
Residual	All six		Exclude Camera 2	
	μ (cm)	σ (cm)	μ (cm)	σ (cm)
x	0.2821	0.2084	0.2905	0.1864
y	0.2830	0.2776	0.2369	0.2144
z	-0.4012	0.2548	-0.4890	0.1827
sum	0.5662	**0.4306**	0.6161	**0.3378**

Table 6.2: Average residual values with standard deviation for 30 points within the 2nd target set (just outside calibrated region).

Residuals far outside calibrated region (cm)				
Residual	All six		Exclude Camera 2	
	μ (cm)	σ (cm)	μ (cm)	σ (cm)
x	0.4274	0.3472	0.3261	0.1202
y	5.1312	1.0323	3.6869	0.6044
z	1.3681	0.4123	-0.6450	0.1343
sum	5.3276	**1.1649**	3.7571	**0.6307**

Table 6.3: Average residual values with standard deviation for 6 points within the 3rd target set (further outside calibrated region).

6.4 Summary

The residuals lead to the conclusion that for triangulation to be done accurately, the entire pixel space in which the triangulation is to be performed must be calibrated and fitted with a pixel mapping. When triangulating points outside of the calibrated region, systematic errors grow high. Inside of the calibrated region, however, the errors are dominated by statistics. In this area an accuracy of 0.1 ± 0.3 cm can be achieved for each of the three dimensions.

The geometry of the cameras in SNO+ will further balance out the residuals between the three dimensions. In these swimming pool tests the y dimension tended to have the highest systematic drift due to the setup of the cameras. In SNO+ this systematic effect will not be present since the cameras will surround the source being photographed rather than view it along one axis.

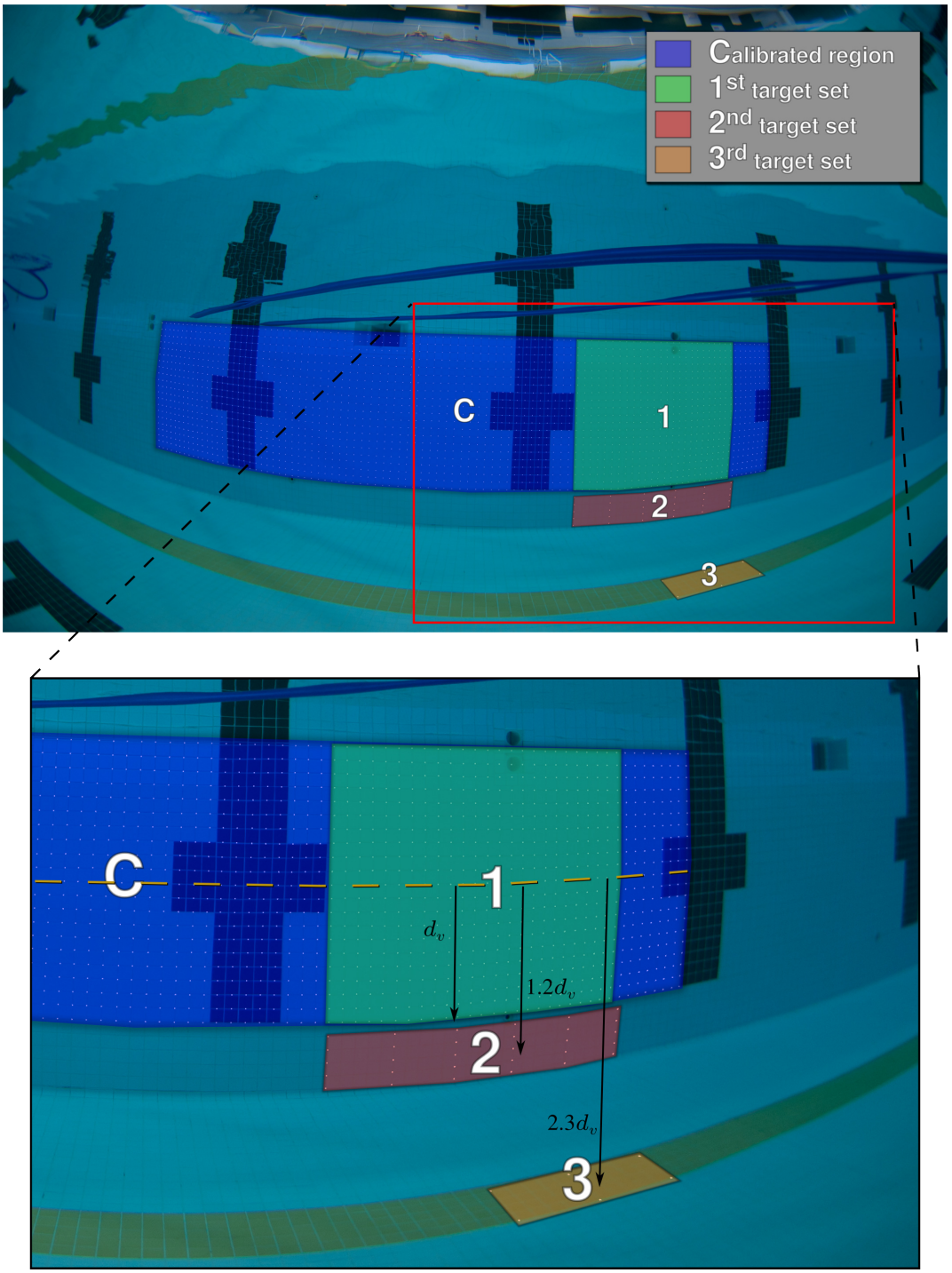
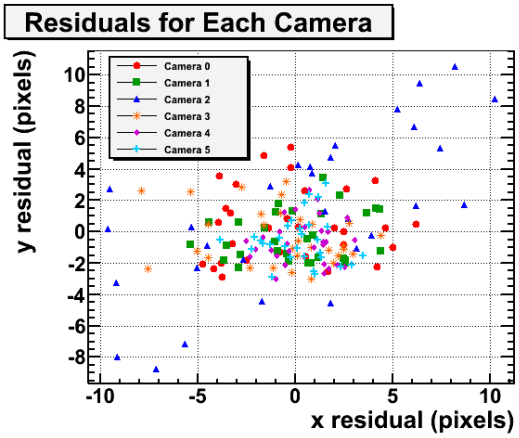
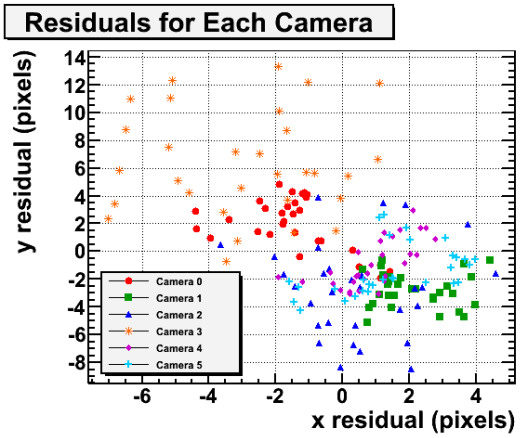


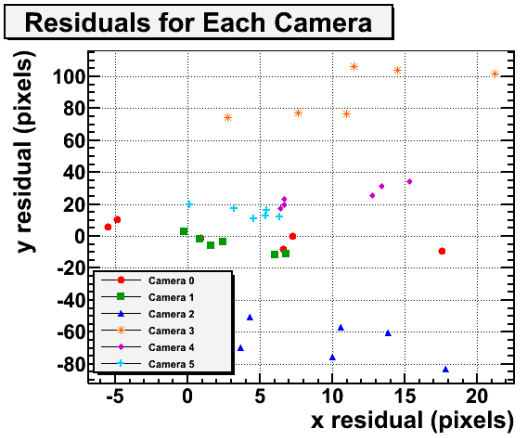
Figure 6.2: Calibrated regions and triangulation regions for Camera 0. Here d_v is the height from the center of the calibrated region to its edge. The center of the second region is located at approximately $1.2d_v$ and the center of the third region is at approximately $2.3d_v$.



(a) Pixel residuals for points inside the calibrated region (1st target set).

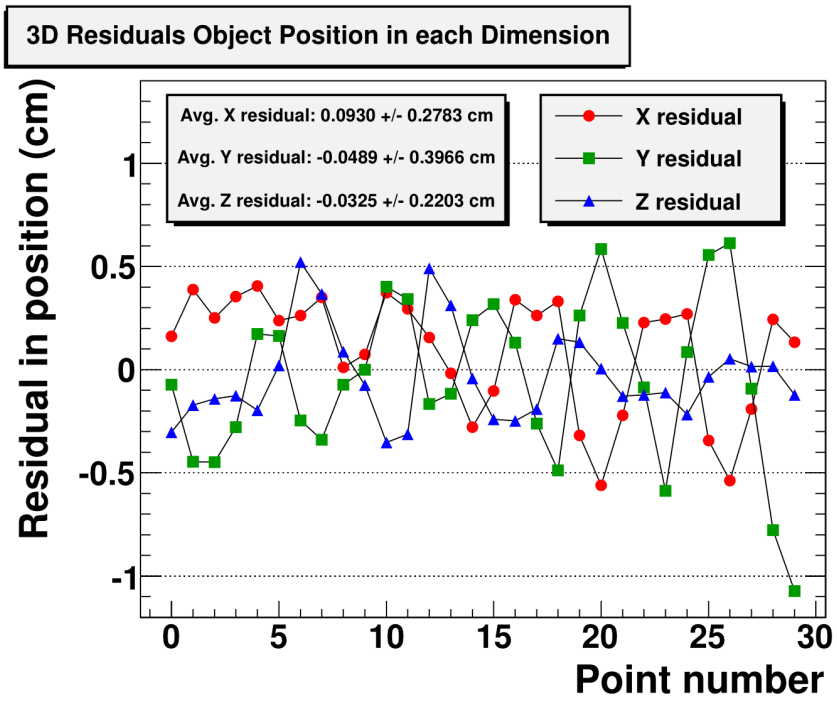


(b) Pixel residuals for points just outside the calibrated region (2nd target set).

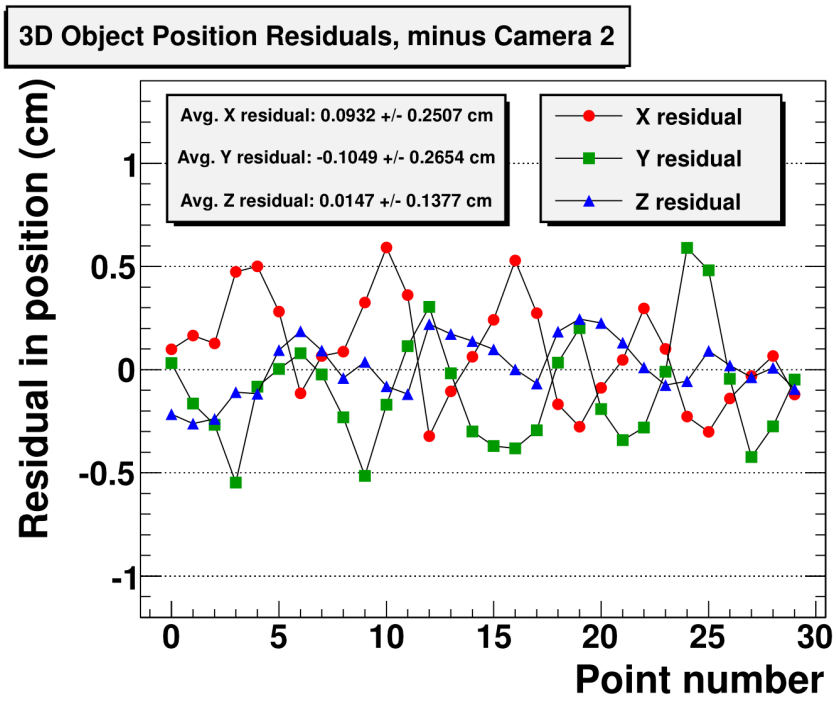


(c) Pixel residuals for points far outside the calibrated region (3rd target set).

Figure 6.3: Pixel residuals for different triangulation data sets showing systematic drift from individual cameras. Fig. 6.3a shows how Camera 2 tends to oppose the location suggested by the other 5 cameras when triangulating the position of a point. Camera 3 does this to a lesser extent. This is due to their fits having a residual that is 3–7 times higher than the other cameras. Figs. 6.3b and 6.3c show separation of different cameras' predictions as systematic effects begin to dominate outside of their calibrated zones. Moving a target outside of this zone causes a different effect based on each camera's individual calibration parameters.



(a) Distance residuals using all six cameras.



(b) Distance residuals excluding camera 2.

Figure 6.4: Residuals of triangulations performed for 30 points within the calibrated region (1^{st} target set). Camera 2 is excluded for the second triangulation, resulting in lower residuals. This shows the effect its distortions have on the analysis.

Chapter 7

Procedure of Calibration and Triangulation

This chapter focuses on the technical detail of performing calibration and triangulation with the camera system and is intended to be a guide for a user operating the system.

Various pieces of code are used to process a photograph to calibrate a camera, from extracting target points in the picture to fitting these points with various functions until the best fit is found. The parameters that make up this fit would then allow the camera system to perform accurate triangulations.

Some additional user interaction is necessary in this process, since operations like picking out targets in a photo require "guiding" the computer at first. The steps undertaken in the entire process will be detailed in this chapter.

All code used in this section was written by the author including `image2hist.c`, `image2spectrum.c`, `slope_sort`, `Intrinsics_Solve_final`, and `Position_Solve_final`. It is available in Appendix B and as an electronic attachment to the thesis.

7.1 Selecting Target Points

The target points used for calibration need to be chosen as a set of features with each position known spanning the largest possible field of view (FOV) in a photograph.

They do not necessarily have to be the same features for all six cameras.

In the June swimming pool test (Sec. 4.1) the target features were the reflectors on the black tarp that was submerged in the pool. The difficulty with using these features is that though they were resolvable, they did not span a wide area of each camera's FOV as a result of their being placed too far away. For the September swimming pool test (Sec. 4.2) the tiles on the swimming pool wall were chosen because they spanned the entire FOV for each camera and were close enough to be resolved. To recalibrate the cameras when they are deployed in SNO+, targets such as the PMTs or the centers of PMT clusters could be chosen as calibration targets. They span the entire FOV of each camera, and the position of each PMT is well known.

The pixel position of an object in a photograph needs to relate to its position in the environment in order to construct a 3D $\rightarrow$ 2D mapping. For the tests done in September, measurements were made inside the pool while it was empty of water. After measuring the distances between pool features such as formations of yellow and green tiles, the location of each tile could be interpolated and stored in a global coordinate system. The coordinate system may be seen in Fig. 6.1.

The targets used for the calibration were chosen to be the intersections of pool tile grout, the "glue" in between the tiles. The thin lines were easy to trace and map out in a graphics editor, and the locations of the intersections were calculated in the global coordinate system. These locations were saved in a .txt file, with points ordering left-to-right, bottom-to-top when viewed in a photograph. The pixel position of the points were ordered in the same way. For reference, here are the first few lines of the 3D coordinate file for Camera 1 in the September pool test, in space-separated (x, y, z) format:

```
$ head -n3 3d_cam.txt
-305.6 545.7 27.12
-300.517 545.7 27.12
-295.434 545.7 27.12
```

To pick out the corresponding points from a photograph, the computer had to first be guided to find the targets. This was done by using Photoshop CS5 and GIMP graphics editing software to draw lines on the pool tile grout and then extract the

intersections of these lines. GIMP can be used for the first step instead of Photoshop, but the paths stored in GIMP are harder to copy and manipulate after they have been set. Photoshop is simpler to work with in this case.

Here are the steps taken in guiding the computer towards finding the calibration targets in a photograph:

1. Open the photograph of the target points in Photoshop CS5. Using the **Path** tool, draw paths across the pool tile grout - first a horizontal path, then a separate vertical path. Once the paths are drawn, **Stroke** each path onto a separate layer with a 3 pixel wide brush (Fig. 7.1). Using two different colors makes it easier to distinguish between paths. *Tip:* Paths may be copied and used on other photographs with slight adjustment to avoid having to redraw them.

2. Save the Photoshop file and re-open it in GIMP. Using the **Layers** dialog, select the topmost layer and set **Mode** to **Difference**. This will cause the intersection of lines to stand out in a different color, as shown in Fig. 7.1b. (The **Subtraction** and **Darken Only** modes will produce similar effects and may be also used.)

3. Hide the background layer by clicking its eye-shaped icon in the **Layers** dialog, leaving only the lines visible. Create a new layer by selecting **Layer** → **New Layer From Visible**.

4. Hide all of the original layers, leaving only the recently created layer visible. To isolate the intersections, select that layer and then open **Colors** → **Threshold**. Adjust the sliders until the intersections appear as white dots and the rest of the lines are black (Fig. 7.2a).

5. Finally, fill in the rest of the background with the **Bucket Fill Tool**, using the **Fill Similar Colors** option (Fig. 7.2b). **Save** the working file and select **Save a Copy** to save the top layer as a .png file.

If we start with the original photograph `Camera1_09.jpg` we get the Photoshop working file `Camera1_09.psd` and our saved copy of the white dots on black background, `Camera1_09_dots.png`.

A 3 pixel wide brush stroke is used to draw the paths because it is the smallest possible brush stroke that preserves symmetry. Theoretically, a 1 pixel wide stroke would be ideal for precision, creating line intersections of exactly 1 pixel in area. The problem with this is that lines sometimes miss each other (due to aliasing when stroking the path) causing gaps to appear in the grid of points. A stroke of 2 pixels causes a row of pixels to be drawn either above or below the path on the photo, leading to asymmetry. A 3 pixel stroke creates symmetry in the lines that are drawn, and the resulting intersection points are still small at approximately 9-11 pixels in area.

When creating these files in environments where the geometry does not follow horizontal and vertical lines as pool tiles do, a different method may be chosen to create the points on the photo. For SNO+, PMTs could be chosen as the calibration targets[1]. Points could be manually applied onto a separate layer over a photograph of SNO+ PMTs, and this layer taken directly to step 4 to create the white-on-black photograph.

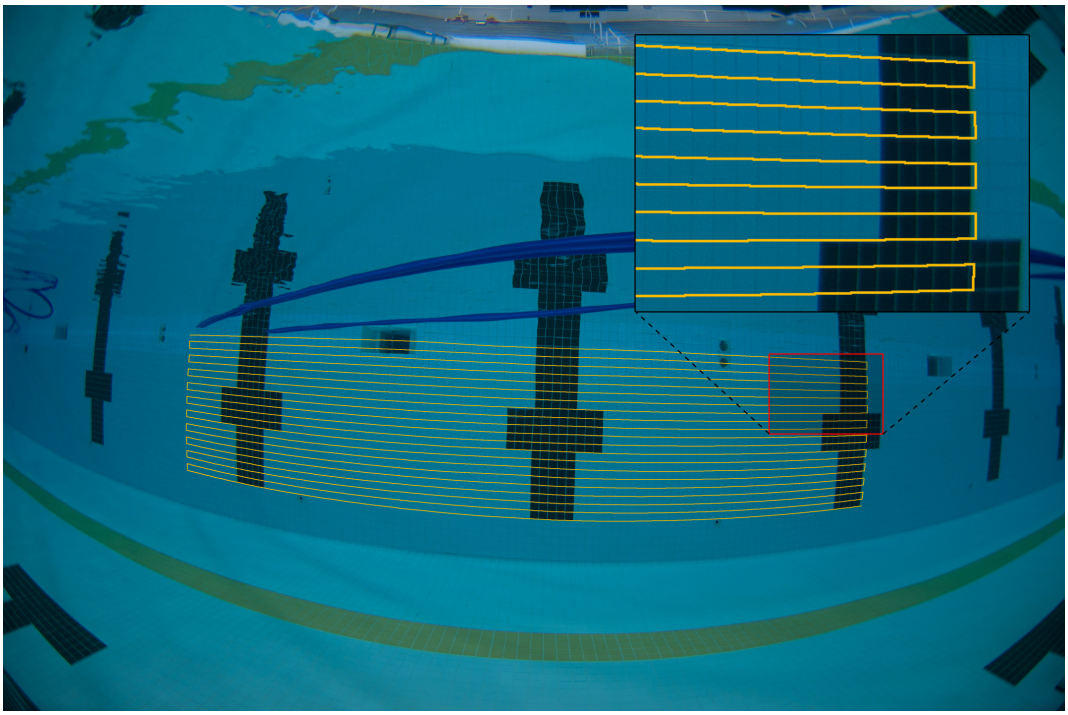
7.2 Extracting Positions Using ROOT

Once the black and white photograph has been generated, the points can be easily extracted by the TSpectrum2 routine in ROOT [29]. This is done by opening ROOT in a terminal and running the macro `image2hist.c` on the photograph that was just generated.

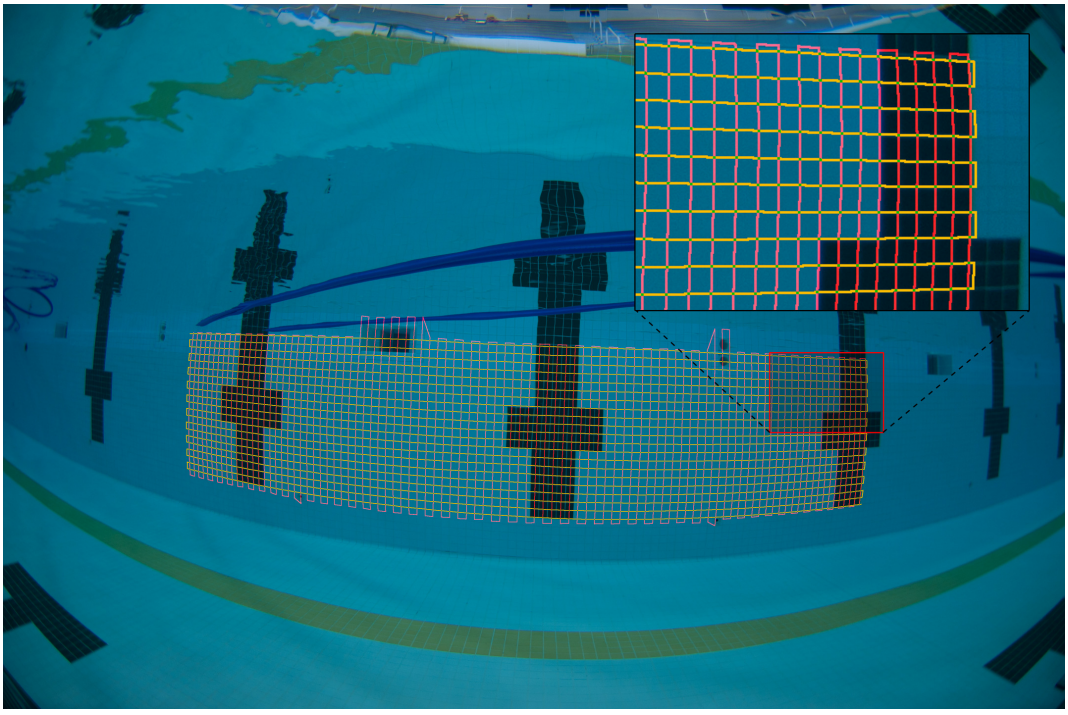
```
$ image2hist(Camera1_09_dots.png)
```

This creates a file titled `Camera1_09_dots.root`. The macro takes an image file and converts it to a two dimensional histogram. Since each pixel of the image contains information on its red, green and blue content, these values can be added together with coefficients to get a grayscale version of the picture. This single grayscale value of each pixel is stored in the histogram, creating peaks in place of the white dots. The pixels have a one-to-one correspondence with the bins in the histogram, preserving the original image dimensions.

[1] Calibrating the installed cameras would have to be done once the AV is filled with scintillator and the cavity is filled with water, since the similar indices of refraction of the fluids and acrylic would lead to less distortion of ray geometry. In fact, calibrating in air and then again after the cavity fill can give a handle on the effect that the acrylic vessel has on refraction.

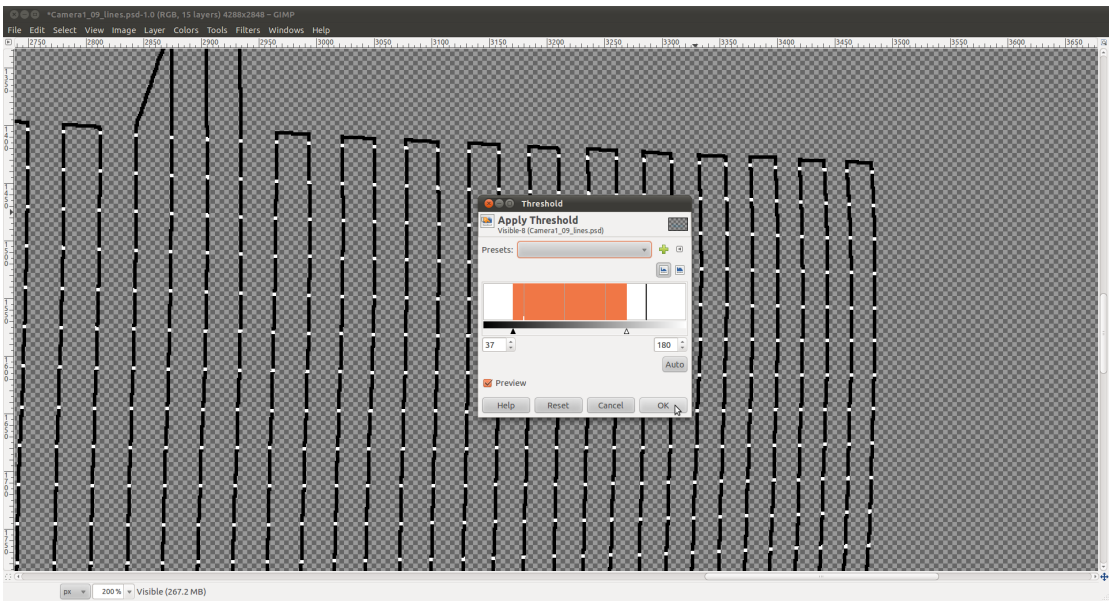


(a) Horizontal lines drawn first.

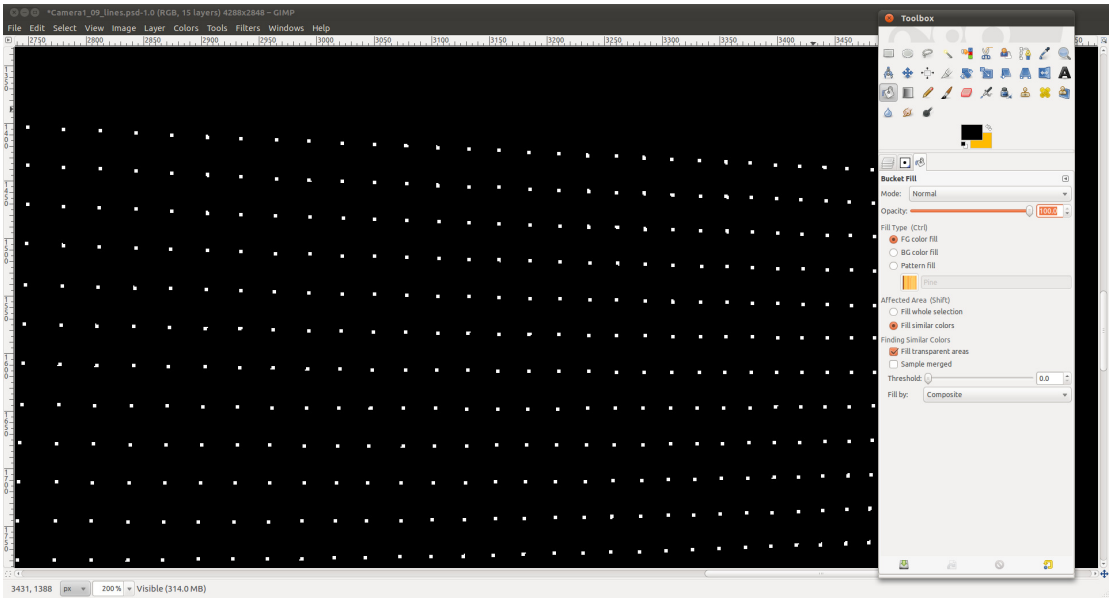


(b) Vertical lines placed overtop, intersections visible as green squares.

Figure 7.1: Example showing the blending of horizontal line layer with a vertical line layer in GIMP using the **Difference** mode.



(a) Using the **Threshold** command to make the intersections appear as white dots.



(b) Applying a **Bucket Fill** to turn the entire background black.

Figure 7.2: After creating a new layer from the overlapping horizontal and vertical lines, the layer is modified to create a high-contrast photograph of the target points.

Using the peak-finding macro

```
$ .L image2spectrum.c
$ image2spectrum(2,"Camera1_09_dots.root",3,10,1)
```

The peaks in this histogram can then be found with ROOT's two-dimensional peak finder, TSpectrum2. This is done by running the macro `image2spectrum.c` on the root file with the following arguments:

```
$ image2spectrum(int dim, string "file", double sigma, double thresh, bool
    write)
```

The first argument `dim` takes an input of either 2 or 3, and changes only whether the histogram is displayed in 2D or 3D format. It has no effect on the stored histogram. The second argument takes the name of the .root file generated by `image2hist.c` with the name in quotations. The third and fourth arguments are inputs to the TSpectrum2 function inside the macro. Inputting `sigma` will tell it the width of the peaks for which it should be looking in units of pixels, and `threshold` is a percentage of the highest peak (up to 100) below which the function will not search for peaks. It is used to reduce false peaks caused by background, but if the image was processed according to Sec. 7.1 to eliminate background, the default value of 10 will suffice. The boolean value `write` decides if the peaks found by TSpectrum2 will be written to file or not.

Here the macro is used on the histogram previously generated. Since most of the dots have a width of 3 units, setting `sigma` to 3 will typically find all of them.

The macro will output how many peaks were found along with a plot of the peaks, providing a visual check that the peaks were found correctly. The locations of the peaks are written to file in (x_{pic}, y_{pic}) format. In this case, the file is named `Camera1_09_dots.txt`.

The `sigma` and `thresh` values can be changed if the picture contains, for example, PMTs that all have a bright central region. The PMTs can then be found directly with this macro, thus allowing a user to bypass steps 1-5 in Sec. 7.1. It can take some creativity and trial-and-error to prepare a picture that can be passed successfully

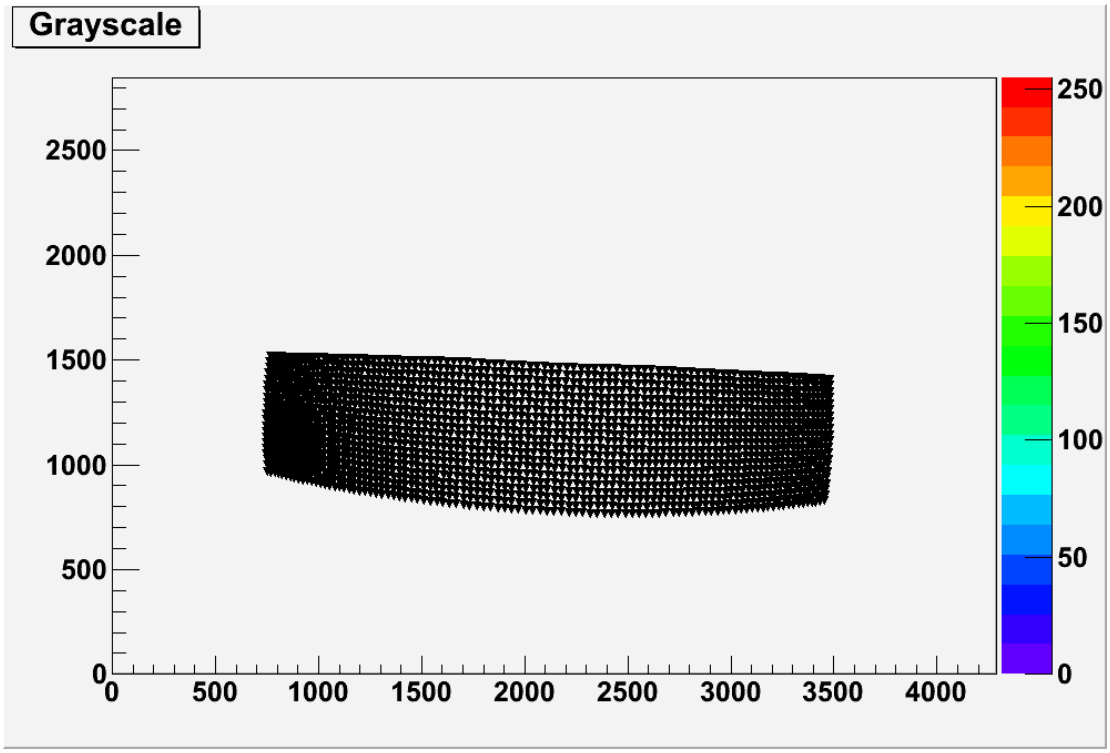


Figure 7.3: TSpetrum2 plot showing all 2037 peaks found overlayed on the histogram `Camera1_09_dots.root`.

through `image2spectrum.c` with all the peaks found; Sec. 7.1 shows only one way of doing it for the case of pool tiles being used as targets.

7.3 Sorting Points

Now that we have found the points, plotting them in Fig. 7.4 shows that they are roughly sorted left-to-right, but are sorted unpredictably in the vertical direction. For the mapping to succeed, the pixel positions must follow the same ordering as the measured 3D locations of these points. The points are stored as space-separated values like the 3D position file `3d_cam.txt`.

Unsorted (x, y) pixel positions

```
$ head -3 Camera1_09_dots.txt
```

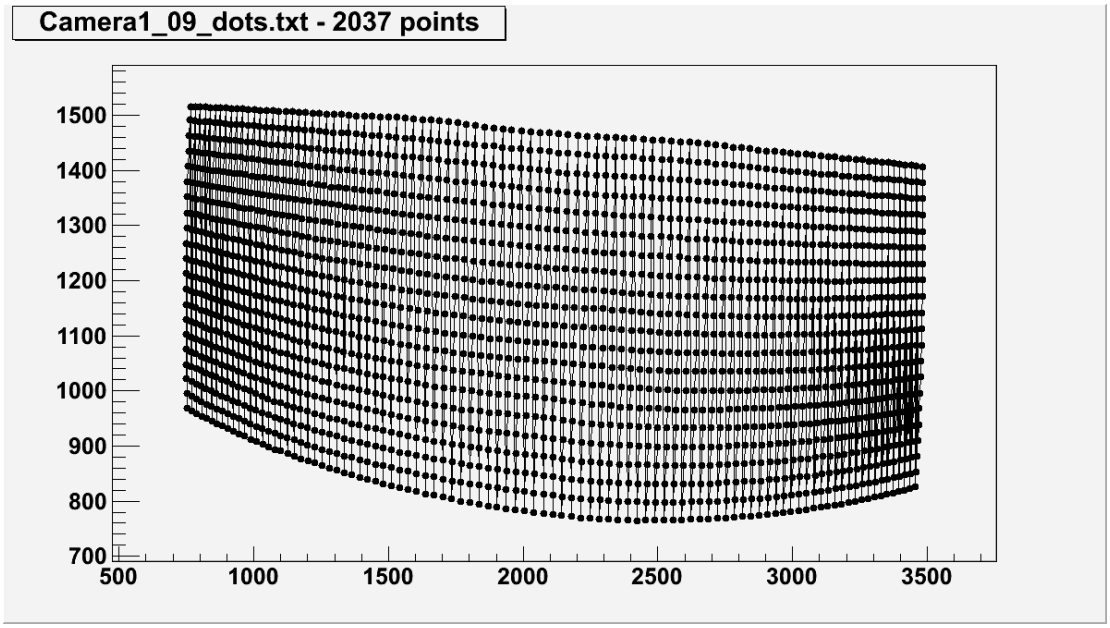


Figure 7.4: Output points from `image2spectrum.c` before sorting. There is a single line connecting them, but the ordering of points is hard to predict or reproduce. The points must be sorted before they can be used.

```
749.038025 1075.972412
749.011230 1102.950684
749.011292 1129.950684
```

The 2D pixel file must be modified to match the order of the 3D position file. To achieve cohesion between the two files, we apply a sorting algorithm to the pixel file.

The algorithm `slope_sort.cpp` was written specifically to sort points that appear in straight rows and columns, and later modified to account for the fisheye effect on the straight lines. This modified version requires more input from the user and can take several attempts to succeed in sorting points, but there are methods in place that can help guide the user to the correct choice of input parameters.

The program `slope_sort` is run in a terminal with the following input:

```
$ slope_sort Camera1_09_dots.txt 97 10 0.2 3000
```

The first argument to the program is the name of the file containing the unsorted pixel positions, here being `Camera1_09_dots.txt`. The second argument, 97, is the number of points per row. As the program scans from left to right, this number dictates when to return to the left of the picture to start scanning again. This makes it necessary that the each row contain the same number of points if the file is to be ordered with this algorithm.

The third argument in the example, 10, is the "x-exclusion width". This is the number of pixels in the x-direction that are skipped when searching for the second point in a row, and is dependent upon the spacing between points. It prevents the scan from hopping into a higher row at the beginning of a line.

The fourth argument, 0.2, is the "y-tolerance" variable. It prevents the scan from hopping into a higher row once more than 2 points have been scanned. Using the slope between the n and $n-1$ points, the algorithm can narrow down where to find the $n+1$ point. A cone of allowed points, starting at the n^{th} point, opens with an angle proportional to the y-tolerance (Fig. 7.6) along the vector $\left|\vec{d}_n - \vec{d}_{n-1}\right|$. This parameter needs to be tuned correctly, as choosing too large of a y-tolerance will create a very large cone of accepted points that may cause the scan to jump to a higher row. However, choosing a y-tolerance that is too small can cause a point in a row to be skipped (Fig. 7.5). Both instances will lead to errors that halt the sorting program due to internal memory errors[2].

The fifth and final parameter is the "stop-and-draw" parameter. This is used for debugging when the program fails to sort the points. When the program displays error messages of bad memory allocation either a point was skipped in a line, or that the scan jumped to a higher line before completing the scan of the current line. Fig. 7.5 shows a scan stopped on a line where a point was missed due to a y-tolerance that was too small. To simplify the debugging process, this parameter is used to stop the scan and display all found points up until the chosen point. Entering any number greater than the number of points in the file will let the program complete its scan of the entire

[2]As the algorithm scans a row from left to right, it shifts the leftmost limit to the last found point while still expecting to find the same fixed amount of points per row. If a point is skipped, it creates the condition where the algorithm reaches the rightmost point in a row before having found the expected number of points (and reseting the left limit). This creates a memory allocation error. Using the "stop-and-draw" parameter allows the user to stop the program before it crashes to see where point was skipped and adjust the tolerances accordingly.

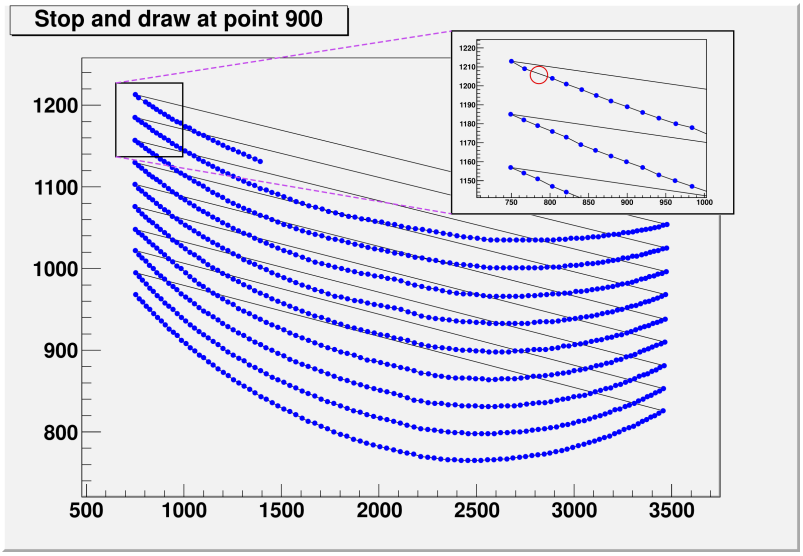


Figure 7.5: Output from `slope_sort` showing a skipped point due to a small y-tolerance. Running the algorithm to the end of the current row of points will cause it to crash. Increasing the y-tolerance from 0.1 to 0.2 fixes this problem.

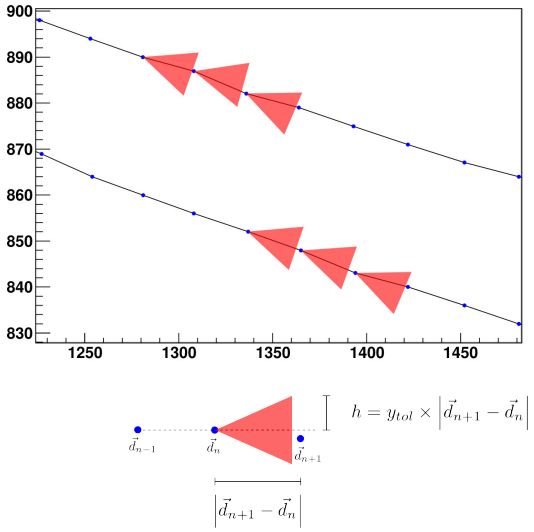


Figure 7.6: Figure showing how the `slope_sort` algorithm uses the slope between previous points to narrow the search for the next point. Increasing the y-tolerance variable y_{tol} will increase the height of the cone h when it reaches a candidate point at $\vec{d}_{n+1}$.

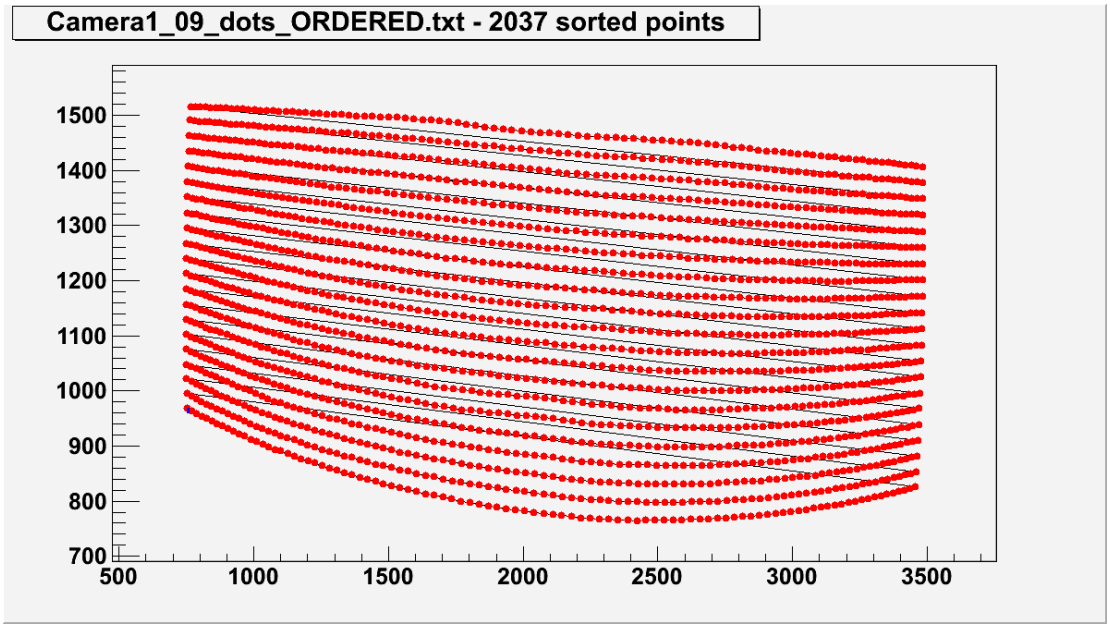


Figure 7.7: Output from `slope_sort` after sorting the points. A line connects the points in the order they appear in the file. It travels left to right, bottom to top.

file without stopping.

A set of sorted points may be seen in Fig. 7.7. Once the points have been sorted to match the 3D position file, the camera is ready to be calibrated. Depending on the targets chosen for calibration in SNO+, a different sorting program may need to be written for the target points.

7.4 Running the Calibration Code

Once the sets of pixel positions and matching object positions have been generated for each camera it is possible to create a mapping which will result in a calibrated camera. A program called `Intrinsics_Solve` was written to run the calibration fit, with the latest version called `Intrinsics_Solve_final`. This program calibrates one camera at a time, taking the pixel positions and object positions as input, along with the camera number. The position of the camera is taken as a fixed variable and is currently stored inside the program, chosen via its number.

The program is run with the following input:

```
$ Intrinsics_Solve_final 3d_cam1.txt Camera1_09_dots_ORDERED.txt 81
```

The first argument is the name of the 3D object position file, and the second is the pixel position file after being ordered in the same way at the 3D file. The third argument chooses which camera to use. The number 81, in this case, signifies the eighth model being used for Camera 1. The program can be changed to take the coordinates of the camera as input instead of a number (1 in this case) as it will allow for greater flexibility in editing the positions. In addition to supplying the three (X, Y, Z) coordinates of the camera, the approximate orientation of the camera in terms of the three Euler angles θ, ϕ, ψ in the global coordinate system should also be supplied. `Intrinsics_Solve_final` uses these as a first guess in the fit it performs, but they are currently hard-coded into the program.

The program uses the Minuit2 fitting algorithm to fit the chosen fitting function to the data in the two files of positions. With each iteration, the fitter compares the given pixel position as it appears in the photograph, (x_{pic}, y_{pic}), with the position that a point at (X, Y, Z) would transform to with a transformation given by the fitting function used. The squared difference between the generated (x_{calc}, y_{calc}) position and given (x_{pic}, y_{pic}) position is summed for each points, and this value is reduced with each successful iteration of the fitter.

Upon completion of the fit, the program will output all of the parameters in the fit with their value and error, along with whether they were fixed or floated in the fit. The total squared residual of the fit is also given, in addition to an average residual per point, in pixels. The overall squared residual is what allows cameras to be easily compared to one another.

The program will also display plots of the main fit and plots of the residuals, separated into x and y residuals. These residual plots are useful for deciding how the model can be changed to reduce residuals and give a better overall fit. This is described in more detail in Chapter 5. Finally, the program also displays a 3D map of the swimming pool shallow end. The map shows the position of the camera used in the fit in its calibrated orientation along with the other cameras. Several different views of this map are shown in Fig. A.4 in Appendix A. This helps confirm that the fit makes

sense physically and the camera is not, for example, facing backwards. It can be useful in the more serious debugging sessions when working with a new fitting model.

The parameters from the fit are written to file as `PAR_Camera1_09_dots_ORDERED.txt`. This fit needs to be carried out on all six cameras, with each one producing a separate parameter file. Once all of the cameras have corresponding parameter files, they are ready to be used in the final step, triangulation.

7.5 Triangulation Process

A second program was written to perform the triangulation minimization, called `Position_Solve_final`. It uses the same minimizer as the calibration program, but is different enough to have its own program. The triangulation code takes pixel locations of a target as seen from each camera as input, along with each camera's set of parameters calculated from calibration. It will then float an (X, Y, Z) location for each point in the calibration and determine which is the most likely location for the object.

The program is run with the following input:

```
$ Position_Solve_final Params.txt Pixels.txt 3d_coords.txt
```

The first argument is a file containing the parameters from all six cameras. For six cameras, the first line would contain all of Camera P's parameters, the second would contain Camera 1's parameters, and so on. A shell script is used to combine all six individual parameter files into this larger file.

The second argument is a collection of the pixel locations of all the points as seen by each camera, and each following the same order. The pixel locations of each target need to be extracted and sorted (Sec. 7.2 and 7.3) as detailed previously. The sorting may be different, however, if the cameras are in place as they will be in SNO+. Essentially, the n^{th} line in a pixel position file for each camera needs to refer to the n^{th} object to be triangulated. A shell script may be used to combine all six of these files into one larger file to be used by the triangulation program.

The third argument to the program is a file containing the actual 3D positions of the object being triangulated, if they are known. This is used to determine the error on the triangulation algorithm by calculating and plotting the residuals of calculated

position versus actual position for each object. These plots are presented in Chapter 5 for 30 points chosen from the swimming pool test pictures.

After running `Position_Solve_final` with these arguments, the position of each point triangulated is displayed on screen and can be written to file. In SNO+ there will likely be no prior position to which the result can be compared if the target is a calibration source for the PMTs. This may be untrue, however, if the camera system result is compared to the result given by measurements from the rope system used to lower the calibration source into the AV.

Since the cameras will be run in near darkness during a PMT calibration run, the only source of light will be the LED situated on the calibration source. Extracting this from each photo taken is a straightforward process, involving only step 4 in the point extraction procedure (Sec. 7.2). If there is only LED present, it will also be unnecessary to sort the points for each camera as they will be singular. The pixel locations can be promptly passed to the triangulation routine and a position calculated within about 10 minutes of taking the photographs. Ideally, the whole process will automated with the user being able to check on the intermediate steps to make sure everything is working correctly.

Chapter 8

Conclusion

The camera system was developed in order to accurately determine the position of radioactive sources used during PMT calibration runs in the SNO+ neutrino experiment, and to monitor the AV over the course of the experiment. Six watertight stainless steel enclosures were developed to house single Nikon D500 cameras behind acrylic dome viewports. The cameras have a fixed focus and are aimed at the center of the acrylic vessel. To date, three have been installed and have demonstrated the ability to be operated remotely from the SNO+ deck.

After being tested in a swimming pool at the University of Alberta, the camera system was shown to triangulate a point 3 m away to an accuracy of 0.5 cm. This extrapolates to an accuracy of 1.5 cm at the center of the AV, approximately 9 meters away from each camera. This result improves upon the previous accuracy of the rope manipulator system which could measure to an accuracy of 5 cm.

Further work includes installing the last 3 cameras during the water-fill phase of SNO+ in late 2012. Once the detector is filled with liquid scintillator a calibration will still need to be done in the same vein as the swimming pool calibration to ensure that the entire field of view of the cameras is mapped. Using the well-known positions of every PMT and the positions of the installed cameras, this should bring the system to full operational readiness for triangulation of calibration sources when the SNO+ experiment begins operation in 2013.

Bibliography

[1] Aharmim B. et al. Low Energy Threshold Analysis of the Phase I and Phase II Data Sets of the Sudbury Neutrino Obersvatory. *Physical Review C*, 81(055504), May 2010.

[2] M. C. Chen. The SNO Liquid Scintillator Project. *Nuclear Physics B (Proc. Suppl.)*, 145:65–68, 2005.

[3] Alex Wright. *Robust Signal Extraction Methods and Monte Carlo Sensitivity Studies for the Sudbury Neutrino Observatory and SNO+ Experiments*. PhD thesis, Queen's University, 2009.

[4] A. Piepke. Final Results from the Palo Verde Neutrino Oscillation Experiment. *Prog. Part. Nucl. Phys.*, 48:113–121, 2002.

[5] A. Hallin. Nonlinear Buckling Analysis of Acrylic Vessel. Internal SNO+ report, 2008.

[6] A.S. Barabash. NEMO 3 Double Beta Decay Experiment: Latest Results. *J. Phys.: Conf. Ser.*, 173(012008):10, 2009.

[7] N. Ackerman et al. Observation of Two-Neutrino Double-Beta Decay in ^{136}Xe with the EXO-200 Detector. *Phys. Rev. Lett.*, 107:212501, Nov 2011.

[8] A.S. Barabash and V.B. Brudanin. Investigation of Double-Beta Decay with the NEMO-3 Detector. *Physics of Atomic Nuclei*, 74(2):312–317, 2011.

[9] C. Peña-Garay and A. Serenelli. Solar Neutrinos and the Solar Composition Problem. *arXiv*, (0811.24-24), 2008.

[10] J. Bahcall, M. Gonzalez-Garcia, and C. Peña-Garay. Does the Sun Shine by pp or CNO Fusion Reactions? *Phys. Rev. Lett.*, 90(131301), 2003.

[11] N.A. Jelley. Results from Solar and Reactor Neutrino Experiments. *Nuclear Physics B - Proceedings Supplements*, 169(0):299–308, 2007. Proceedings of the Ninth International Workshop on Tau Lepton Physics TAU06.

[12] Jean-Claude Mareschal, Claude Jaupart, Catherine Phaneuf, and Claire Perry. Geoneutrinos and the energy budget of the Earth. *Journal of Geodynamics*, 54(0):43–54, 2012.

[13] C. Jaupart, S. Labrosse, and J.-C. Mareschal. 7.06 - Temperatures, Heat and Energy in the Mantle of the Earth. In Gerald Schubert, editor, *Treatise on Geophysics*, pages 253–303. Elsevier, Amsterdam, 2007.

[14] M. Chen. Geo-neutrinos in SNO+. *Earth, Moon, and Planets*, 99:221–228, 2006. 10.1007/s11038-006-9116-4.

[15] E. Guillian. The Sensitivity of SNO+ to Δm^2_{12} Using Reactor Anti-neutrino Data. 2008. arxiv:0809.1649.

[16] S. Peeters. SNO+ Calibration. Dec 2011. SNO+ Document 1252-v5.

[17] M. Walker. SNO+ Detector Calibration Hardware: Umbilical Cables. Poster, June 2012. SNO+ Document 1454-v1.

[18] G. Lefeuvre S. Peters. Laser Ball and TELLIE Status. Mar 2012. SNO+ Document 1378-v1.

[19] I. Coulter. SMELLIE and AMELLIE. Mar 2012. SNO+ Document 1344-v1.

[20] A. Reichold. SMELLIE Overview. Aug 2012. SNO+ Document 1623-v1.

[21] Bryce Anton Moffat. *The Optical Calibration of the Sudbury Neutrino Observatory.* PhD thesis, Queen's University, 2001.

[22] James R.N. Cameron. *The Photomultiplier Tube Calibration of the Sudbury Neutrino Observatory.* PhD thesis, Wolfson College, Oxford, 2001.

[23] Henning Olling Back. *Internal Radioactive Source Calibration of the Borexino Solar Neutrino Experiment*. PhD thesis, Virginia Polytechnic Institute and State University, 2004.

[24] T. Sonley and P. Gorel. personal communication, 2011.

[25] K. Olsen. Improvements to the resolution and efficiency of the deap-3600 dark matter detector and their effects on background studies. Master's thesis, University of Alberta, 2010.

[26] Nikon - Understanding ISO Sensitivity. `http://www.nikonusa.com/Learn-And-Explore/Photography-Techniques/g9mqnyb1/1/Understanding-ISO.html`.

[27] Ellen Schwalbe. Geometric Modelling and Calibration of Fisheye Lens Camera System. 2005.

[28] Frédéric Devernay and Olivier Faugeras. Straight lines have to be straight. *Machine Vision and Applications*, 13:14–24, 2001. 10.1007/PL00013269.

[29] Rene Brun and Fons Rademakers. ROOT - An Object Oriented Data Analysis Framework. *Nucl. Inst. & Meth. in Phys. Res.*, A(389):81–86, 1996.

[30] Eric W. Weisstein. "Euler Angles." From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/EulerAngles.html`.

Appendix A

Pool Test Coordinates

This appendix will give further details on the September 2011 pool test, described in Chapter 4. It will also provide a brief summary of the Euler angle transformations used in the development of the coordinate systems and calibration described in Chapter 5.

A.1 Euler Angles

Chapter 5 discusses the transformation of an object's position from the global coordinate system to the camera coordinate system. The Euler rotation matrix in Eq. 5.1 has nine elements, $(a_{12}, a_{12}, \ldots, a_{33})$. It is constructed by applying three successive rotational transformations to a coordinate system. These three rotations can be represented by a matrix $\mathbf{A} = \mathbf{BCD}$, where each matrix describes a rotation around a different axis. (Fig. A.1).

The rotations are given by the following matrices:

$$\mathbf{D} = \begin{bmatrix} \cos\phi & \sin\phi & 0 \\ -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \mathbf{C} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{A.1}$$

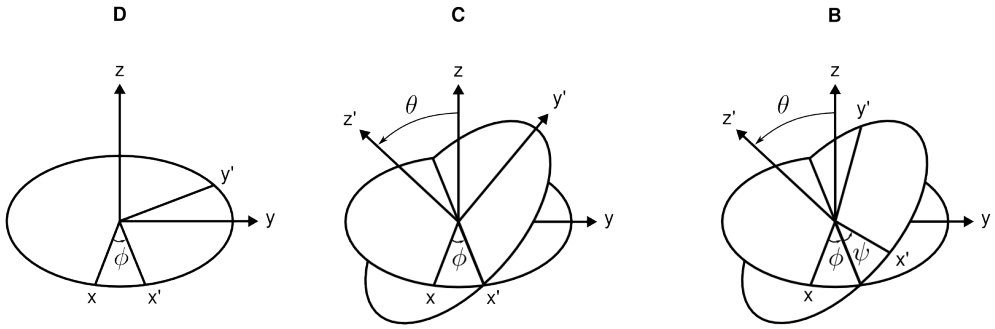


Figure A.1: Diagram demonstrating the effect of the three Euler angles: ϕ, θ, and ψ. The first is a rotation ϕ about the original z-axis (**D**), followed by rotation θ about the new x-axis (**C**), and finally a rotation ψ about the new z-axis (**B**). The matrices are described in Eq. A.1. Figure reproduced from [30].

This results in the rotation matrix **A** having the following elements:

$$\begin{aligned}
a_{11} &= \cos\psi\cos\phi - \cos\theta\sin\phi\sin\psi \\
a_{12} &= \cos\psi\sin\phi + \cos\theta\cos\phi\sin\psi \\
a_{13} &= \sin\psi\sin\theta \\
a_{21} &= -\sin\psi\cos\phi - \cos\theta\sin\phi\cos\psi \\
a_{22} &= -\sin\phi\sin\phi + \cos\theta\cos\phi\cos\psi \\
a_{23} &= \cos\psi\sin\theta \\
a_{31} &= \sin\theta\sin\phi \\
a_{32} &= -\sin\theta\cos\phi \\
a_{33} &= \cos\theta
\end{aligned} \tag{A.2}$$

These matrix elements are used in the program `Intrinsics_Solve_final` (Chapter 7) to transform the position of an object from the global coordinate system into the camera coordinate system in Eq. 5.1. This transformation is shown in Fig. A.2 and has the effect of lining up the x and y axes of the camera coordinate system with that of the photograph. All three angles are allowed to float in the fit, but ϕ and ψ remain close to zero while θ usually takes on a value close to $\pi/2$.

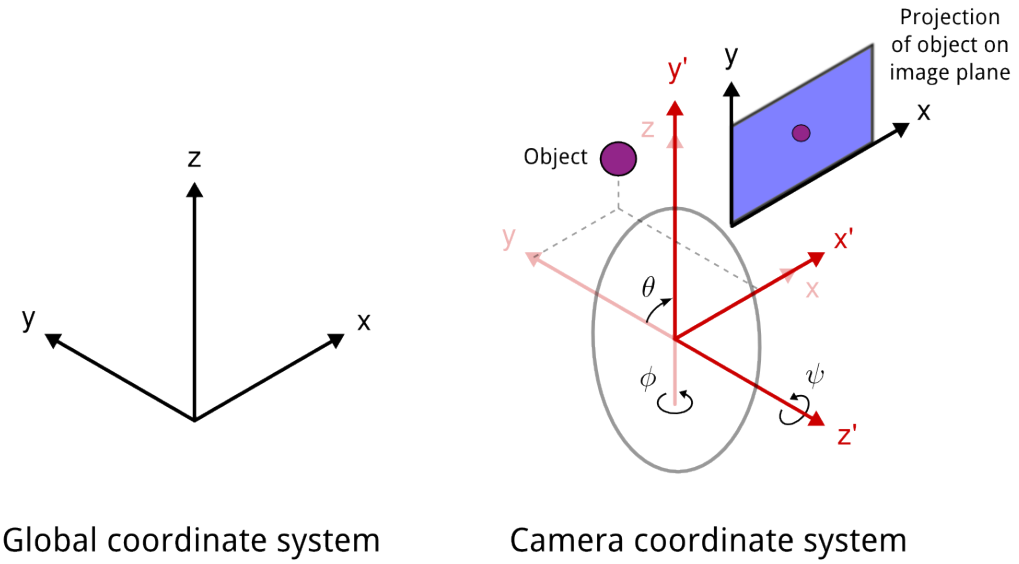


Figure A.2: Conversion from the global coordinate system to the camera coordinate system is done by a transformation of $\pi/2$ through θ. The new coordinates make it simpler to interpret a projection from a camera frame coordinate to a pixel coordinate, as the x and y axes now line up.

A.2 Global Coordinate System and Locations of Cameras

All locations of cameras and pool tiles are given with respect to the global coordinate system. Fig. A.3 shows the origin and setup of the coordinate system. Once the global coordinate system was set up, the locations of cameras and calibration objects could be given in this system. All position measurements are given in centimeters, including in the code. The locations of the cameras are given in Table A.1.

The locations of the tiles were generated by measuring the position of the bottom-left tile and then calculating the position of all other tiles using the known spacing between the tiles, measured as 5.083 ± 0.001 cm[1]. Each camera used a grid of calibration points spanning 97×21 lines, chosen such that it was approximately centered in the camera's field-of-view. This resulted in the best possible resolution of tiles on the edge of the grid.

[1]The average size of a pool tile with grout was calculated by measuring the distance spanned by several meters' worth of tiles and dividing by the number of tiles within that set.

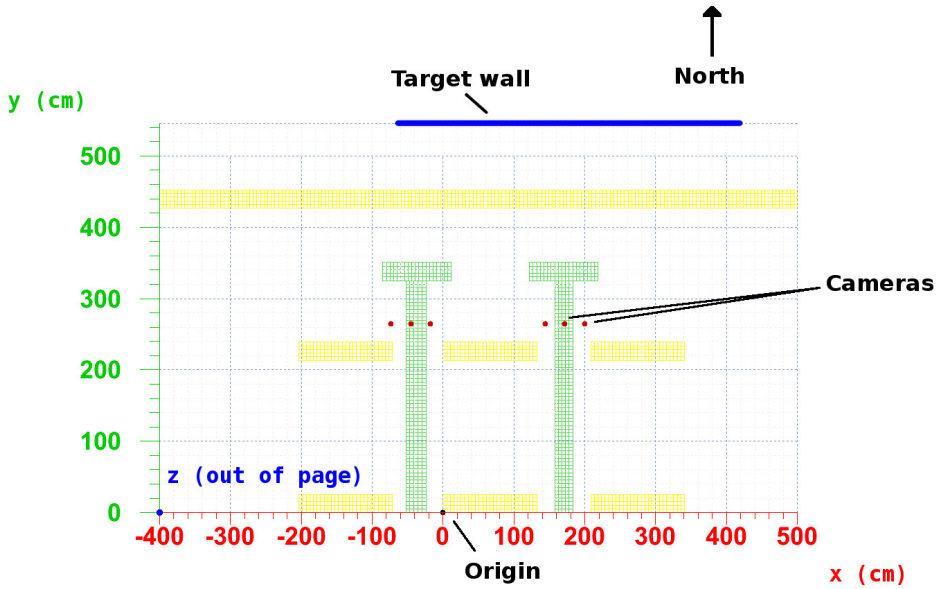


Figure A.3: The global coordinate system used for camera calibrations has its origin located at the bottom-left corner of the second segmented yellow line in the University of Alberta West Pool, 545.7 cm south of the north wall. Due to the slight downward slope of the floor (1.23°) away from the target wall, the origin was chosen to be 11.7 cm above the pool floor so that $z = 0$ would coincide with the bottom of the north wall.

In the code snippet, `x1`, `y1` and `z1` constitute the position of the bottom-left "starting point" from which all other tile positions are extrapolated. These points are given in Table A.2. The positions of all 2037 points are stored in the arrays `xpos[]`, `ypos[]` and `zpos[]` and output the generated points to a .txt file in a left-to-right, bottom-to-top manner as outlined in Sec. 7.3. The generated points were compared to photographs of the target wall to ensure that the extrapolations were still accurate to 1 mm in the last generated column.

An extra visual tool in understanding the calibrations done in the pool is the 3D viewer in ROOT. The locations of walls and colored floor tiles were coded in to accurately show the relative positions of the cameras and calibration points. This

Camera	P	1	2	3	4	5
X_0	-73.5	-45.6	-17.6	143.8	172.0	199.5
Y_0	264.2	264.2	264.2	264.2	264.2	264.2
Z_0	94.5	94.5	94.5	94.5	94.5	94.5

Table A.1: Camera positions from the Sept. 2011 pool test, used in Eq. 5.1. All values are given in centimeters.

Generating a grid of target calibration points

```
1  // Generate a grid in XZ plane (constant Y)
2  int nx = 97;                  // # of markers in x-direction
3  int ny = 1;                   // # of markers in y-direction
4  int nz = 21;                  // # of markers in z-direction
5  int npoints = nx*ny*nz;       // total # of markers
6  double sp = 5.083;            // spacing between points (cm)
7  for(int j=0; j<nz; j++) {
8        for(int i=0; i<nx; i++) {
9                int n = j*nx+i;                 // counter
10               xpos[n] = double(i)*sp + x1;    // single tile spacing in x
11               ypos[n] = y1;                   // constant y (flat wall)
12               zpos[n] = double(j)*sp + z1;    // single tile spacing in z
13       }
14 }
```

system was also an invaluable tool in understanding the Euler rotations of the cameras - including when they went wrong due to a misbehaving program. Several different views of a calibration done for Camera 4 are given in Fig. A.4.

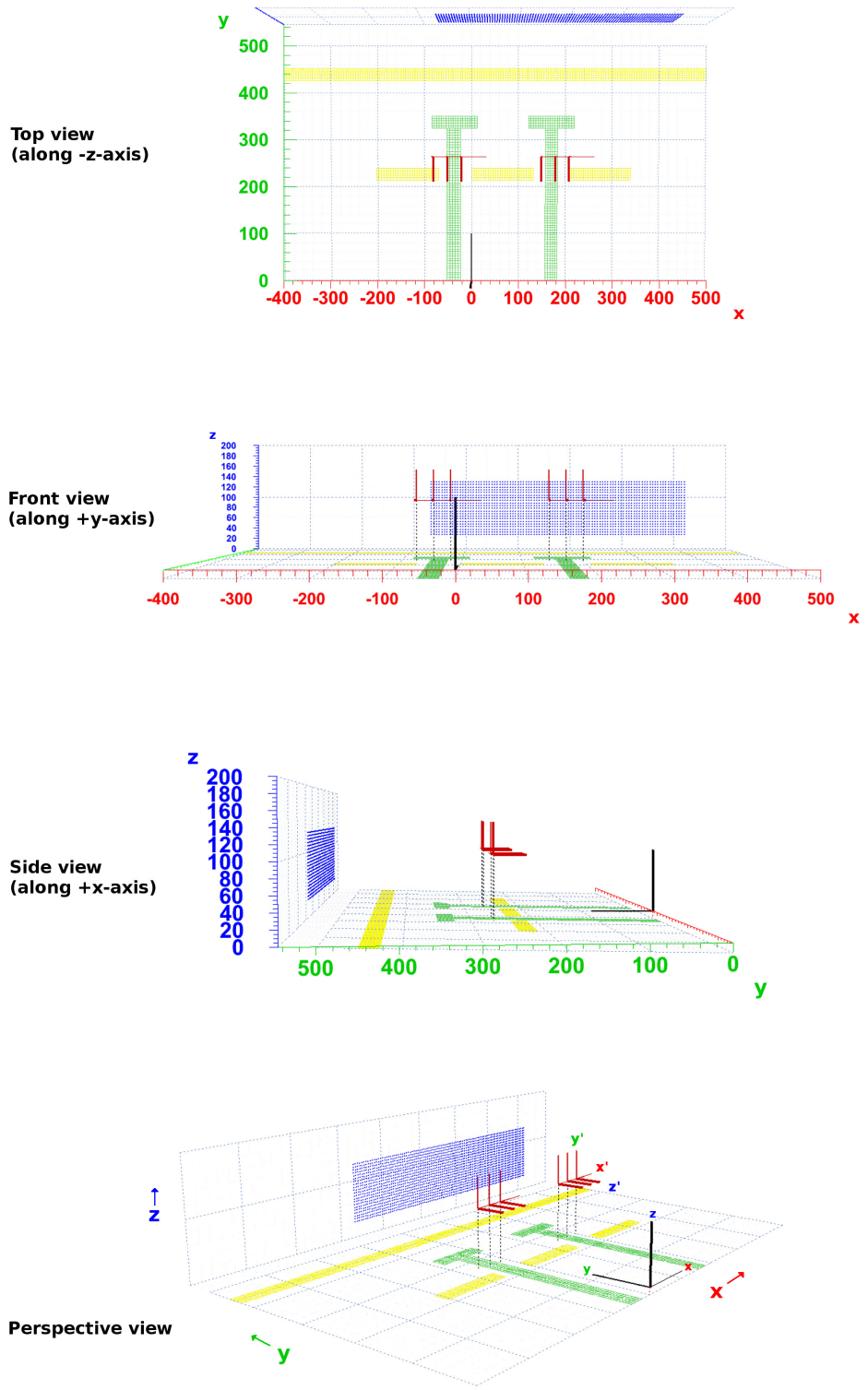


Figure A.4: Graphical representations of the Sept. 2011 pool tests. The six maroon vertices represent the camera coordinate system unique to each camera. The black vertex represents the global coordinate system. Each vertex is centered on its respective coordinate system.

Camera	P	1	2	3	4	5
x1	-331.1	-305.6	-254.8	-66.8	-66.8	-51.5
y1	545.7	545.7	545.7	545.7	545.7	545.7
z1	27.1	27.1	27.1	27.1	27.1	27.1

Table A.2: Coordinates of the bottom-left point, used to generate the entire calibration grid for each camera. All values are given in centimeters.

Appendix B

Algorithms

This section contains the code used in the image editing, calibration and triangulation done in this body of work. Programs were compiled with standard ROOT libraries as well as several header files written by the author, also included in this section.

B.1 image2hist.c

```
// Author: Zachary Petriw
// March 17, 2011 (started Dec. 2010)

// image2hist.c
// Importing an image as grayscale and converting it to a histogram
// for later analysis

#include "TColor.h"
#include "TImage.h"
#include <string>
#include <stdio>

using namespace std;

Double_t getgray (UInt_t col)
  {     //returns a grayscale value of the color
        // ITU color standard
  return (0.299f*((col & 0xff0000) >> 16) +
        0.587f*((col & 0xff00) >> 8) +
        0.114f*(col & 0xff));
  }

Double_t getred (UInt_t col)
  { return ((col & 0xff0000) >> 16); }
Double_t getgreen (UInt_t col)
  { return ((col & 0xff00) >> 8); }
Double_t getblue (UInt_t col)
```

```
  { return (col & 0xff); }


void image2hist (TString fname, UInt_t binning = 10) {
  gStyle->SetPalette(1);

  // Time the process
  TStopwatch *sw = new TStopwatch();

  // *** EDIT this if you are changing DIRECTORIES!!! ***
  //  string dir = "~Desktop/Camera_work/Minimization/"+fname;
  // ************************************************
  TImage *img = TImage::Open(fname.Data());

  UInt_t w = img->GetWidth();
  UInt_t h = img->GetHeight();
  UInt_t *argb = img->GetArgbArray();   // hex colors i.e. 0xffeeddcc
  Double_t grayscale[15000000];         // grayscale version of *argb,
                                        // large enough for ~15Mb

  cout << endl;
  cout << "w = " << w << endl;
  cout << "h = " << h << endl;
  cout << "w*h = " << w*h << endl;

  TH2D *grid = new TH2D("Grayscale", "Grayscale", w, 0, w, h, 0, h);
  grid->SetStats(0);    // remove statistics box because it gets in the way

  // scan all pixels in image
  // make a grayscale version of the image
  for(UInt_t i=0; i<h; i++) {
        for(UInt_t j=0; j<w; j++) {

                Int_t idx = i*w + j;
                grayscale[idx] = getgray(argb[idx]);
                grid->SetBinContent(j,h-i,grayscale[idx]);
        }
  }

  TString s1(".root");
  // replace the 4 char after the last period with s1
  TString filename = fname.Replace(fname.Last('.'),4,s1);
  TFile *f = new TFile(filename.Data(), "RECREATE");
  grid->Write();
  f->Close();
  cout << "Wrote histogram to " << filename << endl;

  TH2 *gridbin = grid->Rebin2D(binning, binning, "Grayscale2");
  TCanvas *cfit = new TCanvas("cfit", "Grayscale Image", 0, 0, 1800, 1000);
  cfit->Divide(2,2);
  cfit->SetFillColor(17);

  cfit->cd(1); grid->Draw("colz");
  cfit->cd(2); gridbin->Draw("surf3");
  cfit->cd(3); gridbin->Draw("lego2");
  cfit->cd(4); img->Draw();

  // Stop timing
  sw->Stop();
```

```
  sw->Print("u");
  cout << endl;
}
```

B.2 image2spectrum.c

```
// Auther: Zachary Petriw
// March 17, 2011 (started Dec. 2010)

// image2spectrum.c
// Find peaks in a 2D histogram - lots borrowed from Src.C from ROOT

#include <iostream>
#include <string>
#include "TSpectrum2"
#include "TH2.h"

void image2spectrum (Int_t dim = 2, TString rootfile, Double_t sigma,
    Double_t thresh, Int_t write = 0) {

TFile *f = new TFile(rootfile.Data());
search = (TH2D*)f->Get("Grayscale");

Int_t i,j,nfound;
Int_t nbinsx = search->GetNbinsX();
Int_t nbinsy = search->GetNbinsY();
cout << "nbinsx = " << nbinsx << endl;
cout << "nbinsy = " << nbinsy << endl;
Double_t xmin = 0;
Double_t xmax = (Double_t)nbinsx;
Double_t ymin = 0;
Double_t ymax = (Double_t)nbinsy;
Float_t fPositionX[5000];
Float_t fPositionY[5000];
Float_t ** source = new Float_t* [nbinsx];
for (i=0; i<nbinsx; i++)
        source[i] = new Float_t[nbinsy];
Float_t ** dest = new Float_t* [nbinsx];
for (i=0; i<nbinsx; i++)
        dest[i] = new Float_t[nbinsy];

TCanvas *Searching = new TCanvas("Searching","High resolution peak searching
    ",10,10,1000,700);
search->SetStats(0); // remove stats box because it is currently useless
gStyle->SetPalette(1); // rainbow color scale

   // Set maximum number of peaks higher than the default 100
   TSpectrum2 *s = new TSpectrum2(5000);

   for (i = 0; i < nbinsx; i++){
     for (j = 0; j < nbinsy; j++){
                source[i][j] = search->GetBinContent(i + 1,j + 1);
            }
   }

  // Time this sucker
  TStopwatch *sw = new TStopwatch();

  nfound = s->SearchHighRes(source, dest, nbinsx, nbinsy, sigma, thresh,
      kTRUE, 3, kFALSE, 3);
```

```
  printf("Found %d candidate peaks\n",nfound);

// set up marker coordinates
Float_t* posx = new Float_t [nfound];   // x-positions from TSpectrum2
        posx = s->GetPositionX();
Float_t* posy = new Float_t [nfound];   // y-positions from TSpectrum2
        posy = s->GetPositionY();
Float_t* bincon = new Float_t [nfound]; // content of each bin
Float_t* ptr = new Float_t [nfound*3];  //  for TPolyMarker3D
Int_t binx, biny;

for (i=0; i<nfound; i++) {
        binx = 1 + Int_t(posx[i]); // for proper rounding
        biny = 1 + Int_t(posy[i]);
        bincon[i] = search->GetBinContent(binx, biny);
        ptr[i*3] = posx[i]; ptr[i*3+1] = posy[i]; ptr[i*3+2] = bincon[i];
}

// WRITE MARKER COORDINATES TO FILE IF LAST PARAM != 0
if(write != 0) {
  TString ext(".txt");
  // replace the 5 char after the last period (including period) with 'ext'
  TString file_out = rootfile.Replace(rootfile.Last('.'),5,ext);
  FILE *fp;
  fp = fopen(file_out.Data(), "w");
  for(Int_t i=0; i<nfound; i++) {
        fprintf(fp, "%f %f\n", posx[i], posy[i]);
  }
  fclose(fp);
  cout << "Wrote marker positions to " << file_out << endl;
}

// 2D CASE ONLY
if(dim == 2) {
search->Draw("colz");
TPolyMarker *poly2d = new TPolyMarker(nfound, posx, posy);
poly2d->SetMarkerStyle(23);
poly2d->SetMarkerSize(1);
poly2d->SetMarkerColor(1);
poly2d->Draw("same");
}

// 3D CASE ONLY
else if(dim == 3) {
search->Draw("surf1");
TPolyMarker3D *poly3d = new TPolyMarker3D(nfound, ptr, 23);
poly3d->SetMarkerStyle(23);
poly3d->SetMarkerSize(1);
poly3d->SetMarkerColor(1);
poly3d->Draw("same");
}

  // Read the stopwatch
  sw->Stop();
  sw->Print("u");
}
```

B.3 Intrinsics_Solve_final.cpp

```
// Zachary Petriw
// May 20, 2011 (day before the Rapture?)

// Intrinsics_Solve_final.cpp

// Changes made on Sept 16, 2011:
// Decoupling of the radial distortion parameters,k1,k2,k3, and
// tangental parameters, p1,p2, from the x,y offsets: xh,yh.

// The variables xh,yh have been replaced with xr,yr when calculating
// radial/tangental distortions.

// [NEW] Dec. 2, 2011
// Added the possibility of using a second fitting fuction, one
// which uses a polynomial expansion for the distortion parameters
// To use the second function, use camera parameter sets 20-25, or
// 30-25.

// The polynomial fit has a different number (15) of parameters than
// the radial distortion model.

// [NEW] Dec. 28, 2011
// Rewritten as Intrinsics_Solve_final. It will include the functions
// Residual and ResidualFCN. Should be more lightweight.


#include <iostream>
#include <vector>
#include <fstream>
#include <sstream>
#include "TStopwatch.h"
#include "TVector2.h"
#include "TVector3.h"
#include "TMath.h"
#include "TStopwatch.h"
#include "TApplication.h"
#include "TCanvas.h"
#include "TGraph.h"
#include "TGraphErrors.h"
#include "TMultiGraph.h"
#include "TAxis.h"
#include "TPolyMarker.h"
#include "TMarker.h"
#include "TArrow.h"
#include "TPolyMarker3D.h"
#include "TPolyLine3D.h"
#include "TView.h"
#include "TH1.h"
#include "TH2.h"
#include "TStyle.h"
#include "DrawFunc.h"
#include "DrawResiduals.h"
#include "Residual.h"
#include "ResidualFCN.h"
#include <Minuit2/FunctionMinimum.h>
```

```
#include <Minuit2/MnPrint.h>
#include <Minuit2/VariableMetricMinimizer.h>
#include <Minuit2/MnMigrad.h>
#include <Minuit2/MnMinos.h>
#include <Minuit2/MnContours.h>
#include <Minuit2/MnPlot.h>
#include "TAxis3D.h"

using namespace TMath;
using namespace ROOT::Minuit2;
using namespace std;


// ********************************************************************** //
// ********************** Main program ************************** //
// ********************************************************************** //

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

  cerr << endl;
//  TStopwatch *sw;

  // Oct 15,2011 - added a 4th argument, removed the option for 5
  // Fourth argument chooses which camera to use, to avoid constantly
      recompiling the code
  if( argc != 4 ) {
        cerr << "Usage: ./Intrinsic_Solve markercoords(3xN).txt pixelcoords
            (2xN).txt Camera_number" << endl;
        cerr << endl;
  return -1;
  }

  string mkname = argv[1];      // Name of file containing marker positions
  string pxname = argv[2];      // Name of file containing pixel positions (
      TSpectrum2 output)
  int camera_num = atoi(argv[3]);       // Camera parameters to use, 0-5 (0
      = P)
                                        // (Nov 8, 2011) 10-15 will be used
                                            for triangulation pic sets

  // Check that the inputs are .txt files
  size_t check1, check2;
  check1 = mkname.find_last_of(".");    // Look for the .txt extension
  check2 = pxname.find_last_of(".");
  string exten1 (mkname, check1+1, 3);  // turn the extension into a string
  string exten2 (pxname, check2+1, 3);

  if((exten1 != "txt") || (exten2 != "txt")) {
        cerr << "Both input files need to be .txt files!" << endl;
        cerr << "Buh-bye!" << endl << endl;
  return -2;
  }

  // Open the marker position file
  ifstream mkfile;
  mkfile.open(mkname.data());

  if(!mkfile) {
        cerr << "...Cannot open markercoord file" << endl;
```

```
  return -3;
  }

  // Open the pixel position file
  ifstream pxfile;
  pxfile.open(pxname.data());

  if(!pxfile) {
        cerr << "...Cannot open pixelcoord file" << endl;
  return -4;
  }

  cerr << "The files we are opening are:" << endl;
  cerr << "     " << mkname << ", and " << endl;
  cerr << "     " << pxname << endl << endl;

  // Get each file's size and make sure they are the same
  Int_t fsize;                  // file size, pending both files have the
     same size
  Int_t mksize = 0;             // marker file size
  Int_t pxsize = 0;             // pixel file size
  string dummy;                 // dummy string to hold lines from file

  // Read in each line of marker file
  while(getline(mkfile,dummy))
        ++mksize;
  // Read in each line of pixel file
  while(getline(pxfile,dummy))
        ++pxsize;

  // Compare the two
  if(mksize == pxsize) {
        fsize = mksize;
        cerr << "Both files have " << mksize << " lines" << endl;
  }
  else {
        cerr << "Files aren't the same size!" << endl;
        cerr << "marker file has " << mksize << " lines" << endl;
        cerr << "pixel file has " << pxsize << " lines" << endl;
  return -5;
  }

  // So far, so good
  mkfile.clear();               // clear the eof() flag that we hit while
     checking the file size
  pxfile.clear();
  mkfile.seekg(0, ios::beg);    // Reset counter back to beginning
  pxfile.seekg(0, ios::beg);
//  cerr << "tellg after reseting counter = " << mkfile.tellg() << endl <<
    endl;
//  cerr << "tellg after reseting counter = " << pxfile.tellg() << endl <<
    endl;



  // Read in the files' lines and append to vectors
  // It needs to be in vector format for Minuit2 to work
  vector<TVector3> Markers;     // Marker coords
  vector<TVector2> Pixels;      // Pixel coords
```

```
  double mx,my,mz,px,py;        // dummy variables to hold Marker, Pixel
      values
  TVector3 mkdum;               // dummy TVector3
  TVector2 pxdum;               // dummy TVector2
  double* xpic = new double [fsize];    // x pixel coords from input file
  double* ypic = new double [fsize];    // y pixel coords from input file
  double* xloc = new double [fsize];    // x coord of 3D marker location
  double* yloc = new double [fsize];    // y coord of 3D marker location
  double* zloc = new double [fsize];    // z coord of 3D marker location
  double* ptr = new double [fsize*3];   // xyz coords of 3D marker locations

  mkfile.precision(9);          // # of sig figs to read (set to max)
  for(int i=0; i<fsize; i++) {

        // Marker coords
        mkfile >> mx >> my >> mz;
//      cerr << "mx = " << mx << ", my = " << my << ", mz = " << mz << endl;
        mkdum.SetXYZ(mx,my,mz);         // Set x,y,z of marker TVector3
//      mkdum.Print();
        Markers.push_back(mkdum);       // Append to marker vector
        xloc[i] = mx;                   // x coord of 3D marker location
        yloc[i] = my;                   // y coord of 3D marker location
        zloc[i] = mz;                   // z coord of 3D marker location

        // xyz of marker locations
        ptr[i*3] = mx;
        ptr[i*3+1] = my;
        ptr[i*3+2] = mz;

        // Pixel coords
        pxfile >> px >> py;

        pxdum.Set(px,py);               // Set x,y of pixel TVector2
//      pxdum.Print();
        Pixels.push_back(pxdum);        // Append to pixel vector
        xpic[i] = px;                   // x-coord of pixel, to be plotted
        ypic[i] = py;                   // y-coord of pixel, to be plotted
  }

  // Close the files
  mkfile.close();
  pxfile.close();

  // Return sizes if you are interested
  cerr << "Markers.size() = " << Markers.size() << endl;
  cerr << "Pixels.size() = " << Pixels.size() << endl;

  // [Dec. 2, 2011] Here we need to decide on which model to use.
  // This will affect the number of parameters we will be using (parsize)
  int parsize;
  if(camera_num < 10)
        parsize = 16;   // radial distortion model
  else if(camera_num < 20)
        parsize = 19;   // polynomial, cameras facing each other
  else if(camera_num < 30)
        parsize = 9;    // simpler model
  else if(camera_num < 40)
        parsize = 10;   // split Rx and Ry
  else if(camera_num < 50)
```

```
      parsize = 14;   // polynomial
else if(camera_num < 70)
      parsize = 19;   // polynomial + radial
else if(camera_num < 80)
      parsize = 24;   // extra polynomial + radial model
else if(camera_num < 90)
      parsize = 25;   // extra polynomial + radial model + sine correction
           to y
else if(camera_num < 100)
      parsize = 27;   // extra polynomial + radial model + extra poly
          corrections

// Initial starting parameters
// [NEW] (09/16/11) r2 is now calculated with xr,yr
// [NEW] (09/16/11) parsize is now 16, up from 14
// X0,Y0,Z0, phi,theta,psi, R,xh,yh, k1,k2,k3, p1,p2, xr,yr
// [NEW] (11/07/11) modified camera y-coords to account for slight
    backwards tilt of sawhorse

double* p = new double [parsize];


double cpos0[] = {-73.5,264.2,94.5,0,1.57,0};                  // shallow
    end positions
double cpos1[] = {-45.6,264.2,94.5,0,1.57,0};
double cpos2[] = {-17.6,264.2,94.5,0,1.57,0};
double cpos3[] = {143.8,264.2,94.5,0,1.57,0};
double cpos4[] = {172.0,264.2,94.5,0,1.57,0};
double cpos5[] = {199.5,264.2,94.5,0,1.57,0};

double cpos10[] = {-101.6,337.3,96.1,-1.57,1.57,3.14};         // deep end
    positions
double cpos11[] = {-101.6,309.4,95.5,-1.57,1.57,0};
double cpos12[] = {-101.6,281.4,94.9,-1.57,1.57,0};
double cpos13[] = {235.5,275.2,94.7,1.57,1.57,0};
double cpos14[] = {235.5,303.4,95.3,1.57,1.57,0};
double cpos15[] = {235.5,330.9,95.9,1.57,1.57,0};

// shallow end, facing wall - old model
double par0[] =
    {-73.5,264.2,94.5,0,1.57,0,2573,2144,1424,0,0,0,0,0,-3952,4109};
    // P
double par1[] =
    {-45.6,264.2,94.5,0,1.57,0,2573,2144,1424,0,0,0,0,0,6567,1303};
    // 1
double par2[] =
    {-17.6,264.2,94.5,0,1.57,0,2573,2144,1424,0,0,0,0,0,-1886,3179};
    // 2
double par3[] =
    {143.8,264.2,94.5,0,1.57,0,2573,2144,1424,0,0,0,0,0,-6901,776};
    // 3
double par4[] =
    {172.0,264.2,94.5,0,1.57,0,2573,2144,1424,0,0,0,0,0,5997,-1402};
    // 4
double par5[] =
    {199.5,264.2,94.5,0,1.57,0,2573,2144,1424,0,0,0,0,0,-4474,887};
    // 5

// shallow end, facing each other - 19 par newest model
```

```
double par10[] =
    {-101.6,337.3,96.1,-1.57,1.57,3.14,2172.92,1435.16,3105.15,
    3242.26,-40.2643,1010.44,-163.938,1718.33,-1.12473e-08,2.51707e-15,
    -1.4499e-22,3956.27,1994.82};
double par11[] =
    {-101.6,309.4,95.5,-1.57,1.57,0,2277.12,1432.38,3121.58,3164.76,
    -117.705,566.093,-192.253,1814.93,-9.63161e-09,1.02525e-15, -6.79423e
    -23,3531.15,1850.72};
double par12[] =
    {-101.6,281.4,94.9,-1.57,1.57,0,2347.23,1624.34,3420.93,3731.89,
    -398.695,2654.86,-848.247,6617.94,-4.7948e-08,5.6652e-15, -2.06951e
    -22,4298.71,1747.79};
double par13[] =
    {235.5,275.2,94.7,1.57,1.57,0,2044.49,1462.85,3138.89,3165.51,
    97.8946,84.9701,39.2842,-1844.25,1.70956e-08,-2.3151e-14, 4.26948e
    -21,2115.72,1589.74};
double par14[] =
    {235.5,303.4,95.3,1.57,1.57,0,2131.4,1456.23,2532.68,2557.07,
    10.4175,8.1809,-50.0431,-473.786,3.40054e-09,3.26501e-17, -5.4104e
    -25,9957.1,-720.805};
double par15[] =
    {235.5,330.9,95.9,1.57,1.57,0,2202.2,1449.21,2983.39,3001.71,
    -36.5552,-133.436,-114.38,-728.171,1.44658e-08,-1.86841e-15, 6.86229e
    -23,121.8,1711.55};

double par20[] = {-73.5,264.2,94.5,0,1.57,0,3020,0,0};                  // P
double par21[] = {-45.6,264.2,94.5,0,1.57,0,3020,0,0};                  // 1
double par22[] = {-17.6,264.2,94.5,0,1.57,0,3020,0,0};                  // 2
double par23[] = {143.8,264.2,94.5,0,1.57,0,3020,0,0};                  // 3
double par24[] = {172.0,264.2,94.5,0,1.57,0,3020,0,0};
double par25[] = {199.5,264.2,94.5,0,1.57,0,3020,0,0};                  // 5

// 10 parameter fit - x0,y0,x1,y1
double par30[] = {-73.5,264.2,94.5,0,1.57,0,3020,3020,0,0};
double par31[] = {-45.6,264.2,94.5,0,1.57,0,3020,3020,0,0};
double par32[] = {-17.6,264.2,94.5,0,1.57,0,3020,3020,0,0};
double par33[] = {143.8,264.2,94.5,0,1.57,0,3020,3020,0,0};
double par34[] = {172.0,264.2,94.5,0,1.57,0,3020,3020,0,0};
double par35[] = {199.5,264.2,94.5,0,1.57,0,3020,3020,0,0};

// 14 parameter fit -  x0,..,x3,y0,..,y3
double par40[] = {-73.5,264.2,94.5,0,1.57,0,0,0,0,0,0,0,0,0};
double par41[] = {-45.6,264.2,94.5,0,1.57,0,0,0,0,0,0,0,0,0};
double par42[] = {-17.6,264.2,94.5,0,1.57,0,0,0,0,0,0,0,0,0};
double par43[] = {143.8,264.2,94.5,0,1.57,0,0,0,0,0,0,0,0,0};
double par44[] = {172.0,264.2,94.5,0,1.57,0,0,0,0,0,0,0,0,0};
double par45[] = {199.5,264.2,94.5,0,1.57,0,0,0,0,0,0,0,0,0};

// 19 parameter fit - x0,..,x3,y0,..,y3, k1,k2,k3,xr,yr
// last two elements need to be estimated correctly - most sensitive to
    bad guesses
double par50[] = {-73.5,264.2,94.5,0,1.57,0,2e3,1500,3e3,3e3
    ,0,0,0,0,0,0,0,1e3,3e3};
double par51[] = {-45.6,264.2,94.5,0,1.57,0,2e3,1500,3e3,3e3
    ,0,0,0,0,0,0,0,5e3,3e3};
double par52[] = {-17.6,264.2,94.5,0,1.57,0,2e3,1500,3e3,3e3
    ,0,0,0,0,0,0,0,6e3,3e3};
double par53[] = {143.8,264.2,94.5,0,1.57,0,2e3,1500,3e3,3e3
    ,0,0,0,0,0,0,0,4e3,3e3};
```

```
double par54[] = {172.0,264.2,94.5,0,1.57,0,2e3,1500,3e3,3e3
    ,0,0,0,0,0,0,0,8e3,1e3};
double par55[] = {199.5,264.2,94.5,0,1.57,0,2e3,1500,3e3,3e3
    ,0,0,0,0,0,0,0,0,3e3};

// 19 parameter fit - x0,..,x3,y0,..,y3, k1,k2,k3,xr,yr
// *** Used for scanning over 2d space spanned by xr,yr ***
double par60[] = {-73.5,264.2,94.5,0,1.57,0,2e3,1500,3e3,3e3
    ,0,0,0,0,0,0,0,0,0};
double par61[] = {-45.6,264.2,94.5,0,1.57,0,2e3,1500,3e3,3e3
    ,0,0,0,0,0,0,0,0,0};
double par62[] = {-17.6,264.2,94.5,0,1.57,0,2e3,1500,3e3,3e3
    ,0,0,0,0,0,0,0,0,0};
double par63[] = {143.8,264.2,94.5,0,1.57,0,2e3,1500,3e3,3e3
    ,0,0,0,0,0,0,0,0,0};
double par64[] = {172.0,264.2,94.5,0,1.57,0,2e3,1500,3e3,3e3
    ,0,0,0,0,0,0,0,0,0};
double par65[] = {199.5,264.2,94.5,0,1.57,0,2e3,1500,3e3,3e3
    ,0,0,0,0,0,0,0,0,0};

// more parameter fit - x0,..,x3,y0,..,y3, k1,k2,k3,xr,yr,
// last two elements need to be estimated correctly - most sensitive to
    bad guesses
double par70[] = {-73.5,264.2,94.5,0,1.57,0,2e3,1500,3e3,3e3
    ,0,0,0,0,0,0,0,1e3,3e3,0,0,0,2.5e3,1e3};
double par71[] = {-45.6,264.2,94.5,0,1.57,0,2e3,1500,3e3,3e3
    ,0,0,0,0,0,0,0,5e3,3e3,0,0,0,2.5e3,1e3};
double par72[] = {-17.6,264.2,94.5,0,1.57,0,2e3,1500,3e3,3e3
    ,0,0,0,0,0,0,0,6e3,3e3,0,0,0,2.5e3,1e3};
double par73[] = {143.8,264.2,94.5,0,1.57,0,2e3,1500,3e3,3e3
    ,0,0,0,0,0,0,0,4e3,3e3,0,0,0,2.5e3,1e3};
double par74[] = {172.0,264.2,94.5,0,1.57,0,2e3,1500,3e3,3e3
    ,0,0,0,0,0,0,0,8e3,1e3,0,0,0,2.5e3,1e3};
double par75[] = {199.5,264.2,94.5,0,1.57,0,2e3,1500,3e3,3e3
    ,0,0,0,0,0,0,0,0,3e3,0,0,0,2.5e3,1e3};

// 19 parameter fit - x0,..,x3,y0,..,y3, k1,k2,k3,xr,yr
// last two elements need to be estimated correctly - most sensitive to
    bad guesses
double par80[] = {-73.5,264.2,94.5,0,1.57,0,2e3,1500,3e3,3e3
    ,0,0,0,0,0,0,0,1e3,3e3,5,2200,500,1000,500,0};
double par81[] = {-45.6,264.2,94.5,0,1.57,0,2e3,1500,3e3,3e3
    ,0,0,0,0,0,0,0,5e3,3e3,5,2200,500,1000,500,0};
double par82[] = {-17.6,264.2,94.5,0,1.57,0,2e3,1500,3e3,3e3
    ,0,0,0,0,0,0,0,6e3,3e3,5,2200,500,1000,500,0};
double par83[] = {143.8,264.2,94.5,0,1.57,0,2e3,1500,3e3,3e3
    ,0,0,0,0,0,0,0,4e3,3e3,5,2200,500,1000,500,0};
double par84[] = {172.0,264.2,94.5,0,1.57,0,2e3,1500,3e3,3e3
    ,0,0,0,0,0,0,0,8e3,1e3,5,2200,500,1000,500,0};
double par85[] = {199.5,264.2,94.5,0,1.57,0,2e3,1500,3e3,3e3
    ,0,0,0,0,0,0,0,0,3e3,5,2200,500,1000,500,0};

// 27 parameter fit - x0,..,x3,y0,..,y3, k1,k2,k3,xr,yr, x1y,x2y,x3y,y1x,
    y2x,y3x,b1x,b1y
// last two elements need to be estimated correctly - most sensitive to
    bad guesses
double par90[] = {-73.5,264.2,94.5,0,1.57,0,2e3,1500,3e3,3e3
    ,0,0,0,0,0,0,0,1e3,3e3,0,0,0,0,0,0,0};
```

```
double par91[] = {-45.6,264.2,94.5,0,1.57,0,2e3,1500,3e3,3e3
    ,0,0,0,0,0,0,0,5e3,3e3,0,0,0,0,0,0,0,0};
double par92[] = {-17.6,264.2,94.5,0,1.57,0,2e3,1500,3e3,3e3
    ,0,0,0,0,0,0,0,6e3,3e3,0,0,0,0,0,0,0,0};
double par93[] = {143.8,264.2,94.5,0,1.57,0,2e3,1500,3e3,3e3
    ,0,0,0,0,0,0,0,4e3,3e3,0,0,0,0,0,0,0,0};
double par94[] = {172.0,264.2,94.5,0,1.57,0,2e3,1500,3e3,3e3
    ,0,0,0,0,0,0,0,8e3,1e3,0,0,0,0,0,0,0,0};
double par95[] = {199.5,264.2,94.5,0,1.57,0,2e3,1500,3e3,3e3
    ,0,0,0,0,0,0,0,0,3e3,0,0,0,0,0,0,0,0};

// Make a larger array that can hold all these parameters, for easier
    accessing
// This may have caused a memory bug - ?
double** pc = new double* [6];
for(int i=0; i<6; i++)
      pc[i] = new double [parsize];
if (camera_num < 10) {
      for(int i=0; i<6; i++) {
              pc[0][i] = cpos0[i];
              pc[1][i] = cpos1[i];
              pc[2][i] = cpos2[i];
              pc[3][i] = cpos3[i];
              pc[4][i] = cpos4[i];
              pc[5][i] = cpos5[i];
      }
}
else if (camera_num < 20) {
      for(int i=0; i<6; i++) {
              pc[0][i] = cpos10[i];
              pc[1][i] = cpos11[i];
              pc[2][i] = cpos12[i];
              pc[3][i] = cpos13[i];
              pc[4][i] = cpos14[i];
              pc[5][i] = cpos15[i];
      }
}

else if (camera_num < 100)
{
      for(int i=0; i<6; i++) {
              pc[0][i] = cpos0[i];
              pc[1][i] = cpos1[i];
              pc[2][i] = cpos2[i];
              pc[3][i] = cpos3[i];
              pc[4][i] = cpos4[i];
              pc[5][i] = cpos5[i];
      }
}

for(int i=0; i<parsize; i++) {
      if(camera_num == 0)             p[i] = par0[i];
      else if(camera_num == 1)        p[i] = par1[i];
      else if(camera_num == 2)        p[i] = par2[i];
      else if(camera_num == 3)        p[i] = par3[i];
      else if(camera_num == 4)        p[i] = par4[i];
      else if(camera_num == 5)        p[i] = par5[i];
      else if(camera_num == 10)       p[i] = par10[i];
      else if(camera_num == 11)       p[i] = par11[i];
```

```
        else if(camera_num == 12)       p[i] = par12[i];
        else if(camera_num == 13)       p[i] = par13[i];
        else if(camera_num == 14)       p[i] = par14[i];
        else if(camera_num == 15)       p[i] = par15[i];
        else if(camera_num == 20)       p[i] = par20[i];
        else if(camera_num == 21)       p[i] = par21[i];
        else if(camera_num == 22)       p[i] = par22[i];
        else if(camera_num == 23)       p[i] = par23[i];
        else if(camera_num == 24)       p[i] = par24[i];
        else if(camera_num == 25)       p[i] = par25[i];
        else if(camera_num == 30)       p[i] = par30[i];
        else if(camera_num == 31)       p[i] = par31[i];
        else if(camera_num == 32)       p[i] = par32[i];
        else if(camera_num == 33)       p[i] = par33[i];
        else if(camera_num == 34)       p[i] = par34[i];
        else if(camera_num == 35)       p[i] = par35[i];
        else if(camera_num == 40)       p[i] = par40[i];
        else if(camera_num == 41)       p[i] = par41[i];
        else if(camera_num == 42)       p[i] = par42[i];
        else if(camera_num == 43)       p[i] = par43[i];
        else if(camera_num == 44)       p[i] = par44[i];
        else if(camera_num == 45)       p[i] = par45[i];
        else if(camera_num == 50)       p[i] = par50[i];
        else if(camera_num == 51)       p[i] = par51[i];
        else if(camera_num == 52)       p[i] = par52[i];
        else if(camera_num == 53)       p[i] = par53[i];
        else if(camera_num == 54)       p[i] = par54[i];
        else if(camera_num == 55)       p[i] = par55[i];
        else if(camera_num == 60)       p[i] = par60[i];
        else if(camera_num == 61)       p[i] = par61[i];
        else if(camera_num == 62)       p[i] = par62[i];
        else if(camera_num == 63)       p[i] = par63[i];
        else if(camera_num == 64)       p[i] = par64[i];
        else if(camera_num == 65)       p[i] = par65[i];
        else if(camera_num == 70)       p[i] = par70[i];
        else if(camera_num == 71)       p[i] = par71[i];
        else if(camera_num == 72)       p[i] = par72[i];
        else if(camera_num == 73)       p[i] = par73[i];
        else if(camera_num == 74)       p[i] = par74[i];
        else if(camera_num == 75)       p[i] = par75[i];
        else if(camera_num == 80)       p[i] = par80[i];
        else if(camera_num == 81)       p[i] = par81[i];
        else if(camera_num == 82)       p[i] = par82[i];
        else if(camera_num == 83)       p[i] = par83[i];
        else if(camera_num == 84)       p[i] = par84[i];
        else if(camera_num == 85)       p[i] = par85[i];
        else if(camera_num == 90)       p[i] = par90[i];
        else if(camera_num == 91)       p[i] = par91[i];
        else if(camera_num == 92)       p[i] = par92[i];
        else if(camera_num == 93)       p[i] = par93[i];
        else if(camera_num == 94)       p[i] = par94[i];
        else if(camera_num == 95)       p[i] = par95[i];
}

// standard minimization using MIGRAD
// create MINUIT parameters with names
MnUserParameters upar;

        upar.Add("X0", p[0], 0.1);      // name, initial value, error;
```

```
      upar.Add("Y0", p[1], 0.1);
      upar.Add("Z0", p[2], 0.1);
      upar.Add("phi", p[3], 0.1);
      upar.Add("theta", p[4], 0.1);
      upar.Add("psi", p[5], 0.1);
// old 16 parameter model
if(camera_num < 10) {
      upar.Add("R", p[6], 0.1);
      upar.Add("xh", p[7], 0.1);
      upar.Add("yh", p[8], 0.1);
      upar.Add("k1", p[9], 1e-9);
      upar.Add("k2", p[10], 1e-16);
      upar.Add("k3", p[11], 1e-24);
      upar.Add("p1", p[12], 1e-7);
      upar.Add("p2", p[13], 1e-7);
      upar.Add("xr", p[14], 0.1);
      upar.Add("yr", p[15], 0.1);
}
// 19 parameter model
else if(camera_num < 20) {
      upar.Add("x0",    p[6], 0.1);
      upar.Add("y0",    p[7], 0.1);
      upar.Add("x1",    p[8], 0.1);
      upar.Add("y1",    p[9], 0.1);
      upar.Add("x2",    p[10], 0.1);
      upar.Add("y2",    p[11], 0.1);
      upar.Add("x3",    p[12], 0.1);
      upar.Add("y3",    p[13], 0.1);
      upar.Add("k1",    p[14], 0.1);
      upar.Add("k2",    p[15], 0.1);
      upar.Add("k3",    p[16], 0.1);
      upar.Add("xr",    p[17], 0.1);
      upar.Add("yr",    p[18], 0.1);
}

else if(camera_num < 30) {
      upar.Add("R", p[6], 0.1);
      upar.Add("xh", p[7], 0.1);
      upar.Add("yh", p[8], 0.1);
}
// 10 parameter model
else if(camera_num < 40) {
      upar.Add("Rx", p[6], 0.1);
      upar.Add("Ry", p[7], 0.1);
      upar.Add("xh", p[8], 0.1);
      upar.Add("yh", p[9], 0.1);
}
// 14 parameter model
else if(camera_num < 50) {
      upar.Add("x0",    p[6], 0.1);
      upar.Add("y0",    p[7], 0.1);
      upar.Add("x1",    p[8], 0.1);
      upar.Add("y1",    p[9], 0.1);
      upar.Add("x2",    p[10], 0.1);
      upar.Add("y2",    p[11], 0.1);
      upar.Add("x3",    p[12], 0.1);
      upar.Add("y3",    p[13], 0.1);
}
// 19 parameter model
```

```
else if(camera_num < 70) {
      upar.Add("x0",    p[6], 0.1);
      upar.Add("y0",    p[7], 0.1);
      upar.Add("x1",    p[8], 0.1);
      upar.Add("y1",    p[9], 0.1);
      upar.Add("x2",    p[10], 0.1);
      upar.Add("y2",    p[11], 0.1);
      upar.Add("x3",    p[12], 0.1);
      upar.Add("y3",    p[13], 0.1);
      upar.Add("k1",    p[14], 0.1);
      upar.Add("k2",    p[15], 0.1);
      upar.Add("k3",    p[16], 0.1);
      upar.Add("xr",    p[17], 0.1);
      upar.Add("yr",    p[18], 0.1);
}
// Extra parameters model
else if(camera_num < 80) {
      upar.Add("x0",    p[6], 0.1);
      upar.Add("y0",    p[7], 0.1);
      upar.Add("x1",    p[8], 0.1);
      upar.Add("y1",    p[9], 0.1);
      upar.Add("x2",    p[10], 0.1);
      upar.Add("y2",    p[11], 0.1);
      upar.Add("x3",    p[12], 0.1);
      upar.Add("y3",    p[13], 0.1);
      upar.Add("xk1",   p[14], 0.1);
      upar.Add("xk2",   p[15], 0.1);
      upar.Add("xk3",   p[16], 0.1);
      upar.Add("xxr",   p[17], 0.1);
      upar.Add("xyr",   p[18], 0.1);
      upar.Add("yk1",   p[19], 0.1);
      upar.Add("yk2",   p[20], 0.1);
      upar.Add("yk3",   p[21], 0.1);
      upar.Add("yxr",   p[22], 0.1);
      upar.Add("yyr",   p[23], 0.1);
}
// 25 parameter model
else if(camera_num < 90) {
      upar.Add("x0",    p[6], 0.1);
      upar.Add("y0",    p[7], 0.1);
      upar.Add("x1",    p[8], 0.1);
      upar.Add("y1",    p[9], 0.1);
      upar.Add("x2",    p[10], 0.1);
      upar.Add("y2",    p[11], 0.1);
      upar.Add("x3",    p[12], 0.1);
      upar.Add("y3",    p[13], 0.1);
      upar.Add("k1",    p[14], 0.1);
      upar.Add("k2",    p[15], 0.1);
      upar.Add("k3",    p[16], 0.1);
      upar.Add("xr",    p[17], 0.1);
      upar.Add("yr",    p[18], 0.1);
      upar.Add("sa",    p[19], 0.1);
      upar.Add("sxo",   p[20], 0.1);
      upar.Add("sxf",   p[21], 0.1);
      upar.Add("syo",   p[22], 0.1);
      upar.Add("syf",   p[23], 0.1);
      upar.Add("sdum",  p[24], 0.1);
}
else if(camera_num < 100) {
```

```
        upar.Add("x0",    p[6], 0.1);
        upar.Add("y0",    p[7], 0.1);
        upar.Add("x1",    p[8], 0.1);
        upar.Add("y1",    p[9], 0.1);
        upar.Add("x2",    p[10], 0.1);
        upar.Add("y2",    p[11], 0.1);
        upar.Add("x3",    p[12], 0.1);
        upar.Add("y3",    p[13], 0.1);
        upar.Add("k1",    p[14], 0.1);
        upar.Add("k2",    p[15], 0.1);
        upar.Add("k3",    p[16], 0.1);
        upar.Add("xr",    p[17], 0.1);
        upar.Add("yr",    p[18], 0.1);
        upar.Add("x1y",   p[19], 0.1);
        upar.Add("x2y",   p[20], 0.1);
        upar.Add("x3y",   p[21], 0.1);
        upar.Add("y1x",   p[22], 0.1);
        upar.Add("y2x",   p[23], 0.1);
        upar.Add("y3x",   p[24], 0.1);
        upar.Add("b1x",   p[25], 0.1);
        upar.Add("b1y",   p[26], 0.1);
  }


  // The function to be minimized, defined in Residual.h,*.cpp and
      ResidualFCN.h,*.cpp
  ResidualFCN func(Markers,Pixels);  // ResidualFCN(vector<TVector3>, vector
      <TVector2>)

  // create MIGRAD minimizer
  MnMigrad migrad(func, upar);          // function, user parameters
  // This creates a function Residual(Markers,Pixels,Params), which can be
      used later
  // to reproduce useful parameter things such as calculated points and
      residuals

  // Fix x,y,z location of camera
  for(int i=0; i<3; i++)
        migrad.Fix(i);

// New [11/24/2011] minimization routine that hopefully works.
// Fix xr, fit yr, then fix yr, fit xr. Repeat a few times.
// Note: [12/28/2011] The parameters xr,yr are flat for most
// of their parameter space, and only spike up at one end. This
// can make it difficult to fit for them.

  gErrorIgnoreLevel = kFatal;           // silences minuit2 output
  TStopwatch s;
  s.Start();

  FunctionMinimum min = migrad();
  int count = 0;

  // Done for the cameras facing each other to get the correct Euler angles
      out of the fit.
  if(camera_num >= 10 && camera_num < 20) {
    for(int i=0; i<parsize; i++) {
        if(i<3 || i>5)                  // fix all but Euler angles
                migrad.Fix(i);
```

```
    }   // end for
    min = migrad();
  }     // end if


  // This is for the 19 parameter model - polynomial/radial mix
  else if(camera_num < 60 && camera_num >= 50) {
  // Here we run the fit 10 times, alternating between fixed 'xr', free 'yr'
      and free 'xr', fixed 'yr'.
  // Alternating like this allows the res to drop and make the fit better.
  for(int i=0; i<5; i++) {
  migrad.Fix("xr");
        min = migrad();
        cerr << endl << "*** min totalres (" << i*2  << ") = " << min.
            UserState().Fval() << " ***" << endl;
        cerr << "xr = " << migrad.Value("xr") << ", yr = " << migrad.Value("
            yr") << endl;
//      cerr << min;
  migrad.Release("xr");
  migrad.Fix("yr");
        min = migrad();
        cerr << endl << "*** min totalres (" << i*2 + 1 << ") = " << min.
            UserState().Fval() << " ***" << endl;
        cerr << "xr = " << migrad.Value("xr") << ", yr = " << migrad.Value("
            yr") << endl;
//      cerr << min;
  migrad.Release("yr");
  count += 2;
  } // end for
  } // end if


  // This section of the code will scan over a large range of xr,yr values
      and
  // create a histogram that will contain the residuals for each pair. It
      should
  // help identify local minima so small ones can be avoided.

  // A fit with the basic parameters will still be performed and displayed
      as output.
  else if(camera_num < 70 && camera_num >= 60) {

    int range = 31;                     // number of iterations in x and y
    double scanstep = 1000;                     // scan step for xr,yr
    double hlim = scanstep*double(range)/2;     // histogram lower and upper
        limits
    double xoff = 0;
    double yoff = 0;
    cerr << "hlim = " << hlim << ", xoff = " << xoff << ", yoff = " << yoff
        << endl;

    size_t pos = pxname.find("/");
    string htitle = "Residual Scan for " + pxname.substr(pos+1,7);
    TH2D *scanres = new TH2D("scanres",htitle.data(),range,-hlim+xoff,hlim+
        xoff,range,-hlim+yoff,hlim+yoff);

    for(int i=0; i<range; i++) {
        for(int j=0; j<range; j++) {
```

```
                ResidualFCN func(Markers,Pixels);       // ResidualFCN(
                    vector<TVector3>, vector<TVector2>)
                MnMigrad migrad(func,upar);             // function, user
                    parameters
                migrad.Fix("X0");       migrad.Fix("Y0");       migrad.Fix("
                    Z0");
                migrad.Fix("xr");       migrad.Fix("yr");

                migrad.SetValue("xr",p[17] + xoff + scanstep*(double(i) -
                    double(range-1)/2));
                migrad.SetValue("yr",p[18] + yoff + scanstep*(double(j) -
                    double(range-1)/2));
                cerr << "i = " << i << ", j = " << j << endl;
                min = migrad();
                cerr << "xr = " << migrad.Value("xr") << ", yr = " << migrad
                    .Value("yr") << endl;
                cerr << "min.UserState.Fval() = " << min.UserState().Fval()
                    << endl;
                scanres->Fill(migrad.Value("xr"),migrad.Value("yr"),min.
                    UserState().Fval());
        }
    }   // end for(i<range)

    string dir = "~/Desktop/Camera_work/Photos/Swimming_Pool_Test_Sept7_2011
        /Analysis/Thesis_Plots/";
    string parout = dir + "Scanhist_" + pxname.substr(pos + 1,10) + ".root";
    TFile *f = new TFile(parout.data(), "RECREATE");
    scanres->Write();
    cerr << "Wrote scanres histogram to " << parout << endl;
    f->Close();

    migrad.Release("xr");
    migrad.Release("yr");

}     // end if

// for the 24 parameter model - polynomial + radial + y-only radial
else if(camera_num < 80 && camera_num >= 70) {
// Here we run the fit several times, alternating between fixed 'xr', free
     'yr' and free 'xr', fixed 'yr'.
// Alternating like this allows the residual to drop and make the fit
    better.
for(int i=0; i<5; i++) {
// float xr
migrad.Fix(18);       migrad.Fix(22); migrad.Fix(23);
      min = migrad();
      cerr << endl << "*** min totalres (" << i*4  << ") = " << min.
          UserState().Fval() << " ***" << endl;
      cerr << "xxr = " << migrad.Value("xxr") << ", xyr = " << migrad.
          Value("xyr") << endl;
      cerr << "yxr = " << migrad.Value("yxr") << ", yyr = " << migrad.
          Value("yyr") << endl;
migrad.Release(18);   migrad.Release(22);     migrad.Release(23);
// float yr
migrad.Fix(17);       migrad.Fix(22); migrad.Fix(23);
      min = migrad();
      cerr << endl << "*** min totalres (" << i*4 + 1 << ") = " << min.
          UserState().Fval() << " ***" << endl;
```

```
        cerr << "xxr = " << migrad.Value("xxr") << ", yr = " << migrad.Value
            ("xyr") << endl;
        cerr << "yxr = " << migrad.Value("yxr") << ", yyr = " << migrad.
            Value("yyr") << endl;
  migrad.Release(17);   migrad.Release(22);     migrad.Release(23);
  // float yxr
  migrad.Fix(17);       migrad.Fix(18); migrad.Fix(22);
        min = migrad();
        cerr << endl << "*** min totalres (" << i*4 + 2 << ") = " << min.
            UserState().Fval() << " ***" << endl;
        cerr << "xxr = " << migrad.Value("xxr") << ", xyr = " << migrad.
            Value("xyr") << endl;
        cerr << "yxr = " << migrad.Value("yxr") << ", yyr = " << migrad.
            Value("yyr") << endl;
  migrad.Release(17);   migrad.Release(18);     migrad.Release(22);
  // float yyr
  migrad.Fix(17);       migrad.Fix(18); migrad.Fix(23);
        min = migrad();
        cerr << endl << "*** min totalres (" << i*4 + 3 << ") = " << min.
            UserState().Fval() << " ***" << endl;
        cerr << "xxr = " << migrad.Value("xxr") << ", yr = " << migrad.Value
            ("xyr") << endl;
        cerr << "yxr = " << migrad.Value("yxr") << ", yyr = " << migrad.
            Value("yyr") << endl;
  migrad.Release(17);   migrad.Release(18);     migrad.Release(23);
  count += 4;
  } // end for
  } // end if

  // other 24 (25) parameter model - polynomial + radial + sin correction to
      y
  else if(camera_num < 90 && camera_num >= 80) {
  // Here we run the fit 10 times, alternating between fixed 'xr', free 'yr'
      and free 'xr', fixed 'yr'.
  // Alternating like this allows the res to drop and make the fit better.
  for(int i=0; i<5; i++) {
  migrad.Fix("xr");
        min = migrad();
        cerr << endl << "*** min totalres (" << i*2  << ") = " << min.
            UserState().Fval() << " ***" << endl;
        cerr << "xr = " << migrad.Value("xr") << ", yr = " << migrad.Value("
            yr") << endl;
//      cerr << min;
  migrad.Release("xr");
  migrad.Fix("yr");
        min = migrad();
        cerr << endl << "*** min totalres (" << i*2 + 1 << ") = " << min.
            UserState().Fval() << " ***" << endl;
        cerr << "xr = " << migrad.Value("xr") << ", yr = " << migrad.Value("
            yr") << endl;
//      cerr << min;
  migrad.Release("yr");
  count += 2;
  } // end for
  } // end if

  // polynomial + radial + extra polynomial
  else if(camera_num < 100 && camera_num >= 90) {
```

```
  // Here we run the fit 10 times, alternating between fixed 'xr', free 'yr'
       and free 'xr', fixed 'yr'.
  // Alternating like this allows the res to drop and make the fit better.
  for(int i=0; i<5; i++) {
  migrad.Fix("xr");
        min = migrad();
        cerr << endl << "*** min totalres (" << i*2  << ") = " << min.
            UserState().Fval() << " ***" << endl;
        cerr << "xr = " << migrad.Value("xr") << ", yr = " << migrad.Value("
            yr") << endl;
//      cerr << min;
  migrad.Release("xr");
  migrad.Fix("yr");
        min = migrad();
        cerr << endl << "*** min totalres (" << i*2 + 1 << ") = " << min.
            UserState().Fval() << " ***" << endl;
        cerr << "xr = " << migrad.Value("xr") << ", yr = " << migrad.Value("
            yr") << endl;
//      cerr << min;
  migrad.Release("yr");
  count += 2;
  } // end for
  } // end if


  // Fitting one last time with xr, yr free gives the correct number of free
       parameters
  min = migrad();
  cerr << endl << "*** min totalres (" << count << ") = " << min.UserState()
      .Fval() << " ***" << endl;
  cerr << min;

  // Initialize parameters for residual plots
  vector<double> Params;
  for(int i=0; i<parsize; i++)
        Params.push_back(min.UserState().Value(i));

  // Initialize the function so we can use its stored values
  Residual Res(Markers,Pixels,Params);

  double* xcalc = new double [fsize];
  double* ycalc = new double [fsize];
  double* xres  = new double [fsize];           // xcalc-xpic
  double* yres  = new double [fsize];           // xcalc-xpic
  double totalres = Res.ResidualSum2();         // sum of x^2 + y^2
      residuals
  double xrestot = 0;                           // sum of x^2 residuals
  double yrestot = 0;                           // sum of y^2 residuals


  for(int i=0; i<fsize; i++) {
        xcalc[i] = Res.Xcalc()[i];
        ycalc[i] = Res.Ycalc()[i];
        xres[i]  = Res.Xres()[i];
        yres[i]  = Res.Yres()[i];
        xrestot += xres[i]*xres[i];
        yrestot += yres[i]*yres[i];
  }
```

```
  // Count number of free parameters to calculate Chi2 correctly
  int nfreepar = 0;
  for(int i=0; i<parsize; i++) {
        if(min.UserState().Parameter(i).IsFixed()) {
        }
        else {
                nfreepar += 1;
        }
  }

  cerr << "*******************************************************" << endl;
  cerr << "Total residual = " << totalres << endl;
  // Find average residual
  cerr << "*******************************************************" << endl;
  cerr << "Average residual r.m.s. length (pixels) = " << sqrt(totalres/
      double(fsize)) << endl;

  // Find the average x and y residuals
  double xresrms = sqrt(xrestot/double(fsize));         // r.m.s. of x-
      residual
  double yresrms = sqrt(yrestot/double(fsize));         // r.m.s. of y-
      residual
  cerr << "xres r.m.s. = " << xresrms << ", yres r.m.s. = " << yresrms <<
      endl;
  cerr << "*******************************************************" << endl;


  // *********************************************************
  // ********** Begin: Save parameters to file ************
  // *********************************************************
{
  // Write the parameters to a file, and name it accordingly
  string directory = "/home/zack/Desktop/Camera_work/Photos/
      Swimming_Pool_Test_Sept7_2011/Analysis/PAR_directory/";
  size_t pos = pxname.find("/");
  string parout = directory + "PAR_" + pxname.substr(pos + 1);
  ofstream prfile (parout.data());
  if (prfile.is_open()) {
        for(int i=0; i<parsize; i++) {
                prfile << min.UserState().Value(i) << " ";
        }
        prfile << endl << totalres << " " << sqrt(totalres/double(fsize)) <<
             " ";
        prfile << xresrms << " " << yresrms << endl;
        // needs a space to be prettyyyyy
        cerr << "Wrote to " << parout << endl;
        prfile.close();
  }
  else cerr << "Unable to open write-file" << endl;
}
  // *********************************************************
  // ************ End: Save parameters to file ************
  // *********************************************************


  // Display graphs and all else
  TApplication *tapp = new TApplication("Graph", &argc, argv);

  // *********************************************************
```

```
  // ******* Begin: Mainfit and Marker Locations **********
  // ******************************************************
{

  TCanvas *c3 = new TCanvas("c3", "Marker locations", 100, 100, 1600, 600);
          // Bonus graphs
        c3->Divide(3,1);
  TCanvas *c2 = new TCanvas("c2", "Main fit", 100, 100, 1800, 1000);
             // Main picture


  TGraph *gpic = new TGraph(fsize, xpic, ypic);         // coords from
     picture
  TGraph *gcalc = new TGraph(fsize, xcalc, ycalc);      // predicted coords
     after fit
  TGraph *gloc = new TGraph(fsize, xloc, zloc);         // x-z coords from 3
     D locations
  gpic->SetTitle("Picture marker locations (from TSpectrum2)");
  gcalc->SetTitle("Fitted marker locations (reproduced after fitting for
     unknowns)");
  gloc->SetTitle("x,z coordinates from 3D marker locations");
  gloc->GetXaxis()->SetTitle("x location (cm)");
  gloc->GetYaxis()->SetTitle("z location (cm)");
  gloc->GetYaxis()->SetLabelSize(0.03);
  gloc->GetXaxis()->SetLabelSize(0.03);
  gloc->GetYaxis()->SetTitleSize(0.03);
  gloc->GetYaxis()->SetTitleOffset(1.5);
  gloc->GetXaxis()->SetTitleSize(0.03);
  gpic->SetMarkerStyle(20);
  gpic->SetMarkerColor(3);              // pic = green
  gcalc->SetMarkerStyle(20);
  gcalc->SetMarkerColor(2);             // calc = red
  gloc->SetMarkerStyle(20);
  gloc->SetMarkerColor(4);              // x,y from 3D = blue

  // Draw the multigraph on the large canvas
  TMultiGraph *mg = new TMultiGraph();
  mg->Add(gpic);
  mg->Add(gcalc);
  c2->cd();      mg->Draw("AP");
  mg->SetTitle("Picture locations (green) vs. Fitted locations (red)");
  mg->GetXaxis()->SetTitle("x (pixels)");
  mg->GetYaxis()->SetTitle("y (pixels)");
  c2->cd();      mg->Draw("AP");

  // Mark the principal (x,y) point with a black cross
  c2->cd();
  TMarker *mr = new TMarker(Params[6],Params[7],1);
  mr->SetMarkerStyle(34);
  mr->SetMarkerColor(1);
  mr->Draw("same");

  // Draw arrows from pic[i] to calc[i] points
  TArrow** arrow = new TArrow* [fsize];
  for(Int_t i=0; i<fsize; i++) {
        arrow[i] = new TArrow(xpic[i],ypic[i],xcalc[i],ycalc[i],0.01,"|-|");
        arrow[i]->SetAngle(50);
        arrow[i]->SetLineWidth(2);
        arrow[i]->Draw();
```

```
  }

  // Draw the individual graphs on the small canvas
  c3->cd(1);    gpic->Draw("APL");
  c3->cd(2);    gcalc->Draw("APL");
  c3->cd(3);    gloc->Draw("APL");

}
  // ******************************************************
  // **********End: Mainfit and Marker Locations **********
  // ******************************************************

  //  ************************************************************ //
  //  ****************** Shallow end ***************************** //
  //  ************************************************************ //
{
  // draw_vertices(TCanvas, x, y, z, phi, theta, psi, length, color)
  // draw_grid(TCanvas, x1, x2, y1, y2, z1, z2, nlinesx, nlinesy, color,
      style)
  // draw_line(TCanvas, x1, y1, z1, x2, y2, z2, color, style)
  // draw_marker(TCanvas, x, y, z)

  TCanvas *c = new TCanvas("c3D","3D Shallow End",1600,1000);
  TView *view2 = TView::CreateView(1);          // some vaguely defined ROOT
       stuff from TPolyLine3D
  // SetRange(x1,y1,z1, x2,y2,z2) I think
  double ran2[] = {-400,0,0,500,545.7,200};     // Range to be used in view
      ->SetRange()
  view2->SetRange(ran2[0],ran2[1],ran2[2],ran2[3],ran2[4],ran2[5]);     //
      what is this I don't even

  // Draws the entire shallow end
  draw_shallowend(c);

  // Draw camera location and direction it's facing
  //  draw_vertices(c,Params[0],Params[1],Params[2],Params[3],Params[4],
      Params[5],50,2);

/*
  // Draw other cameras
  for(int i=0; i<6; i++) {
        draw_line(c,pc[i][0],pc[i][1],pc[i][2],pc[i][0],pc[i][1],flht(pc[i
            ][1]),1,2);
        if(camera_num-10 != i)          // So we don't redraw the fitted
            camera, only surrounding cameras
                draw_vertices(c,pc[i][0],pc[i][1],pc[i][2],pc[i][3],pc[i
                    ][4],pc[i][5],50,633);
  }
*/

  // Draw markers on the floor where the cameras would be
  for(int i=0; i<6; i++) {
        draw_marker(c,pc[i][0],pc[i][1],flht(pc[i][1]));
  }


  // Draw a line showing object height
  draw_line(c,0,0,0,0,0,flht(0),2,2);           // origin vertex to -11.75
```

```
  // Draw markers for object positions
  TPolyMarker3D *polymark = new TPolyMarker3D(fsize, ptr, 20);
  polymark->SetMarkerColor(4);
  polymark->SetMarkerSize(0.20);
  polymark->Draw();

//  view2->SetPerspective();                    // gives it a proper 3D
    perspective view
  view2->ShowAxis();
  TAxis3D *axis = TAxis3D::GetPadAxis();
  axis->SetAxisColor(kGreen+1,"Y");
  axis->SetLabelColor(kGreen+1,"Y");
  axis->SetXTitle("");
//  axis->ToggleZoom();                         // lets you adjust the axes
//  axis->SetLabelColor(kBlack);
//  axis->SetAxisColor(kBlack);
}
  //  ************************************************************* //
  //  ******************** Residual Plots ************************* //
  //  ************************************************************* //
{

  TCanvas *c = new TCanvas("cres","Residuals",100,100,1900,1000);
  draw_residuals(c,fsize,xcalc,ycalc,xres,yres,xpic,ypic);

}
  cerr << "Real time = " << s.RealTime() << " and Cpu time = " << s.CpuTime
      () << endl;
  tapp->Run();

  return 0;
}
```

B.4 Position_Solve_final.cpp

```
// Zachary Petiw
// Jan 30, 2012

// Position_Solve_final.cpp

// This program takes camera position, angle and distortion parameters
// that we found before using Intrinsic_Solve, and together with x,y pixel
// coordinates of an object (objects), will solve for that object's 3-space
// position.

// Modification of Intrinsics_Solve_decouple.cpp.

// [NEW] (01/30/11) Changes made from Intrinsics_Solve_decouple.cpp to
    include
// models with different sizes. Minuit2 functions must be rewritten to allow
// this.

#include <iostream>
#include <vector>
#include <fstream>
#include <sstream>
#include <stdio.h>
#include <iomanip>
#include "TVector2.h"
#include "TVector3.h"
#include "TRotation.h"
#include "ChiFcnSolve.h"
#include "TMath.h"
#include "TStopwatch.h"
#include "TF2.h"
#include "TApplication.h"
#include "TCanvas.h"
#include "TPad.h"
#include "TGraph.h"
#include "TMultiGraph.h"
#include "TAxis.h"
#include "TPolyMarker.h"
#include "TMarker.h"
#include "TArrow.h"
#include "TPolyMarker3D.h"
#include "TPolyLine3D.h"
#include "TView.h"
#include "TH1.h"
#include "DrawFunc.h"
#include "DrawResiduals.h"
#include "Triang.h"
#include "TriangFCN.h"
#include "gStyles.h"                                // my own gStyles
#include "TPaveText.h"
#include "TLegend.h"
#include "TAxis3D.h"
#include "TROOT.h"
#include <Minuit2/FunctionMinimum.h>
#include <Minuit2/MnPrint.h>
#include <Minuit2/VariableMetricMinimizer.h>
```

```
#include <Minuit2/MnMigrad.h>
#include <Minuit2/MnMinos.h>
#include <Minuit2/MnContours.h>
#include <Minuit2/MnPlot.h>

using namespace ROOT::Minuit2;
using namespace std;

// This function will create a stats box for the x,y,z residuals of a TGraph
// Graphstats(number_of_points, array_x_residuals, array_y_residuals,
    array_z_residuals)

TPaveText* graphstats(int npoints, double* x, double* y, double* z) {

  TPaveText *pt = new TPaveText(0.19,0.69,0.57,0.87,"NDC");
  string pts[3];
  pts[0] = "Avg. X residual: ";
  pts[1] = "Avg. Y residual: ";
  pts[2] = "Avg. Z residual: ";
  double** v = new double* [3];
  v[0] = x;     v[1] = y;       v[2] = z;

  for(int i=0; i<3; i++) {
        ostringstream mean,rms;
        mean << fixed << setprecision(4) << Mean(npoints,v[i]);
        rms << fixed << setprecision(4) << RMS(npoints,v[i]);
        pts[i] = pts[i] + mean.str() + " +/- " + rms.str() + " cm";
        pt->AddText(pts[i].data());
        pt->SetBorderSize(3);
  }

  return pt;

}       // end TPaveText statistics box

// Some marker styles - 20,21,22,31,33,34
int mstyle(int c) {
        if(c%6==0)      return 20;
        else if(c%6==1) return 21;
        else if(c%6==2) return 22;
        else if(c%6==3) return 31;
        else if(c%6==4) return 33;
        else if(c%6==5) return 34;
        else return 20;
}

// Some colors
int mcolor(int c) {
        if(c%6==0)      return 632;             // kRed
        else if(c%6==1) return 418;             // kGreen+2
        else if(c%6==2) return 600;             // kBlue
        else if(c%6==3) return 807;             // kOrange+7
        else if(c%6==4) return 617;             // kMagenta+1
        else if(c%6==5) return 870;             // kAzure+10
        else return 1;
}


// ****************************************** //
```

```
// ************ Main program ****************** //
// ******************************************** //

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

  cerr << endl;

  if(argc != 4) {
        cerr << "Usage: Position_Solve Parameters.txt Pixels_P12345.txt 3
            d_coords.txt (if known)" << endl;
        cerr << endl;
  return -1;
  }

  string parname = argv[1];     // File containing rows of camera parameters
  string pxname = argv[2];      // File containing rows of pixel positions
           (TSpectrum2 output)
  string coname = argv[3];      // File of 3d coordinates if known

  // Open the camera parameters file
  ifstream parfile;
  parfile.open(parname.data());

  if(!parfile) {
        cerr << "...Cannot open camera parameters file" << endl;
  return -3;
  }

  // Open the pixel position file
  ifstream pxfile;
  pxfile.open(pxname.data());

  if(!pxfile) {
        cerr << "...Cannot open pixelcoord file" << endl;
  return -4;
  }

  cerr << "The files we are opening are:" << endl;
  cerr << "     " << parname << ", and " << endl;
  cerr << "     " << pxname << endl << endl;

  // Get each file's size and make sure they are the same
  string dummy;                 // dummy string to hold lines from file
  double dpar;                  // dummy parameter holder
  int ncam = 0;                 // number of cameras used = number of lines
      in parameter file
  int npar = 0;                 // number of parameters in a line (or in the
       model)
  int totalpar = 0;             // total parameters in file, used as a check
       I suppose
  vector<double> Params;                        // Camera parameters, npar*
      ncam

  // Read in the number of lines in parameter file, and the parameters as
      well
  // can use is.str() to view contents of istringstream object, or view the
      components separately
  while(getline(parfile,dummy)) {
        ++ncam;
```

```
      istringstream is(dummy);
      while(is >> dpar) {
              Params.push_back(dpar);
              ++totalpar;
              if(ncam == 1) ++npar;
      }
}
cerr << "npar = " << npar << endl;

int pxsize = 0;                                 // pixel file size
vector<TVector2> Pixels;                        // Pixel coords[ n_cam *
    n_points ]
double xdum,ydum;                               // dummy x,y pixel
    coordinates
TVector2 pxdum;                                 // dummy TVector2

// Input all the 2d pixel coords to the Pixels vector
// This will contain ncam*npoints entries, i.e. 24 lines for all 6 cameras
     and 4 points
// Should be structured: CameraP coords, Camera1 coords, etc.

while(getline(pxfile,dummy)) {
      ++pxsize;
      istringstream is(dummy);
      is >> xdum >> ydum;
      pxdum.Set(xdum,ydum);
      Pixels.push_back(pxdum);
}

int npoints = pxsize/ncam;                              // number of points
    used in the fit

cerr << " Our dear parameters: " << endl;
for(int i=0; i<totalpar; i++) {
      cerr << Params[i] << " ";
      if((i+1)%npar==0)
              cerr << endl;
}
cerr << endl;

cerr << "ncam = " << ncam << " and npoints = " << npoints << endl;

cerr << "Parameter file has " << ncam << " lines" << endl;
cerr << "Pixel file has " << pxsize << " entries for " << npoints << "
    point(s)" << endl;

if(pxsize%ncam != 0) {
      cerr << "Pixel file and parameter file don't add up! Remainder of "
          << pxsize%ncam << endl;
      cerr << "Exiting.... try again, foo" << endl;
      return -1;
}

if(totalpar%ncam != 0) {
      cerr << "Parameter file and number of cameras doesn't add up!
          Remainder of " << npar%ncam << endl;
      cerr << "Exiting.... try again, foo" << endl;
      return -1;
}
```

```
cerr << "npar = " << npar << endl;

// We also read in the 3d values so that we can compare them later
int n3d = 0;                                    // number of entries in 3d
    file
vector<double> Realpos;                         // actual 3d positions from
    file

ifstream cofile;
cofile.open(coname.data());
while(getline(cofile,dummy)) {
      ++n3d;
      istringstream is(dummy);
      while(is >> dpar)
              Realpos.push_back(dpar);
}

// Return sizes if you are interested
cerr << "Params.size() = " << Params.size() << endl;
cerr << "Pixels.size() = " << Pixels.size() << endl;
cerr << "Realpos.size() = " << Realpos.size() << endl;
cerr << endl;

TApplication *app = new TApplication("Viewer", &argc, argv);
TCanvas *c2 = new TCanvas("c2", "Viewer", 100, 100, 1600, 900);       //
    our 3d viewer
draw_shallowend(c2);                                    // draw shallow end
    in 3D on chosen canvas


// We need to make a loop here that will select a set of 6 (x,y) locations
     and send
// them to a function that will triangulate the point off of which they
    are based.

vector<double> Pos;                     // This will store the positions
    from each triangulation
gErrorIgnoreLevel = kFatal;             // silences Minuit2 output

TGraph** gcam = new TGraph* [ncam];     // One graph for each camera's pixel
     residuals, all-camera fit
TGraph** gres = new TGraph* [3];        // One graph for each dimension, all
    -camera fit
TGraph** gres5 = new TGraph* [ncam*3];          // One graph for each
    dimension * each camera in 5 camera fits
for(int i=0; i<ncam; i++)
      gcam[i] = new TGraph(npoints);
for(int i=0; i<3; i++)
      gres[i] = new TGraph(npoints);
for(int i=0; i<ncam*3; i++)
      gres5[i] = new TGraph(npoints);

double f5min = 1e9;                     // global variable for highest point
     in 5 cam graphs
double f5max = -1e9;                    // global variable for lowest point
    in 5 cam graphs

for(int k=0; k<npoints; k++) {
```

```
// First we create a subset of the total Pixels vector to isolate
    one set with 'ncam' values
vector<TVector2> Pixcopy;
Pixcopy.clear();
for(int i=0;i<ncam;i++) {
        Pixcopy.push_back(Pixels[(i*npoints)+k]);
}

// Initial starting parameters - X,Y,Z position of object
double p[] = {100,500,100};

MnUserParameters upar;
// name, initial value, error;
upar.Add("X", p[0], 0.1);
upar.Add("Y", p[1], 0.1);
upar.Add("Z", p[2], 0.1);

TriangFCN func(Params,Pixcopy);         // Triang(vector<double>,
    vector<TVector2>)
MnMigrad migrad(func, upar);            // migrad minimizer(
    function, user parameters )
FunctionMinimum min = migrad();         // minimize

for(int i=0; i<3; i++)
        Pos.push_back(min.UserState().Value(i));                //
            store the results

vector<double> pos (Pos.begin()+k*3,Pos.begin()+(k+1)*3);       //
    vector of this result

Triang Result(Params,Pixcopy,pos);

double res = Result.ResidualSum();
double* xcalc = new double [ncam];              // calculated x-
    pixel point for a camera
double* ycalc = new double [ncam];              // calculated y-
    pixel point for a camera
double* xres  = new double [ncam];              // xcalc-xpic
double* yres  = new double [ncam];              // xcalc-xpic

for(int i=0; i<ncam; i++) {
        xcalc[i] = Result.Xcalc()[i];
        ycalc[i] = Result.Ycalc()[i];
        xres[i]  = Result.Xres()[i];
        yres[i]  = Result.Yres()[i];
}

for(int i=0; i<ncam; i++) {
        gcam[i]->SetPoint(k,Result.Xres()[i],Result.Yres()[i]);
}

// Fill the graphs of X,Y, and Z residuals
for(int i=0; i<3; i++)
        gres[i]->SetPoint(k,k,pos[i]-Realpos[k*3+i]);

// Draw the real position from the 3d.txt file
draw_marker(c2,Realpos[k*3],Realpos[k*3+1],Realpos[k*3+2],4);
```

```
        draw_line(c2,Realpos[k*3],Realpos[k*3+1],Realpos[k*3+2],Realpos[k
            *3],Realpos[k*3+1],flht(Realpos[k*3+1]));
        // Draw the points we solved for
        draw_marker(c2,pos[0],pos[1],pos[2],2);
        draw_line(c2,pos[0],pos[1],pos[2],pos[0],pos[1],flht(pos[1]));

        delete xcalc;
        delete ycalc;
        delete xres;
        delete yres;

        cerr << k << "... ";

        // Another loop where we exlude one camera and repeat the test
        for(int m=0; m<ncam; m++) {

                vector<TVector2> Pixels5 (Pixcopy);             // Copy of
                    pixel coords
                vector<double> Params5 (Params);                // Copy of
                    camera parameters, npar*ncam

                // Need to remove a line from Pixels and Params in order to
                    exclude a camera
                Pixels5.erase (Pixels5.begin()+m);
                Params5.erase (Params5.begin()+m*npar, Params5.begin()+(m+1)
                    *npar);

                TriangFCN func(Params5, Pixels5);
                MnMigrad migrad(func, upar);            // function, user
                    parameters
                FunctionMinimum min5 = migrad();

                vector<double> pos5;
                pos5.clear();
                for(int i=0; i<3; i++)
                        pos5.push_back(min5.UserState().Value(i));      //
                            store the resulting positions

                // Fill the graph of X,Y,Z residuals for each of the 5-
                    camera fits
                for(int i=0; i<3; i++) {
                        gres5[m*3+i]->SetPoint(k,k,pos5[i]-Realpos[k*3+i]);
                        // record if a new fmin or fmax was exceeded
                        if((pos5[i]-Realpos[k*3+i]) > f5max)
                                f5max = pos5[i]-Realpos[k*3+i];
                        if((pos5[i]-Realpos[k*3+i]) < f5min)
                                f5min = pos5[i]-Realpos[k*3+i];
                }

        }       // end short m for-loop

}               // end longer k for-loop

// *** Draw the multigraph
{
MyThesisStyle();                        // This sets the graph display style
     - all white, more formal
```

```
  TCanvas *cres1 = new TCanvas("cres1", "Pixel Residuals for Each Camera"
      ,100,50,600,500); // px residuals, all 6
  TCanvas *cres2 = new TCanvas("cres2", "cm Residuals for Each Camera"
      ,700,50,600,500); // cm residuals, all 6

  TMultiGraph *mg = new TMultiGraph();                  // store the 2d
      camera-specific residuals
  TMultiGraph *mgres = new TMultiGraph();               // store the 3d
      residuals

  for(int i=0; i<ncam; i++) {
        gcam[i]->SetMarkerColor(mcolor(i));
        gcam[i]->SetMarkerStyle(mstyle(i));
//      gcam[i]->SetFillColor(424);                     // defines box color
     in TLegend
//      gcam[i]->SetFillStyle(3003);                    // box style in
    TLegend - sparse dots
        ostringstream os;
        os << i;
        string gname = "Camera " + os.str();
        gcam[i]->SetTitle(gname.data());
        mg->Add(gcam[i]);
  }

  cres1->cd();
  gPad->SetGrid();                      // gPad = pointer to current pad
  mg->Draw("AP");
  mg->SetTitle("Residuals for Each Camera");
  mg->GetXaxis()->SetTitle("x residual (pixels)");
  mg->GetYaxis()->SetTitle("y residual (pixels)");

  TLegend *legr = new TLegend(0.20,0.65,0.45,0.88);
  for(int i=0; i<ncam; i++)
        legr->AddEntry(gcam[i],gcam[i]->GetTitle(),"pl");
  legr->SetMargin(0.4);
  legr->Draw();

  for(int i=0; i<3; i++) {
        gres[i]->SetMarkerColor(mcolor(i));
        gres[i]->SetMarkerStyle(mstyle(i));
        mgres->Add(gres[i]);
  }
        gres[0]->SetTitle("X residual");
        gres[1]->SetTitle("Y residual");
        gres[2]->SetTitle("Z residual");

  cres2->cd();
  gPad->SetGridy();                     // gPad = pointer to current pad
  gPad->SetTicky();
  mgres->Draw("APL");
  mgres->SetTitle("Calculated Position Residuals");
  mgres->GetXaxis()->SetTitle("Point number");
  mgres->GetYaxis()->SetTitle("Residual in position (cm)");
  double fmin,fmax;                             // min/max values on axis,
      so that they can be shifted down
  fmin = mgres->GetYaxis()->GetXmin();
  fmax = mgres->GetYaxis()->GetXmax();
  mgres->GetYaxis()->SetRangeUser(fmin, (fmax - 0.3*fmin)/0.7);
```

```
// change this to build a legend

TLegend *leg = new TLegend(0.61,0.69,0.88,0.87);
for(int i=0; i<3; i++)
      leg->AddEntry(gres[i],gres[i]->GetTitle(),"pl");
leg->SetMargin(0.4);
leg->Draw();

// graphstats(npoints, x_array, y_array, z_array)
graphstats( npoints, gres[0]->GetY(), gres[1]->GetY(), gres[2]->GetY() )->
    Draw();
}     // end create main multigraph

/*********************************************************/

{     // begin create 5 camera multigraphs

TCanvas *cress = new TCanvas("cress", "cm residuals for a single camera"
    ,1000,50,600,500); // cm residuals, single graph to duplicate
TCanvas *cres5 = new TCanvas("cres5", "Residual for Each Camera, minus one
    ",100,100,1600,900);
cres5->Divide(3,2);
TMultiGraph** mg5 = new TMultiGraph* [ncam];
for(int i=0; i<ncam; i++)
      mg5[i] = new TMultiGraph();

int chosen_cam = 2;
for(int i=0; i<ncam; i++) {
      for(int j=0; j<3; j++) {
              gres5[i*3+j]->SetMarkerColor(mcolor(j));
              gres5[i*3+j]->SetMarkerStyle(mstyle(j));
              gres5[i*3+j]->SetFillColor(424);                        //
                  defines box color in TLegend
              gres5[i*3+j]->SetFillStyle(3003);                       //
                  box style in TLegend - sparse dots
              mg5[i]->Add(gres5[i*3+j]);
      }

      gres5[i*3]->SetTitle("X residual");
      gres5[i*3+1]->SetTitle("Y residual");
      gres5[i*3+2]->SetTitle("Z residual");

      // do some ostringstream magic here
      ostringstream os, oss;
      os << i;
      oss << i+1;
      string rtitle = "Calculated Position Residuals, minus Camera " + os.
          str();

      string pname = cres5->GetName();
      pname += "_" + oss.str();

      TPad *cpad = (TPad*) cres5->GetListOfPrimitives()->FindObject(pname.
          data());

      cpad->cd();
      cpad->SetGridy();               // gPad = pointer to current pad
      cpad->SetTicky();
      mg5[i]->Draw("APL");
```

```
      mg5[i]->SetTitle(rtitle.data());
      mg5[i]->GetXaxis()->SetTitle("Point number");
      mg5[i]->GetYaxis()->SetTitle("Residual in position (cm)");
      mg5[i]->GetYaxis()->SetRangeUser(f5min,(f5max-0.3*f5min)/0.7);

      TLegend *leg = new TLegend(0.61,0.69,0.88,0.87);
      for(int n=0; n<3; n++)
              leg->AddEntry(gres5[i*3+n],gres5[i*3+n]->GetTitle(),"pl");
      leg->SetMargin(0.4);
      leg->Draw();

      // draw an extra graph on a separate canvas if this is the chosen
          graph
      if(i == chosen_cam) {
      cress->cd();
      cress->SetGridy();
      cress->SetTicky();\
      TMultiGraph *mg5copy(mg5[i]);   // create a copy so we can edit
          RangeUser
      mg5copy->GetYaxis()->SetRangeUser(-1,1.5);
      mg5copy->Draw("APL");
      leg->Draw();
      }

      // draw the custom stats box
      graphstats( npoints, gres5[i*3]->GetY(), gres5[i*3+1]->GetY(), gres5
          [i*3+2]->GetY() )->Draw();
}     // end of ncam-loop
}     // end create 5 camera multigraphs

// ***********************************************************
// ******* Draw stuff ********************************************
// ***********************************************************

// draw_grid(TCanvas, x1, x2, y1, y2, z1, z2, nlinesx, nlinesy, color) -
    no style
// draw_line(TCanvas, x1, y1, z1, x2, y2, z2, color, style)
// draw_vertices(TCanvas, x, y, z, phi, theta, psi, length, color)

// Draw the x,y,z axes for each camera's internal coordinate system
// draw_vertices(TCanvas, X0, Y0, Z0, phi, theta, psi, magnitude = 1,
    color = 2)
for(int i=0; i<ncam; i++) {
      draw_vertices(c2,Params[i*npar],Params[i*npar+1],Params[i*npar+2],
          Params[i*npar+3],Params[i*npar+4],Params[i*npar+5],50);
      draw_line(c2,Params[i*npar],Params[i*npar+1],Params[i*npar+2],Params
          [i*npar],Params[i*npar+1],flht(Params[i*npar+1]),1,2);
}

// Draw the superior coordinate system axes
draw_vertices(c2,0,0,0,0,0,0,100,1);
draw_line(c2,0,0,0,0,0,flht(0),1,2);

TView *view = TView::CreateView(1);                  // you need this
view->SetRange(-200,0,0,300,500,200);
view->ShowAxis();
TAxis3D *axis = TAxis3D::GetPadAxis();
axis->SetLabelColor(kBlack);
axis->SetAxisColor(kBlack);
```

```
/*
  // Draw the actual position
  draw_marker(c2,realx[0],realx[1],realx[2],3);
  draw_line(c2,realx[0],realx[1],realx[2],realx[0],realx[1],flht(realx[1]));

  // Draw the other points from solving 5 at a time
  for(int i=0; i<ncam; i++) {
        draw_marker(c2,result5[i][0],result5[i][1],result5[i][2],4);
        draw_line(c2,result5[i][0],result5[i][1],result5[i][2],result5[i
            ][0],result5[i][1],flht(result5[i][1]));
  }
*/

  app->Run();

}       // end of main()
```

B.5 slope_sort.cpp

```
// Zachary Petriw
// Based on corner_sort.cpp from May 31, 2011

// slope_sort.cpp

// Adjustments made on Aug 26, 2011 - allows use of slope to find next point

// Works like corner_sort, looking for the lowest x+y sum to find the bottom
    -
// left corner, but it also uses the slope between points n and n-1 to
    choose
// a suitable n+1 y-value. Once it hits the end of the row, the slope is
    reset
// and it starts again. The value "exclusion_width" will therefore only be
// critical for finding the correct 2nd point in a row and setting up a
// suitable slope. It will no longer be responsble for finding every point
// after the 1st in a row.

// To be used in conjunction with with the other camera calibration software
    .

// Usage:
// ./slope_sort.cpp file.txt rowsize exclusion_width ytolerance
    stop_and_draw

#include <string>
#include <vector>
#include <iostream>
#include <fstream>
#include <sstream>
#include <math.h>
#include "TMath.h"
#include "TApplication.h"
#include "TCanvas.h"
#include "TGraph.h"
#include "TImage.h"
#include "TASImage.h"
#include "TMarker.h"
#include "TPolyMarker.h"
#include "TROOT.h"
#include "TColor.h"
#include "TBox.h"

using namespace std;
using namespace TMath;

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

  if(argc != 6) {
        cout << "Usage: ./slope_sort.c file.txt rowsize x_exclusion_width y-
            tolerance stop_and_draw " << endl;
  return -1;
  }

  string fname = argv[1];               // Name of file containing x,y pairs
```

```
int rowsize = atoi(argv[2]);          // Rowsize, for sorting purposes
double exc = atof(argv[3]);           // Size of x-exclusion range
double ytol = atof(argv[4]);          // y-tolerance: should take values
    of about 1-5 pixels, but not 30
int stop = atoi(argv[5]);             // Stop and draw after this many
    iterations

// Display things
TApplication *graphwin = new TApplication("Graph", &argc, argv);

// Check that the input is actually a .txt file
size_t check;
check = fname.find_last_of(".");      // find where the .txt or .csv
    extension is
string exten (fname, check+1, 3);     // turn the extension into a string

if((exten != "txt") && (exten != "csv")) {
      cout << "You need to input a .txt or .csv file!" << endl;
      cout << "Stop trying to break things." << endl;
return -2;
}

cout << endl << "The file you are trying to open is = " << fname << endl;

// Open the file
ifstream infile;
infile.open(fname.data(), ios::in);

if(!infile) {
      cout << "...Cannot open file" << endl;
      return 1;
}

// Get file size before you store the values
Int_t fsize = 0;                      // Number of x,y pairs in file
string dummy;                         // dummy string to hold a line from
    the file

// Read in each line
while(getline(infile,dummy)) {
      ++fsize;
}

// Reset eof bit so that tellg() and seekg() will actually work
// If the eof bit remains at 1, seekg() and tellg() won't work properly!!!
infile.clear();

double* xval = new double [fsize];            // Array of x values
double* yval = new double [fsize];            // Array of y values

// Read the file and store the values
infile.seekg(0, ios::beg);                    // Reset counter back to
    beginning
infile.precision(9);
cout.precision(9);
for(Int_t i=0; i<fsize; i++) {
      infile >> xval[i] >> yval[i];           // Read in the values
}
```

```
// Stick the points into vectors
// This will allow points to be removed as they are sorted out
vector<double> xvec (xval, xval + fsize);
vector<double> yvec (yval, yval + fsize);

// Debugging canvas
TCanvas *cdebug = new TCanvas("cdebug","cdebug",100,100,1200,850);


// To sort, we need to first find the bottom left corner by
// summing x and y and finding the minimum value. Then we move
// across to the left until we have found npoints == rowsize,
// and reset to the left.
double* sortxval = new double [fsize];        // Sorted x coords
double* sortyval = new double [fsize];        // Sorted y coords

double tempsum;                         // to be compared against xysum
double leftlim;                         // left x-limit of searchable area
double prevd;                           // previous distance between last
    point and current point
double currd;                           // current distance between last
    point and current point
int idx;                                // index of current lowest sum

// Made functional Aug 26, 2011.
// This variable will place a tolerance on the next y-value that can
// be accepted. The x-value already has a tolerance that it must be
// larger than the previous x-value (unless we are at the end of a row)

double slope;                           // (y2 - y1)/(x2 - x1)
double dy,dx;                           // (y2 - y1), (x2 - x1)
double yhi;                             // slope*(xn+1 - xn) + ytol
double ylo;                             // slope*(xn+1 - xn) - ytol
double ytols;                           // ytol scaled by current distance:
    ytol*currd

// Sort through one row at a time
for(int i=0; i<(fsize/rowsize); i++) {

  leftlim = -9999;                      // Reset the searchable area
  for(int j=0; j<rowsize; j++) {        // Sort through a single row

      // it's over 9000!!one1
      double xysum = 9001;                    // current lowest x,y sum
      idx = 9999;                             // resets value, can be used
           to debug
      prevd = 99998;                          // resets previous distance
          to something large for a new row

      // Find the bottom left corner of available points
      // Sum over current vector size

      if(j>1) {               // Using the slope condition for points 3
          and up
              dy = sortyval[j + i*rowsize -1] - sortyval[j + i*rowsize
                  -2];   // y2 - y1
              dx = sortxval[j + i*rowsize -1] - sortxval[j + i*rowsize
                  -2];   // x2 - x1
              slope = dy/dx;
```

```
        cout << "slope for (" << i << "," << j << ") = " << slope <<
             endl;

  // Choose an (x,y) pair from the remaining pairs for examination
  for(unsigned int k=0; k<xvec.size(); k++) {

  currd = sqrt(pow(xvec[k] - sortxval[j+i*rowsize-1],2) + pow(yvec[k
      ] - sortyval[j+i*rowsize-1],2));

  if(currd < prevd) {

    // Check that it's to the right of the previous point
    if(xvec[k] > leftlim) {

        // Check that it passes the slope condition
        ytols = currd*ytol;
        yhi = sortyval[j+i*rowsize -1] + slope*(xvec[k]-sortxval[j+i
            *rowsize -1]) + ytols;
        ylo = sortyval[j+i*rowsize -1] + slope*(xvec[k]-sortxval[j+i
            *rowsize -1]) - ytols;
        if((yvec[k] < yhi) && (yvec[k] > ylo)) {

        prevd = currd;          // Store the lowest distance so far
        idx = k;                // Store the index of the lowest sum
        }       // end if &&
    }           // end if xvec[k] > leftlim
  }             // end if currd < prev
 }              // end for k loop
}               // end for point 2 and up

else {                  // No slope condition for points 1 and 2 in
    row

// if we are on point j=0 or j=1, we can't get a slope yet from two
    previous points
  for(unsigned int k=0; k<xvec.size(); k++) {
        tempsum = 0.7*xvec[k] + yvec[k];// Oct.25/11 - reducing x-
            weight of corner, can remove later
        if((xvec[k] > leftlim) && (tempsum < xysum)) {
                xysum = tempsum;        // Store the lowest sum
                idx = k;                // Store the index of the
                    lowest sum
        }       // end if &&
  }             // end for k loop

} // end for points 0 and 1

// Once the corner point is found, record it and remove it from this
     vector
sortxval[j + i*rowsize] = xvec[idx];            // record
sortyval[j + i*rowsize] = yvec[idx];
xvec.erase(xvec.begin() + idx);                 // remove
yvec.erase(yvec.begin() + idx);

// Shift the searchable area to the right of the last found point
leftlim = sortxval[j + i*rowsize] +exc;

// Make and draw a graph a million goddamn times to find bugs
if(j+i*rowsize == stop) {
```

```
                TGraph *gde = new TGraph(j+i*rowsize+1,sortxval,sortyval);
                ostringstream oss;
                oss << stop;
                string title = "Stop and draw at point " + oss.str();
                cdebug->cd();
                gde->SetTitle(title.data());
                gde->SetMarkerStyle(20);
                gde->SetMarkerColor(4);
                gde->Draw("APL");
                graphwin->Run();
        }       // End if rowsize
    }           // End of row
}               // End of all points

// Modify original file name to get a new file name, to which we will
    write
size_t pos;
pos = fname.find_last_of(".");        // find where the .txt or .csv
    extension is
string writename = fname.insert(pos, "_ORDERED");
cout << "my write-file will be: " << writename << endl;

// Write to a new file
ofstream writefile (writename.data());
writefile.precision(9);
if (writefile.is_open()) {            // and it should be...
      for(Int_t i=0; i<fsize; i++) {
              writefile << sortxval[i] << " " << sortyval[i] << endl;
      }
      writefile.close();
}
else cout << "Unable to open file" << endl;   // which would be
    unfortunate

TCanvas *c3 = new TCanvas("c3", "c3", 100, 10, 1600, 1000);
TCanvas *c2 = new TCanvas("c2", "c2", 200, 10, 1200, 1000);
TCanvas *c2sep = new TCanvas("c2sep", "c2sep", 200, 10, 1200, 700);   //
    for thesis presentation
c2->Divide(1,2);
TGraph *g1 = new TGraph(fsize, xval, yval);
TGraph *g2 = new TGraph(fsize, sortxval, sortyval);
ostringstream os;
os << fsize;
string title = fname + " - " + os.str() + " sorted points";
g1->SetTitle("Unsorted Points");
g2->SetTitle(title.data());
g1->SetMarkerStyle(20);
g1->SetMarkerColor(2);
g2->SetMarkerStyle(20);
g2->SetMarkerColor(2);

// Draw the graphs
c2->cd(1);    g1->Draw("APL");
c2->cd(2);    g2->Draw("APL");
c2sep->cd();  g2->Draw("APL");

// A box to show where the algorithm cannot look for points after the
    first corner
```

```
  TBox *box = new TBox(sortxval[0],sortyval[0],sortxval[0]+exc,sortyval
      [0]-10);
  box->SetFillColor(4);
  box->Draw("same");

  // Import the original picture and overlay the points
  string picname = fname;
  picname.erase(pos);
  picname.append(".png");
  cout << "picname = " << picname << endl;

  TImage *img = TImage::Open(picname.data());
  if (!img) {
        cout << "Could not open the image... bye bye" << endl;
        return -3;
  }

  c3->cd();
  img->Draw();
  for(Int_t i=0; i<fsize; i++) {
        img->PutPixel(sortxval[i],img->GetHeight()-sortyval[i],"#ff00ff");
        img->PutPixel(xval[i],img->GetHeight()-yval[i],"#008080");
  }
  img->Draw();
  graphwin->Run();

return 0;
}
```

B.6 DrawFunc.h

```
// Zachary Petriw
// Dec. 22, 2011

// DrawFunc.h

// This header file contains useful drawing functions for drawing dots,
// lines, grids, vertices and other things in 3D plots. Good for drawing
// the swimming pool when doing fits.

// Originally written in Intrinsics_Solve_decouple.cpp

#include "TVector2.h"
#include "TVector3.h"
#include "TRotation.h"
#include "TMath.h"
#include "TPolyMarker.h"
#include "TMarker.h"
#include "TArrow.h"
#include "TPolyMarker3D.h"
#include "TPolyLine3D.h"
#include "TStyle.h"
#include "TCanvas.h"

// This function will draw the x,y,z axes of each camera so that we can see
// where it is looking. It will make a TVector3 for each axis, and rotate
// each one by phi, theta and psi (Euler angles). It will then draw it on
    our
// canvas of choice.

void draw_vertices(TCanvas* c, double x, double y, double z, double phi = 0,
     double theta = 0, double psi = 0, double mag = 1, int color = 2) {

  c->cd();
  // Unit x,y,z axes in 3-space
  TVector3* vec = new TVector3 [3];
  vec[0].SetXYZ(1,0,0);                 // x-axis
  vec[1].SetXYZ(0,1,0);                 // y-axis
  vec[2].SetXYZ(0,0,1);                 // z-axis

  // Rotate the vector with Euler angles
  TRotation a,b;
  a.SetXEulerAngles(-phi,-theta,-psi);
  b = a.Inverse();              // read about TRotation class to see why
      this is done
  for(int i=0; i<3; i++)
        vec[i].Transform(b);

  TPolyLine3D** ax = new TPolyLine3D* [3];
  for(int i=0; i<3; i++) {
        vec[i].SetMag(mag);
        ax[i] = new TPolyLine3D(2);
        ax[i]->SetPoint(0,x,y,z);                                       //
            start point
        ax[i]->SetPoint(1, x+vec[i].X(), y+vec[i].Y(), z+vec[i].Z());   //
            end point
```

```
        ax[i]->SetLineColor(color);
        ax[i]->SetLineWidth(double(i)*2);
        ax[i]->Draw();
  }
}       // end of draw_vertices

void draw_marker(TCanvas* c, double x, double y, double z, int color = 2) {
  c->cd();
  // Draw a polymarker showing where the object is
  double* obj = new double [3];
  obj[0] = x;
  obj[1] = y;
  obj[2] = z;

  TPolyMarker3D *polyobj = new TPolyMarker3D(1,obj,20);
  polyobj->SetMarkerColor(color);                        // red color by
      default
  polyobj->Draw();
}       // end of draw_marker


void draw_line(TCanvas* c, double x1, double y1, double z1, double x2,
    double y2, double z2, int color = 1, int style = 1) {
  c->cd();
  // Draw a line from x1,y1,z1 to x2,y2,z2
  TPolyLine3D* line = new TPolyLine3D(2);
  line->SetPoint(0,x1,y1,z1);
  line->SetPoint(1,x2,y2,z2);
  line->SetLineColor(color);            // ROOT color
  line->SetLineStyle(style);            // ROOT line style
  line->Draw();
}       // end of draw_line
// A function to draw a grid defined by two opposite corners, and the amount
     of lines to use in x- and y-directions

void draw_grid(TCanvas *c, double x1, double x2, double y1, double y2,
    double z1, double z2, int xlines, int ylines, int color = 38, int style =
     2) {
  double dx = (x2-x1)/xlines;                   // x-increment of grid
  double dy = (y2-y1)/ylines;                   // y-increment of grid
  double dz = (z2-z1)/ylines;                   // z-increment of grid

  for(double i=0; i<xlines+1; i++)
        draw_line(c,x1+i*dx,y1,z1,x1+i*dx,y2,z2,color,style);   // parallel
            to y-axis
  for(double i=0; i<ylines+1; i++)
        draw_line(c,x1,y1+i*dy,z1+i*dz,x2,y1+i*dy,z1+i*dz,color,style);
            // parallel to x-axis
}       // end of draw_grid

// Draws a grid in a perpendicular orientation compared to the first grid
    type
void draw_grid2(TCanvas *c, double x1, double x2, double y1, double y2,
    double z1, double z2, int xlines, int ylines, int color = 38, int style =
     2) {
  double dx = (x2-x1)/xlines;                   // x-increment of grid
  double dy = (y2-y1)/ylines;                   // y-increment of grid
  double dz = (z2-z1)/xlines;                   // z-increment of grid
  for(double i=0; i<xlines+1; i++)
```

```
        draw_line(c,x1+i*dx,y1,z1+i*dz,x1+i*dx,y2,z1+i*dz,color,style);   //
             parallel to y-axis
  for(double i=0; i<ylines+1; i++)
        draw_line(c,x1,y1+i*dy,z1,x2,y1+i*dy,z2,color,style);     //
            parallel to x-axis
}       // end of draw_grid

// Calculates floor height (flht) of the shallow end of the swimming pool at
    a given y-value
double flht(double y) {
  return -11.75 + y*0.02152;
}

// Draw the shallow end of the swimming pool using the above functions
void draw_shallowend (TCanvas *c)
{
  // draw_grid(canvas,x1,x2,y1,y2,z1,z2,n_xlines,n_ylines,color,style)

  // Draw the yellow and green lines for reference
  draw_grid(c,-400,500,427.0,452.4,flht(427.0),flht(452.4),177,5,5,1);
             // long yellow line

  draw_grid(c,0,132.2,213.5,238.9,flht(213.5),flht(238.9),26,5,5,1);
              // middle x yellow line - big y
  draw_grid(c,-203.3,-71.2,213.5,238.9,flht(213.5),flht(238.9),26,5,5,1);
          // neg x yellow line - big y
  draw_grid(c,208.4,340.6,213.5,238.9,flht(213.5),flht(238.9),26,5,5,1);
           // pos x yellow line - big y

  draw_grid(c,0,132.2,0,25.4,flht(213.5),flht(238.9),26,5,5,1);           //
      middle x yellow line - low y
  draw_grid(c,-203.3,-71.2,0,25.4,flht(213.5),flht(238.9),26,5,5,1);      //
      neg x yellow line - low y
  draw_grid(c,208.4,340.6,0,25.4,flht(213.5),flht(238.9),26,5,5,1);       //
      pos x yellow line - low y

  draw_grid(c,122.0,218.6,325.3,350.7,flht(325.3),flht(350.7),19,5,8,1);
           // pos green top line
  draw_grid(c,-84.9,11.7,325.3,350.7,flht(325.3),flht(350.7),19,5,8,1);
            // neg green top line
  draw_grid(c,157.6,183.0,0,325.3,flht(0),flht(325.3),5,64,8,1);
                   // pos green long line
  draw_grid(c,-52.4,-23.9,0,325.3,flht(0),flht(325.3),5,64,8,1);
                   // neg green long line

  // Draw a grids for the shallow end floor and wall
  double yshl = 545.7;
  draw_grid(c,-400,500,0,540,flht(0),flht(540),45,27,17);      // shallow
      end floor (minor)
  draw_grid(c,-400,500,0,500,flht(0),flht(500),9,5);           // shallow
      end floor (major)
  draw_grid(c,-400,500,yshl,yshl,0,200,45,10,17);              // shallow
      end wall (minor)
  draw_grid(c,-400,500,yshl,yshl,0,200,9,2);                   // shallow
      end wall (major)

  // Draw the superior coordinate system axes
//  draw_vertices(c,0,0,0,0,0,0,100/*mag*/,1/*color*/);
//  draw_line(c,0,0,0,0,0,flht(0),1,2);
```

```
  draw_marker(c,0,0,0,1);
}
```

B.7 DrawResiduals.h

```
// Zachary Petriw
// Dec 31, 2011 - last day of the year to write code

// DrawResiduals.h

// This header file will draw residuals plots given arrays
// of x and y residuals.

#include "TMath.h"
#include "TH2.h"
#include "TStyle.h"
#include "TGraph.h"
#include "TMultiGraph.h"
#include "TFile.h"
#include <iostream>
#include <sstream>
#include <string>
#include <iomanip>
#include "gStyles.h"            // my custom paint job for histograms and
    canvases

using namespace TMath;
using namespace std;

// Returns the mainfit multigraph
TMultiGraph* gmain (int fsize, double *xcalc, double *ycalc, double *xpic,
    double *ypic)
{
  TMultiGraph *mg = new TMultiGraph("mg","");
  TGraph *gcalc = new TGraph(fsize,xcalc,ycalc);
  TGraph *gpic = new TGraph(fsize,xpic,ypic);

  gcalc->SetMarkerColor(2);
  gcalc->SetMarkerStyle(20);
  gcalc->SetMarkerSize(0.5);
  gpic->SetMarkerColor(4);
  gpic->SetMarkerStyle(20);
  gpic->SetMarkerSize(0.5);
  mg->Add(gpic);
  mg->Add(gcalc);

  mg->Draw("AP");
  mg->SetTitle("Picture locations (blue) vs. Fitted locations (red)");
  mg->GetXaxis()->SetTitle("x (pixels)");
  mg->GetYaxis()->SetTitle("y (pixels)");
  mg->Draw("AP");

return mg;
}

void draw_residuals (TCanvas *c, int fsize, double *xcalc, double *ycalc,
    double *xres, double *yres, double *xpic, double *ypic)
{
  MyThesisStyle2();                             // load my custom style
  double* xres2 = new double [fsize];           // (xcalc-xpic)^2
```

```
double* yres2 = new double [fsize];           // (ycalc-ypic)^2
double* rres  = new double [fsize];           // (xcalc-xpic)^2 + (ycalc-
    ypic)^2
double totalres = 0;                          // total residual before
    normalization
double yrestot = 0;                           // total y-residual before
double xrestot = 0;                           // total x-residual before

// Populate the residual arrays and calculate total residual values
for(int i=0; i<fsize; i++) {
      xres2[i] = xres[i]*xres[i];
      yres2[i] = yres[i]*yres[i];
      rres[i]  = xres2[i] + yres2[i];
      xrestot += xres[i]*xres[i];
      yrestot += yres[i]*yres[i];
      totalres += xres[i]*xres[i] + yres[i]*yres[i];
}

// Set up the titles of histograms to include the total residuals they
    represent
// 2 digits after decimal
ostringstream osx;    osx << fixed << setprecision(3) << sqrt(xrestot/
    double(fsize));
ostringstream osy;    osy << fixed << setprecision(3) << sqrt(yrestot/
    double(fsize));
ostringstream ost;    ost << fixed << setprecision(3) << sqrt(totalres/
    double(fsize));
string xtit = "X r.m.s. = ";  xtit += osx.str();
string ytit = "Y r.m.s. = ";  ytit += osy.str();
ytit += ", Total r.m.s. = ";  ytit += ost.str();

TH2D** hres = new TH2D* [6];
int bins = 30;

int xmin = LocMin(fsize,xcalc);
int xmax = LocMax(fsize,xcalc);
int ymin = LocMin(fsize,ycalc);
int ymax = LocMax(fsize,ycalc);

double dimB[] = {xcalc[xmin]-100,xcalc[xmax]+100,ycalc[ymin]-100,ycalc[
    ymax]+100};
hres[0] = new TH2D("hx",xtit.data(),bins,dimB[0],dimB[1],bins,dimB[2],dimB
    [3]);
hres[1] = new TH2D("hy",ytit.data(),bins,dimB[0],dimB[1],bins,dimB[2],dimB
    [3]);
hres[2] = new TH2D("hr","Total r (xpic-xcalc)^2 + (ypic-ycalc)^2",bins,
    dimB[0],dimB[1],bins,dimB[2],dimB[3]);
hres[3] = new TH2D("hx2","X residuals^2 ",bins,dimB[0],dimB[1],bins,dimB
    [2],dimB[3]);
hres[4] = new TH2D("hy2","Y residuals^2 ",bins,dimB[0],dimB[1],bins,dimB
    [2],dimB[3]);
hres[5] = new TH2D("hw","Weight of bins",bins,dimB[0],dimB[1],bins,dimB
    [2],dimB[3]);

for(int i=0; i<6; i++) {
      hres[i]->GetXaxis()->SetTitle("x (pixels)");
      hres[i]->GetYaxis()->SetTitle("y (pixels)");
      hres[i]->UseCurrentStyle();
```

```
  }

  hres[0]->FillN(fsize,xcalc,ycalc,xres);              // Fill with x
      residuals
  hres[1]->FillN(fsize,xcalc,ycalc,yres);              // Fill with y
      residuals
  hres[2]->FillN(fsize,xcalc,ycalc,rres);              // Fill with x^2+y^2
       residuals^2
  hres[3]->FillN(fsize,xcalc,ycalc,xres2);             // Fill with x
      residuals^2
  hres[4]->FillN(fsize,xcalc,ycalc,yres2);             // Fill with y
      residuals^2

  // This records bin entries for normalization
  double *ones = new double [fsize];
  for(int i=0; i<fsize; i++) ones[i] = 1.;
        hres[5]->FillN(fsize,xcalc,ycalc,ones);

  for(int i=0; i<5; i++)
        hres[i]->Divide(hres[5]);

  // Note: SURF1 is a nice draw option, but hard to see the whole field of
      view
  TMultiGraph *mg(gmain(fsize,xcalc,ycalc,xpic,ypic));
  c->Clear();
  c->Divide(3,2);
  c->cd(1);     hres[0]->Draw("COLZ");
  c->cd(2);     hres[1]->Draw("COLZ");
  c->cd(3);     mg->Draw("AP");
  cerr << "Graph name is: " << mg->GetName() << endl;
  c->cd(4);     hres[0]->Draw("LEGO2Z 0");
  c->cd(5);     hres[1]->Draw("LEGO2Z 0");
  c->cd(6);     hres[2]->Draw("COLZ");

  // Write (append) the first three plots to a file
  string dir = "~/Desktop/Camera_work/Photos/Swimming_Pool_Test_Sept7_2011/
      Analysis/Thesis_Plots/";
  string fname = dir + "ResHistograms.root";
  TFile *f = new TFile(fname.data(), "UPDATE");
  hres[0]->Write();     cerr << "Wrote xres hist..." << endl;
  hres[1]->Write();     cerr << "Wrote yres hist..." << endl;
  mg->Write();          cerr << "Wrote mg..." << endl << " to " << fname <<
      endl;
  f->Close();
}

TH2D* reshist (int fsize, double *xcalc, double *ycalc, double *res)
{
  int bins = 30;
  int xmin = LocMin(fsize,xcalc);
  int xmax = LocMax(fsize,xcalc);
  int ymin = LocMin(fsize,ycalc);
  int ymax = LocMax(fsize,ycalc);

  TH2D* hresr = new TH2D();
  TH2D* hresw = new TH2D();

  double dim[] = {xcalc[xmin]-100,xcalc[xmax]+100,ycalc[ymin]-100,ycalc[ymax
      ]+100};
```

```
  hresr->SetBins(bins,dim[0],dim[1],bins,dim[2],dim[3]);
  hresw->SetBins(bins,dim[0],dim[1],bins,dim[2],dim[3]);
  hresr->GetXaxis()->SetTitle("x (pixels)");
  hresr->GetYaxis()->SetTitle("y (pixels)");
  hresr->GetYaxis()->SetTitleOffset(1.3);
  hresr->SetStats(0);
  hresr->FillN(fsize,xcalc,ycalc,res); // Fill with x residuals

  // This records bin entries for normalization
  double *ones = new double [fsize];
  for(int i=0; i<fsize; i++) ones[i] = 1.;
  hresw->FillN(fsize,xcalc,ycalc,ones);
  hresr->Divide(hresw);  // Normalize

return hresr;
}

void draw_yres (TCanvas *c, int pad, int fsize, double *xcalc, double *ycalc
    , double *yres)
{
  TH2D** hres = new TH2D* [2];
  int bins = 30;
  int xmin = LocMin(fsize,xcalc);
  int xmax = LocMax(fsize,xcalc);
  int ymin = LocMin(fsize,ycalc);
  int ymax = LocMax(fsize,ycalc);

  double dimB[] = {xcalc[xmin]-100,xcalc[xmax]+100,ycalc[ymin]-100,ycalc[
      ymax]+100};
  hres[0] = new TH2D("hy","Y residuals (ycalc-ypic)",bins,dimB[0],dimB[1],
      bins,dimB[2],dimB[3]);
  hres[1] = new TH2D("hw","Weight of bins",bins,dimB[0],dimB[1],bins,dimB
      [2],dimB[3]);

  for(int i=0; i<2; i++) {
        hres[i]->GetXaxis()->SetTitle("x (pixels)");
        hres[i]->GetYaxis()->SetTitle("y (pixels)");
        hres[i]->GetYaxis()->SetTitleOffset(1.3);
        hres[i]->SetStats(0);
  }

  hres[0]->FillN(fsize,xcalc,ycalc,yres); // Fill with y residuals

  // This records bin entries for normalization
  double *ones = new double [fsize];
  for(int i=0; i<fsize; i++) ones[i] = 1.;
        hres[1]->FillN(fsize,xcalc,ycalc,ones);

  // Normalize
  hres[0]->Divide(hres[1]);

  // Note: SURF1 is a nice draw option, but hard to see the whole field of
      view
  gStyle->SetPalette(1);
  c->cd(pad);     hres[0]->Draw("COLZ");
}
```

B.8 gStyles.h

```
// Zachary Petriw
// Feb. 6, 2011

// gStyles.h

// This header file contains some of the gStyle settings that I often use.
#ifndef _gStyles_H_                     // This prevents the errors from
    including the file more than once
#define _gStyles_H_
#include "TText.h"
#include "TString.h"
#include "TColor.h"
#include "TStyle.h"

void MyStyle() {

  // graph and histogram stuff
  gStyle->SetPalette(1);
  gStyle->SetOptStat(1111);
  gStyle->SetLineWidth(2.);
  gStyle->SetTextSize(1.1);
  gStyle->SetTitleSize(0.07,"");
  gStyle->SetLabelSize(0.05,"xyz");
  gStyle->SetTitleSize(0.06,"xyz");
  gStyle->SetTitleOffset(0.9,"x");
  gStyle->SetTitleOffset(1.2,"y");

  // Pad size and border
  gStyle->SetPadTopMargin(0.11);
  gStyle->SetPadRightMargin(0.07);
  gStyle->SetPadBottomMargin(0.13);
  gStyle->SetPadLeftMargin(0.16);
  gStyle->SetPadBorderMode(0);
  gStyle->SetPadColor(38);

  // the background area of a graph or hist, "inside" the pad region
  gStyle->SetFrameFillColor(424);

  // Canvas color and border
  gStyle->SetCanvasBorderMode(0);
  gStyle->SetCanvasColor(0);

  // Extra axes drawn for graphs
  gStyle->SetPadGridY(1);
  gStyle->SetPadTickY(1);

  // Legend settings
  gStyle->SetLegendBorderSize(3);
}       // end MyStyle()

void MyThesisStyle() {

  // graph and histogram stuff
  gStyle->SetPalette(1);
  gStyle->SetOptStat(1111);
```

```
  gStyle->SetLineWidth(2.);
  gStyle->SetTextSize(1.1);
  gStyle->SetTitleSize(0.07,"");
  gStyle->SetLabelSize(0.05,"xyz");
  gStyle->SetTitleSize(0.06,"xyz");
  gStyle->SetTitleOffset(0.9,"x");
  gStyle->SetTitleOffset(1.2,"y");

  // Pad size and border
  gStyle->SetPadTopMargin(0.11);
  gStyle->SetPadRightMargin(0.07);
  gStyle->SetPadBottomMargin(0.13);
  gStyle->SetPadLeftMargin(0.16);
  gStyle->SetPadBorderMode(0);
  gStyle->SetPadColor(0);

  // the background area of a graph or hist, "inside" the pad region
  gStyle->SetFrameFillColor(0);

  // Canvas color and border
  gStyle->SetCanvasBorderMode(0);
  gStyle->SetCanvasColor(0);

  // Extra axes drawn for graphs
  gStyle->SetPadGridY(1);
  gStyle->SetPadTickY(1);

  // Legend settings
  gStyle->SetLegendBorderSize(3);
}

// Same as before, but meant for DrawResidual for x- and y-residual plots
// - no horizontal guide lines
// - better right side margins
void MyThesisStyle2() {

  // graph and histogram stuff
  gStyle->SetPalette(1);
  gStyle->SetOptStat(0);
  gStyle->SetLineWidth(2.);
  gStyle->SetTextSize(1.1);
  gStyle->SetTitleSize(0.05,"");
  gStyle->SetLabelSize(0.05,"xyz");
  gStyle->SetTitleSize(0.06,"xyz");
  gStyle->SetTitleOffset(0.9,"x");
  gStyle->SetTitleOffset(1.2,"y");

  // Pad size and border
  gStyle->SetPadTopMargin(0.11);
  gStyle->SetPadRightMargin(0.11);
  gStyle->SetPadBottomMargin(0.13);
  gStyle->SetPadLeftMargin(0.16);
  gStyle->SetPadBorderMode(0);
  gStyle->SetPadColor(0);

  // the background area of a graph or hist, "inside" the pad region
  gStyle->SetFrameFillColor(0);

  // Canvas color and border
```

```
  gStyle->SetCanvasBorderMode(0);
  gStyle->SetCanvasColor(0);
}
#endif
```

B.9 Triang.h

```
// Zachary Petriw
// Jan. 31, 2012

// Triang.h

// This is a header file that will work with Triang.cpp to minimize
// the position of an object during a triangulation. It will initialize
// the variables and possibly return them if needed.

#ifndef Triang_H
#define Triang_H
#include <vector>
#include <iostream>
#include "TMath.h"
#include "TVector2.h"
#include "TVector3.h"

using namespace std;

class Triang {

private:

  // inputs to object
  vector<double> Params;                          // ncamera*npar (specific to
       model)
  vector<TVector2> Pixels;                        // ncamera
  vector<double> Coords;                          // (X,Y,Z)

  // from initialization
  double X0,Y0,Z0,phi,theta,psi;
  double R,xh,yh,k1,k2,k3,p1,p2,xr,yr,Rx,Ry;      // original model parameters
  double x0,x1,x2,x3,y0,y1,y2,y3;                 // polynomial parameters

  vector<double> euler_elem;
  vector<TVector3> rotmark;                       // rotated marker position,
      one for each cam. coord. system

  // constructed with ResidualSum()
  vector<double> xcalc;
  vector<double> ycalc;
  vector<double> xres;
  vector<double> yres;
  double totalres;
  int fcalls;                   // number of times a function was called, to
       be inserted somewhere
  int npar;                     // number of parameters used in the model

public:

  // default contructor with all 3 arguments
  Triang(const vector<double>& params, const vector<TVector2>& pixels,
        const vector<double>& coords) :
        Params(params), Pixels(pixels), Coords(coords)
  {
```

```
totalres = 0;                                   // total residual of the fit
fcalls = 0;                                     // incremented when a
    specific function is called
npar = Params.size()/Pixels.size();

double X = Coords[0];         // x-coordinate of target triangulation
    point, sup. coord. system
double Y = Coords[1];         // y-coordinate of target triangulation
    point, sup. coord. system
double Z = Coords[2];         // z-coordinate of target triangulation
    point, sup. coord. system

for(unsigned int i=0; i<Pixels.size(); i++) {
      X0 = Params[i*npar];                      // Camera x position in
          superior coordinate system
      Y0 = Params[i*npar+1];                    // Camera y position in
          superior coordinate system
      Z0 = Params[i*npar+2];                    // Camera z position in
          superior coordinate system
      phi = Params[i*npar+3];                   // Camera orientation -
          first Euler angle
      theta = Params[i*npar+4];                 // Camera orientation -
          second Euler angle
      psi = Params[i*npar+5];                   // Camera orientation -
          third Euler angle

      // constructs Euler matrix elements
      euler_elem.clear();                       // 2-3 days to find this bug
          .
      euler_elem.push_back( cos(psi)*cos(phi) - cos(theta)*sin(phi)*sin(
          psi) );
      euler_elem.push_back( cos(psi)*sin(phi) + cos(theta)*cos(phi)*sin(
          psi) );
      euler_elem.push_back( sin(psi)*sin(theta) );
      euler_elem.push_back( -sin(psi)*cos(phi) - cos(theta)*sin(phi)*cos(
          psi) );
      euler_elem.push_back( -sin(psi)*sin(phi) + cos(theta)*cos(phi)*cos(
          psi) );
      euler_elem.push_back( cos(psi)*sin(theta) );
      euler_elem.push_back( sin(theta)*sin(phi) );
      euler_elem.push_back( -sin(theta)*cos(phi) );
      euler_elem.push_back( cos(theta) );

      // make a vector of the target position, rotated into each camera's
          coordinate system
      double Xc,Yc,Zc;
      Xc =  euler_elem[0]*(X-X0) + euler_elem[1]*(Y-Y0) + euler_elem[2]*(Z
          -Z0);
      Yc =  euler_elem[3]*(X-X0) + euler_elem[4]*(Y-Y0) + euler_elem[5]*(Z
          -Z0);
      Zc =  euler_elem[6]*(X-X0) + euler_elem[7]*(Y-Y0) + euler_elem[8]*(Z
          -Z0);
      TVector3 dum(Xc,Yc,Zc);
      rotmark.push_back(dum);
}     // end of for-loop
}     // *** end of constructor

~Triang() {}
```

```
  double ResidualSum();                                    // residual
      calculations, main function

  vector<double> GetParams() const {return Params;}
  vector<TVector2> GetPixels() const {return Pixels;}
  vector<double> GetCoords() const {return Coords;}
  vector<double> Xcalc() const {return xcalc;}
  vector<double> Ycalc() const {return ycalc;}
  vector<double> Xres() const {return xres;}
  vector<double> Yres() const {return yres;}
  double TotalRes() const {return totalres;}
  int Fcalls() {return fcalls;}

};
#endif // Triang_H
```

B.10 Triang.cpp

```
// Zachary Petriw
// Jan. 31, 2012

// Triang.cpp

// This code contains the calculations called by TriangFCN.cpp and TriangFCN
    .h.
// It is used with the triangulation code, and should allow for multiple
    models
// to be used, just like the Residual/ResidualFCN functions. The model used
// will be chosen based on the size of the Params input/ number of cameras,
// or Params.size()/Pixels.size()

#include "Triang.h"
#include "TF2.h"
#include <iostream>
#include <math.h>

using namespace std;

//_____________________________________________________________________
// This function figures out what model is being used and does the brunt
// of the residual calculation needed for triangulation.
double Triang::ResidualSum()
{
  totalres = 0;

//  cerr << "Params.size()/Pixels.size() = " << Params.size()/Pixels.size()
    << endl;
//  cerr << "npar = " << npar << endl;

  // model with x0,y0,x1,y1,x2,y2,x3,y3,k1,k2,k3,xr,yr
  if(npar==19) {

  // R removed from this part
  // x = Xc, y = Yc, [0] = Zc
  TF2 *fxt = new TF2("xp", "x/TMath::Abs(x)*2/TMath::Pi()*TMath::ATan(TMath
      ::Sqrt(x^2+y^2)/-[0])/TMath::Sqrt((y/x)^2+1)", -50000, 50000, -50000,
      50000);
  TF2 *fyt = new TF2("yp", "y/TMath::Abs(y)*2/TMath::Pi()*TMath::ATan(TMath
      ::Sqrt(x^2+y^2)/-[0])/TMath::Sqrt((x/y)^2+1)", -50000, 50000, -50000,
      50000);

  double xt,yt,xcal,ycal,r2;

  xcalc.clear();        ycalc.clear();
  xres.clear();         yres.clear();
  for(unsigned int i=0; i<Pixels.size(); i++) {

        x0 = Params[i*npar+6];       y0 = Params[i*npar+7];
        x1 = Params[i*npar+8];       y1 = Params[i*npar+9];
        x2 = Params[i*npar+10];      y2 = Params[i*npar+11];
        x3 = Params[i*npar+12];      y3 = Params[i*npar+13];
```

```
        k1 = Params[i*npar+14];      k2 = Params[i*npar+15];        k3 =
            Params[i*npar+16];
        xr = Params[i*npar+17];      yr = Params[i*npar+18];

        // single parameter setting
        fxt->SetParameter(0,rotmark[i].Z());
        fyt->SetParameter(0,rotmark[i].Z());

        xt = fxt->Eval(rotmark[i].X(),rotmark[i].Y());                  //
            xt, or x'
        yt = fyt->Eval(rotmark[i].X(),rotmark[i].Y());                  //
            yt, or y'

        r2 = pow(xt*x1 +x0 -xr,2) + pow(yt*y1 +y0 -yr,2);               //
            Centered on x0,y0

        xcal = x0 + x1*xt + x2*pow(xt,2) + x3*pow(xt,3) + xt*x1*(k1*r2 + k2*
            r2*r2 + k3*pow(r2,3));
        ycal = y0 + y1*yt + y2*pow(yt,2) + y3*pow(yt,3) + yt*y1*(k1*r2 + k2*
            r2*r2 + k3*pow(r2,3));

        xcalc.push_back(xcal);                          // calculated values
             accessible
        ycalc.push_back(ycal);
        xres.push_back(xcal-Pixels[i].X());             // unsquared
            residuals (with directionality)
        yres.push_back(ycal-Pixels[i].Y());

        totalres += pow(Pixels[i].X() - xcal,2) + pow(Pixels[i].Y() - ycal
            ,2);
    }     // end for
    }     // end if Params == 19

    return totalres;

}       // end of Triang::ResidualSum()
```

B.11 TriangFCN.h

```
// Zachary Petriw
// Jan 31, 2012

// TriangFCN.h

// This header file is used in conjuction with TriangFCN.cpp and a main
// program to fit using Minuit2.

#ifndef MN_TriangFCN_H_
#define MN_TriangFCN_H_
#include "Minuit2/FCNBase.h"
#include <vector>
#include "TVector2.h"

namespace ROOT {
namespace Minuit2 {

class TriangFCN : public FCNBase {

public:
TriangFCN(const std::vector<double>& params,
        const std::vector<TVector2>& pixels) :
       Params(params), Pixels(pixels), theErrorDef(1.) {}

~TriangFCN() {}

double Up() const {return theErrorDef;}
double operator()(const std::vector<double>&) const;

std::vector<double> params() const {return Params;}
std::vector<TVector2> pixels() const {return Pixels;}

void setErrorDef(double def) {theErrorDef = def;}

private:

std::vector<double> Params;
std::vector<TVector2> Pixels;
double theErrorDef;

};

} // Minuit2
} // ROOT
#endif //MN_TriangFCN_H_
```

B.12 TriangFCN.cpp

```
// Zachary Petriw
// Jan. 30, 2012

// TriangFCN.cpp

// This program contains the operator () as required by Minuit2.
// It works with Triang.h, Triang.cpp and TriangFCN.h to return
// an (x,y,z) location for a target, the position of which is being
// minimized.

#include "TriangFCN.h"
#include "Triang.h"
#include <iostream>

namespace ROOT {
namespace Minuit2 {

double TriangFCN::operator()(const std::vector<double>& coords) const  {

  // create the function that will do the hard work
  Triang func(Params,Pixels,coords);
  return func.ResidualSum();

} // that's it

} // Minuit2
} // ROOT
```

B.13 Residual.h

```
// Zachary Petriw
// Dec. 20, 2011

// Residual.h

// Accompanies cpp file, Residual.cpp
// This is a header file that will do the brute calculation work for
    minimization.
// It will store all the relevant variables, hopefully, and return them when
     needed.
// It can be called and constructed by the main FCNBase type function that
    is needed
// by MnMigrad.

#ifndef Residual_H
#define Residual_H
#include <vector>
#include <cassert>
#include "TMath.h"
#include "TVector2.h"
#include "TVector3.h"

using namespace std;

class Residual {

private:

  // inputs to object
  vector<TVector3> Markers;                     // npoints*(X,Y,Z)
  vector<TVector2> Pixels;                      // npoints*(x_pic,y_pic)
  vector<double> Params;                        // npar (specific to model)

  // from initialization
  double X0,Y0,Z0,phi,theta,psi;
  double R,xh,yh,k1,k2,k3,p1,p2,xr,yr,Rx,Ry;    // original model parameters
  double x0,x1,x2,x3,y0,y1,y2,y3;               // polynomial parameters
//  double xy1,yacx,ypcx,yasy,ypsy;             // additional sin/cos
    parameters - they kinda suck
  double yk1,yk2,yk3,yxr,yyr;                   // radial correction to y
  double xk1,xk2,xk3,xxr,xyr;                   // radial correction to x
  double sa,sxo,sxf,syo,syf,sdum;               // sine corrections to y (
      camera 2 specific)
  double x1y,x2y,x3y,y1x,y2x,y3x,b1x,b1y;               // more polynomial
      corrections

  vector<double> euler_elem;
  vector<TVector3> rotmark;                     // rotated marker positions

  // constructed with ResidualSum2()
  vector<double> xcalc;
  vector<double> ycalc;
  vector<double> xres;
  vector<double> yres;
  double totalres;
```

```
  int fcalls;                   // number of times a function was called, to
       be inserted somewhere

public:

  // default constructor with all 3 arguments
  Residual(const vector<TVector3>& markers, const vector<TVector2>& pixels,
        const vector<double>& params) :
        Markers(markers), Pixels(pixels), Params(params)
  {
  totalres = 0;                         // total residual of the fit
  fcalls = 0;                           // incremented when the function
      using it is called

  assert(Markers.size() == Pixels.size());

  X0 = Params[0];           // Camera x position in superior coordinate
      system
  Y0 = Params[1];           // Camera y position in superior coordinate
      system
  Z0 = Params[2];           // Camera z position in superior coordinate
      system
  phi = Params[3];          // Camera orientation - first Euler angle
  theta = Params[4];        // Camera orientation - second Euler angle
  psi = Params[5];          // Camera orientation - third Euler angle

  // constructs Euler matrix elements
  euler_elem.push_back( cos(psi)*cos(phi) - cos(theta)*sin(phi)*sin(psi) );
  euler_elem.push_back( cos(psi)*sin(phi) + cos(theta)*cos(phi)*sin(psi) );
  euler_elem.push_back( sin(psi)*sin(theta) );
  euler_elem.push_back( -sin(psi)*cos(phi) - cos(theta)*sin(phi)*cos(psi) );
  euler_elem.push_back( -sin(psi)*sin(phi) + cos(theta)*cos(phi)*cos(psi) );
  euler_elem.push_back( cos(psi)*sin(theta) );
  euler_elem.push_back( sin(theta)*sin(phi) );
  euler_elem.push_back( -sin(theta)*cos(phi) );
  euler_elem.push_back( cos(theta) );

  // make a vector of rotated Marker positions
  for(unsigned int i=0; i<Markers.size(); i++) {
        double Xc,Yc,Zc;
        double X = Markers[i].X();
        double Y = Markers[i].Y();
        double Z = Markers[i].Z();
        Xc =  euler_elem[0]*(X-X0) + euler_elem[1]*(Y-Y0) + euler_elem[2]*(Z
            -Z0);
        Yc =  euler_elem[3]*(X-X0) + euler_elem[4]*(Y-Y0) + euler_elem[5]*(Z
            -Z0);
        Zc =  euler_elem[6]*(X-X0) + euler_elem[7]*(Y-Y0) + euler_elem[8]*(Z
            -Z0);
        TVector3 dum(Xc,Yc,Zc);
        rotmark.push_back(dum);
  }     // end of for-loop

  }     // *** end of constructor


  ~Residual() {}
```

```
                                        // ***** Functions needed for
                                            calculating pixel locations

  vector<double> Rotate(unsigned int n);        // returns vector of Xc,Yc,
      Zc
  double FX(unsigned int n);                    // x'
  double FY(unsigned int n);                    // y'
  double GX(double xt, double yt);              // x_calc
  double GY(double xt, double yt);              // y_calc
  double Res(double p, double c);               // (pic-calc)^2 for a single
       event
  void ResidualSum();                           // residual calculations and
       assignment
  double ResidualSum1();                        // residual calculations
      done inline
  double ResidualSum2();                        // residual calculations
      done inline

                                        // a few simple function
                                            declarations
  vector<TVector3> GetMarkers() const {return Markers;}
  vector<TVector3> GetRotMarkers() const {return rotmark;}
  vector<TVector2> GetPixels() const {return Pixels;}
  vector<double> GetParams() const {return Params;}
  vector<double> EulerMat() const {return euler_elem;}
  vector<double> Xcalc() const {return xcalc;}          // calculated x-
      points
  vector<double> Ycalc() const {return ycalc;}          // calculated y-
      point
  vector<double> Xres() const {return xres;}            // x-residuals,
      unsquared for direction
  vector<double> Yres() const {return yres;}            // y-residuals,
      unsquared for direction
  double TotalRes() const {return totalres;}            // total residual
      after running ResidualSum*()
  int Fcalls() {return fcalls;}                         // insert in a
      function you want to count

};
#endif // Residual_H
```

B.14 Residual.cpp

```
// Zachary Petriw
// Dec. 20, 2011

// Residual.cpp

// Accompanying cpp file to go with the header file, Residual.h
// Here the class functions will be defined. It will calculated the
// residual given a set of 3d point, 2d pixels, and parameters[parsize].

// I hope it works.

#include "Residual.h"
#include "TF2.h"
#include <iostream>
#include <math.h>

using namespace std;

//_____________________________________________________________________________
double Residual::ResidualSum2()
{
  fcalls++;
  totalres = 0;

  // Model with R,xh,yh,k1,k2,k3,p1,p2,xr,yr
  if(Params.size()==16) {

  // x = Xc, y = Yc, [0] = Zc, [1] = R
  TF2 *fxt = new TF2("xp", "x/TMath::Abs(x)*2*[1]/TMath::Pi()*TMath::ATan(
      TMath::Sqrt(x^2+y^2)/-[0])/TMath::Sqrt((y/x)^2+1)", -50000, 50000,
      -50000, 50000);
  TF2 *fyt = new TF2("yp", "y/TMath::Abs(y)*2*[1]/TMath::Pi()*TMath::ATan(
      TMath::Sqrt(x^2+y^2)/-[0])/TMath::Sqrt((x/y)^2+1)", -50000, 50000,
      -50000, 50000);


  // These need to be initialized separately, as they will change
  // from model to model
  R = Params[6];            // Radius of image
  xh = Params[7];           // principal (central) x point
  yh = Params[8];           // principal (central) y point
  k1 = Params[9];           // 1st radial distortion parameter
  k2 = Params[10];          // 2nd radial distortion parameter
  k3 = Params[11];          // 3rd radial distortion parameter
  p1 = Params[12];          // 1st tangental distortion parameter
  p2 = Params[13];          // 2nd tangental distortion parameter
  xr = Params[14];          // x-center of radial/tang distortion
  yr = Params[15];          // y-center of radial/tang distortion


  xcalc.clear();        ycalc.clear();
  xres.clear();         yres.clear();
  for(unsigned int i=0; i<Markers.size(); i++) {
        double xt,yt,r2,xcal,ycal;
```

```
      fxt->SetParameters(rotmark[i].Z(),R);
      fyt->SetParameters(rotmark[i].Z(),R);

      xt = fxt->Eval(rotmark[i].X(),rotmark[i].Y());             // xt,
          or x'
      yt = fyt->Eval(rotmark[i].X(),rotmark[i].Y());             // yt,
          or y'
      r2 = pow(xt-xr,2) + pow(yt-yr,2);       // Note: the spread of xt
          centers on the origin

      xcal = xt + xt*(k1*r2 + k2*pow(r2,2) + k3*pow(r2,3)) + 2*p1*yt + p2
          *(r2+2*pow(xt,2)) + xh;
      ycal = yt + yt*(k1*r2 + k2*pow(r2,2) + k3*pow(r2,3)) + p1*(r2+2*pow(
          yt,2)) + 2*p2*xt + yh;

      xcalc.push_back(xcal);                          // calculated values
           accessible
      ycalc.push_back(ycal);
      xres.push_back(xcal-Pixels[i].X());             // unsquared
          residuals (with directionality)
      yres.push_back(ycal-Pixels[i].Y());

      totalres += pow(Pixels[i].X() - xcal,2) + pow(Pixels[i].Y() - ycal
          ,2);
} // end for

delete fxt;                           // you need to include this or it
    will eat RAM forever
delete fyt;

} // end if Params == 16

// model with R,xh,yh
else if(Params.size()==9) {

R = Params[6];            // Radius of image
xh = Params[7];           // principal (central) x point
yh = Params[8];           // principal (central) y point

// x = Xc, y = Yc, [0] = Zc, [1] = R
TF2 *fxt = new TF2("xp", "x/TMath::Abs(x)*2*[1]/TMath::Pi()*TMath::ATan(
    TMath::Sqrt(x^2+y^2)/-[0])/TMath::Sqrt((y/x)^2+1)", -50000, 50000,
    -50000, 50000);
TF2 *fyt = new TF2("yp", "y/TMath::Abs(y)*2*[1]/TMath::Pi()*TMath::ATan(
    TMath::Sqrt(x^2+y^2)/-[0])/TMath::Sqrt((x/y)^2+1)", -50000, 50000,
    -50000, 50000);

xcalc.clear();        ycalc.clear();
xres.clear();         yres.clear();
for(unsigned int i=0; i<Markers.size(); i++) {
      double xt,yt,xcal,ycal;

      fxt->SetParameters(rotmark[i].Z(),R);
      fyt->SetParameters(rotmark[i].Z(),R);

      xt = fxt->Eval(rotmark[i].X(),rotmark[i].Y());             // xt,
          or x'
      yt = fyt->Eval(rotmark[i].X(),rotmark[i].Y());             // yt,
          or y'
```

```
      xcal = xt + xh;
      ycal = yt + yh;

      xcalc.push_back(xcal);                          // calculated values
           accessible
      ycalc.push_back(ycal);
      xres.push_back(xcal-Pixels[i].X());             // unsquared
          residuals (with directionality)
      yres.push_back(ycal-Pixels[i].Y());

      totalres += pow(Pixels[i].X() - xcal,2) + pow(Pixels[i].Y() - ycal
          ,2);
} // end for

delete fxt;                           // you need to include this or it
    will eat RAM forever
delete fyt;

} // end if Params == 9

// model with Rx,Ry,xh,yh
else if(Params.size()==10) {

Rx = Params[6];       // x radius of image
Ry = Params[7];       // y radius of image
xh = Params[8];       // principal (central) x point
yh = Params[9];       // principal (central) y point

// x = Xc, y = Yc, [0] = Zc, [1] = R
TF2 *fxt = new TF2("xp", "x/TMath::Abs(x)*2*[1]/TMath::Pi()*TMath::ATan(
    TMath::Sqrt(x^2+y^2)/-[0])/TMath::Sqrt((y/x)^2+1)", -50000, 50000,
    -50000, 50000);
TF2 *fyt = new TF2("yp", "y/TMath::Abs(y)*2*[1]/TMath::Pi()*TMath::ATan(
    TMath::Sqrt(x^2+y^2)/-[0])/TMath::Sqrt((x/y)^2+1)", -50000, 50000,
    -50000, 50000);

xcalc.clear();        ycalc.clear();
xres.clear();         yres.clear();
for(unsigned int i=0; i<Markers.size(); i++) {
      double xt,yt,xcal,ycal;

      fxt->SetParameters(rotmark[i].Z(),Rx);
      fyt->SetParameters(rotmark[i].Z(),Ry);

      xt = fxt->Eval(rotmark[i].X(),rotmark[i].Y());              // xt,
          or x'
      yt = fyt->Eval(rotmark[i].X(),rotmark[i].Y());              // yt,
          or y'

      xcal = xt + xh;
      ycal = yt + yh;

      xcalc.push_back(xcal);                          // calculated values
           accessible
      ycalc.push_back(ycal);
      xres.push_back(xcal-Pixels[i].X());             // unsquared
          residuals (with directionality)
      yres.push_back(ycal-Pixels[i].Y());
```

```
      totalres += pow(Pixels[i].X() - xcal,2) + pow(Pixels[i].Y() - ycal
          ,2);
} // end for

delete fxt;                           // you need to include this or it
    will eat RAM forever
delete fyt;

} // end if Params == 10

// model with x0,y0,x1,y1,x2,y2,x3,y3
else if(Params.size()==14) {

x0 = Params[6];
y0 = Params[7];
x1 = Params[8];
y1 = Params[9];
x2 = Params[10];
y2 = Params[11];
x3 = Params[12];
y3 = Params[13];

// R removed from this part
// x = Xc, y = Yc, [0] = Zc
TF2 *fxt = new TF2("xp", "x/TMath::Abs(x)*2/TMath::Pi()*TMath::ATan(TMath
    ::Sqrt(x^2+y^2)/-[0])/TMath::Sqrt((y/x)^2+1)", -50000, 50000, -50000,
    50000);
TF2 *fyt = new TF2("yp", "y/TMath::Abs(y)*2/TMath::Pi()*TMath::ATan(TMath
    ::Sqrt(x^2+y^2)/-[0])/TMath::Sqrt((x/y)^2+1)", -50000, 50000, -50000,
    50000);

xcalc.clear();        ycalc.clear();
xres.clear();         yres.clear();
for(unsigned int i=0; i<Markers.size(); i++) {
      double xt,yt,xcal,ycal;

      // single parameter setting
      fxt->SetParameter(0,rotmark[i].Z());
      fyt->SetParameter(0,rotmark[i].Z());

      xt = fxt->Eval(rotmark[i].X(),rotmark[i].Y());              // xt,
          or x'
      yt = fyt->Eval(rotmark[i].X(),rotmark[i].Y());              // yt,
          or y'

      xcal = x0 + x1*xt + x2*pow(xt,2) + x3*pow(xt,3);
      ycal = y0 + y1*yt + y2*pow(yt,2) + y3*pow(yt,3);

      xcalc.push_back(xcal);                          // calculated values
           accessible
      ycalc.push_back(ycal);
      xres.push_back(xcal-Pixels[i].X());             // unsquared
          residuals (with directionality)
      yres.push_back(ycal-Pixels[i].Y());

      totalres += pow(Pixels[i].X() - xcal,2) + pow(Pixels[i].Y() - ycal
          ,2);
} // end for
```

```
  delete fxt;                           // you need to include this or it
      will eat RAM forever
  delete fyt;

  } // end if Params == 14

  // model with x0,y0,x1,y1,x2,y2,x3,y3,k1,k2,k3,xr,yr
  else if(Params.size()==19) {

  x0 = Params[6];       y0 = Params[7];
  x1 = Params[8];       y1 = Params[9];
  x2 = Params[10];      y2 = Params[11];
  x3 = Params[12];      y3 = Params[13];

  k1 = Params[14];      k2 = Params[15];        k3 = Params[16];
  xr = Params[17];      yr = Params[18];

  // R removed from this part
  // x = Xc, y = Yc, [0] = Zc
  TF2 *fxt = new TF2("xp", "x/TMath::Abs(x)*2/TMath::Pi()*TMath::ATan(TMath
      ::Sqrt(x^2+y^2)/-[0])/TMath::Sqrt((y/x)^2+1)", -50000, 50000, -50000,
      50000);
  TF2 *fyt = new TF2("yp", "y/TMath::Abs(y)*2/TMath::Pi()*TMath::ATan(TMath
      ::Sqrt(x^2+y^2)/-[0])/TMath::Sqrt((x/y)^2+1)", -50000, 50000, -50000,
      50000);

  xcalc.clear();        ycalc.clear();
  xres.clear();         yres.clear();
  for(unsigned int i=0; i<Markers.size(); i++) {
        double xt,yt,xcal,ycal,r2;

        // single parameter setting
        fxt->SetParameter(0,rotmark[i].Z());
        fyt->SetParameter(0,rotmark[i].Z());

        xt = fxt->Eval(rotmark[i].X(),rotmark[i].Y());              // xt,
            or x'
        yt = fyt->Eval(rotmark[i].X(),rotmark[i].Y());              // yt,
            or y'

        r2 = pow(xt*x1 +x0 -xr,2) + pow(yt*y1 +y0 -yr,2);               //
            Centered on x0,y0
//      if(i%100==0) cerr << xt*x0 << " " << yt*y0 << " ";

        xcal = x0 + x1*xt + x2*pow(xt,2) + x3*pow(xt,3) + xt*x1*(k1*r2 + k2*
            r2*r2 + k3*pow(r2,3));
        ycal = y0 + y1*yt + y2*pow(yt,2) + y3*pow(yt,3) + yt*y1*(k1*r2 + k2*
            r2*r2 + k3*pow(r2,3));

        xcalc.push_back(xcal);                          // calculated values
             accessible
        ycalc.push_back(ycal);
        xres.push_back(xcal-Pixels[i].X());             // unsquared
            residuals (with directionality)
        yres.push_back(ycal-Pixels[i].Y());

        totalres += pow(Pixels[i].X() - xcal,2) + pow(Pixels[i].Y() - ycal
            ,2);
```

```
  } // end for

  delete fxt;                           // you need to include this or it
      will eat RAM forever
  delete fyt;

  } // end if Params == 19

  // model with x0,y0,x1,y1,x2,y2,x3,y3,k1,k2,k3,xr,yr, yk1,yk2,yk3,yxr,yyr
  else if(Params.size()==24) {

  x0 = Params[6];       y0 = Params[7];
  x1 = Params[8];       y1 = Params[9];
  x2 = Params[10];      y2 = Params[11];
  x3 = Params[12];      y3 = Params[13];

  xk1 = Params[14];     xk2 = Params[15];       xk3 = Params[16];
  xxr = Params[17];     xyr = Params[18];

  yk1 = Params[19];     yk2 = Params[20];       yk3 = Params[21];
  yxr = Params[22];     yyr = Params[23];

  // R removed from this part
  // x = Xc, y = Yc, [0] = Zc
  TF2 *fxt = new TF2("xp", "x/TMath::Abs(x)*2/TMath::Pi()*TMath::ATan(TMath
      ::Sqrt(x^2+y^2)/-[0])/TMath::Sqrt((y/x)^2+1)", -50000, 50000, -50000,
      50000);
  TF2 *fyt = new TF2("yp", "y/TMath::Abs(y)*2/TMath::Pi()*TMath::ATan(TMath
      ::Sqrt(x^2+y^2)/-[0])/TMath::Sqrt((x/y)^2+1)", -50000, 50000, -50000,
      50000);

  xcalc.clear();        ycalc.clear();
  xres.clear();         yres.clear();
  for(unsigned int i=0; i<Markers.size(); i++) {
        double xt,yt,xcal,ycal,xr2,yr2;

        // single parameter setting
        fxt->SetParameter(0,rotmark[i].Z());
        fyt->SetParameter(0,rotmark[i].Z());

        xt = fxt->Eval(rotmark[i].X(),rotmark[i].Y());              // xt,
            or x'
        yt = fyt->Eval(rotmark[i].X(),rotmark[i].Y());              // yt,
            or y'

        xr2 = pow(xt*x1 +x0 -xxr,2) + pow(yt*y1 +y0 -xyr,2);            //
            Centered on x0,y0
        yr2 = pow(xt*x1 +x0 -yxr,2) + pow(yt*y1 +y0 -yyr,2);            //
            Centered on x0,y0
//      if(i%100==0) cerr << xt*x0 << " " << yt*y0 << " ";

        xcal = x0 + x1*xt + x2*pow(xt,2) + x3*pow(xt,3) + xt*x1*(xk1*xr2 +
            xk2*xr2*xr2 + xk3*pow(xr2,3));
        ycal = y0 + y1*yt + y2*pow(yt,2) + y3*pow(yt,3) + yt*y1*(yk1*yr2 +
            yk2*yr2*yr2 + yk3*pow(yr2,3));
//      ycal += yt*y1*(yk1*yr2 + yk2*yr2*yr2 + yk3*pow(yr2,3));         //
    extra radial correction to y
```

```
        xcalc.push_back(xcal);                          // calculated values
             accessible
        ycalc.push_back(ycal);
        xres.push_back(xcal-Pixels[i].X());             // unsquared
            residuals (with directionality)
        yres.push_back(ycal-Pixels[i].Y());

        totalres += pow(Pixels[i].X() - xcal,2) + pow(Pixels[i].Y() - ycal
            ,2);
  } // end for

  delete fxt;                           // you need to include this or it
      will eat RAM forever
  delete fyt;

  } // end if Params == 24

  // model with x0,y0,x1,y1,x2,y2,x3,y3,k1,k2,k3,xr,yr, sa,sxo,sxf,syo,syf,
      sdum
  else if(Params.size()==25) {

  x0 = Params[6];       y0 = Params[7];
  x1 = Params[8];       y1 = Params[9];
  x2 = Params[10];      y2 = Params[11];
  x3 = Params[12];      y3 = Params[13];

  k1 = Params[14];      k2 = Params[15];        k3 = Params[16];
  xr = Params[17];      yr = Params[18];

  sa = Params[19];      sxo = Params[20];       sxf = Params[21];
  syo = Params[22];     syf = Params[23];       sdum = Params[24];

  // R removed from this part
  // x = Xc, y = Yc, [0] = Zc
  TF2 *fxt = new TF2("xp", "x/TMath::Abs(x)*2/TMath::Pi()*TMath::ATan(TMath
      ::Sqrt(x^2+y^2)/-[0])/TMath::Sqrt((y/x)^2+1)", -50000, 50000, -50000,
      50000);
  TF2 *fyt = new TF2("yp", "y/TMath::Abs(y)*2/TMath::Pi()*TMath::ATan(TMath
      ::Sqrt(x^2+y^2)/-[0])/TMath::Sqrt((x/y)^2+1)", -50000, 50000, -50000,
      50000);

  xcalc.clear();        ycalc.clear();
  xres.clear();         yres.clear();
  for(unsigned int i=0; i<Markers.size(); i++) {
        double xt,yt,xcal,ycal,r2;

        // single parameter setting
        fxt->SetParameter(0,rotmark[i].Z());
        fyt->SetParameter(0,rotmark[i].Z());

        xt = fxt->Eval(rotmark[i].X(),rotmark[i].Y());              // xt,
            or x'
        yt = fyt->Eval(rotmark[i].X(),rotmark[i].Y());              // yt,
            or y'

        r2 = pow(xt*x1 +x0 -xr,2) + pow(yt*y1 +y0 -yr,2);               //
            Centered on x0,y0
//      yr2 = pow(xt*x1 +x0 -xr,2) + pow(yt*y1 +y0 -yr,2);              //
    Centered on x0,y0
```

```
//      if(i%100==0) cerr << xt*x0 << " " << yt*y0 << " ";

        xcal = x0 + x1*xt + x2*pow(xt,2) + x3*pow(xt,3) + xt*x1*(k1*r2 + k2*
            r2*r2 + k3*pow(r2,3));
        ycal = y0 + y1*yt + y2*pow(yt,2) + y3*pow(yt,3) + yt*y1*(k1*r2 + k2*
            r2*r2 + k3*pow(r2,3));
        ycal += sa*sin(pow((xt*x1 - sxo)/sxf,2) + (yt*y1 - syo)/syf);
                    // extra radial correction to y

        xcalc.push_back(xcal);                          // calculated values
             accessible
        ycalc.push_back(ycal);
        xres.push_back(xcal-Pixels[i].X());             // unsquared
            residuals (with directionality)
        yres.push_back(ycal-Pixels[i].Y());

        totalres += pow(Pixels[i].X() - xcal,2) + pow(Pixels[i].Y() - ycal
            ,2);
  } // end for

  delete fxt;                           // you need to include this or it
      will eat RAM forever
  delete fyt;

  } // end if Params == 25

  else if(Params.size()==27) {

  x0 = Params[6];       y0 = Params[7];
  x1 = Params[8];       y1 = Params[9];
  x2 = Params[10];      y2 = Params[11];
  x3 = Params[12];      y3 = Params[13];

  k1 = Params[14];      k2 = Params[15];        k3 = Params[16];
  xr = Params[17];      yr = Params[18];

  x1y = Params[19];     x2y = Params[20];       x3y = Params[21];
  y1x = Params[22];     y2x = Params[23];       y3x = Params[24];
  b1x = Params[25];     b1y = Params[26];

  // R removed from this part
  // x = Xc, y = Yc, [0] = Zc
  TF2 *fxt = new TF2("xp", "x/TMath::Abs(x)*2/TMath::Pi()*TMath::ATan(TMath
      ::Sqrt(x^2+y^2)/-[0])/TMath::Sqrt((y/x)^2+1)", -50000, 50000, -50000,
      50000);
  TF2 *fyt = new TF2("yp", "y/TMath::Abs(y)*2/TMath::Pi()*TMath::ATan(TMath
      ::Sqrt(x^2+y^2)/-[0])/TMath::Sqrt((x/y)^2+1)", -50000, 50000, -50000,
      50000);

  xcalc.clear();        ycalc.clear();
  xres.clear();         yres.clear();
  for(unsigned int i=0; i<Markers.size(); i++) {
        double xt,yt,xcal,ycal,r2;

        // single parameter setting
        fxt->SetParameter(0,rotmark[i].Z());
        fyt->SetParameter(0,rotmark[i].Z());
```

```
        xt = fxt->Eval(rotmark[i].X(),rotmark[i].Y());              // xt,
            or x'
        yt = fyt->Eval(rotmark[i].X(),rotmark[i].Y());              // yt,
            or y'

        r2 = pow(xt*x1 +x0 -xr,2) + pow(yt*y1 +y0 -yr,2);               //
            Centered on x0,y0
//      yr2 = pow(xt*x1 +x0 -xr,2) + pow(yt*y1 +y0 -yr,2);              //
    Centered on x0,y0
//      if(i%100==0) cerr << xt*x0 << " " << yt*y0 << " ";

        xcal = x0 + x1*xt + x2*pow(xt,2) + x3*pow(xt,3) + xt*x1*(k1*r2 + k2*
            r2*r2 + k3*pow(r2,3));
        ycal = y0 + y1*yt + y2*pow(yt,2) + y3*pow(yt,3) + yt*y1*(k1*r2 + k2*
            r2*r2 + k3*pow(r2,3));
        xcal += x1y*yt + x2y*pow(yt,2) + x3y*pow(yt,3) + b1x*xt*yt;
        ycal += y1x*xt + y2x*pow(xt,2) + y3x*pow(xt,3) + b1y*xt*yt;

        xcalc.push_back(xcal);                          // calculated values
             accessible
        ycalc.push_back(ycal);
        xres.push_back(xcal-Pixels[i].X());             // unsquared
            residuals (with directionality)
        yres.push_back(ycal-Pixels[i].Y());

        totalres += pow(Pixels[i].X() - xcal,2) + pow(Pixels[i].Y() - ycal
            ,2);
  } // end for

  delete fxt;                           // you need to include this or it
      will eat RAM forever
  delete fyt;

  } // end if Params == 25

  return totalres;
}
```

B.15 ResidualFCN.h

```
// Zachary Petriw
// Dec. 21, 2011

// ResidualFCN.h

// This header file is used in conjuction with ResidualFCN.cpp and a main
// program to fit using Minuit2.

#ifndef MN_ResidualFCN_H_
#define MN_ResidualFCN_H_
#include "Minuit2/FCNBase.h"
#include <vector>
#include "TVector3.h"
#include "TVector2.h"

namespace ROOT {
namespace Minuit2 {

class ResidualFCN : public FCNBase {

public:
ResidualFCN(const std::vector<TVector3>& marker_pos,
        const std::vector<TVector2>& found_points) :
       Marker_Pos(marker_pos), Points(found_points), theErrorDef(1.) {}

~ResidualFCN() {}

double Up() const {return theErrorDef;}
double operator()(const std::vector<double>&) const;

std::vector<TVector3> marker() const {return Marker_Pos;}
std::vector<TVector2> points() const {return Points;}

void setErrorDef(double def) {theErrorDef = def;}

private:

std::vector<TVector3> Marker_Pos;
std::vector<TVector2> Points;
double theErrorDef;

};

} // Minuit2
} // ROOT
#endif //MN_ResidualFCN_H_
```

B.16 ResidualFCN.cpp

```
// Zachary Petriw
// Dec. 21, 2011

// ResidualFCN.cpp

// This program is structured as required by the Minuit2 minimizer to
// take use the operator() and return a residual. The argument to ()
// is tweaked until the residual is at a minimium.

#include "ResidualFCN.h"
#include "Residual.h"
#include <iostream>

namespace ROOT {
namespace Minuit2 {

double ResidualFCN::operator()(const std::vector<double>& par) const  {

  // create the function that will do the hard work
  Residual func(Marker_Pos,Points,par);
  return func.ResidualSum2();

} // that's it

} // Minuit2
} // ROOT
```