

LASER INTERFEROMETER GRAVITATIONAL WAVE OBSERVATORY
- LIGO -
CALIFORNIA INSTITUTE OF TECHNOLOGY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LIGO-Paper-

2008/09/24

**Development of a Digital Camera
Network and Associated Analytical
Tools for the Caltech 40m LIGO
Prototype**

Eric Mintun

Mentors: Rana Adhikari, Joseph Betzwieser, Alan Weinstein

Draft

California Institute of Technology
LIGO Project, MS 18-34
Pasadena, CA 91125
Phone (626) 395-2129
Fax (626) 304-9834
E-mail: info@ligo.caltech.edu

Massachusetts Institute of Technology
LIGO Project, Room NW22-295
Cambridge, MA 02139
Phone (617) 253-4824
Fax (617) 253-7014
E-mail: info@ligo.mit.edu

LIGO Hanford Observatory
Route 10, Mile Marker 2
Richland, WA 99352
Phone (509) 372-8106
Fax (509) 372-8137
E-mail: info@ligo.caltech.edu

LIGO Livingston Observatory
19100 LIGO Lane
Livingston, LA 70754
Phone (225) 686-3100
Fax (225) 686-7189
E-mail: info@ligo.caltech.edu

Abstract

The Laser Interferometer Gravitational-Wave Detector (LIGO) uses a network of analog cameras to image the laser at various stages in the device. This network is old and lacks functionality and expandability. This paper details the upgrade of this camera network to one that uses digital cameras on a Gigabit-Ethernet network, as well as the development of a number of software tools that aid in archiving and analyzing the images. This software determines the position and width of the beam profile using both a center of mass calculation and a chi-squared minimization fit to the beams theoretical Gaussian shape. It can then pass this information to LIGO's primary control software (EPICS), which can actuate on it in order to realign the beam position. The fitting software was also modified to determine the peak parameters of resonances in the interferometers pre-mode cleaner. This fit would provide a method to determine what percentage of the output beam is comprised of each mode, making it easier remove higher order modes from the laser in later stages of the interferometer, but the fit was not successful due to thermal effects on surfaces of the cavity's mirrors. Possibilities for future work on the GigE camera network is discussed, including other potential analytical tools that can build off this software.

Contents

1	Background and Introduction	3
1.1	The Purpose of the Camera Network	3
1.2	Limitations of the Existing Network	3
1.3	Outline of New Digital Network	4
2	The Camera Network	4
2.1	Hardware	4
2.2	Software	6
3	Beam Profile Fitting Software	7
3.1	Two-Dimensional Reduced-Chi Squared Fit	8
3.2	Center of Mass Calculation	9
3.3	Testing the Fit	9
3.4	Converting to Physical Dimensions	12
4	Feedback Control	17
5	Determining Cavity Power Loss	19
5.1	Cavity Axis Misalignment	19
5.2	Energy Absorption as Heat by the Cavity Mirrors	21
6	Scanning the Pre-mode Cleaner	22
7	Conclusions	24
7.1	Results	24
7.2	Future Work	26
A	Source Code	27
A.1	Fitting Code	27
A.1.1	Matlab Code	27
A.1.2	C Code	34
A.2	Code for Testing the Gaussian Fit on ETMX Images	46
A.3	Rotation Transform Code	50

1 Background and Introduction

1.1 The Purpose of the Camera Network

The Laser Interferometer Gravitational-Wave Observatory (LIGO) is a detector designed to measure gravitational waves produced primarily by neutron stars which inspiral towards each other and merge. Other sources include supernovae, inspiraling black holes, and the big bang. Since a passing gravitational wave will change the length of space in one dimension compared to another very slightly, LIGO is designed to measure very small differential changes in length. In order to do this, LIGO employs an extremely sensitive interferometer. The laser is split and passed into two high fidelity resonance cavities perpendicular to each other. Light leaving the two cavities returns to the beam splitter and interferes either constructively or destructively, depending on the relative lengths of the cavities. By measuring this interference signal, it is possible to record extremely small changes in the relative lengths of the two resonance cavities.[1]

In order to keep track of the beam in each of the optical parts, LIGO deploys a series of closed circuit analog cameras. Each camera either measures the scatter of laser light off one of the mirrors, or a portion of the laser directly after it passes through an important optical component. These measurements provides data on the position of the beam, the mode of the laser beam, and to some extent, the beam's intensity. This information aids in aligning the laser beam on the optics, checking if the resonance cavities are resonating in the proper mode, and debugging any number of other problems associated with the laser's mode and position.

In order to test new hardware and software without disrupting the main detectors, LIGO maintains a much smaller (40m long arms, instead of 4km), prototype detector. Both the main detectors and the 40m detector possess camera networks. While the eventual goal is to update the cameras at the main sites, all work done on the network was done at the 40m detector. All calculations made apply to the 40m detector unless otherwise explicitly stated.

1.2 Limitations of the Existing Network

The existing closed circuit analog camera network suffers from a number of problems. Firstly, while the analog system can save data to a file, the process to do so is very unwieldy. The existing method of digitally saving camera images is very slow and can only take one image from one camera at a time. This significantly limits the ability to run automated analysis of anything involving the camera images. There exists no other method of interfacing a computer with the camera network. Additionally, it is difficult to change settings on the cameras, such as exposure time. Since the laser's intensity can vary over several orders of magnitude, the inflexibility of the exposure time means that the image of the laser is often either saturated or nearly zero.

1.3 Outline of New Digital Network

In order to solve the problems discussed above, a new GigE digital camera network will be designed to replace the existing camera network. This network will be more expandable than the previous system, as well as be easier to interface with LIGO's control computers. This paper will first describe the basic hardware and software implemented to run the new camera network. It will then discuss the a couple of software tools designed to determine the position of the beam from the image, and use this data to automatically realign the beam in the two resonance cavities. These tools are expected to act as the first steps towards integrating the new camera network into LIGO's feedback loops; it is expected that other tools for other purposes will be developed later.

2 The Camera Network

2.1 Hardware

The camera network was initially set-up as two Prosilica GC GigE Digital Cameras (<http://www.prosilica.com>) connected via GigE Cat6 cables to the computer network of the LIGO lab. Each camera is assigned its own Internet Protocol and can be plugged directly into the network. One camera was connected to a lens and positioned to look at stray light scattering off ETMX (End Test Mass X, or the end mirror in the east arm of the interferometer). The other was positioned without a lens at a laser pick-off immediately after the PMC (Pre-Mode Cleaner, a resonant cavity designed to filter out unwanted laser modes). In this case, the laser beam was directly incident on the camera's screen. Eventually, a third camera was added to the network, but not set-up to face anything in particular and was used to test other features of the cameras. The specific camera models are discussed below in the following paragraphs. While these three cameras were set-up for preliminary testing, the eventual goal is to replace every analog camera in LIGO interferometer with digital cameras. This involves placing cameras that measure the reflection and transmission off both arms of the LIGO interferometer, all three mode cleaners (premode cleaner, mode cleaner, and the output mode cleaner), off of the central beam splitter, and from other various pick-off points. This requires at least 12 cameras. Since the system will be designed to be easily expandable, more cameras may be used to measure the beam profile in other locations as well. This network will be setup using three Netgear 24 Port 10/100/1000 Mbps Smart Switch connected by GigE Cat6 cables. There will be one switch down each arm of the interferometer, and one near the central beamsplitter. This will allow every camera to be accessible from a centrally located control computer.

The camera model that is most likely to be used is the GC 750, which is a CMOS (Complementary-symmetry Metal-Oxide Semiconductor) camera. However, it has two sources of imperfections that distort the resulting data. One is dark noise, which is independent of how much light is incident on the camera. The other is that the pixels are not all uniform: some pixels yield different amounts of current per unit of light than other pixels. In order to get the most accurate results, these imperfections need to be measured so that they can be subtracted out of the image. The dark noise was measured by taking pictures with the

lens cap on the camera. The resulting signal was on average 4.1% of the saturation intensity, and was reasonably consistent from pixel to pixel and from measurement to measurement (standard deviation of 0.1%), with the exception of the occasional 'hot' pixel that has a signal which was substantially larger than normal dark noise. Also, the signal of the hot pixels increased with increased exposure time, unlike the normal pixels. These pixels were rare and are not expected to cause significant error.

To measure the second source of error, diffuse laser light was shined over the entire CMOS screen. As a basic test, this was done by diffusing the beam with a piece of paper. 300 images were taken and then averaged, as shown in Figure 1. There is a very clear pattern in the resulting image, which could possibly be due to a component behind the CMOS screen that reflects infrared light. The pattern was only visible in infrared light. This pattern occurs in more than one of the GC 750s, so it is not a defect of a single camera. This measurement will need to be redone with the beam focused evenly over the camera's entire screen in order to get a more accurate image. Since the pattern is consistent from camera to camera and scales with image intensity, it should not be difficult to divide this pattern out of other images.

One other possibility exists for use in the camera network. The GC 650 is a CCD (Charge-Coupled Device) camera that does not have the diffuse light problem described above. It also has a better frame rate (90 fps versus 62 fps) and digitization (12 bits versus 10 bits). However, the pixels 'leak' vertically, creating streaks in the image. The intensity of leaking

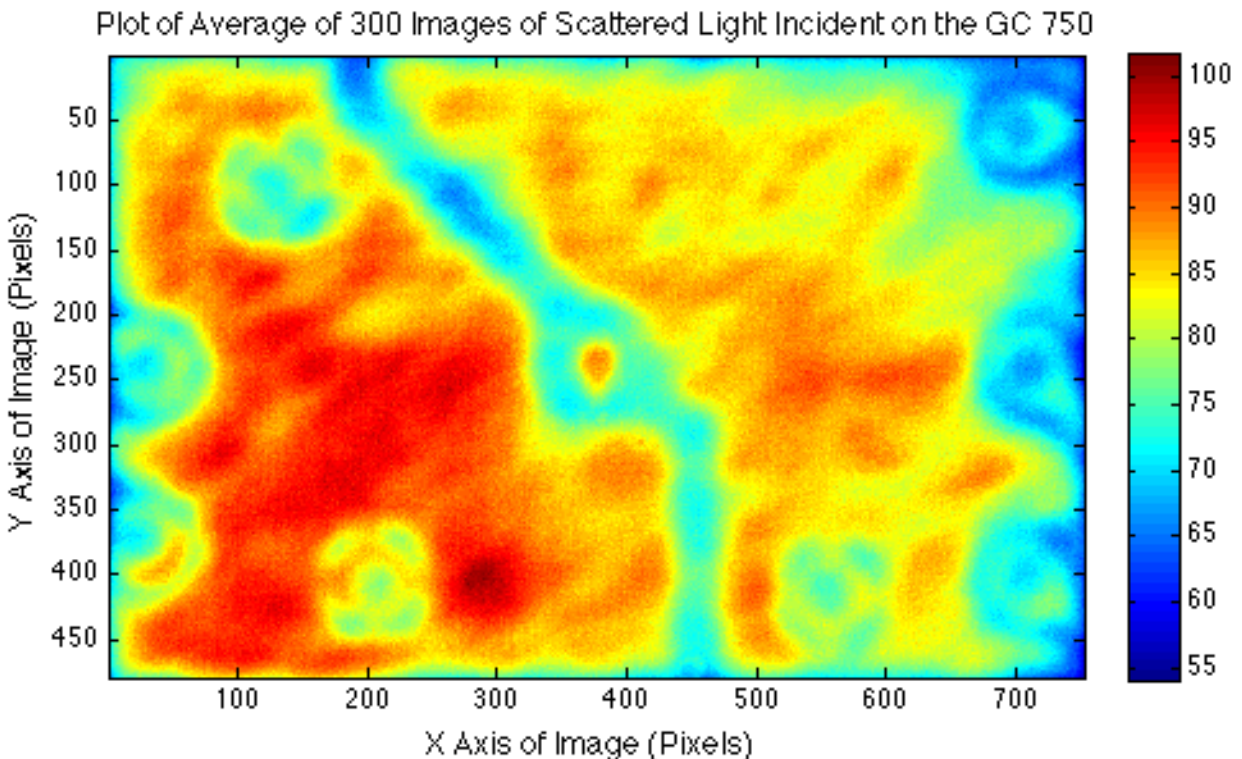


Figure 1: The result of averaging 300 images of diffuse laser light on the CMOS screen.

Table 1: Prosilica GC 750 and GC 650 Specifications[2]

Specification	GC750	GC650
Sensor Type	Micron MT9V022 CMOS	Sony ICX424AL CCD
Sensor Shutter Type	Snapshot (freeze frame)	Progressive Interline
Image Resolution	752 x 480 pixels	659 x 493 pixels
Pixel Size	6.0 μ m x 6.0 μ m	7.4 μ m x 7.4 μ m
Optical Format	1/3 inch	1/3 inch
Lens Mount	CS-Mount	C-mount
Full Resolution Frame Rate	62 fps	90 fps
Power Requirements	2.2W	3.0W
Digitization	10 bits	12 bits
Trigger Latency	43 μ m	2.8 μ m
Trigger Jitter	\pm 0.5 μ m	\pm 0.5 μ m
Tpd	1.3 μ m	1.3 μ m
Operating Temperature	0 to 50 Celsius	0 to 50 Celsius
Operating Humidity	20 to 80% non-condensing	20 to 80% non-condensing
Size	33mm x 46mm x 45mm	33mm x 46mm x 59mm
Weight	85g	99g

does not scale with exposure time, so this problem can be in part solved by using longer exposure times. However, since the leaking effect is very dependent on the pattern of the light incident on the camera, it is thus much harder to divide out. Due to this problem, the GC 750 will most likely be the primary camera used in the network, though the GC 650 may be used for situations where the leaking is not significant (high exposure time images, for instance). For reference, the specifications of the GC 750 and GC 650 are compared in Table 1.

2.2 Software

The primary software used to manage the camera network is a single program based on Prosilica's sample camera control software. It is written in C++ and uses Prosilica's camera API (Application Programming Interface) to communicate with the cameras. It runs in the Linux command line and takes options that set which camera to operate, which settings to use (such as exposure time, bit depth, etc.) and how many images to take (and at what time intervals). The software's standard output for an image is TIFF (Tagged Image File Format) which requires roughly 500kB per image and would likely require too much space once the entire camera network was running continuously. However, Joseph Betzwieser modified the software to give it the capacity to interface with gstreamer (<http://www.gstreamer.net/>), allowing a series of images to be saved in OGM (Ogg Media File). Not only does this allow for the use of codecs that can significantly compress the files, but it makes it easier to navigate sets of images, as they can be easily played-back as a movie. The current implementation uses the Theora codec to encode the data, but gstreamer makes changing codecs fairly straightforward, should a better codec be found. The ability to save individual or sets of TIFF files was retained, since this format is easier to use for data manipulation.

The software also internally converts the raw image data into a GSL matrix (Gnu Scientific Library; specifics can be found at <http://www.gnu.org/software/gsl/>). This GSL matrix, along with associated GSL functions, are used to do any calculations that must be done in real-time. This includes, for example, the laser beam position finding and rotation transform calculations discussed below in Section 3. Any image analysis that does not need to be run in real-time is written in MatLab. These scripts do not interface directly with the camera software, but instead load the TIFF files saved by the camera software. At the moment, these scripts are all run manually; if necessary, they will be automated in the future.

The eventual goal is to develop camera control software that operates as two parts. One would be a server-side program that would run on a single computer and communicate directly with all the cameras in the network. It would constantly run every camera and archive the resulting images as OGM files, incase they needed to be accessed at a later date. The second program would be a client-side program that could run from any of the lab's control computers. If a camera setting needed to be changed, or realtime data needed to be obtained, the program would query the server-side program, which would execute the requested commands. This system is beneficial for a number of reasons. Firstly, it helps control the flow of network traffic, since all of the heavy traffic occurs between the cameras and just one other computer (which would likely have a network card dedicated to the cameras alone). Secondly, it prevents two users from interfering with each other, since each camera is only capable of interacting with one computer at a time. Thirdly, it allows archives to be saved constantly, even if a user is also requesting realtime camera data for a different purpose. At the moment, the code for these programs has not been developed.

3 Beam Profile Fitting Software

The modes of a laser beam profile looked at head-on are given by the Hermite-Gaussian equations

$$\phi = H_m \left(\sqrt{2} \cdot \frac{x}{w} \right) \cdot H_n \left(\sqrt{2} \cdot \frac{y}{w} \right) \cdot e^{-j \cdot \left(P + \frac{k}{2q} \cdot (x^2 + y^2) \right)} \quad (1)$$

where H_m and H_n are Hermite polynomials of order m and n , w is the width (or waist) of the beam, k is the wavenumber of the beam, x and y are the distances from the beam's center perpendicular to the direction of propagation, P represents a complex phase shift, and q (which is also complex) represents a change intensity and phase front curvature for different distances from the center axis. Both P and q are independent of x and y . [6] Since the cameras measure the intensity of the light, the relevant equation is $I = |\phi|^2$ or

$$I = H_m \left(\sqrt{2} \cdot \frac{x}{w} \right)^2 \cdot H_n \left(\sqrt{2} \cdot \frac{y}{w} \right)^2 \cdot e^{2 \cdot \Im \{ P \} + \frac{2k \cdot \Im \{ q \}}{|q|^2} \cdot (x^2 + y^2)} \quad (2)$$

or

$$I = H_m \left(\sqrt{2} \cdot \frac{x}{w} \right)^2 \cdot H_n \left(\sqrt{2} \cdot \frac{y}{w} \right)^2 \cdot A \cdot e^{-\frac{(x^2 + y^2)}{\sigma^2}} \quad (3)$$

with $A = e^{2 \cdot \Im \{ P \}}$ and $\sigma^2 = \frac{|q|^2}{2k \cdot \Im \{ q \}}$. m and n give the mode's number, and the modes are often referred to as TEM_{mn} . The fundamental mode, TEM_{00} , is the mode that the LIGO interferometer operates in. In this case, $H_0 = 1$, and the equation for the intensity of the laser light reduces to a two dimensional gaussian.

The goal of the methods described in this section is to extract information about the beam from the camera images, assuming that TEM₀₀ is the dominant mode. The most important piece of information is the position of the center of the beam, since it will be used to realign the beam. This data must be extracted from the picture both accurately and rapidly (ideally, in less than half a second or so), since it will be passed to a feedback loop used to realign the beam. Other data, such as the width of the beam and some measurement of the conformity of the image to the TEM₀₀ mode, is also useful to extract from the images, but may be done much more slowly.

3.1 Two-Dimensional Reduced-Chi Squared Fit

This approach to extracting information from the images uses nonlinear regression to minimize a χ^2 term. It assumes that the data is of the form

$$I = I_{max} \cdot e^{-\frac{(x-\mu_1)^2}{\sigma_1} - \frac{(y-\mu_2)^2}{\sigma_2}} + I_{offset} \quad (4)$$

and attempts to choose the best possible values for the parameters I_{max} , μ_1 , σ_1 , μ_2 , σ_2 , and I_{offset} , where μ_1 and μ_2 give the position of the beam in the image, σ_1 and σ_2 give the width of the beam, I_{max} gives the maximum intensity, and I_{offset} is a scalar intensity added to the beam. Although the TEM₀₀ mode ideally has perfect radial symmetry, this equation allows for different beam widths in each dimension in case the actual beam is not perfectly symmetric. The benefit of using a reduced chi-squared fit is that it provides a method for obtaining position, maximum intensity, beam width, and the noise floor all at once. It also provides a method of measuring how ideal the beam profile is: the larger the value for χ^2 is, the more the image differs from the expected two dimensional gaussian. This method can potentially provide a way of estimating the precision of various calculated parameters, though the code for this has not yet be implemented. The downside to this approach is that it is slow, usually requiring at least five seconds to complete (the exact time depends on the size of the image and the ideality of the image). Thus, this approach will not be used in the feedback loop, but only as a method for extracting data for processes that do not need to operate quickly.

To implement the fit, C code was written that employs the Levenberg-Marquardt algorithm for nonlinear chi-squared minimization as described in *Numerical Recipes in C*. [7] This method is an iterative process that, at each step, attempts to calculate a change to the parameters that produces the largest decrease in χ^2 . It requires initial guesses at the parameters in order to begin iterating. These estimations are made in the following way: μ_1 and μ_2 are guessed using a center of mass calculation on the intensities, I_{max} is taken as the intensity at $x = \mu_1$ and $y = \mu_2$, I_{offset} is taken as the minimum intensity, and σ_1 and σ_2 are calculated as the distance along each axis to the point with the intensity $I_{max} \cdot e^{-1}$. Since a chi-squared minimization requires an estimation of error on each data point, multiple images are taken and averaged together to obtain an estimated standard deviation for each pixel. The iteration process continues until χ^2 is only improving by a negligible amount (less than 0.01, for this software). This method is capable of 'getting caught' in local minimums while missing the global minimum, though if the initial assumption is a good assumption (TEM₀₀

dominates the profile), the initial estimates almost always good enough that this does not happen. The source code for this fitting software is shown in Appendix A.1.

An example of the results of the fitting code is shown in Figure 2. The first graph shows an average of 100 images taken of the scattered laser light off the end mirror of the east facing arm of the Caltech 40m LIGO interferometer (labelled ETMX). Figure 3 shows one such image. The image is mostly obscured by spikes of intensity that occur due to small imperfections in the mirror that scatter more light than the rest of the mirror. The second image shows the gaussian generated by the parameters of the fit. The fit generally succeeds in ignoring the intensity spikes, though it likely distorts the calculated center by some amount. A potential use of this fit is to use the residuals of the fit to construct a map of the imperfections on the mirror. The degree of error introduced by these spikes is estimated in Section 3.3. This map could then be used to either find smoothest portion of the mirror or to cancel out the intensity spikes when data is taken later for other purposes.

3.2 Center of Mass Calculation

Since the chi-squared minimization is very slow, another method was developed that attempts to extract only the data regarding the position of the center of the beam much more quickly. C code was written to 'project' the data onto the x-axis and y-axis by summing along the rows and columns. Data near the noise floor of the gaussian (below 10% of the maximum intensity in the image) is not included in the summation in order to prevent long tails on one side of the gaussian from skewing the calculation). Any point above a certain threshold is truncated to the threshold in order to prevent large, systematic errors (such as those seen in the figure above) from skewing the calculation. The center is then calculated by weighting the positions by the summed intensities, and then dividing by the total sum of the intensities. The source code is in Appendix A.1. This approach is easily fast enough, running almost instantaneously. The resulting values are less accurate than those obtained from the fit, but are still accurate enough to run a simple feedback loop, as discussed in Section: 4.

3.3 Testing the Fit

In order to test if these methods provide reasonable results despite the noise in the end mirror images, a simple model was developed to try to simulate the noise from the mirror. An image full of fake defects was generated, where each defect was modeled with the equation

$$I = I_{max} \frac{\sigma_0}{\sqrt{\frac{(x-x_0)^2}{\sigma_x} + \frac{(y-y_0)^2}{\sigma_y} + \sigma_0}} \quad (5)$$

where I_{max} , σ_x , σ_y , and σ_0 are variable parameters that control the shape of the defect and x_0 and y_0 differ from defect to defect to change the position. The equation was chosen because it led to defects that looked fairly similar to the existing defects. The defects were generated with positions (x_0 and y_0) distributed uniformly over a 225 by 275 pixel image. The intensity of each defect was distributed randomly between 5000 and 35000, σ_x and σ_y

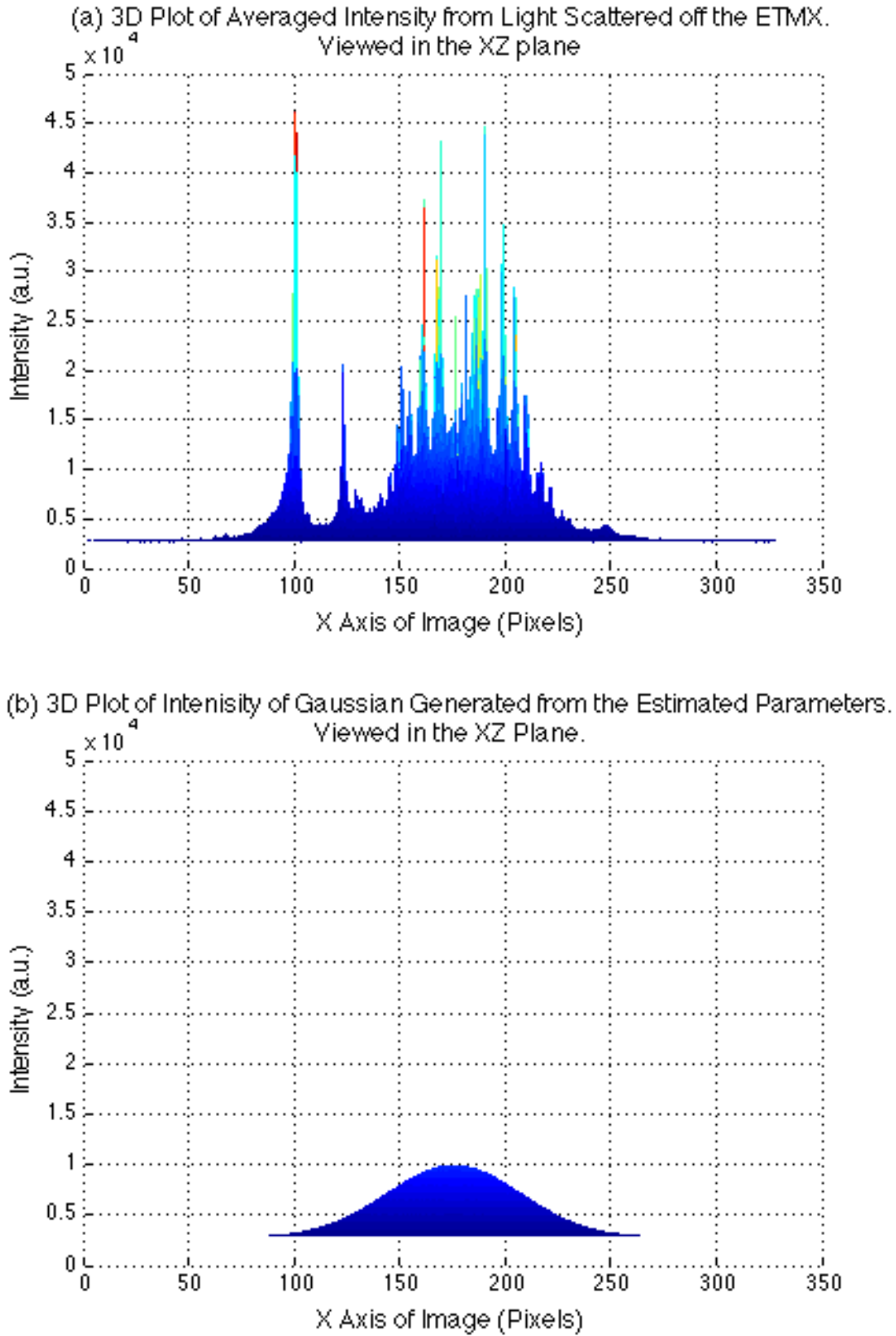


Figure 2: (a) shows a picture of the light scattered off the ETMX, while (b) shows the gaussian generated by the 6 parameters determined by the fit.

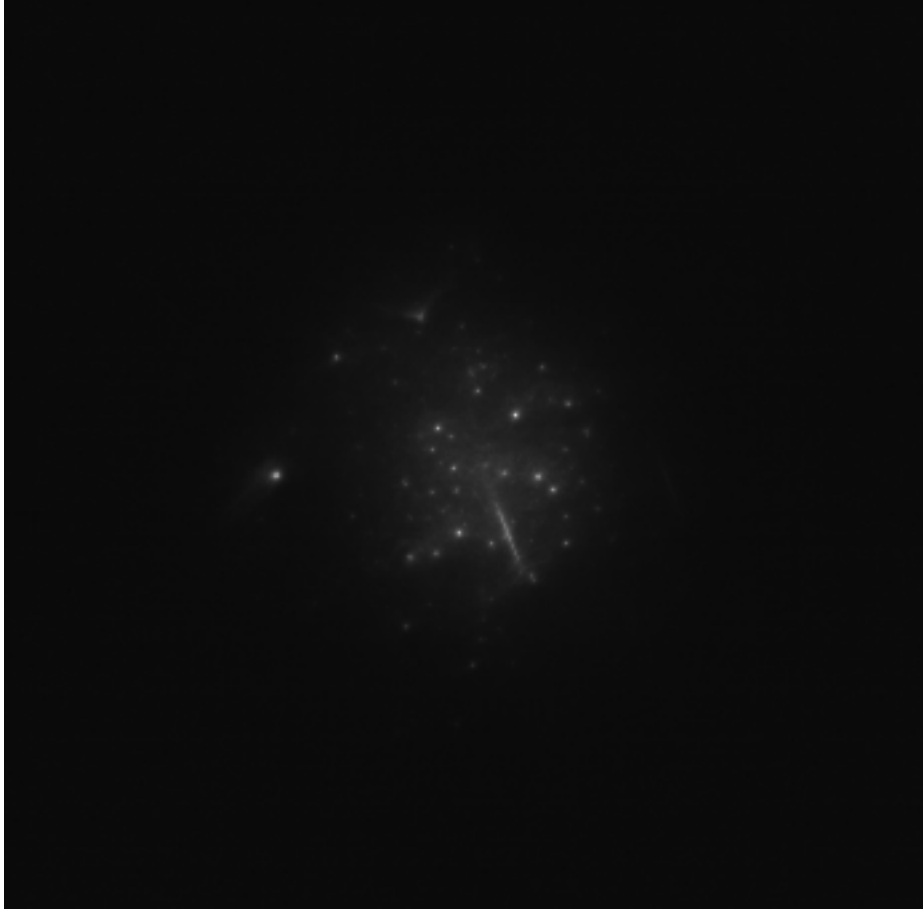


Figure 3: An image of the scattered light from the laser beam off the Caltech 40m ETMX mirror. The bright spots are small defects in the mirror which scatter a large amount of light. The exposure time was 10ms.

were distributed normally around 1 with a standard deviation of 0.2, and σ_0 was fixed at 0.35. These parameters, while somewhat arbitrary, were chosen to make the fake defects look similar to the actual defects. These fake defects were all added to each other (each individual one was cropped to a $12\sigma_x$ by $12\sigma_y$ rectangle, to reduce computation time). The result was cropped at 50000 to prevent accidental layering of points on top of each other from resulting in very high intensities. A two-dimensional gaussian was then generated with random parameters designed to be roughly similar to the gaussian profile in the real images. Each point of this gaussian included noise with a standard deviation of $\frac{128}{\sqrt{12}}$, which is the error associated with the real image’s digitization noise. This gaussian is then normalized, and the intensity of fake defects is then scaled by the normalized gaussian and added to the original gaussian, since the power scattered by the defect is dependent on the power incident on the defect. This was done for ten different gaussians (all with the same parameters), which were then averaged. The MatLab code used to create and test the model in this manner is shown in Appendix A.2. A comparison between these 10 averaged simulated gaussians and 10 averaged real beam profiles (from the 40m ETMX) is shown in Figure 4. The simulated data is not completely accurate: points near (but not directly on top of) a simulated defect are made slightly too intense. Additionally, this model cannot handle larger gashes such as the one seen in Figure 4a, or that the beam profile is not perfectly gaussian. Nevertheless, it should be reasonable for estimating the basic effect of the defects on the fit.

In order to test the fit with this model, it was repeated 20 times for different gaussian parameters and defects (each generated randomly using the same method described above). The gaussian parameters were all distributed uniformly, with $I_{max} \in [8000, 11000]$, $I_{offset} \in [1000, 4000]$, $x_0 \in [125, 175]$, $y_0 \in [100, 150]$, and $\sigma_x, \sigma_y \in [35, 45]$. The two-dimensional gaussian fit and the center-of-mass calculation (with a threshold of 13000) were run on the result each time, and the error between the actual gaussian parameter and the estimated gaussian parameter was recorded and then averaged across all 20 runs. The results are shown in Table 2. These values suggest that the fit is generally reliable for estimating the beam’s center and waist size and can be trusted within a pixel of the location it provides. The defects ruin the estimation of intensity, however, which is too high by 20% to 25% on average. Additionally, for defect ridden images, the center of mass calculation does a significantly better job of estimating the center than the fit does. In situations with defects such as these, fitting should only be used when the beam waist size needs to be calculated, or when error values on beam position need to be known.

3.4 Converting to Physical Dimensions

Both methods of determining the position provide the result in pixels, which are not a very useful unit. In order to convert from pixels to distances, the angle of the camera with respect to the surface that it is imaging must be taken into account. This can be done most accurately using perspective projection. This method requires six parameters (three for the camera’s position and three for its angle) and is unnecessarily messy. Two assumptions can be made that simplify the transformation. One is that the camera is pointed at the center of the optic. The other is that the camera is far enough away that the differences in position along the camera’s axis of different components of the image are negligible. These

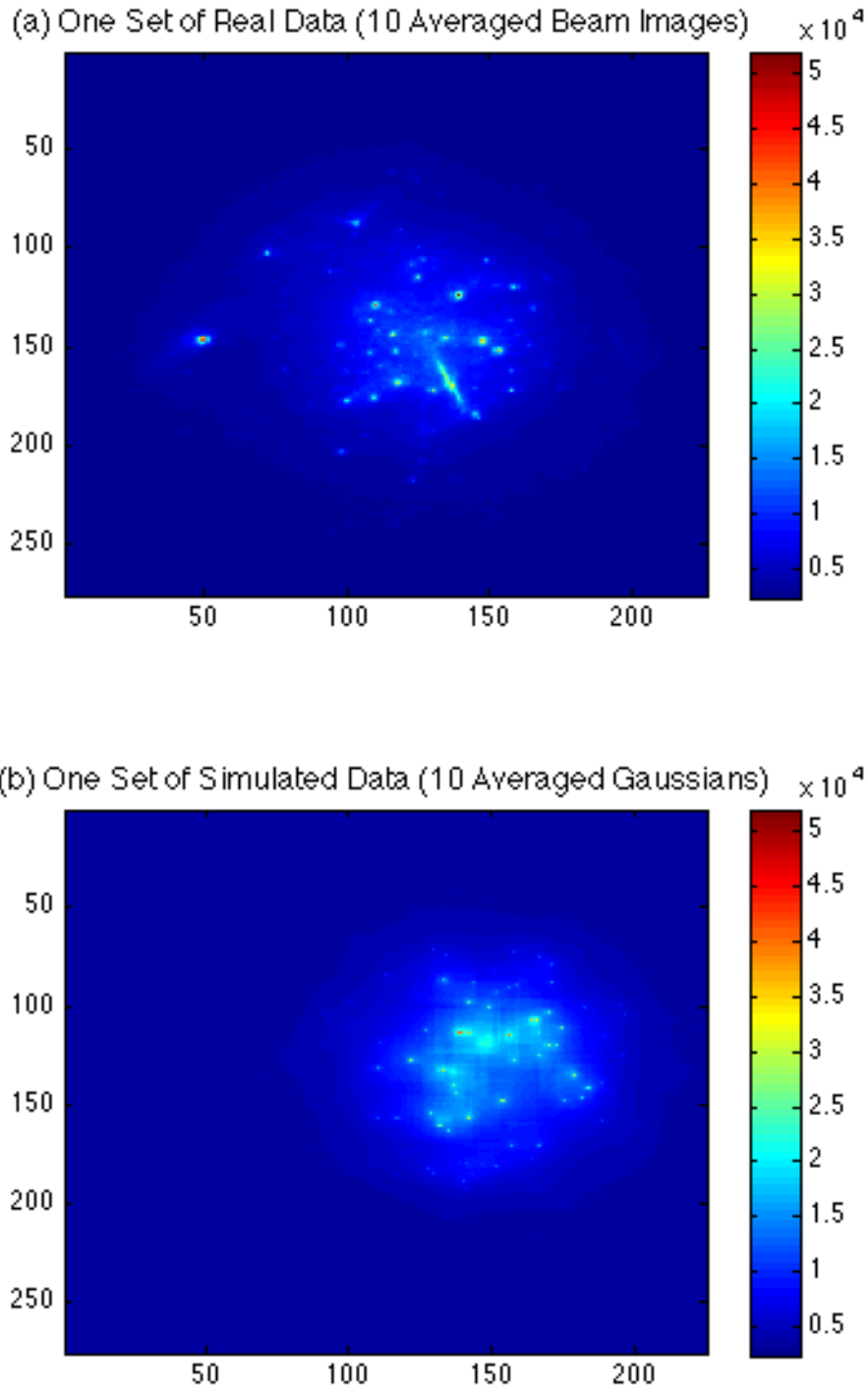


Figure 4: A comparison between the simulated gaussian beam with error and the real gaussian beam with error. Both images are an average of 10 data sets. (a) shows an actual image of the beam, while (b) shows the simulated image.

Table 2: Average Error of Fit from Model Parameters

Method	Parameter	Average Error
Fit	I_{max}	2767.4
	I_{offset}	12.7705
	x_0	0.9401
	y_0	0.9997
	σ_x	1.3406
	σ_y	1.3059
Center of Mass	x_0	0.0087
	y_0	0.0286

assumptions reduce the system to only four parameters, three of which are the angle of the camera, and one of which is a scaling factor related to the distance between the camera and the optic.

Since the system now depends only on angle, the transformation can be described a set of Euler angles. The Euler transformation is given by

$$\mathbf{R} = \begin{bmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \beta & -\sin \beta \\ 0 & \sin \beta & \cos \beta \end{bmatrix} \cdot \begin{bmatrix} \cos \gamma & -\sin \gamma & 0 \\ \sin \gamma & \cos \gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (6)$$

where α is a rotation around the z-axis, β is a rotation around the new x-axis (after the first rotation), and γ is a rotation around the new z-axis (after the first two rotations). By assuming that the camera is attached to some point on the z-axis, an undistorted image (where the axis that the camera lies on is orthogonal to the optic's plane) has the optic in the xy-plane. To distort this image so that it looks like the actual image, the axes must be transformed by three angles that align the z-axis with the actual camera's position. This is done using the transformation

$$\begin{bmatrix} x & y & 0 \end{bmatrix} \cdot \mathbf{R} = c \cdot \begin{bmatrix} x' & y' & z' \end{bmatrix} \quad (7)$$

where x and y give the actual position of the light on the optic, x' , y' , and z' give the position in the image, and c is a scaling factor that converts pixels to distances based on the distance between the optic and the camera. In this equation, \mathbf{R} and c are both known from the camera's position, and x' and y' are known from the image taken. This leaves three equations and three unknowns (x , y , and z'), which can be solved for.

There are two ways of obtaining the values for α , β , γ , and c . If the position of the camera is known, they can be calculated from this value. Say the camera is at (x_c, y_c, z_c) . To convert this to Euler angles, the xy-plane must first be rotated until the camera's new x-position is zero. This gives $\alpha = \arctan \frac{x_c}{y_c}$. It then must be rotated so that the camera lies on the new z-axis, giving $\beta = \arctan \frac{y'_c}{z'_c}$, where z'_c and y'_c are the new z and y positions (after rotation by α). Since this rotation occurred around the z-axis, $z'_c = z_c$, and $y'_c = y_c \cdot \cos \alpha = \sqrt{x^2 + y^2}$.

Then, γ is given by the camera's rotation around the axis orthogonal to its screen, which can simply be measured. The scaling factor, c , depends on internal parameters of the camera and its lens. Since the assumption has been made that the distance from the camera to the optic is much larger than distances on the optic, c is taken as the same for every point on the optic, regardless of z' . Thus, $c = \frac{d}{a \cdot e_z} \cdot \ell$, where d is the distance between the focal point of the camera lens and the optic, e_z is the distance between the focal point of the lens and the camera's screen, a is the zoom ratio of the lens, and ℓ is the length of one pixel on the camera's screen. For the GC 750, $\ell = 6 \frac{\mu\text{m}}{\text{pixel}}$, and for the lens currently in use $e_z = 24$ mm and $a = 1.8$.

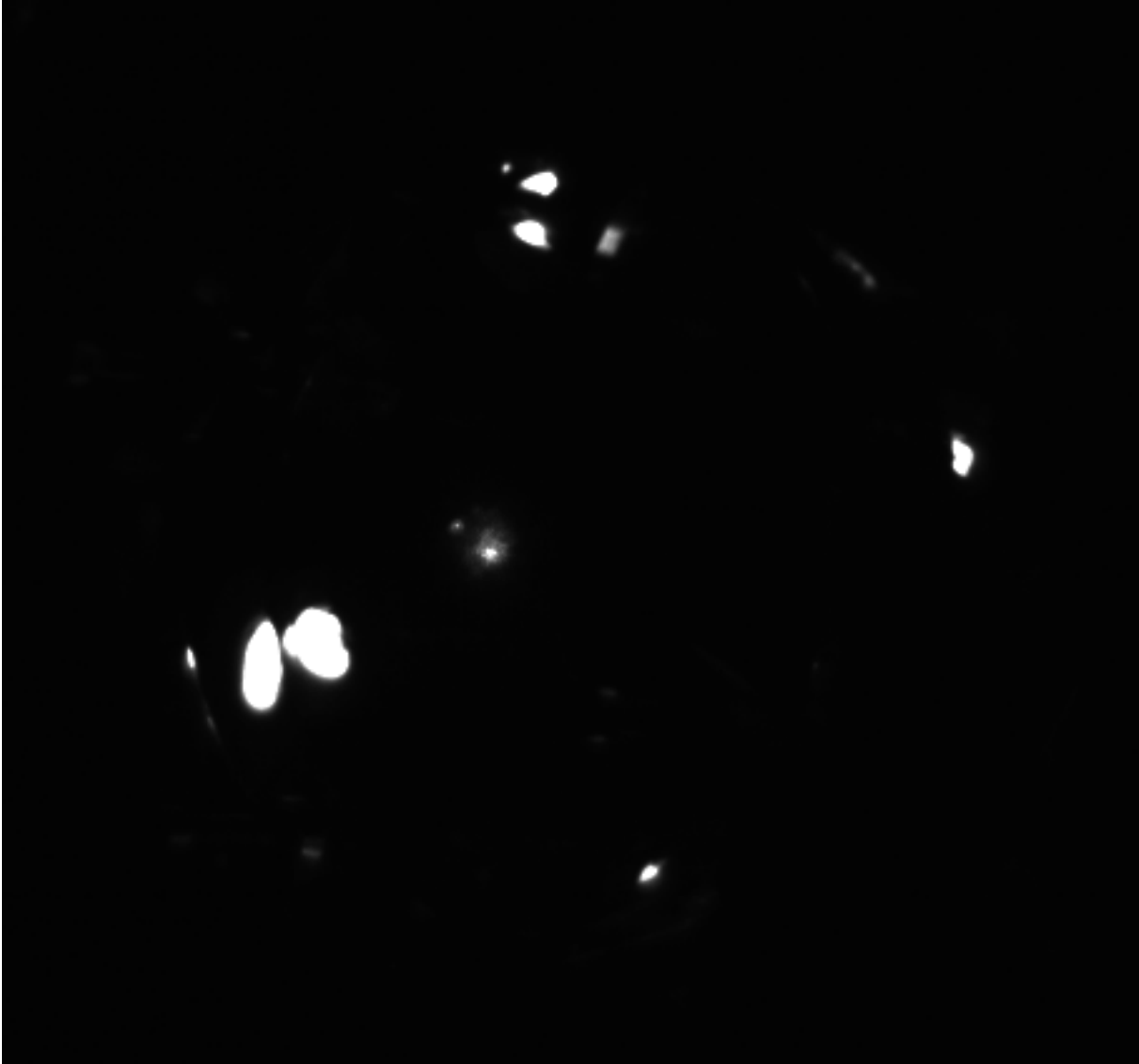


Figure 5: A picture of the OSEMs on the ETMX end mirror. The four bright areas are light from the OSEMs, while the dimmer spot in the middle is the scatter from the laser beam.

Since there is a reasonable chance that some cameras will be frequently moved around and adjusted, and the vacuum chamber around the optics makes it difficult to measure distances,

the values used in the above paragraph may not be known. Instead, they can be calculated from the apparent positions of known light sources. In this case, x , y , x' , and y' are all known, and α , β , γ , and c must be solved for. Since z' is still unknown, one point will yield three equations for five unknowns, while two points will yield six equations for six unknowns. This should be solvable, though there may be ambiguities that arise because sine and cosine are not one-to-one. These can be solvable through logical argument or the inclusion of additional points. The optics in the arms of LIGO have four optical shadow sensor and magnetic actuators (OSEMs) attached to their backs that produce light and are easily visible in the image of the laser beam (an example is shown in Figure 5). These will be used as light sources with known positions. For this particular mirror, the OSEMs are located at $(45.2548, 45.2548, 0)$, $(45.2548, -45.2548, 0)$, $(-45.2548, 45.2548, 0)$, and $(-45.2548, -45.2548, 0)$, where $(0, 0, 0)$ is the middle of the back of the optic (all distances in millimeters). Since the light reflects off the front of the optic and not the back, a correction of -53mm needs to be included due to the thickness of the optic. This correction can be included by calculating the camera's position in Cartesian coordinates by reversing the method described above, then subtracting 53mm and re-calculating the Euler angles. Once α , β , γ , and c are found, the normal transformation can be applied to the beam's apparent position to obtain its actual position. The MatLab code used to calculate these parameters and apply the transformation is shown in Appendix A.3.

An image was taken of the ETMX OSEMs using a 16mm focal point, 1.4 zoom lens. The positions of the OSEMs in this image were taken as $(523, 201)$, $(381, 118)$, $(275, 285)$, and $(412, 376)$. The result of applying this transformation to the image of the OSEMs is shown in Figure 6. The resulting four parameters were $\alpha = 1.5525\text{rad}$, $\beta = -0.6441\text{rad}$, $\gamma = -0.9904\text{rad}$, and $c = 0.4351 \frac{\text{mm}}{\text{pixel}}$. The transformation worked fairly well; from the figure, it is clear that the image has been rotated to be upright and scaled to have the correct lengths. To determine the accuracy of the transformation more precisely, the results of applying the transform to the four OSEMs in the image was compared to their actual values in real space. This comparison is shown in Table 3. (These numbers differ slightly from the ones in Figure 6. The image in the figure is after the correction for the thickness of the mirror, while the numbers in the table are before the correction. The correction causes the imaged points to not agree with their actual locations, but at that stage, it is no longer important since the relevant plane is the front of the optic, not the back.) The positions of the transformed points, though all within 1.5mm of their actual values, are not quite square. Instead, they form a slight parallelogram. This suggests that the rotation in the z dimension was incorrect. This could occur either because β was misestimated (the plane was not rotated the correct amount) or because α was misestimated (the axis around which the rotation by β occurred was not angled correctly). In either case, the error is likely due to misestimation of the OSEMs location on the original image. Since the OSEMs saturate the image and create reflections on the edges of mirror, it is difficult to determine their exact position extremely accurately by eye. This is one aspect of the calculation that could likely be improved upon.

Once the parameters of the rotation have been calculated, they can be loaded into the camera software so that when it calculates the position of the beam, it automatically applies

Table 3: Results of OSEMs Image Rotation

Image Position	Transformed Image Position	Actual Position
(523, 201)	(46.2993, 46.2577)	(45.2548, 45.2548)
(381, 118)	(-44.1926, 44.1529)	(-45.2548, 45.2548)
(275, 285)	(-44.2088, -44.1691)	(-45.2548, -45.2548)
(412, 376)	(46.3185, -46.2769)	(45.2548, -45.2548)

the transformation to convert that value into meaningful units. Since beam's position are in terms of something independent on the particular set-up of the camera, it allows the camera software to communicate beam position data to other software and devices. Potentially, this transformation could be applied to the beam waist size and even the whole beam profile, although this has not yet been implemented.

4 Feedback Control

Once the center of the beam has been calculated, this information needs to be standardized so that it can be accessed by other scripts in the LIGO control structure. To this end, the data is written to an EPICS (Experimental Physics and Industrial Control System) channel using the EZCA (Easy Channel Access) API (Application Programming Interface). Each dimension requires one channel, so there are two channels per camera. The image-capturing code averages every 10 images to prevent extremely brief fluctuations or glitches from affecting the data and updates the channels with the calculated position. Since the camera is capable of taking images very rapidly, this allows other scripts almost real-time access to the position of the beam.

Realignment of the beam is one potential use of the beam position data. In order to move the position of the beam on ETMX, the angle of the mirror on the input end of the arm (Initial Test Mass X, or ITMX) must be changed. This is done by changing the values in two EPICs channels associated with the pitch and yaw of ITMX. To automate this process, these EPICs channels and the channels containing the beam position data were passed to two servo scripts, one for each dimension. The servo script simply modifies the angle channel of the ITMX by some gain factor times the output of the beam position, then repeats the process once the beam position has updated. To run the servo on the ITMX, a modified version of the servo script had to be developed that included a minimum, a maximum, and a slew rate limiter in order to protect the mirror. Since the mirror's position must be very precisely controlled, and its suspensions are fairly delicate and can be damaged by large or rapid movements. Initially, the servo was tested in just the x-dimension (corresponding to the yaw control for the ITMX). The imaging software was set manually to crop the image so that the OSEMs were not included and would not interfere with the center of mass calculation. The rotation transformation values as calculated in Section 3.4 for the ETMX were used to correct for the angle of the camera. When the servo script was run, it successfully changed the x-position of the beam to the new, desired location. However, the servo also changed the y-position of the beam by about half as much. This would suggest that, somewhere in the

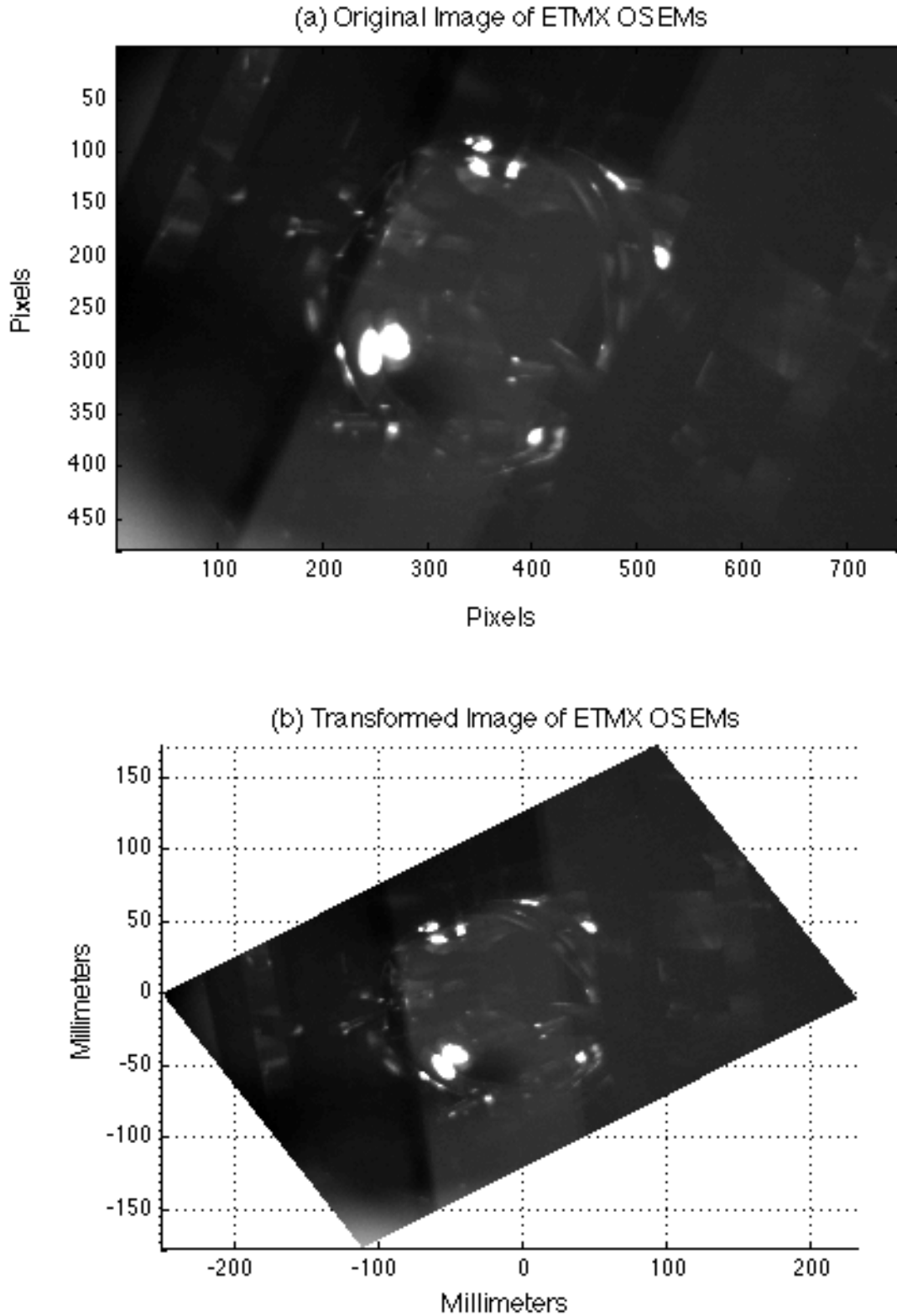


Figure 6: A comparison between the original image of the ETMX OSEMs and the rotated, transformed one. (a) shows the original image, and is in units of pixels measured from the top left of the image, while (b) shows the rotated image, which is in units of millimeters measured from the center of the optic. The units are only valid on the face of the optic.

feedback loop, the two dimensions are being coupled together. This coupling is problematic because when both the x-dimension and y-dimension servos are running at the same time, it is possible that the change induced by one on the other could lead to oscillations around the desired point or other unusual behavior. This error could potentially come from the angle of the camera with respect to the plane of the optic, but the transformation discussed in Section 3.4 should have rotated the image to decouple the two dimensions. Another possibility is that a separate servo that controls the mirrors via optical levers off their backs was resisting the change in position and has coupled x and y dimensions. Thus, when it sensed the change in the x direction, its attempt to correct it by changed both the x and y directions. The servo needs to be run again with this servo disabled. A third possibility is that the controls of the ITMX are coupled in pitch and yaw, so that modifying the yaw was also moving the beam's position in the y dimension. In any of these cases, however, the problem can be corrected by inserting a rotation matrix between the channel output and the servo that would decouple the two dimensions. This approach has not yet been tested, however.

5 Determining Cavity Power Loss

One of the potential uses of the new digital cameras is to indirectly measure power losses due to misalignment of beam cavities and slight changes to the radius of curvature of the beam cavity mirrors. The power losses likely cannot be detected directly with the camera, but they produce indirect changes to the beam's position and waist size that can be monitored by the cameras. The goal is to have a sensitivity of to a power loss of at least 1%. The limiting factors in this sensitivity include the cameras' resolution, the camera's stability over time, how the camera is set-up with regards to the beam it is measuring, and potentially non-camera related effects such as thermal vibrations in the cameras' mounts. In the case of the ETMX, this level of error is roughly 1 pixel, or, using the conversion calculated in Section 3.4, roughly 0.5mm ($c = 0.4351 \frac{\text{mm}}{\text{pixel}}$).

5.1 Cavity Axis Misalignment

In order to know if these sensitivity values are good enough, the minimum measurable power loss must be calculated from these minimum measurable changes in beam parameters. For the case of cavity misalignment, there are two possibilities that a change in beam position could signify: either the beam's position inside the cavity is displaced from the cavity axis, or the beam path inside the cavity is not parallel to the cavity axis. Both produce coupling with the mode U_{01} or U_{10} (the Hermite-Gaussian modes 01 and 10, see Section 3 for a more in depth description) depending on which dimension the misalignment occurs in; however, the modes resulting from displacement and angular misalignment occur at different phases and have different scaling factors. Thus, if the original, aligned waveform is given by

$$\psi = A \cdot U_{00} \tag{8}$$

where U_{00} is the fundamental mode and A is a scalar constant, then the equation for a waveform misaligned in one dimension is

$$\psi = A'(U_{00} + \alpha U_{10} + i\beta U_{10}) \tag{9}$$

where A' is a different scaling constant, α is the factor associated with displacement, and β is the factor associated with angular displacement.[3] Since power is given by the square of the amplitude, these equations become

$$P = A^2 \quad (10)$$

for the aligned beam, and

$$P = A'^2(1 + \alpha^2 + \beta^2) \quad (11)$$

for the misaligned beam. The U_{00}^2 and U_{10}^2 terms become 1 because the Hermite-Gaussian modes are normalized, while the $U_{00}U_{10}$ term becomes 0 since different Hermite-Gaussian modes are orthogonal to each other. Since the amount of power entering the system has remained unchanged, the total powers for the two above equations must be the same. Only the fundamental mode is useful to LIGO however, so power loss is measured in relation to this alone. The percentage power loss is thus given by $\frac{A^2 - A'^2}{A^2}$, or

$$\text{Loss} = \frac{\alpha^2 + \beta^2}{1 + \alpha^2 + \beta^2} \quad (12)$$

The relationship between the loss factors and the degree of misalignment is given by D. Anderson as $\alpha = x/\omega_0$ and $\beta = \frac{\pi\theta\omega_0}{\lambda}$. [3] In these equations, ω_0 is the minimum waist size of the beam in the cavity, x is the displacement from the cavity axis, θ is the angular difference between the beam path and the cavity axis, and λ is the wavelength of laser light in the cavity (1064nm for the LIGO interferometers). The beam waist is given by

$$\omega_0^2 = \frac{L\lambda}{\pi} \sqrt{\frac{g_1g_2(1 - g_1g_2)}{(g_1 + g_2 - 2g_1g_2)^2}} \quad (13)$$

where $g_1 = 1 - \frac{L}{R_1}$, $g_2 = 1 - \frac{L}{R_2}$, R_1 and R_2 are the radii of curvatures of the two mirrors in the cavity, and L is the distance between the two mirrors.[4] In order to determine x and θ , the change in position of the beam's center must be measured on both mirrors of the cavity. If x_1 is the change in one image, and x_2 is the change in the other, then $x = \frac{1}{2}(x_1 + x_2)$ and $\theta = \arctan(\frac{x_1 - x_2}{L}) \approx \frac{x_1 - x_2}{L}$. The y-dimension can be found similarly and is added in quadrature. The minimum measurable percentage power loss will be taken as the maximum power loss that can occur with the measurements on the two mirrors constrained to their minimum sensitivities. This gives

$$\alpha^2 = \frac{\pi}{4L\lambda}(x_1 + x_2)^2 \frac{1}{\sqrt{\frac{g_1g_2(1 - g_1g_2)}{(g_1 + g_2 - 2g_1g_2)^2}}} \quad (14)$$

and

$$\beta^2 = \frac{\pi}{L\lambda}(x_1 - x_2)^2 \sqrt{\frac{g_1g_2(1 - g_1g_2)}{(g_1 + g_2 - 2g_1g_2)^2}} \quad (15)$$

for one dimension. Since the sensitivity for the two mirrors is the same, either $x_1 = x_2$ or $x_1 = -x_2$ can be assumed when calculating the minimum measurable error. In this case, the values for α^2 and β^2 only differ by $4G^2$, where G is defined as

$$G = \sqrt{\frac{g_1g_2(1 - g_1g_2)}{(g_1 + g_2 - 2g_1g_2)^2}} \quad (16)$$

If $4G^2 < 1$, a larger power loss comes from displacement; otherwise, a greater power loss comes from angular misalignment. For the X-arm of the LIGO interferometer, the cavity parameters are $R_1 = 57.375\text{m}$, $R_2 = \infty$ (the mirror is flat), and $L = 38.55\text{m}$. This gives $g_1 = 0.3281$, $g_2 = 1$, $\omega_0 = 3.02\text{mm}$, and $G = 0.6988$, so $4G^2 = 1.95329 > 1$. Angular misalignment will thus be used to calculate the minimum measurable power loss. With $x_1 = 0.5\text{mm}$ and $x_2 = -0.5\text{mm}$ the minimum measurable fractional power loss is roughly 0.0508. Assuming a similar sensitivity to changes in position at the main LIGO sites, which have cavity parameters of $R_1 = 7400\text{m}$, $R_2 = 14540\text{m}$, and $L = 3999\text{m}$, $4G^2 = 3.30974$, so angular misalignment still produces a larger power loss. The resulting minimum measurable power loss is 0.000671139. Measurement sensitivity in the 40m is worse than 1%, while for the main sites, it is significantly better than 1%. It is worth noting, however, that the power loss scales with the change in beam position squared, so if other factors limit the measurement of beam position further, it will significantly increase the minimum measurable power loss.

5.2 Energy Absorption as Heat by the Cavity Mirrors

Energy absorption by the cavity mirrors as heat are another form of energy loss. This heating produces a change in the radius of curvature of the mirror, which results in the beam waist having a different size. The equation for the conversion from energy absorbed to the change in beam waist size is given by the first order approximation

$$\omega_2(P_a) = \omega_2(0) - \left(\frac{\alpha\sqrt{\lambda}}{8\kappa\omega_1^2} \left(\frac{L}{\pi} \right)^{\frac{3}{2}} \frac{1}{g_1^{\frac{3}{4}}g_2^{\frac{1}{4}}(1-g_1g_2)^{\frac{5}{4}}} \right) P_a \quad (17)$$

or $\Delta\omega_2 = \Upsilon P_a$ where

$$\Upsilon = \frac{\alpha\sqrt{\lambda}}{8\kappa\omega_1^2} \left(\frac{L}{\pi} \right)^{\frac{3}{2}} \frac{1}{g_1^{\frac{3}{4}}g_2^{\frac{1}{4}}(1-g_1g_2)^{\frac{5}{4}}} \quad (18)$$

In these equations, P_a is the power absorbed at one mirror of the resonant cavity, $\omega_2(P_a)$ is the new beam waist size at the *opposite* mirror of the cavity, ω_1 is the normal (without considering absorption) beam waist size at the mirror absorbing the power, $\omega_2(0)$ is the normal beam waist size at the opposite mirror, α is the coefficient of thermal expansion, κ is the thermal conductivity of the material, L is the length of the cavity, λ is the wavelength of the laser, g_1 is the g-factor of the mirror absorbing the light, while g_2 is the g-factor of the mirror where the change is being measured.[5] For the 40m interferometer, $\alpha = 0.55 \cdot 10^{-6}\text{K}^{-1}$, $\kappa = 1.38 \frac{\text{W}}{\text{m}\cdot\text{K}}$, and g_1 , g_2 , L , and λ are the same values as given in the previous section. The waist size can be calculated using the equation

$$\omega_1^2 = \frac{L\lambda}{\pi} \sqrt{\frac{g_2}{g_1(1-g_1g_2)}} \quad (19)$$

These values give $\Upsilon = 16.6 \frac{\text{mm}}{\text{W}}$, yielding a minimum measurable power loss of roughly 3mW. This uses the thermal conductivity and coefficient of thermal expansion for fused silica, which is the substrate the mirror is made of. For the main sites, $\Upsilon = 3.6 \frac{\text{mm}}{\text{W}}$, or a minimum measurable power loss of roughly 14mW. Rough estimates of the total power in the cavities are 40kW for the 40m detector and 75kW for the main sites. These are calculated by multiplying the arm cavity gain, the recycling cavity gain, and the initial laser power together for

each location. The cavity gains are taken from http://www.ligo.caltech.edu/~cit40m/Docs/40m_params.pdf, the 40m laser power is estimated at 3W, while the main site laser power is estimated at 12W. This leads to a measurable fractional power loss of $\sim 8 \cdot 10^{-8}$ for the 40m and $\sim 2 \cdot 10^{-7}$ for the main sites. In both cases, these sensitivities are significantly better than 1%. This is a significantly more accurate measurement than for power losses due to misalignments.

6 Scanning the Pre-mode Cleaner

The pre-mode cleaner (PMC) is an optical instrument that attempts to 'clean' the laser modes before they enter the interferometer; i.e., it attempts to reduce the intensity of higher order modes in the laser by as much as possible, leaving only the fundamental (TEM_{00}). It is a three mirror resonant cavity in which one mirror is connected to a PZT (Piezoelectric Transducer). The PZT allows the position of the mirror to be changed very precisely, which changes the length of the resonant cavity. When the cavity is the correct length for a given mode, that mode resonates inside the cavity and is greatly amplified over the other modes. In normal operation, control loops connected to voltage of the PZT lock the length of the cavity to the length that resonates with the fundamental mode, causing the output to be dominated by the fundamental mode. It is also possible to unlock the PMC and scan the length range providing a relationship between the voltage applied to the PZT and the transmitted intensity. The result is a series of resonance peaks. If the peaks could be fitted to equations, where each peak is given by the equation

$$I(x) = I_{max} \cdot \frac{\left(\frac{\Gamma}{2}\right)^2}{(x - x_0)^2 + \left(\frac{\Gamma}{2}\right)^2} + I_{offset} \quad (20)$$

the resulting parameters could be used to find the finesse (essentially, narrowness) of the fundamental peak, the number of volts between two identical peaks (the free spectral range), and the positions of higher order peaks in relation to the fundamental peak. Knowing the free spectral range of the PMC allows it to be calibrated: it is not difficult to calculate the theoretical distance between two fundamental peaks in hertz, so this would give a relationship between PZT volts and hertz. The finesse yields information regarding the effectiveness of the PMC. Though this is unrelated to the camera network, the code developed to fit the images was applicable to fitting these peaks with only a few modifications.

To actually collect the data, a few modifications to the standard set-up for the PMC must be made. First, the control loop which locks the PMC to a transmission peak must be disconnected from the PZT voltage, or else the system will resist any attempt to scan past a peak. Second, the laser power must be reduced significantly due to thermal effects. Since the laser has a very high power, whenever a peak is encountered the power becomes large enough to significantly heat the mirror connected to the PZT, causing it to change shape slightly and modify the length of the cavity. This greatly distorts the shape of the resonance peak. However, if the laser power is decreased too much, higher order peaks with smaller maximums may become so small that they are lost inside the noise floor. Once these modifications are made, the PZT voltage is modified by running a script that feeds a slow

triangle wave into a PZT voltage offset EPICs channel. Then, both the PZT voltage and the transmission intensity are read out from separate channels.

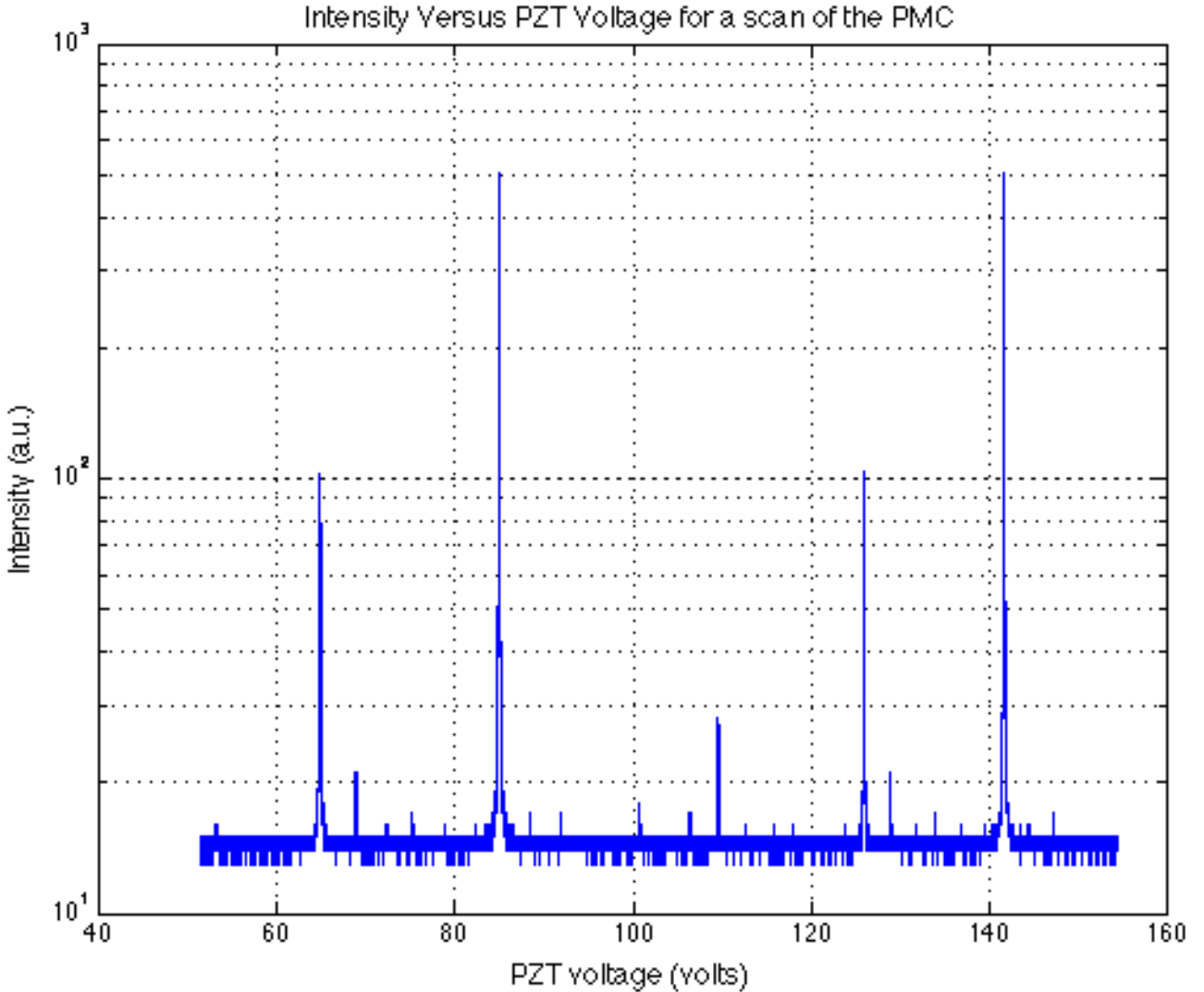


Figure 7: Graph of the a scan of the PMC. An arbitrary offset of 200 ADC counts was added to all the data in order to make all points positive so it could be plotted on a log scale.

The results of one such scan of the PMC are shown in Figures 7 and 8. The peak pattern repeats, as seen in Figure 7. The spacing between these repetitions is dependent on the frequency of the laser and can be easily calculated, which provides a way of calibrating the relationship between PZT voltage and peak position. However, viewed closely, the peaks' shapes are clearly distorted. When the PZT voltage is rising (Figure 8a) the peak is too sharp for a lorentzian, while when the PZT voltage is falling (Figure 8b), the peak plateaus. This error is consistent with the thermal effects discussed earlier. Increasing the PZT voltage shrinks the cavity, and when the mirror thermally expands, it further shrinks the cavity, causing the peak to be transverse more rapidly. Decreasing the PZT voltage expands the cavity, so the thermal expansion (which shrinks the cavity) instead opposes the expected

change in the cavity size and causes the peak to become wider. To solve this problem, the laser power was lowered by much more significantly (a factor of ~ 1000) in an attempt to remove this error. This lowered the maximum intensity of the fundamental resonant peak below the digitization noise of the Data Acquisition (DAQ) hardware that collected the data. An SR650 Programmable Filter was placed between the photodiode and the DAQ. No filter was applied, but the gain on the SR650 was set to 200. The PMC was scanned again, but the thermal effects were still present in the peak's shape. Additionally, extra noise had been introduced into the signal (presumably by the SR650, though it is possible that it simply amplified some other noise inherent in the system) that had an amplitude on the same order of magnitude as the resonant peak height. This noise could likely be reduced significantly through averaging, but since thermal effects were still clearly present in the peak and reducing laser power further wouldn't be possible without additions to the optical set-up, this was not attempted.

7 Conclusions

7.1 Results

The camera network, while not yet constructed, has had the groundwork laid to make creating the full network a much simpler task. Software has been written that communicates with the Prosilica cameras and allows settings for individual cameras to be controlled remotely with a computer. It is possible to save images either individually or as a set in which they are automatically time-stamped. Additionally, a single setting allows the software to switch from controlling one camera to another, making it easy to add more cameras to the network. Individual camera settings can also be saved to a configuration file that can be recalled, so that separate settings for separate cameras can be stored simply. These features should allow the network to both be easily put together once the required equipment has been ordered, but should also make it easy to expand later if necessary.

In addition to setting up the network, it has been shown that the cameras can be used as sensors in control loops within the LIGO interferometer. The specific example of a control loop developed in this paper is a servo that provides control of the alignment of the laser beam within the X-arm of the interferometer. The necessary calculations were added to the camera software so that it could determine the laser beam's position from its Gaussian profile on one mirror of the resonant cavity in the X-arm. These calculations were shown to work fairly well (within 1 to 2 pixels error) despite significant noise from defects in the mirror's surface. Additional software was developed that uses points in the image with known physical positions (the OSEMs of the mirror, in this case) to convert positions in the image to positions in physical space with fairly good accuracy (this could likely be improved upon, however, as discussed briefly in Section 3.4). This allows the camera software to communicate the positions of relevant features to other software that knows nothing of the camera's orientation. Additionally, the software was set-up to be capable of outputting these calculated values to EPICS channels, so that it is accessible by the servo script which runs the cavity alignment, or by any other program that may wish to access the information.

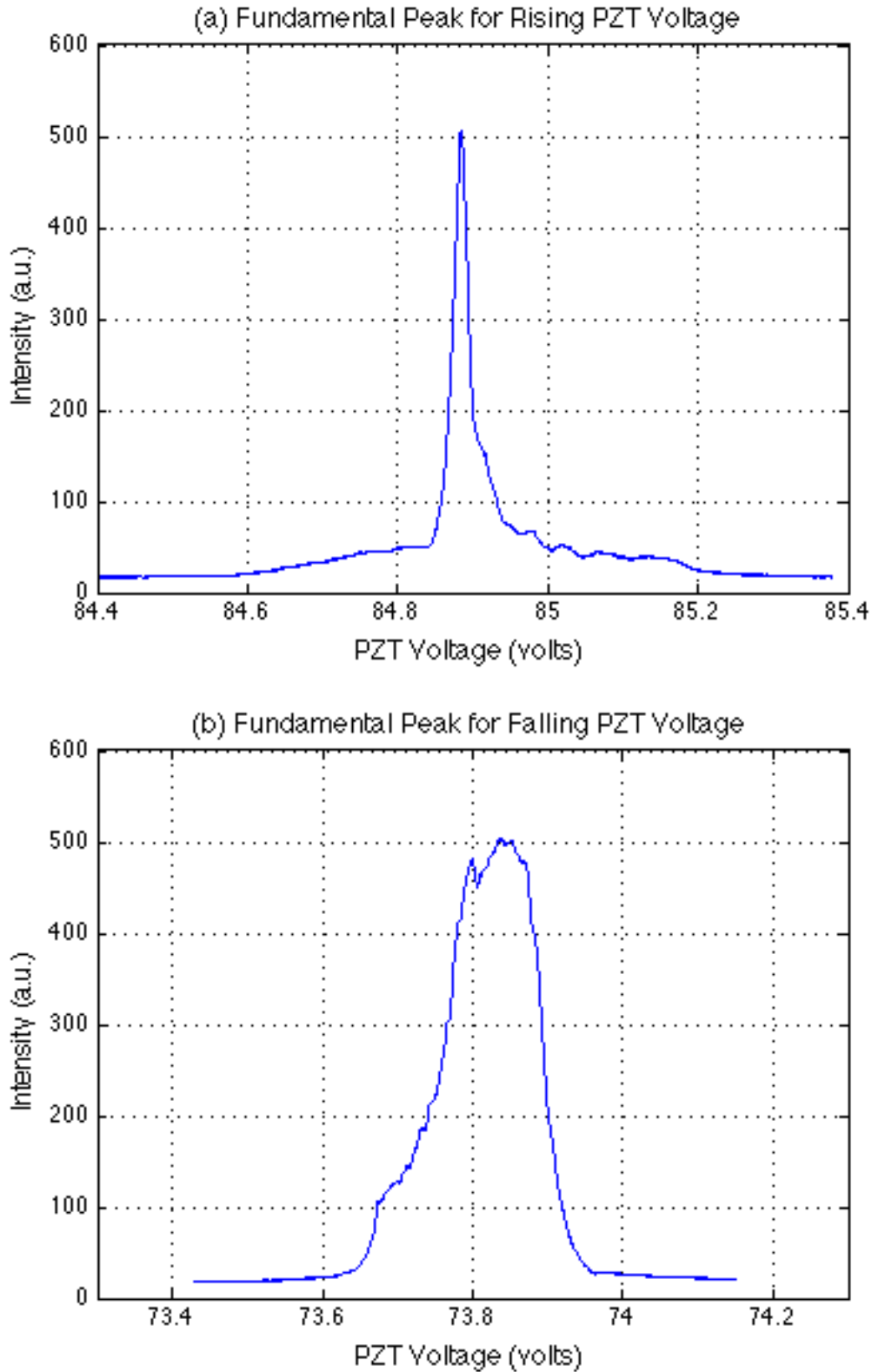


Figure 8: The fundamental peaks from the same scan as above. (a) shows the shape of the peak when the PZT voltage is increasing, while (b) shows the shape of the peak when the PZT voltage is decreasing.

The preliminary step towards using the cameras to detect power losses in resonant cavities due to cavity misalignment and power absorption was also accomplished. Calculations were done that, using the bounds on the cameras' ability to detect changes in beam position and waist size, set bounds on the minimum power losses that could be detected in this fashion. The results suggest that the cameras can potentially be used to measure power losses due to absorption by the mirrors. Sensitivity to misalignments is much worse, especially in the 40m cavity, though it is still manageable in the cavities at the main sites. Once developed, this could be another way the camera network is integrated into LIGO's control systems.

The attempt to scan the Pre-Mode Cleaner and use a chi-squared minimization fit to determine the frequencies of its resonances was not successful. Thermal effects that changed the curvature of the cavity's mirrors occurred whenever the cavity approached a resonance, which greatly distorted the resonance peak. The laser power was turned down in an attempt to compensate for this, but even at very low laser powers, the effect persisted. Although the laser power could be reduced further, the resonance peaks were already almost within the noise floor. However, obtaining a good scan could still be possible, either by attempting to average out the noise and add optical components to further reduce the laser power, or by attempting to predict and remove the influence of the absorbed power, as described below.

7.2 Future Work

The purpose of this project was partly to show that the new digital cameras could be used within control loops in LIGO, and partly to begin setting up this control network. Much more work needs to be done to flesh out this system, however. At the moment, only three cameras are connected to the network, only two of which are set up to image parts of the interferometer. This network needs to be expanded so that there are cameras viewing every important optical device (on the order of 20 cameras). Along with this expansion, the camera software needs to be developed into a client-side program and a server-side program, in order to better manage network traffic and user requests (this is discussed more in Section 2.2). Additionally, the software that manages the cameras, saves image files, and updates the EPICS channels is run in the Linux command line and is very unpolished. When the camera network is expanded to include many more cameras, a more user friendly interface will need to be designed.

It is also necessary to begin finding other useful applications of the cameras' ability to interface with computers in real time. One application is to monitor the power loss in the major cavities using the equations described in Section 5. This could be done almost immediately in the two-mirror resonant cavities, though for cavities with different optical arrangements, the calculations would first need to be redone. Potentially, it may be possible to measure the effects of thermal expansion of the mirrors in the PMC (a three mirror cavity). Were this expansion modeled, it would be possible to run scans of the PMC without decreasing the power for large peaks, making this a much simpler process. It would also allow for a better understanding of the mechanisms within the cavity when it is locked on the high power fundamental peak, providing better information on the magnitudes of higher order modes that clutter the beam profile.

While this project has shown that it is possible to use the cameras as sensors for servos in LIGO's control loops, the camera network needs to become much more integrated into the control system. As more cameras are added to the network, beam alignment servos can be set-up for other optical components and optical cavities. By adding software that does fitting to higher order Gaussian modes, the cameras could potentially help improve the auto-locking scripts that attempt to lock various cavities (such as the PMC) to the fundamental mode. Additionally, the digital cameras provide a much finer control over image exposure, position, and capturing speed that could lead to other uses that the old analog would be unable to do.

A Source Code

A.1 Fitting Code

A.1.1 Matlab Code

```

1 % TwoDGaussianFit
2 % Author: Eric Mintun
3 % Date: May 31, 2008
4
5 % Fits the input data to a two dimensional Gaussian and outputs the
6 % resulting Chi-Squared goodness-of-fit value, as well as the fit
7 % parameters. The fit currently takes two two-dimensional matrices, where
8 % inputMatrix gives the y values and errorMatrix gives the errors on these
9 % y values. The matrix position is taken as the x1 and x2 values;
10 % non-integer, non-successive independent variables are not currently
11 % supported. Uses the Levenberg-Marquardt algorithm from "Numerical
12 % Recipes in C".
13
14 function [reducedChiSquared, fitParameters]=TwoDGaussianFit(inputMatrix,
    errorMatrix)
15
16 if size(inputMatrix) ~= size(errorMatrix)
17     error('The sizes of the data matrix and the error matrix do not match')
18 end;
19
20 if min(min(errorMatrix)) <= 0
21     error('At least one point has a zero or negative error value')
22 end;
23
24 inputSize = size(inputMatrix);
25 fitParameters = estimateParameters(inputMatrix); %Initializes the six fit
    parameters, and sets them equal to the estimated parameters
26 % Order of parameters is: Y_max, mu1, sigma1, mu2, sigma2, Y_offset
27 hessian = zeros(6,6); %Initializes the Hessian matrix for the fit
28 betaParameters = zeros(6,1); %Initializes beta parameters used in the fit
29 lambda = 0.001; %Initializes lambda
30 fitTolerance = 0.01; %When the change in chi-Squareds is less than this value,
    the algorithm halts
31 finished = 0; %Set to 1 when fitting loop finishes
32 counter = 0; %Increased every time a loop finishes
33 countTolerance = 1000; %Fitting times out after this many iterations

```

```

34
35 % Sets up the 2D Gaussian function and creates a vector of its derivatives
36 % in its 6 fit parameters
37 %syms yMax x1 mu1 sigma1 x2 mu2 sigma2 yOffset y;
38 %twoDGaussian = yMax*exp(-(x1 - mu1)^2/sigma1^2)*exp(-(x2-mu2)^2/sigma2^2)+
    yOffset;
39 %derivativeVector = [diff(twoDGaussian, yMax), diff(twoDGaussian, mu1), diff(
    twoDGaussian, sigma1), diff(twoDGaussian, mu2), diff(twoDGaussian, sigma2), diff
    (twoDGaussian, yOffset)];
40
41 while finished == 0
42
43     %Generates the sums of derivatives necessary for the various terms of the
44     %Hessian function
45     counter
46     'Generating_Hessian'
47     for L=1:6
48         %Sets the first term of the Hessian element to the derivative of that
49         %particular fit paramter and plugs in the current fit parameters
50         %firstTerm = subs(subs(subs(subs(subs(derivativeVector(L), yMax,
            fitParameters(1)), mu1, fitParameters(2)), sigma1, fitParameters
            (3)), mu2, fitParameters(4)), sigma2, fitParameters(5)), yOffset,
            fitParameters(6));
51
52     for K=1:6
53         %Sets the second term of the Hessian element to the derivative of
54         %that particular fit parameter and plugs in the current fit
55         %parameters
56         %secondTerm = subs(subs(subs(subs(subs(derivativeVector(K),
            yMax, fitParameters(1)), mu1, fitParameters(2)), sigma1,
            fitParameters(3)), mu2, fitParameters(4)), sigma2,
            fitParameters(5)), yOffset, fitParameters(6));
57
58         %Creates the Hessian element by summing over all independent
59         %variables
60         temporaryElement = 0;
61         for x1Value = 1:inputSize(1)
62             for x2Value = 1:inputSize(2)
63                 if K==1
64                     firstTerm = twoDGaussianDyMax(x1Value, x2Value,
                        fitParameters);
65                 elseif K==2
66                     firstTerm = twoDGaussianDmu1(x1Value, x2Value,
                        fitParameters);
67                 elseif K==3
68                     firstTerm = twoDGaussianDsigma1(x1Value, x2Value,
                        fitParameters);
69                 elseif K==4
70                     firstTerm = twoDGaussianDmu2(x1Value, x2Value,
                        fitParameters);
71                 elseif K==5
72                     firstTerm = twoDGaussianDsigma2(x1Value, x2Value,
                        fitParameters);
73                 else
74                     firstTerm = twoDGaussianDyOffset(x1Value, x2Value,
                        fitParameters);

```

```

75         end;
76         if L==1
77             secondTerm = twoDGaussianDyMax(x1Value, x2Value,
78                 fitParameters);
79         elseif L==2
80             secondTerm = twoDGaussianDmu1(x1Value, x2Value,
81                 fitParameters);
82         elseif L==3
83             secondTerm = twoDGaussianDsigma1(x1Value, x2Value,
84                 fitParameters);
85         elseif L==4
86             secondTerm = twoDGaussianDmu2(x1Value, x2Value,
87                 fitParameters);
88         elseif L==5
89             secondTerm = twoDGaussianDsigma2(x1Value, x2Value,
90                 fitParameters);
91         else
92             secondTerm = twoDGaussianDyOffset(x1Value, x2Value,
93                 fitParameters);
94         end;
95         temporaryElement = temporaryElement + firstTerm *
96             secondTerm / errorMatrix(x1Value, x2Value)^2;
97     end;
98     end;
99     hessian(L,K) = temporaryElement;
100 end;
101 hessian
102 'Generating_beta_parameters'
103 %Generates the beta parameters
104 for K=1:6
105     %Sets up the function that independent and dependent values will be
106     %substituted into
107     %betaFunction = subs(subs(subs(subs(subs(subs((y - twoDGaussian) *
108         derivativeVector(K), yMax, fitParameters(1)), mu1, fitParameters
109         (2)), sigma1, fitParameters(3)), mu2, fitParameters(4)), sigma2,
110         fitParameters(5)), yOffset, fitParameters(6));
111     temporaryElement = 0;
112     for x1Value = 1:inputSize(1)
113         for x2Value = 1:inputSize(2)
114             %Sums over all independent variables
115             %temporaryElement = temporaryElement + subs(subs(subs(
116                 betaFunction, x1, x1Value), x2, x2Value), y, inputMatrix(
117                 x1Value, x2Value)) / errorMatrix(x1Value, x2Value)^2;
118         if K==1
119             betaFunction = twoDGaussianDyMax(x1Value, x2Value,
120                 fitParameters);
121         elseif K==2
122             betaFunction = twoDGaussianDmu1(x1Value, x2Value,
123                 fitParameters);

```

```

117         elseif K==3
118             betaFunction = twoDGaussianDsigma1(x1Value, x2Value,
119                 fitParameters);
120         elseif K==4
121             betaFunction = twoDGaussianDmu2(x1Value, x2Value,
122                 fitParameters);
123         elseif K==5
124             betaFunction = twoDGaussianDsigma2(x1Value, x2Value,
125                 fitParameters);
126         else
127             betaFunction = twoDGaussianDyOffset(x1Value, x2Value,
128                 fitParameters);
129         end;
130         temporaryElement = temporaryElement + (inputMatrix(x1Value,
131             x2Value) - twoDGaussian(x1Value, x2Value, fitParameters)) *
132             betaFunction / errorMatrix(x1Value, x2Value)^2;
133     end;
134     end;
135     betaParameters(K) = temporaryElement;
136 end;
137 betaParameters
138
139 %Solves for modifications to the h
140 %parameters using the hessian matrix
141 %and the betaParameters
142 'Solving system of equations'
143 deltaFitParameters = linsolve(hessian, betaParameters);
144
145 deltaFitParameters
146
147 'Calculating Old Chi-Squared'
148 %Determines the chi-squared value for the previous fit parameters
149 temporaryElement = 0;
150 for x1Value = 1:inputSize(1)
151     for x2Value = 1:inputSize(2)
152         %Sums over all independent variables
153         temporaryElement = temporaryElement + (inputMatrix(x1Value, x2Value)
154             - twoDGaussian(x1Value, x2Value, fitParameters))^2 /
155             errorMatrix(x1Value, x2Value)^2;
156     end;
157 end;
158 oldChiSquared = temporaryElement;
159
160 'Calculating New Chi-Squared'
161 %Updates the fit parameters and determines the new chi-squared value
162 newFitParameters = fitParameters + deltaFitParameters;
163 %chiSquaredFunction = subs(subs(subs(subs(subs(subs((y - twoDGaussian)^2,
164     yMax, newFitParameters(1)), mu1, newFitParameters(2)), sigma1,
165     newFitParameters(3)), mu2, newFitParameters(4)), sigma2,
166     newFitParameters(5)), yOffset, newFitParameters(6));
167 temporaryElement = 0;
168 for x1Value = 1:inputSize(1)
169     for x2Value = 1:inputSize(2)
170         %Sums over all independent variables

```

```

162         temporaryElement = temporaryElement + (inputMatrix(x1Value, x2Value
                ) - twoDGaussian(x1Value, x2Value, newFitParameters))^2 /
                errorMatrix(x1Value, x2Value)^2;
163     end;
164 end;
165
166 newChiSquared = temporaryElement;
167
168 %Updates values for next iteration
169 oldChiSquared
170 newChiSquared
171 if (newChiSquared - oldChiSquared) > 0
172     'Increasing  $\lambda$ '
173     lambda = lambda * 10;
174 else
175     'Decreasing  $\lambda$ '
176     lambda = lambda / 10;
177     fitParameters = newFitParameters;
178     %Checks to see if iterating should stop
179     if abs(newChiSquared - oldChiSquared) < fitTolerance
180         %Sets the output to the calculated chi-squared
181         reducedChiSquared = newChiSquared/(inputSize(1)*inputSize(2) - 6);
182         %Tells the loop to stop
183         finished = 1;
184     end;
185 end;
186 fitParameters
187
188 %Throws an error if fitting fails due to time out.
189 counter = counter + 1;
190 if counter > countTolerance
191     error('Fitting timed out after too many iterations without reaching
            the desired precision')
192 end;
193 end;
194
195 function value=twoDGaussian(x1, x2, parameters)
196
197 yMax = parameters(1);
198 mu1 = parameters(2);
199 sigma1 = parameters(3);
200 mu2 = parameters(4);
201 sigma2 = parameters(5);
202 yOffset = parameters(6);
203
204 value = yMax * exp(-(x1 - mu1)^2/sigma1^2 - (x2 - mu2)^2/sigma2^2) + yOffset;
205
206 function value=twoDGaussianDyOffset(x1, x2, parameters)
207
208 yMax = parameters(1);
209 mu1 = parameters(2);
210 sigma1 = parameters(3);
211 mu2 = parameters(4);
212 sigma2 = parameters(5);
213 yOffset = parameters(6);
214

```

```

215 value = 1;
216
217 function value=twoDGaussianDyMax(x1,x2,parameters)
218
219 yMax = parameters(1);
220 mu1 = parameters(2);
221 sigma1 = parameters(3);
222 mu2 = parameters(4);
223 sigma2 = parameters(5);
224 yOffset = parameters(6);
225
226 value = exp(-(x1 - mu1)^2/sigma1^2 - (x2 - mu2)^2/sigma2^2);
227
228 function value=twoDGaussianDmu1(x1,x2,parameters)
229
230 yMax = parameters(1);
231 mu1 = parameters(2);
232 sigma1 = parameters(3);
233 mu2 = parameters(4);
234 sigma2 = parameters(5);
235 yOffset = parameters(6);
236
237 value = yMax * exp(-(x1 - mu1)^2/sigma1^2 - (x2 - mu2)^2/sigma2^2) * 2 * (x1 -
    mu1) /sigma1^2;
238
239 function value=twoDGaussianDmu2(x1,x2,parameters)
240
241 yMax = parameters(1);
242 mu1 = parameters(2);
243 sigma1 = parameters(3);
244 mu2 = parameters(4);
245 sigma2 = parameters(5);
246 yOffset = parameters(6);
247
248 value = yMax * exp(-(x1 - mu1)^2/sigma1^2 - (x2 - mu2)^2/sigma2^2) * 2 * (x2 -
    mu2) /sigma2^2;
249
250 function value=twoDGaussianDsigma1(x1,x2,parameters)
251
252 yMax = parameters(1);
253 mu1 = parameters(2);
254 sigma1 = parameters(3);
255 mu2 = parameters(4);
256 sigma2 = parameters(5);
257 yOffset = parameters(6);
258
259 value = yMax * exp(-(x1 - mu1)^2/sigma1^2 - (x2 - mu2)^2/sigma2^2) * 2 * (x1 -
    mu1)^2 /sigma1^3;
260
261 function value=twoDGaussianDsigma2(x1,x2,parameters)
262
263 yMax = parameters(1);
264 mu1 = parameters(2);
265 sigma1 = parameters(3);
266 mu2 = parameters(4);
267 sigma2 = parameters(5);

```

```

268 yOffset = parameters(6);
269
270 value = yMax * exp(-(x1 - mu1)^2/sigma1^2 - (x2 - mu2)^2/sigma2^2) * 2 * (x2 -
      mu2)^2 /sigma2^3;

1  %Two Dimensional Gaussian Parameter Estimation
2  %Author: Eric Mintun
3  %Date: May 31, 2008
4
5  %Uses some simple methods to estimate the six parameters of a two
6 %dimensional gaussian. These are used primarily for starting estimates for
7 %the 2DGaussian fit function, though they may be useful elsewhere as well.
8 %Since they're meant as initial estimates, they're accuracy cannot be
9 %relied on. Returns a column vector of parameters in the order: yMax, mu1,
      sigma1, mu2,
10 %sigma2, yOffset
11
12 function parameterEstimates=estimateParameters(inputMatrix)
13
14 inputSize = size(inputMatrix);
15
16 %The offset is taken as the minimum value found
17 yOffset = min(min(inputMatrix));
18
19 fakeYMax = max(max(inputMatrix));
20
21 % Calculates centroid of the gaussian using a center of mass calculation
22
23 % Sums row vectors
24 rowSums = zeros(inputSize(1),1);
25 for x1=1:inputSize(1)
26     for i=1:inputSize(2)
27         if inputMatrix(x1,i) - yOffset > 0.1 * fakeYMax
28             rowSums(x1) = rowSums(x1) + inputMatrix(x1,i) - yOffset;
29         end;
30     end;
31 end;
32
33 % Calculates weighted average for row position based on summed intensity
34 runningTotal = 0;
35 for currentPosition = 1:length(rowSums)
36     runningTotal = runningTotal + currentPosition * rowSums(currentPosition);
37 end;
38
39 %Divides by total weight
40 mul = runningTotal / sum(rowSums);
41
42 %Sums column vectors
43 columnSums = zeros(inputSize(2),1);
44 for x2=1:inputSize(2)
45     for i=1:inputSize(1)
46         if inputMatrix(i,x2) - yOffset > 0.1 * fakeYMax
47             columnSums(x2) = columnSums(x2) + inputMatrix(i,x2) - yOffset;
48         end;
49     end;
50 end;

```

```

51
52 % Calculates weighted average for column position based on summed intensity
53 runningTotal = 0;
54 for currentPosition = 1:length(columnSums)
55     runningTotal = runningTotal + currentPosition * columnSums(currentPosition
56         );
57 end;
58 %Divides by total weight
59 mu2 = runningTotal / sum(columnSums);
60
61 %yMax is taken as the value at the centroid minus the offset
62 yMax = inputMatrix(round(mu1),round(mu2)) - yOffset;
63
64 %sigma values are taken as 1/e along the column and row
65 %vectors defined by the centroid (rounds to nearest integer for row/column
66 %position)
67 currentMin = abs(yMax - yOffset - yMax * exp(-1));
68 currentMinPosition = 0;
69
70 for L = 1:inputSize(1)
71     if abs(inputMatrix(L,round(mu2)) - yOffset - yMax * exp(-1)) < currentMin
72         currentMin = abs(inputMatrix(L,round(mu2)) - yOffset - yMax * exp(-1))
73         ;
74         currentMinPosition = L;
75     end;
76 end;
77 sigma1 = abs(currentMinPosition - mu1);
78
79 currentMin = abs(yMax - yOffset - yMax * exp(-1));
80 currentMinPosition = 0;
81 for L = 1:inputSize(2)
82     if abs(inputMatrix(round(mu1),L) - yOffset - yMax * exp(-1)) < currentMin
83         currentMin = abs(inputMatrix(round(mu1),L) - yOffset - yMax * exp(-1))
84         ;
85         currentMinPosition = L;
86     end;
87 end;
88 sigma2 = abs(currentMinPosition - mu2);
89
90 parameterEstimates = [yMax; mu1; sigma1; mu2; sigma2; yOffset];

```

A.1.2 C Code

```

1 //EMfitTwoDGaussian
2 //Author: Eric Mintun
3 //Date: 6/8/08
4 //Fits the input data to a two dimensional Gaussian and outputs the
5 //resulting Chi-Squared goodness-of-fit value. The fit currently takes two
6 //two dimensional matrices, where inputMatrix gives the y values and
7 //errorMatrix gives the errors on these y values. The matrix position is
8 //taken as the x1 and x2 values; non-integer, non-successive independent
9 //variables are not currently supported. Uses the Levenberg-Marquardt
10 //algorithm. The output chi-squared is reduced.
11
12 #include <stdio.h>

```

```

13 #include <math.h>
14 #include <gsl/gsl_linalg.h>
15 #include <gsl/gsl_blas.h>
16 #include "EMfitTwoDGaussian.hpp"
17 #include <gsl/gsl_randist.h>
18 #include <time.h>
19 #include "EMauxiliaryGSLMatrixFunctions.hpp"
20
21 //Comment out to remove any debug print commands
22 // #define DEBUG
23
24 //Primary function. Output is success or failure. If initialEstimates is
  NULL it will attempt to guess its own starting points. Otherwise, it uses
  the values in initialEstimates
25
26 int EMfitTwoDGaussian(gsl_matrix * dataMatrix, gsl_matrix * errorMatrix,
  EMtwoDGaussianFitParameters * fitParameters, EMtwoDGaussianFitParameters *
  initialEstimates)
27 {
28   if(dataMatrix->size1 != errorMatrix->size1 || dataMatrix->size2 !=
  errorMatrix->size2)
29   {
30     fprintf(stderr, "Error in EMfitTwoDGaussian: The size of the data matrix
  does not match the size of the error matrix\n");
31     return 0;
32   }
33   if(gsl_matrix_min(errorMatrix) <= 0)
34   {
35     fprintf(stderr, "Error in EMfitTwoDGaussian: One or more of the error values
  is less than or equal to zero\n");
36     return 0;
37   }
38
39   //Initialize pointers for the 6 parameters.
40   double yMax;
41   double mul;
42   double sigma1;
43   double mu2;
44   double sigma2;
45   double yOffset;
46
47   //Initializes various values related to the fitting process.
48   gsl_matrix * hessian = gsl_matrix_calloc(6,6);
49   gsl_matrix * hessianDuplicate = gsl_matrix_calloc(6,6); //Since the linear
  equation solver destroys the matrix, this copy is used in the fit
50   gsl_vector * betaParameters = gsl_vector_calloc(6);
51   gsl_vector * changeInParameters = gsl_vector_calloc(6);
52   double lambda = 0.001;
53   double oldChiSquared;
54   double newChiSquared;
55   double fitTolerance = 0.01; //Iterative process ends successfully when the
  improvement in chi-squared is less than this
56   int counter = 0; //Keeps track of number of iterations done
57   int countTolerance = 50; //The function times out after this many of
  iterations
58   //Various counters used in the main process

```

```

59  int L;
60  int K;
61  int x1;
62  int x2;
63  //Temporary variables used in loops;
64  double firstFitTerm;
65  double secondFitTerm;
66  double runningTotal;
67  //Struct for calling the parameter estimation function
68  EMtwoDGaussianFitParameters parameterEstimates;
69
70  //Loads initial guesses into the parameters.  Fails if
      EMestimateTwoDGaussianParameters fails.
71  fprintf(stderr, "Estimating Parameters\n"); //Debug output
72  if(initialEstimates == NULL)
73  {
74      if(EMestimateTwoDGaussianParameters(dataMatrix, &parameterEstimates) == 0)
75      {
76          fprintf(stderr, "Error in EMfitTwoDGaussian:
      EMestimateTwoDGaussianParameters failed.\n");
77          return 0;
78      }
79      yMax = parameterEstimates.yMax;
80      mu1 = parameterEstimates.mu1;
81      sigma1 = parameterEstimates.sigma1;
82      mu2 = parameterEstimates.mu2;
83      sigma2 = parameterEstimates.sigma2;
84      yOffset = parameterEstimates.yOffset;
85  }
86  else
87  {
88      yMax = initialEstimates->yMax;
89      mu1 = initialEstimates->mu1;
90      sigma1 = initialEstimates->sigma1;
91      mu2 = initialEstimates->mu2;
92      sigma2 = initialEstimates->sigma2;
93      yOffset = initialEstimates->yOffset;
94  }
95
96
97  //Output estimate values for debug.
98  #ifdef DEBUG
99  fprintf(stderr, "Value estimates: \nyMax: %lf \nmu1: %lf \nsigma1: %lf \nmu2:
      %lf \nsigma2: %lf \nyOffset: %lf \n", yMax, mu1, sigma1, mu2, sigma2,
      yOffset);
100 #endif
101
102 //Primary iterative loop.  Finishes when too many iterations have occurred or
      when the chi-squared is changing by very little.
103 int finished = 0;
104 while (finished == 0)
105 {
106
107     fprintf(stderr, "Currently on iteration %d.\n", counter+1);
108     if(counter > countTolerance)
109     {

```

```

110     fprintf(stderr, "Error in EMfitTwoDGaussian: The function was unable to
        reach the required level of precision in the maximum allowed number of
        iterations\n");
111     return 0;
112 }
113
114 //Constructs the Hessian matrix used for the fit
115 #ifdef DEBUG
116 fprintf(stderr, "Constructing Hessian\n"); //Debug output
117 #endif
118 for(K = 0; K < 6; K++)
119 {
120     for(L = 0; L < 6; L++)
121     {
122         runningTotal = 0;
123         for(x1 = 0; x1 < dataMatrix->size1; x1++)
124         {
125             for(x2 = 0; x2 < dataMatrix->size2; x2++)
126             {
127                 //Determines the correct derivative of the 2D Gaussian to use for the
                    two parts of the term
128                 if(K==0)
129                     firstFitTerm = EMtwoDGaussianDyMax(x1, x2, yMax, mu1, sigma1, mu2,
                        sigma2, yOffset);
130                 else if(K==1)
131                     firstFitTerm = EMtwoDGaussianDmu1(x1, x2, yMax, mu1, sigma1, mu2,
                        sigma2, yOffset);
132                 else if(K==2)
133                     firstFitTerm = EMtwoDGaussianDsigma1(x1, x2, yMax, mu1, sigma1, mu2,
                        sigma2, yOffset);
134                 else if(K==3)
135                     firstFitTerm = EMtwoDGaussianDmu2(x1, x2, yMax, mu1, sigma1, mu2,
                        sigma2, yOffset);
136                 else if(K==4)
137                     firstFitTerm = EMtwoDGaussianDsigma2(x1, x2, yMax, mu1, sigma1, mu2,
                        sigma2, yOffset);
138                 else
139                     firstFitTerm = EMtwoDGaussianDyOffset(x1, x2, yMax, mu1, sigma1, mu2,
                        sigma2, yOffset);
140                 if(L==0)
141                     secondFitTerm = EMtwoDGaussianDyMax(x1, x2, yMax, mu1, sigma1, mu2,
                        sigma2, yOffset);
142                 else if(L==1)
143                     secondFitTerm = EMtwoDGaussianDmu1(x1, x2, yMax, mu1, sigma1, mu2,
                        sigma2, yOffset);
144                 else if(L==2)
145                     secondFitTerm = EMtwoDGaussianDsigma1(x1, x2, yMax, mu1, sigma1, mu2,
                        sigma2, yOffset);
146                 else if(L==3)
147                     secondFitTerm = EMtwoDGaussianDmu2(x1, x2, yMax, mu1, sigma1, mu2,
                        sigma2, yOffset);
148                 else if(L==4)
149                     secondFitTerm = EMtwoDGaussianDsigma2(x1, x2, yMax, mu1, sigma1, mu2,
                        sigma2, yOffset);
150                 else

```

```

151     secondFitTerm = EMtwoDGaussianDyOffset(x1, x2, yMax, mu1, sigma1, mu2,
152         sigma2, yOffset);
153     //Adds the term to the running total
154     runningTotal = runningTotal + firstFitTerm * secondFitTerm / pow(
155         gsl_matrix_get(errorMatrix, x1, x2),2);
156 }
157 //Adds in lambda if appropriate
158 if(L==K)
159 {
160     runningTotal = runningTotal * (1 + lambda);
161 }
162 //Sets the appropriate value of the Hessian to the calculated value
163 gsl_matrix_set(hessian, K, L, runningTotal);
164 }
165 }
166 //Debug output for Hessian matrix
167 #ifdef DEBUG
168 fprintf(stderr, "Hessian:\n");
169 for(K = 0; K < 6; K++)
170 {
171     for(L = 0; L < 6; L++)
172     {
173         fprintf(stderr, "%lf\t", gsl_matrix_get(hessian, K, L));
174     }
175     fprintf(stderr, "\n");
176 }
177 #endif
178
179 //Constructs the beta parameter vector
180 #ifdef DEBUG
181 fprintf(stderr, "Constructing_beta_parameters.\n");
182 #endif
183 for(K = 0; K < 6; K++)
184 {
185     runningTotal = 0;
186     for(x1 = 0; x1 < dataMatrix->size1; x1++)
187     {
188         for(x2 = 0; x2 < dataMatrix->size2; x2++)
189         {
190             //Determines the correct derivative to use and loads the related value
191             if(K==0)
192                 firstFitTerm = EMtwoDGaussianDyMax(x1, x2, yMax, mu1, sigma1, mu2,
193                     sigma2, yOffset);
194             else if(K==1)
195                 firstFitTerm = EMtwoDGaussianDmu1(x1, x2, yMax, mu1,
196                     sigma1, mu2, sigma2, yOffset);
197             else if(K==2)
198                 firstFitTerm = EMtwoDGaussianDsigma1(x1, x2, yMax,
199                     mu1, sigma1, mu2, sigma2, yOffset);
200             else if(K==3)
201                 firstFitTerm = EMtwoDGaussianDmu2(x1, x2, yMax, mu1,
202                     sigma1, mu2, sigma2, yOffset);
203             else if(K==4)

```

```

200         firstFitTerm = EMtwoDGaussianDsigma2(x1, x2, yMax, mu1,
201         signal, mu2, sigma2, yOffset);
202     else
203         firstFitTerm = EMtwoDGaussianDyOffset(x1, x2, yMax,
204         mu1, signal, mu2, sigma2, yOffset);
205     secondFitTerm = gsl_matrix_get(dataMatrix, x1, x2) - EMtwoDGaussian(x1,
206         x2, yMax, mu1, signal, mu2, sigma2, yOffset);
207     runningTotal = runningTotal + firstFitTerm * secondFitTerm / pow(
208         gsl_matrix_get(errorMatrix, x1, x2), 2);
209     }
210     }
211     gsl_vector_set(betaParameters, K, runningTotal);
212     }
213     //Debug output for beta parameters
214     #ifdef DEBUG
215     fprintf(stderr, "Beta parameters:\n");
216     for(K = 0; K < 6; K++)
217     {
218         fprintf(stderr, "%f\n", gsl_vector_get(betaParameters, K));
219     }
220     #endif
221     //Solves the system of linear equations "hessian * change in parameters =
222     beta parameters" to determine the change to the parameters
223     #ifdef DEBUG
224     fprintf(stderr, "Solving linear system of equations.\n"); //Debug output
225     #endif
226     gsl_matrix_memcpy(hessianDuplicate, hessian);
227     if(gsl_linalg_HH_solve(hessianDuplicate, betaParameters, changeInParameters)
228         != 0)
229     {
230         fprintf(stderr, "Error in EMfitTwoDGaussian: The GSL linear system solver
231         failed... This is often because the input data is very, very different
232         from Gaussian, though this may not be the case\n");
233     }
234     return 0;
235     }
236     //Debug output
237     #ifdef DEBUG
238     fprintf(stderr, "Change in parameters:\n");
239     for(K = 0; K < 6; K++)
240     {
241         fprintf(stderr, "%f\n", gsl_vector_get(changeInParameters, K));
242     }
243     #endif
244     //Calculates the old chi-squared value (before the change to the values)
245     #ifdef DEBUG
246     fprintf(stderr, "Calculating old chi-squared\n");
247     #endif
248     runningTotal = 0;
249     for(x1 = 0; x1 < dataMatrix->size1; x1++)
250     {
251         for(x2 = 0; x2 < dataMatrix->size2; x2++)
252         {

```

```

248     runningTotal = runningTotal + pow((gsl_matrix_get(dataMatrix,x1,x2) -
      EMTwoDGaussian(x1, x2, yMax, mu1, sigma1, mu2, sigma2, yOffset)) /
      gsl_matrix_get(errorMatrix ,x1 ,x2) ,2);
249   }
250 }
251 oldChiSquared = runningTotal;
252
253 //Calculates the new chi-squared value
254 #ifdef DEBUG
255 fprintf(stderr , " Calculating new chi-squared\n"); //Debug output
256 #endif
257 runningTotal = 0;
258 for(x1 = 0; x1 < dataMatrix->size1; x1++)
259 {
260     for(x2 = 0; x2 < dataMatrix->size2; x2++)
261     {
262         runningTotal = runningTotal + pow((gsl_matrix_get(dataMatrix,x1,x2) -
          EMTwoDGaussian(x1, x2, yMax + gsl_vector_get(changeInParameters,0) ,
          mu1 + gsl_vector_get(changeInParameters,1) , sigma1 + gsl_vector_get(
          changeInParameters,2) , mu2 + gsl_vector_get(changeInParameters,3) ,
          sigma2 + gsl_vector_get(changeInParameters,4) , yOffset +
          gsl_vector_get(changeInParameters,5))) / gsl_matrix_get(errorMatrix ,x1
          ,x2) ,2);
263     }
264 }
265 newChiSquared = runningTotal;
266
267 //Compares chi-squareds , updates lambda as necessary
268 #ifdef DEBUG
269 fprintf(stderr , " Old Chi-squared: %lf\n" , oldChiSquared); //Debug output
270 fprintf(stderr , " New Chi-squared: %lf\n" , newChiSquared); //Debug output
271 #endif
272 if(newChiSquared > oldChiSquared)
273 {
274     #ifdef DEBUG
275     fprintf(stderr , " Increasing lambda\n");
276     #endif
277     lambda = lambda * 10;
278 }
279 else
280 {
281     #ifdef DEBUG
282     fprintf(stderr , " Decreasing lambda\n");
283     #endif
284     lambda = lambda / 10;
285     yMax = yMax + gsl_vector_get(changeInParameters,0);
286     mu1 = mu1 + gsl_vector_get(changeInParameters,1);
287     sigma1 = sigma1 + gsl_vector_get(changeInParameters,2);
288     mu2 = mu2 + gsl_vector_get(changeInParameters,3);
289     sigma2 = sigma2 + gsl_vector_get(changeInParameters,4);
290     yOffset = yOffset + gsl_vector_get(changeInParameters,5);
291     //Check if iteration should stop
292     if(abs(newChiSquared - oldChiSquared) < fitTolerance)
293     {
294         finished = 1;
295     }

```

```

296     }
297     #ifdef DEBUG
298     fprintf(stderr, "Lambda: %lf\n", lambda);
299     #endif
300     counter = counter + 1;
301 }
302
303 fprintf(stderr, "Finished Fitting\n");
304 //Sets the output variables to their calculated values
305 fitParameters->yMax = yMax;
306 fitParameters->mul = mul;
307 fitParameters->sigma1 = sigma1;
308 fitParameters->mu2 = mu2;
309 fitParameters->sigma2 = sigma2;
310 fitParameters->yOffset = yOffset;
311 fitParameters->chiSquared = newChiSquared;
312 fitParameters->reducedChiSquared = newChiSquared / (dataMatrix->size1 *
    dataMatrix->size2 - 6);
313
314 //Sets uncalculated output variables to -1
315 fitParameters->yMaxError = -1;
316 fitParameters->mulError = -1;
317 fitParameters->sigma1Error = -1;
318 fitParameters->mu2Error = -1;
319 fitParameters->sigma2Error = -1;
320 fitParameters->yOffsetError = -1;
321
322
323 //Cleans up variables
324 gsl_matrix_free(hessian);
325 gsl_matrix_free(hessianDuplicate);
326 gsl_vector_free(betaParameters);
327 gsl_vector_free(changeInParameters);
328
329 return 1;
330 }
331
332 //Estimates the parameters of a 2D gaussian function using center-of-mass for
    the two means, the minimum for yOffset, the value at the center-of-mass
    minus
333 //yOffset for yMax, and the position with the value closest to yMax / e +
    yOffset. Loads these values into the 6 variables passed to it. Returns 1
    on success.
334 //Currently has no failure conditions.
335 int EMestimateTwoDGaussianParameters(gsl_matrix * dataMatrix,
    EMtwoDGaussianFitParameters * fitParameters)
336 {
337     //Loop variables used
338     int x1;
339     int x2;
340     //Used as an initial guess for yMax to help crop tail data.
341     double initialYMax;
342     initialYMax = gsl_matrix_max(dataMatrix);
343     //Temporary variables used in calculating the means.
344     double runningTotal;
345     double weightedRunningTotal;

```

```

346 double currentSum;
347 //Rounded values of mu1 and mu2 used to calculate yMax and sigma
348 int roundedMu1;
349 int roundedMu2;
350 //Temporary vectors used in calculating the sigmas
351 gsl_vector * rowDifferences = gsl_vector_calloc(dataMatrix->size1);
352 gsl_vector * columnDifferences = gsl_vector_calloc(dataMatrix->size2);
353 //Temporary values for storing estimates
354 double yMax;
355 double mu1;
356 double sigma1;
357 double mu2;
358 double sigma2;
359 double yOffset;
360
361 //Sets yOffset by taking the minimum of the data set.
362 yOffset = gsl_matrix_min(dataMatrix);
363 #ifdef _DEBUG
364 fprintf(stderr, "Mu1\n");
365 #endif
366 //Calculates a guess for mu1
367 runningTotal = 0;
368 weightedRunningTotal = 0;
369 for(x1 = 0; x1 < dataMatrix->size1; x1++)
370 {
371     currentSum = 0;
372     //Calculates the sums of each row of the matrix
373     for(x2 = 0; x2 < dataMatrix->size2; x2++)
374     {
375         //Ignores the value if it is less than 1% of the max. This helps crop the
           image so that large tails don't throw off the center-of-mass
           calculation for non-centered peaks
376         if(gsl_matrix_get(dataMatrix, x1, x2) - yOffset > 0.1 * (initialYMax -
           yOffset))
377         {
378             currentSum = currentSum + gsl_matrix_get(dataMatrix, x1, x2);
379         }
380     }
381     runningTotal = runningTotal + currentSum;
382     weightedRunningTotal = weightedRunningTotal + x1 * currentSum;
383 }
384 mu1 = weightedRunningTotal / runningTotal;
385
386 #ifdef _DEBUG
387 fprintf(stderr, "Mu2\n");
388 #endif
389 //Calculates a guess for mu2
390 runningTotal = 0;
391 weightedRunningTotal = 0;
392 for(x2 = 0; x2 < dataMatrix->size2; x2++)
393 {
394     currentSum = 0;
395     //Calculates the sums of each row of the matrix
396     for(x1 = 0; x1 < dataMatrix->size1; x1++)
397     {

```

```

398 //Ignores the value if it is less than 1% of the max. This helps crop the
      image so that large tails don't throw off the center-of-mass
      calculation for non-centered peaks
399 if(gsl_matrix_get(dataMatrix, x1, x2) - yOffset > 0.1 * (initialYMax -
      yOffset))
400 {
401     currentSum = currentSum + gsl_matrix_get(dataMatrix, x1, x2);
402 }
403 }
404 runningTotal = runningTotal + currentSum;
405 weightedRunningTotal = weightedRunningTotal + x2 * currentSum;
406 }
407 mu2 = weightedRunningTotal / runningTotal;
408
409 #ifdef DEBUG
410 fprintf(stderr, "Round\n");
411 #endif
412 //Calculates rounded values of mu1 and mu2
413 if(fabs(ceil(mu1) - mu1) > fabs(floor(mu1) - mu1))
414 {
415     roundedMu1 = floor(mu1);
416 }
417 else
418 {
419     roundedMu1 = ceil(mu1);
420 }
421 if(fabs(ceil(mu2) - mu2) > fabs(floor(mu2) - mu2))
422 {
423     roundedMu2 = floor(mu2);
424 }
425 else
426 {
427     roundedMu2 = ceil(mu2);
428 }
429
430 //Correct for accidentally rounding off the side of the matrix
431 if(roundedMu1 < 0)
432 {
433     roundedMu1 = 0;
434 }
435 if(roundedMu1 > dataMatrix->size1 - 1)
436 {
437     roundedMu1 = dataMatrix->size1 - 1;
438 }
439 if(roundedMu2 < 0)
440 {
441     roundedMu2 = 0;
442 }
443 if(roundedMu2 > dataMatrix->size2 - 1)
444 {
445     roundedMu2 = dataMatrix->size2 - 1;
446 }
447
448 //Sets yMax using the value at (roundedMu1, roundedMu2)
449 yMax = gsl_matrix_get(dataMatrix, roundedMu1, roundedMu2) - yOffset;
450

```

```

451 #ifdef DEBUG
452 fprintf(stderr, "Sigma1\n");
453 #endif
454 //Determines sigma1 by finding the point along the line ( *, roundedMu2)
      which has a value closest to yMaxGuess / e + yOffsetGuess
455 //Finds the difference for each point
456 for(x1 = 0; x1 < rowDifferences->size; x1++)
457 {
458     gsl_vector_set(rowDifferences, x1, fabs(gsl_matrix_get(dataMatrix, x1,
      roundedMu2) - yOffset - yMax * exp(-1)));
459 }
460 sigma1 = fabs(gsl_vector_min_index(rowDifferences) - mu1);
461
462 #ifdef DEBUG
463 fprintf(stderr, "Sigma2\n");
464 #endif
465 //Determines sigma2 in the same way
466 //Finds the difference for each point
467 for(x2 = 0; x2 < columnDifferences->size; x2++)
468 {
469     gsl_vector_set(columnDifferences, x2, fabs(gsl_matrix_get(dataMatrix,
      roundedMu1, x2) - yOffset - yMax * exp(-1)));
470 }
471 sigma2 = fabs(gsl_vector_min_index(columnDifferences) - mu2);
472
473 //Loads calculated values into struct
474 fitParameters->yMax = yMax;
475 fitParameters->mu1 = mu1;
476 fitParameters->sigma1 = sigma1;
477 fitParameters->mu2 = mu2;
478 fitParameters->sigma2 = sigma2;
479 fitParameters->yOffset = yOffset;
480
481 //Loads uncalculated values with -1
482 fitParameters->yMaxError = -1;
483 fitParameters->mu1Error = -1;
484 fitParameters->sigma1Error = -1;
485 fitParameters->mu2Error = -1;
486 fitParameters->sigma2Error = -1;
487 fitParameters->yOffsetError = -1;
488 fitParameters->chiSquared = -1;
489 fitParameters->reducedChiSquared = -1;
490
491 //Cleans up temporary variables
492 gsl_vector_free(rowDifferences);
493 gsl_vector_free(columnDifferences);
494
495 return 1;
496 }
497
498 //Takes parameters and inputs to the 2D gaussian and outputs either the result
      or the result from some derivative of the gaussian
499 double EMtwoDGaussian(double x1, double x2, double yMax, double mu1, double
      sigma1, double mu2, double sigma2, double yOffset)
500 {

```

```

501  return yMax * exp(-pow((x1 - mu1) / sigma1, 2) - pow((x2 - mu2) / sigma2, 2))
      + yOffset;
502 }
503
504 double EMtwoDGaussianDyMax(double x1, double x2, double yMax, double mu1,
      double sigma1, double mu2, double sigma2, double yOffset)
505 {
506  return exp(-pow((x1 - mu1) / sigma1, 2) - pow((x2 - mu2) / sigma2, 2));
507 }
508
509 double EMtwoDGaussianDyOffset(double x1, double x2, double yMax, double mu1,
      double sigma1, double mu2, double sigma2, double yOffset)
510 {
511  return 1;
512 }
513
514 double EMtwoDGaussianDmu1(double x1, double x2, double yMax, double mu1,
      double sigma1, double mu2, double sigma2, double yOffset)
515 {
516  return yMax * exp(-pow((x1 - mu1) / sigma1, 2) - pow((x2 - mu2) / sigma2, 2))
      * 2 * (x1 - mu1) / pow(sigma1, 2);
517 }
518
519 double EMtwoDGaussianDmu2(double x1, double x2, double yMax, double mu1,
      double sigma1, double mu2, double sigma2, double yOffset)
520 {
521  return yMax * exp(-pow((x1 - mu1) / sigma1, 2) - pow((x2 - mu2) / sigma2, 2))
      * 2 * (x2 - mu2) / pow(sigma2, 2);
522 }
523
524 double EMtwoDGaussianDsigma2(double x1, double x2, double yMax, double mu1,
      double sigma1, double mu2, double sigma2, double yOffset)
525 {
526  return yMax * exp(-pow((x1 - mu1) / sigma1, 2) - pow((x2 - mu2) / sigma2, 2))
      * 2 * pow((x2 - mu2), 2) / pow(sigma2, 3);
527 }
528
529 double EMtwoDGaussianDsigma1(double x1, double x2, double yMax, double mu1,
      double sigma1, double mu2, double sigma2, double yOffset)
530 {
531  return yMax * exp(-pow((x1 - mu1) / sigma1, 2) - pow((x2 - mu2) / sigma2, 2))
      * 2 * pow((x1 - mu1), 2) / pow(sigma1, 3);
532 }
533
534 //Sets all values above the given threshold equal to that threshold and
      estimates the center using a center of mass calculation.
535 //Useful if noise peaks are substantially larger than the gaussian peak.
536 int EMcenterOfMassWithMaxThreshold(gsl_matrix * dataMatrix, double * mu1,
      double * mu2, double threshold)
537 {
538  //Loads data into a new matrix so applying the threshold doesn't damage the
      original data
539  gsl_matrix * newDataMatrix = gsl_matrix_alloc(dataMatrix->size1, dataMatrix->
      size2);
540  gsl_matrix_memcpy(newDataMatrix, dataMatrix);
541  EMtwoDGaussianFitParameters parameterEstimates;

```

```

542
543 int x1;
544 int x2;
545 for(x1 = 0; x1 < newDataMatrix->size1; x1++)
546 {
547     for(x2 = 0; x2 < newDataMatrix->size2; x2++)
548     {
549         if(gsl_matrix_get(newDataMatrix, x1, x2) > threshold)
550         {
551             gsl_matrix_set(newDataMatrix, x1, x2, threshold);
552         }
553     }
554 }
555 if(EMestimateTwoDGaussianParameters(newDataMatrix, &parameterEstimates) == 0)
556 {
557     fprintf(stderr, "Error in EMcenterOfMassWithMaxThreshold:
558         EMestimateTwoDGaussianParameters failed.\n");
559     return 0;
560 }
561 *mu1 = parameterEstimates.mu1;
562 *mu2 = parameterEstimates.mu2;
563
564 gsl_matrix_free(newDataMatrix);
565
566 return 1;
567 }

```

A.2 Code for Testing the Gaussian Fit on ETMX Images

```

1  %EMtestFitOnEndMirrorError
2  %Author: Eric Mintun
3  %Date: 9/2/08
4  %This script attempts to simulate the defects on that scatter light on the
5  %LIGO end mirrors using 'reciprocal points' (each defect of the form
6  %I * gamma / (r + gamma) from EMgenerateManyReciprocalPoints. For each set
7  %of simulated defects, it generates a set of slightly noisy gaussians, all
8  %with the same parameters, to which it applies the simulated error. It
9  %then averages the set of gaussians, runs a fit and a threshold center of
10 %mass calculation on them, and calculates the error between the actual
11 %gaussian parameters and the guessed ones. It returns the average of the
12 %error for each parameter across the whole set of simulated defects. It
13 %also plots the average of the last set of gaussians averaged. This
14 %script is very, very slow.
15
16 %Change these parameters to change the simulated error and gaussian
17 numGaussians = 10;
18 numRepetitions = 20;
19 imageSize=[275, 225];
20 minI = 8000;
21 maxI = 11000;
22 minMu1 = 125;
23 maxMu1 = 175;
24 minMu2 = 100;
25 maxMu2 = 150;
26 minSigma1 = 35;
27 maxSigma1 = 45;

```

```

28 minSigma2 = 35;
29 maxSigma2 = 45;
30 minOffset = 1000;
31 maxOffset = 4000;
32 noise = 128/sqrt(12);
33 threshold = 13000;
34 minISpikes = 5000;
35 maxISpikes = 35000;
36 sigmaSpikesAvg = 1;
37 sigmaSpikesStd = 0.2;
38 gammaSpikes = 0.35;
39 numSpikes = 1500;
40 maximumSpikeValue = 50000;
41
42 %Initializes necessary vectors and matrices
43 intensityErrorFit = zeros(numRepetitions, 1);
44 mu1ErrorFit = zeros(numRepetitions, 1);
45 mu2ErrorFit = zeros(numRepetitions, 1);
46 sigma1ErrorFit = zeros(numRepetitions, 1);
47 sigma2ErrorFit = zeros(numRepetitions, 1);
48 offsetErrorFit = zeros(numRepetitions, 1);
49 mu1ErrorThreshold = zeros(numRepetitions, 1);
50 mu2ErrorThreshold = zeros(numRepetitions, 1);
51 thresholdMatrix = zeros(imageSize);
52 %Loops through sets of simulated defects
53 for j=1:numRepetitions
54     j
55     gaussians = zeros([imageSize, numGaussians]);
56     %Randomly generate the parameters for the gaussian
57     parameters = [random('unif', minI, maxI), random('unif', minMu1, maxMu1),
58                 random('unif', minSigma1, maxSigma1), random('unif', minMu2, maxMu2),
59                 random('unif', minSigma2, maxSigma2), random('unif', minOffset,
60                 maxOffset), 0];
61     %Generate the defects
62     spikes = EMgenerateManyRandomReciprocals(imageSize, [minISpikes,
63                 maxISpikes], [sigmaSpikesAvg, sigmaSpikesStd], gammaSpikes, numSpikes,
64                 maximumSpikeValue);
65
66     %Generate the set of gaussians and apply the defects
67     for i=1:numGaussians
68         gaussians(:, :, i) = EMsimulateEndMirrorError(Generate2DGaussian(size,
69                 parameters, noise), spikes, parameters(1), parameters(6));
70     end;
71
72     %Average the gaussians, run the fit, then threshold the matrix and run
73     %the center-of-mass calculation.
74     averaged = estimateMeanAndSigma(gaussians);
75     [fit, fitParameters] = TwoDGaussianFit(averaged(:, :, 1), averaged(:, :, 2));
76     for x=1:imageSize(1)
77         for y=1:imageSize(2)
78             if averaged(x,y,1) > threshold;
79                 thresholdMatrix(x,y) = threshold;
80             else
81                 thresholdMatrix(x,y) = averaged(x,y,1);
82             end;
83         end;
84     end;

```

```

78     end;
79     thresholdParameters = estimateParameters(thresholdMatrix);
80
81     %Calculate errors
82     intensityErrorFit(j) = fitParameters(1) - parameters(1);
83     mu1ErrorFit(j) = fitParameters(2) - parameters(2);
84     sigma1ErrorFit(j) = fitParameters(3) - parameters(3);
85     mu2ErrorFit(j) = fitParameters(4) - parameters(4);
86     sigma2ErrorFit(j) = fitParameters(5) - parameters(5);
87     offsetErrorFit(j) = fitParameters(6) - parameters(6);
88     mu1ErrorThreshold = thresholdParameters(2) - parameters(2);
89     mu2ErrorThreshold = thresholdParameters(4) - parameters(4);
90 end
91
92 surf(averaged(:, :, 1), 'CDataMapping', 'scaled', 'EdgeColor', 'none')
93
94 %Average and output errors
95 intensityAvgError = sum(abs(intensityErrorFit))/numRepetitions
96 mu1AvgError = sum(abs(mu1ErrorFit))/numRepetitions
97 sigma1AvgError = sum(abs(sigma1ErrorFit))/numRepetitions
98 mu2AvgError = sum(abs(mu2ErrorFit))/numRepetitions
99 sigma2AvgError = sum(abs(sigma2ErrorFit))/numRepetitions
100 offsetAvgError = sum(abs(offsetErrorFit))/numRepetitions
101 mu1AvgThreshError = sum(abs(mu1ErrorThreshold))/numRepetitions
102 mu2AvgThreshError = sum(abs(mu2ErrorThreshold))/numRepetitions

```



```

1 %EMgenerate2DReciprocalPoint
2 %Author: Eric Mintun
3 %Date: 9/2/08
4 %Generates a 'reciprocal' point of the form
5 % $I * \gamma / (\sqrt{(x - \mu_1)^2 / \sigma_1^2 + (y - \mu_2)^2 / \sigma_2^2} + \gamma) +$ 
6 % offset
7 %Noise with a standard deviation of 'noise' is added to each point.
8
9 function outputMatrix=EMgenerate2DReciprocalPoint(size, parameters, noise)
10
11 outputMatrix = zeros(size);
12
13 for x1Position=1:size(1)
14     for x2Position=1:size(2)
15
16         outputMatrix(x1Position, x2Position) = random('norm', parameters(1) *
17             parameters(6) / (sqrt((x1Position - parameters(2))^2 / parameters(3)^2
18                 + (x2Position - parameters(4))^2 / parameters(5)^2) + parameters(6)
19                 ) + parameters(7), noise);
20
21     end;
22 end;

```



```

1 %EMgenerateManyRandomReciprocals
2 %Author: Eric Mintun
3 %Date: 9/2/08
4 %Calls EMgenerate2DReciprocal multiple times to produce try to simulate the
5 %defects on the end mirror. The positions of the points are uniformly
6 %random over the size of the image, the intensity is uniformly random over
7 %the range givem, while the sigma value is normally distributed about the

```

```

8  %given mean.  Each defect is only calculated out to 12 sigma in each
9  %dimension, both to improve computation time and to prevent the tails of
10 %many defects adding up to make a significant floor.
11
12 function outputMatrix=EMgenerateManyRandomReciprocals(matrixSize ,
    intensityWithError , sigmaWithError , gamma, number , maximum)
13
14 outputMatrix = zeros(matrixSize);
15 sigmaFactorLimit = 12;
16
17 for i=1:number
18     mu1 = random('unif', 1, matrixSize(1));
19     mu2 = random('unif', 1, matrixSize(2));
20     sigma1 = random('norm', sigmaWithError(1), sigmaWithError(2));
21     sigma2 = random('norm', sigmaWithError(1), sigmaWithError(2));
22     intensity = random('unif', intensityWithError(1), intensityWithError(2));
23     offset = 0;
24     currentGaussian = EMgenerate2DReciprocalPoint([2*sigmaFactorLimit*floor(
        sigma1), 2*sigmaFactorLimit*floor(sigma2)], [intensity ,
        sigmaFactorLimit*floor(sigma1), sigma1, sigmaFactorLimit*floor(sigma2)
        , sigma2, gamma, offset], 0);
25     for x=1:matrixSize(1)
26         for y=1:matrixSize(2)
27             if x > floor(mu1) - sigmaFactorLimit*floor(sigma1) && x < floor(
                mu1) + sigmaFactorLimit*floor(sigma1) && y > floor(mu2) -
                sigmaFactorLimit*floor(sigma2) && y < floor(mu2) +
                sigmaFactorLimit*floor(sigma2)
28                 outputMatrix(x,y) = outputMatrix(x,y) + currentGaussian(x - (
                    floor(mu1) - sigmaFactorLimit*floor(sigma1)), y - (floor(
                    mu2) - sigmaFactorLimit*floor(sigma2)));
29                 if outputMatrix(x,y) > maximum
30                     outputMatrix(x,y) = maximum;
31                 end;
32             end;
33         end;
34     end;
35     clear currentGaussian;
36 end;
37
38 end

```

```

1  %Generate2DGaussian
2  %Author: Eric Mintun
3  %Date: 8/28/08
4  %Takes a size vector of two components, a parameter vector of 7 components,
5  %and a scalar noise parameter and generates a 2D Gaussian image. The 7
6  %parameters are [intensityMax, mu1, sigma1, mu2, sigma2, intensityOffset,
7  %angleOfRotation]. The noise variable is the standard deviation of random
8  %gaussian change per point. Set to 0 to remove noise.
9
10 function outputMatrix=Generate2DGaussian(size , parameters , noise)
11
12 outputMatrix = zeros(size);
13
14 for x1Position=1:size(1)
15     for x2Position=1:size(2)

```

```

16     outputMatrix(x1Position , x2Position) = random('norm', parameters(1) *
        exp(-((x1Position - parameters(2))*cos(parameters(7)) + (x2Position
            - parameters(4))*sin(parameters(7)))^2/parameters(3)^2) * exp(-((
            x2Position - parameters(4))*cos(parameters(7)) - (x1Position -
            parameters(2))*sin(parameters(7)))^2/parameters(5)^2) + parameters
            (6), noise);
17     end;
18 end;

1  %EMsimulateEndMirrorError
2  %Author: Eric Mintun
3  %Date: 8/28/08
4  %This small function attempts to simulate the error in the images that look
5  %at scattered light off the test masses. The error pattern is generated
6  %elsewhere and passed as a matrix in 'spikes'. This function just
7  %normalizes the gaussian beam using the intensity and offset parameters,
8  %then scales the noise by the result and adds it to the original gaussian.
9
10 function outputMatrix=EMsimulateEndMirrorError(gaussian , spikes , intensity ,
        offset)
11
12 %Normalize the intensity of the gaussian
13 normGaus = (gaussian - offset)/intensity;
14 matrixSize = size(normGaus);
15
16 %Prevent the normalized gaussian from containing negative numbers.
17 for x=1:matrixSize(1)
18     for y=1:matrixSize(2)
19         if normGaus(x,y) < 0
20             normGaus(x,y) = 0;
21         end;
22     end;
23 end;
24
25 %Scale the noise by the normalized gaussian
26 scaledNoise = spikes.*normGaus;
27
28 %Add the scaled noise to the original gaussian
29 outputMatrix = gaussian + scaledNoise;
30
31 end

```

A.3 Rotation Transform Code

```

1  %EMcorrectImageRotation
2  %Author: Eric Mintun
3  %Date: 8/29/08
4  %Given a set of 4 points in real space and four points on an image, this
5  %function corrects the image for being rotated. The points must be given
6  %'around' the quadrilateral they form (i.e. point3 is opposite point1 and
7  %point2 is opposite point4). Image offset is a 3 component vector which
8  %gives the offset from the plane of the four points to the plane the image
9  %you care about is in. E.g, if the center of the plane of the relevant
10 %object is closer to the camera than the four points, z in imageOffset will
11 %be negative. The initial estimate vector in the nonlinear solve is [pi/6,
12 %pi/6, pi/6, 0.75, 0, 0, 0, 0] in order to assure the rotations have

```

```

13 %reasonable (i.e., between  $-\pi/2$  and  $\pi/2$ ) values. This function is
14 %effectively a combination of EMcenterImagePoints, rotationSolve,
15 %EMtranslateEulerRotation, and unrotateImage for convenience
16
17 function [alpha, beta, gamma, c, center, xPos, yPos]=EMcorrectImageRotation(
    imageData, point1Real, point1Image, point2Real, point2Image, point3Real,
    point3Image, point4Real, point4Image, imageOffset, zoom, focalLength,
    distancePerPixel)
18
19 [realCenter, centered1Real, centered2Real, centered3Real, centered4Real]=
    EMcenterImagePoints(point1Real, point2Real, point3Real, point4Real, 0, 0);
20 [imageCenter, centered1Image, centered2Image, centered3Image, centered4Image]=
    EMcenterImagePoints(point1Image, point2Image, point3Image, point4Image, 0,
    1);
21 points=[centered1Real, 0, centered1Image; centered2Real, 0, centered2Image;
    centered3Real, 0, centered3Image; centered4Real, 0, centered4Image];
22 rotation=rotationSolve(points, [pi/6, pi/6, pi/6, 0.75, 0, 0, 0, 0]);
23 [alpha, beta, gamma, c]=EMtranslateEulerRotation(rotation(1), rotation(2),
    rotation(3), rotation(4), imageOffset, zoom, focalLength, distancePerPixel
    );
24 center(1) = imageCenter(1) + imageOffset(1);
25 center(2) = imageCenter(2) + imageOffset(2);
26 figure;
27 subplot(2,1,1);
28 image(imageData, 'CDataMapping', 'scaled');
29 axis image;
30 colormap gray
31 subplot(2,1,2);
32 [xPos, yPos]=unrotateImage(imageData, alpha, beta, gamma, c, 0, center);
33 view(2);
34 axis image;
35 colormap gray

```

```

1 %EMcenterImagePoints
2 %Author: Eric Mintun
3 %Date: 8/28/08
4 %Takes four points in a plane and calculated their center by calculating
5 %their diagonals and solving for the intersection. It then outputs the
6 %center, along with the four points minus the center. xFlip and yFlip are
7 %booleans that cause the points to be mirrored around the new axes. The
8 %points must be given going around the quadrilateral (i.e. point1 is
9 %opposite point3, and point2 is opposite point4).
10
11 function [center, centered1, centered2, centered3, centered4]=
    EMcenterImagePoints(point1, point2, point3, point4, xFlip, yFlip)
12
13 %Calculate the center
14 m1 = (point3(2)-point1(2))/(point3(1)-point1(1));
15 b1 = point1(2) - m1 * point1(1);
16 m2 = (point4(2)-point2(2))/(point4(1)-point2(1));
17 b2 = point2(2) - m2 * point2(1);
18 center(1) = (b2 - b1)/(m1 - m2);
19 center(2) = m1 * center(1) + b1;
20
21 %Center the points
22 centered1 = point1 - center;

```

```

23 centered2 = point2 - center;
24 centered3 = point3 - center;
25 centered4 = point4 - center;
26
27 %Mirror if necessary
28 if xFlip == 1
29     centered1(1) = -centered1(1);
30     centered2(1) = -centered2(1);
31     centered3(1) = -centered3(1);
32     centered4(1) = -centered4(1);
33 end;
34 if yFlip == 1
35     centered1(2) = -centered1(2);
36     centered2(2) = -centered2(2);
37     centered3(2) = -centered3(2);
38     centered4(2) = -centered4(2);
39 end

1 %unrotateImage
2 %Author: Eric Mintun
3 %Date: 8/28/08
4 %Takes an image and corrects for a 3D rotation, given that the original
5 %image is assumed to be on a flat plane with a z-value z. Alpha, beta, and
6 %gamma are Euler angles (of the form zxz), while c is a scaling constant.
7 %The center of the rotation is taken to be at center. This function
8 %produces plane at a strange angle in 3D space with the image on it (it
9 %wouldn't show up without the angle. Only view from above.
10
11 %Setting the z-value to anything other than 0 doesn't seem to work very
12 %well.
13
14 function [xPos, yPos]=unrotateImage(imageData, alpha, beta, gamma, c, z,
    center)
15
16 matrixIndex = zeros(size(imageData, 1) * size(imageData, 2), 3);
17
18 %Coverts to a list of 3D points
19 for i=1:size(imageData,1)
20     for j=1:size(imageData,2)
21         matrixIndex((i-1) * size(imageData,2) + j,2) = -(i - center(2));
22         matrixIndex((i-1) * size(imageData,2) + j,1) = (j - center(1));
23     end;
24 end;
25
26 matrixIndex(:,3) = z;
27
28 %The rotation matrix
29 R1 = [cos(alpha),-sin(alpha),0; sin(alpha), cos(alpha), 0; 0,0,1];
30 R2 = [1,0,0;0,cos(beta), -sin(beta); 0, sin(beta), cos(beta)];
31 R3 = [cos(gamma), -sin(gamma), 0; sin(gamma), cos(gamma), 0; 0,0,1];
32 R = R1 * R2 * R3;
33
34 %Modifies the rotation matrix since x, y, z' are unknown while x', y', and
35 %z are known.
36 modified = zeros(3);
37

```

```

38 modified(1,1) = (R(1,1) - R(3,1)*R(1,3)/R(3,3))/c;
39 modified(2,1) = (R(2,1) - R(3,1)*R(2,3)/R(3,3))/c;
40 modified(3,1) = R(3,1)/R(3,3);
41 modified(1,2) = (R(1,2) - R(3,2)*R(1,3)/R(3,3))/c;
42 modified(2,2) = (R(2,2) - R(3,2)*R(2,3)/R(3,3))/c;
43 modified(3,2) = R(3,2)/R(3,3);
44 modified(1,3) = -R(1,3)/R(3,3);
45 modified(2,3) = -R(2,3)/R(3,3);
46 modified(3,3) = c/R(3,3);
47
48 %Solves for the unknowns.
49 unrotated = matrixIndex * modified^-1;
50
51 %Loads data into a form readable by surf.
52 xPos = zeros(size(imageData));
53 yPos = zeros(size(imageData));
54 zPos = zeros(size(imageData));
55
56 for i=1:size(imageData,1)
57     for j=1:size(imageData,2)
58         xPos(i,j) = unrotated((i-1) * size(imageData,2) + j,1);
59         yPos(i,j) = unrotated((i-1) * size(imageData,2) + j,2);
60         zPos(i,j) = unrotated((i-1) * size(imageData,2) + j,3);
61     end;
62 end;
63
64
65 %Plots result on an arbitrarily angled plane. View from above to see image.
66 surf(xPos, yPos, xPos, double(imageData), 'CDataMapping','scaled','EdgeColor',
        , 'none')
67 view(2)
68 colormap jet

1  %rotationSolveFunc
2  %Author: Eric Mintun
3  %Date: 8/28/08
4  %Takes a set of points along with a vector of initial guesses for
5  %parameters and attempts to use them to determine the rotation parameters.
6  %'points' consists of a n-by-5 matrix, where each row contains the
7  %data for a single point. The first 3 columns are the (x,y,z) coordinates
8  %of the actual position of the point. The second two columns are the (x,y)
9  %coordinates of the apparent position of the point. The 'params' vector
10 %contains the initial guesses for the parameters of the function. There
11 %are 4 for the rotation and 1 for each point given (the apparent z
12 %position). Thus, this vector must be 4 components longer than the number
13 %of rows in 'points'. The function outputs a vector which contains alpha,
14 %beta, gamma, and c of the rotation, followed by the apparent z positions.
15
16 %Setting the z-offset value of the actual point positions to anything other
17 %than 0 doesn't seem to work very well.
18
19 function y=rotationSolve(points, params)
20
21 if size(points, 1) ~= max(size(params)) - 4
22     error('The number of parameters needs to be four greater than the number
        of columns of points')

```

```

23 end;
24
25 options = optimset('Display', 'iter');
26 y=fsolve(@rotationSolveFunc,params, options);
27
28     function minimizeVector=rotationSolveFunc(params)
29         R1 = [cos(params(1)),-sin(params(1)),0; sin(params(1)), cos(params(1))
30             , 0; 0,0,1];
31         R2 = [1,0,0;0,cos(params(2)), -sin(params(2)); 0, sin(params(2)), cos(
32             params(2))];
33         R3 = [cos(params(3)), -sin(params(3)), 0; sin(params(3)), cos(params
34             (3)), 0; 0,0,1];
35         R = R1 * R2 * R3;
36
37         original = zeros(size(points, 1), 3);
38         modified = zeros(size(points, 1), 3);
39         for i=1:size(points,1)
40             original(i,1) = points(i,1);
41             original(i,2) = points(i,2);
42             original(i,3) = points(i,3);
43             modified(i,1) = points(i,4);
44             modified(i,2) = points(i,5);
45             modified(i,3) = params(i + 4);
46         end;
47         minimize = original * R - params(4) * modified;
48
49         minimizeVector = zeros(3*size(minimize,1),1);
50         for i=1:size(minimize,1)
51             minimizeVector(3*i-2) = minimize(i,1);
52             minimizeVector(3*i-1) = minimize(i,2);
53             minimizeVector(3*i) = minimize(i,3);
54         end;
55     end
56 end

```

```

1  %EMcorrectForZOffset
2  %Author: Eric Mintun
3  %Date: 8/29/08
4  %For a set of Euler rotation angles (and scaling factor) around the origin,
5  %this function determines what set of angles are required to move the
6  %camera to the same point, given a different center.
7
8  function [newAlpha, newBeta, newGamma, newC] = EMtranslateEulerRotation(alpha,
9      beta, gamma, c, newCenter, zoom, focalLength, distancePerPixel)
10 y = c / (distancePerPixel/(zoom * focalLength) * sqrt((1+1/(tan(beta))^2)*(1+(
11     tan(alpha))^2)));
12 x = tan(alpha)*y;
13 z = sqrt(x^2 + y^2)/tan(beta);
14 newX = x - newCenter(1);
15 newY = y - newCenter(2);
16 newZ = z - newCenter(3);
17
18 newAlpha = atan(newX/newY);
19 newBeta = atan(sqrt(newX^2+newY^2)/newZ);

```

```

20 newGamma = gamma - (newAlpha - alpha);
21 newC = distancePerPixel / (zoom * focalLength) * sqrt(newX^2 + newY^2 + newZ
    ^2);

1  %EMcenterImagePoints
2  %Author: Eric Mintun
3  %Date: 8/28/08
4  %Takes four points in a plane and calculated their center by calculating
5  %their diagonals and solving for the intersection. It then outputs the
6  %center, along with the four points minus the center. xFlip and yFlip are
7  %booleans that cause the points to be mirrored around the new axes. The
8  %points must be given going around the quadrilateral (i.e. point1 is
9  %opposite point3, and point2 is opposite point4).
10
11 function [center, centered1, centered2, centered3, centered4]=
    EMcenterImagePoints(point1, point2, point3, point4, xFlip, yFlip)
12
13 %Calculate the center
14 m1 = (point3(2)-point1(2))/(point3(1)-point1(1));
15 b1 = point1(2) - m1 * point1(1);
16 m2 = (point4(2)-point2(2))/(point4(1)-point2(1));
17 b2 = point2(2) - m2 * point2(1);
18 center(1) = (b2 - b1)/(m1 - m2);
19 center(2) = m1 * center(1) + b1;
20
21 %Center the points
22 centered1 = point1 - center;
23 centered2 = point2 - center;
24 centered3 = point3 - center;
25 centered4 = point4 - center;
26
27 %Mirror if necessary
28 if xFlip == 1
29     centered1(1) = -centered1(1);
30     centered2(1) = -centered2(1);
31     centered3(1) = -centered3(1);
32     centered4(1) = -centered4(1);
33 end;
34 if yFlip == 1
35     centered1(2) = -centered1(2);
36     centered2(2) = -centered2(2);
37     centered3(2) = -centered3(2);
38     centered4(2) = -centered4(2);
39 end

```

References

- [1] A. Weinstein. *Lectures on the Physics of LIGO*. June 2008.
- [2] Prosilica Inc. *User Manual: GC650/GC650C* and *User Manual: GC750/GC750C*. Available from www.prosilica.com
- [3] D. Anderson. *Alignment of Resonant Optical Cavities*. Applied Optics, Vol. 23, Issue 17, pp. 2944-2949. September 1st, 1984.

- [4] A. Siegman. *Lasers*. University Science Books, 1986.
- [5] R. Adhikari, J. Betzwieser, A. Brooks, E. Mintun. *Gigabit Digital Cameras*. Currently unpublished.
- [6] H. Kogelnik, T. Li. *Laser Beams and Resonators*. Applied Optics, Vol. 5, Issue 10, pp. 1550-1567. October 1966.
- [7] W. Press, B. Flannery, S. Teukolsky, W. Vetterling. *Numerical Recipes in C, Second Edition*. Cambridge University Press, 1995.