High Performance Computing for Adaptive Optics and the Victoria Open Loop Testbed

by

Michael Fischer
Bachelor of Engineering, University of Victoria, 2001

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of

MASTER OF APPLIED SCIENCE

in the department of Mechanical Engineering

Supervisory Committee

High Performance Computing for Adaptive Optics and the Victoria Open Loop Testbed

by

Michael Fischer
Bachelor of Engineering, University of Victoria, 2001

Supervisory Committee

Dr. Colin Bradley (Department of Mechanical Engineering)
Supervisor

Dr. Jean-Pierre Véran (Department of Physics and Astronomy)
Co-Supervisor or Departmental Member

Dr. Peter Wild (Department of Mechanical Engineering)
Departmental Member

Dr. Kim Venn (Department of Physics and Astronomy)
Outside Member

Abstract

Supervisory Committee
Dr. Colin Bradley (Department of Mechanical Engineering)
Supervisor
Dr. Jean-Pierre Véran (Department of Physics and Astronomy)
Co-Supervisor or Departmental Member
Dr. Peter Wild (Department of Mechanical Engineering)
Departmental Member
Dr. Kim Venn (Department of Physics and Astronomy)
Outside Member

This thesis addresses high performance computing in Adaptive Optics (AO) simulation and the development and demonstration of a prototype AO instrument for future Extremely Large Telescopes (ELTs). Adaptive Optics systems are used on astronomical telescopes for correcting the blurring effects of atmospheric turbulence on incoming starlight, improving image quality to that of the diffraction limit of the telescope. Extremely Large Telescopes will have primary mirror diameters in the 20 - 40 m range, driving the need for technology development in two key areas, among others: 1) adaptive optics simulation, and 2) wide field adaptive optics (WFAO).

The Linear Adaptive Optics Simulator (LAOS) is at the forefront of adaptive optics simulation, opening up the capability to simulate ELTs with integrated AO systems on a single computer. This is computationally expensive and time consuming, and thus simulator performance is very important and can determine the feasibility of simulating such systems at all. Efforts were made to improve the existing LAOS performance and bring a larger range of problem sizes and AO instrument concepts including WFAO into the realm of possibility.

WFAO will take advantage of the larger light collection and spatial resolution capabilities of ELTs. One WFAO instrument approach that addresses this is Multi-Object Adaptive Optics (MOAO), which will provide localized correction around a number (5 - 40) of selected science objects spread around the field of view, enabling extragalactic studies otherwise very costly to implement with other WFAO techniques. However, there are several risks that need to be retired. Many elements of an MOAO system, such as the use

of atmospheric tomography, MEMS mirrors, and woofer-tweeter control have all been demonstrated to work in different lab settings and are included in advanced instrument concepts. Open loop control, however, is perhaps the greatest risk to MOAO, introducing unique requirements on the AO system. The Victoria Open Loop Testbed (VOLT) serves as a demonstration of open loop control – both on-sky at the Dominion Astrophysical Observatory's 1.2 m telescope and in the lab – to facilitate the future development of MOAO. Our goal was to demonstrate open loop control with a simple on-axis natural guide star testbed.

Table of Contents

List of Equations

List of Figures

List of Tables

List of Acronyms

ADC	Atmospheric Dispersion Corrector
ALTAIR	ALTitude-conjugate Adaptive optics for the InfraRed
AO	Adaptive Optics
CCD	Charge Coupled Device
CMOS	Complementary Metal Oxide Semiconductor
CAOS	Code for Adaptive Optics Systems
CPU	Central Processing Unit
DAC	Digital to Analog Converter
DAO	Dominion Astrophysical Observatory
DM	Deformable Mirror
DMA	Direct Memory Access
DN	Data Number
ELT	Extremely Large Telescope
FALCON	Fiber optics spectrograph with Adaptive optics on Large fields to Correct at Optical and Near-infrared
FOV	Field Of View
FPGA	Field Programmable Gate Array
FWHM	Full Width at Half Maximum
HIA	Herzberg Institute of Astrophysics
IFU	Integral Field Unit
IRMOS	Near-Infrared Multi-Object Spectrograph
IRQ	Interrupt ReQuest
LAOG	Laboratoire Astrophysique de l'Observatoire de Grenoble
LAOS	Linear Adaptive Optics Simulator
LED	Light Emitting Diode
LGS	Laser Guide Star
MAD	Multi-conjugate Adaptive optics Demonstrator
MCAO	Multi-Conjugate Adaptive Optics
MEMS	Micro-ElectroMechanical System
MOAO	Multi-Object Adaptive Optics
MOS	Multi-Object Spectrograph
NCP	Non-Common Path
NFIRAOS	Narrow-Field Infrared Adaptive Optics System
NGS	Natural Guide Star
PAOLA	Performance of Adaptive Optics for Large Apertures
PCI	Parallel Communication Interface
PS	Phase Screen
PSD	Power Spectral Density
PSF	Point Spread Function
QE	Quantum Efficiency
RAM	Random Access Memory
RMS	Root Mean Square

ROI	Region Of Interest
RTAI	Real Time Application Interface
RTC	Real-Time Computer
RTDSC	Real Time Stamp Counter
RTF	Rejection Transfer Function
SCSI	Small Computer System Interface
SMP	Symmetric MultiProcessor
SNR	Signal-to-Noise Ratio
TMT	Thirty Meter Telescope
TTM	Tip-Tilt Mirror
UML	Unified Modeling Language
VOLT	Victoria Open Loop Testbed
WFAO	Wide Field Adaptive Optics
WFE	WaveFront Error
WFS	WaveFront Sensor

Acknowledgments

I would like to thank the many people who provided me with support and encouragement – in my work and in my personal life – throughout the time I spent working toward my Master of Applied Science. It was a time of great gains in knowledge and experience, but also a time of great loss for me with the passing of my mother, Rita Irene Fischer, on December 27, 2007. I will not forget the compassion, patience, and understanding that was shown toward me by my colleagues, friends, and family during those difficult times, and I feel very fortunate to have shared these past three years in their company. In particular I would like to thank Dr. David Andersen for his excellent guidance and partnership throughout the development of the Victoria Open Loop Testbed. I thoroughly enjoyed the time we spent working toward this goal together. I would like to thank Dr. Jean-Pierre Véran for his expert advice and leadership all along the way, and the enthusiastic support he always gave for me and my work. I am indebted to Dr. Colin Bradley for allowing me the chance to pursue my graduate studies with the University of Victoria Adaptive Optics Laboratory, and his very enabling management style. As well, I greatly appreciate the expert assistance and insight of Dr. Rodolphe Conan on all areas related to adaptive optics, and the wealth of support that he and the rest of the UVic AO team, particularly Aaron Hilton, gave me in solving the many technical problems I encountered. I am also very appreciative for the help I received from Malcolm Smith on countless programming issues. Finally, I would like to thank the many others in the Astronomy Technology Research Group that helped us with the various challenges of VOLT and everyone else at the Herzberg Institute of Astrophysics – an extremely talented and professional bunch – who made me feel very welcome up on the mountain and made daily work there a pleasure. I feel proud to have been a part of such a first-class institution, and proud of the great recognition HIA brings to my hometown of Victoria.

This is dedicated to my parents, to whom I owe everything

Rita Irene Fischer

Donald Joseph Fischer

Chapter 1
Introduction

Adaptive Optics systems are used on astronomical telescopes for correcting the blurring effects of atmospheric turbulence on incoming starlight, improving image quality to that of the diffraction limit of the telescope. Such systems have been successfully implemented on many current-generation observatories around the world with primary mirrors up to 10 m in diameter. Future telescopes under study, known as Extremely Large Telescopes (ELTs), will move to primary mirror diameters in the 20 - 40 m range. This drives the need for a more advanced and science-application-specific suite of adaptive optics (AO) instruments in order to fully take advantage of the large increase in light collecting ability and spatial resolution ELTs will provide, and to best utilize precious and expensive telescope observing time. This leads to tighter optical tolerances from higher degrees of optical component and subsystem integration, and thus the need for novel optical designs and control schemes. As well, these systems will be driven to a much larger degree of complexity in numerical and control software, resulting in a sharp rise in the demand for real-time computing power.

We will begin in Chapter 1 with a brief introduction to the terminology and theory of atmospheric turbulence as it relates to light propagation through the atmosphere from distant sources. We will then introduce Adaptive Optics (AO) and describe the architecture and operation of a classical AO system, from which all the latest generation of AO systems has evolved. Then we will discuss Wide Field Adaptive Optics (WFAO), the emphasis of next-generation AO systems currently under study. Chapter 2 will describe one of two main areas that formed the bulk of my thesis work: the Linear

Adaptive Optics Simulator (LAOS), an end-to-end telescope and adaptive optics software simulation package, and the performance improvements I made to it. Chapter 3 will describe the design, prototyping, and on-sky testing of the Victoria Open Loop Testbed (VOLT), a technology demonstration aimed at addressing one key issue identified as a major risk factor to one kind of WFAO – open loop control.

1.1 The Goal Of Adaptive Optics: Diffraction Limited Performance

An image of a object far enough away to be considered a point source of light that is formed by an optical system – be it onto photographic film, a detector, or other imaging surface – is described by the object convolved with the point spread function (PSF) of the imaging system. The PSF undergoes diffraction from the optics in the system, an effect inherent to the wave nature of light, producing what is called a *diffraction-limited* image. This is the theoretical 'best possible image' of an optical system, as depicted in Figure 1. The pattern of rings are of a decreasing intensity distribution produced by Fraunhofer diffraction around a circular aperture called an Airy disk, which is proportional to

$$I(r) \propto \left[\frac{J_1(r)}{r}\right]^2 \qquad (1)$$

where $J_1(r)$ is a Bessel function of the first kind. The first dark ring of the Airy Disk is located at an angular distance from the centre of

$$\theta = \frac{1.22\lambda}{D} \qquad (2)$$

where θ is the angle in radians, λ the wavelength, and D the telescope diameter. The Full-Width Half Maximum (FWHM) of a diffraction-limited PSF is 1.02 λ/D. This is the ideal resolution of a telescope.

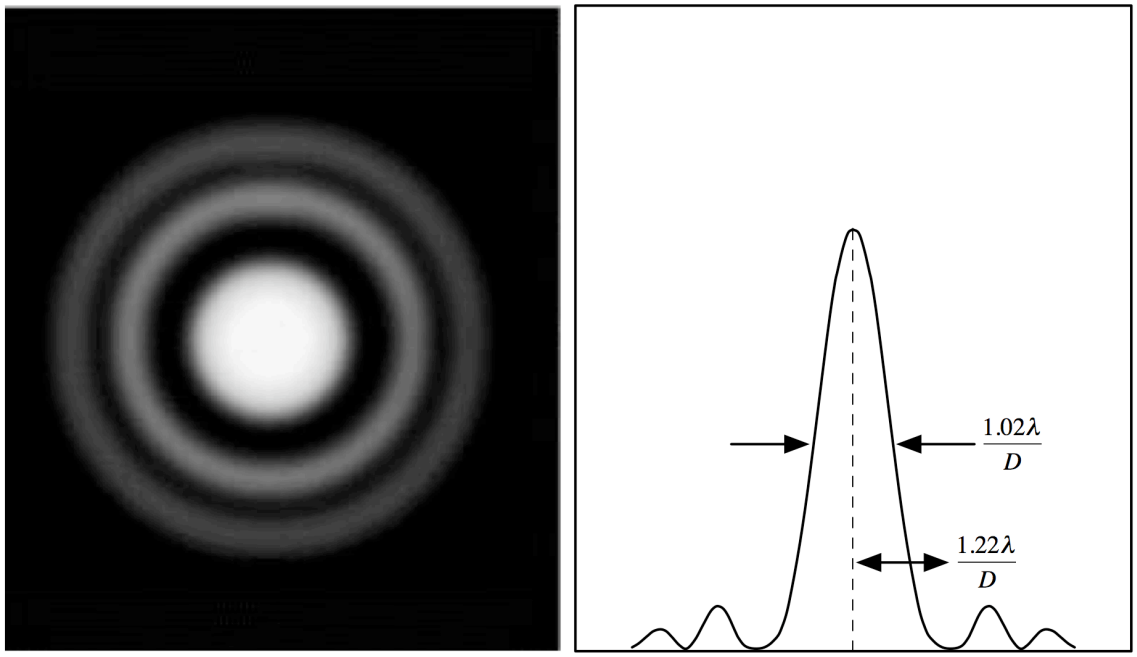


Figure 1: Left: A diffraction-limited image of a point source – a pattern known as an Airy disk – is composed of the intense core and Airy rings which decrease in intensity as one moves radially out from the core. Right: a cross section of a typical diffraction-limited image; the angular width at FWHM is 1.02λ/D [1].

Wavefronts of light emanating spherically from distant light sources are effectively planar by the time they reach us. Upon entering Earth's atmosphere these planar wavefronts are perturbed by turbulence and pockets of air of varying temperatures that cause varying phase distortion across the wavefront, or aberrations. The typical length scale over which the turbulence becomes significant is defined by the Fried parameter, $r_0(\lambda)$[1]. The Fried parameter, r_0, typically has values between 20 cm (at good sites under very good conditions) and 3 cm (under very poor conditions). The seeing-limited FWHM of a PSF is given by λ/r_0, as shown in Figure 2. Thus, there is no spatial

[1] The Fried parameter varies with wavelength by $r_0 \propto \lambda^{6/5}$ under the assumption of that the turbulence is well described by the Kolmogorov model. The convention in the AO community is to quote r_0 at λ=500 nm. See Appendix A.

resolution advantage to be gained from larger telescopes unless the effects of atmospheric turbulence can be compensated.

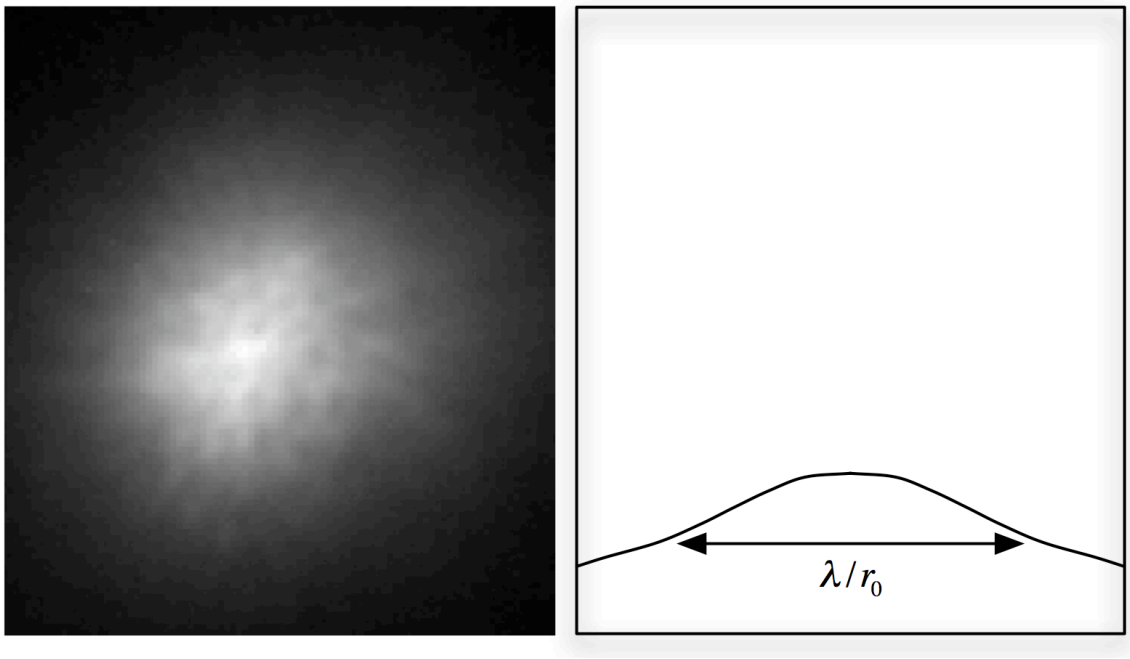


Figure 2: Left: An example of a seeing-limited image with no AO correction. Right: A cross section of a typical seeing-limited PSF; it is a Gaussian with FWHM = λ/r_0 [2].

The goal of AO systems is to recover the diffraction-limited spatial resolution of a telescope by essentially 'pulling' light into the central core that would otherwise be scattered. In general, however, AO will only provide a partial correction of the atmospheric turbulence. A well-corrected AO PSF will have a core with a FWHM roughly equal to λ/D and a broad halo around the core proportional to λ/r_0, as illustrated in Figure 3. One measure of the quality of the AO correction is the *Strehl ratio*. It is defined as the ratio between the peak intensity of an image divided by the peak intensity of a diffraction-limited image with the same total light flux. Strehl ratios on a telescope without AO are typically only a few percent, but can be improved to over 90% with AO (although 20-40% at near-infrared wavelengths is more common). Strehl ratio can be related to RMS wavefront error (WFE) using the Maréchal approximation:

$$S \sim e^{-(2\pi\sigma/\lambda)^2} \quad (3)$$

where S is the Strehl ratio, σ is the RMS WFE, and λ is the wavelength.

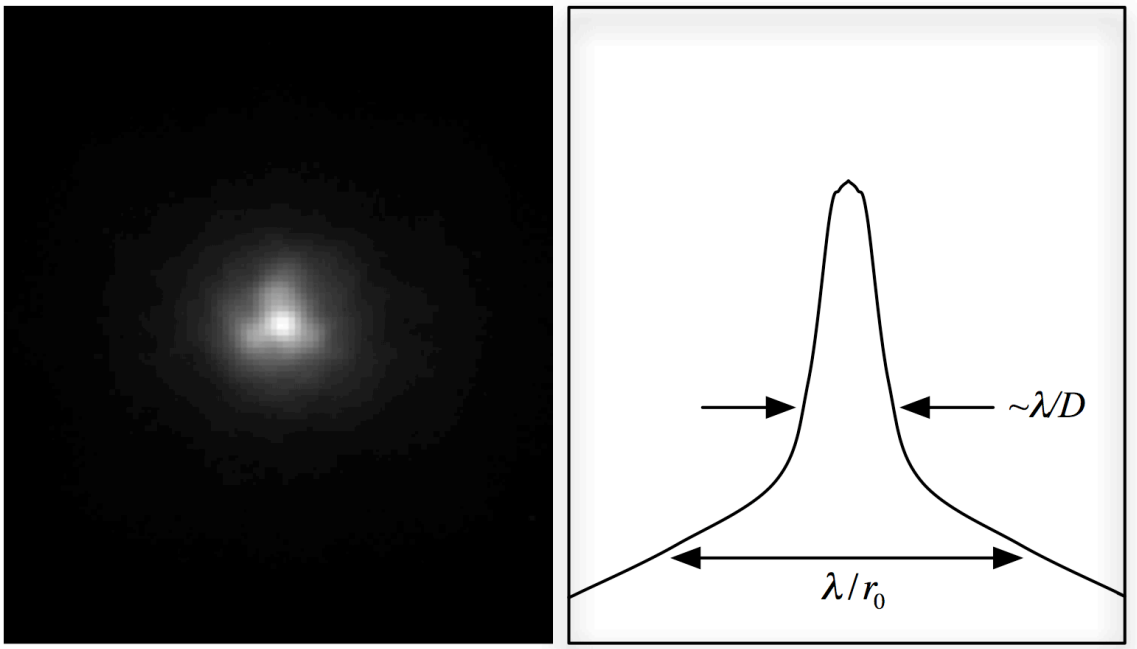


Figure 3: Left: An example image of an AO-corrected image. Right: A generalised cross section of an AO-corrected image PSF. The FWHM in arcseconds of the central core is proportional to λ/D and the halo has a width with a size of roughly λ/r_0 [2].

1.2 Classical Adaptive Optics

It is the ability of an AO system to measure wavefront aberrations and optically compensate for them that allows for the effects of atmospheric turbulence to be corrected. Figure 4 shows a conceptual diagram of an AO system placed optically 'downstream' of the telescope.

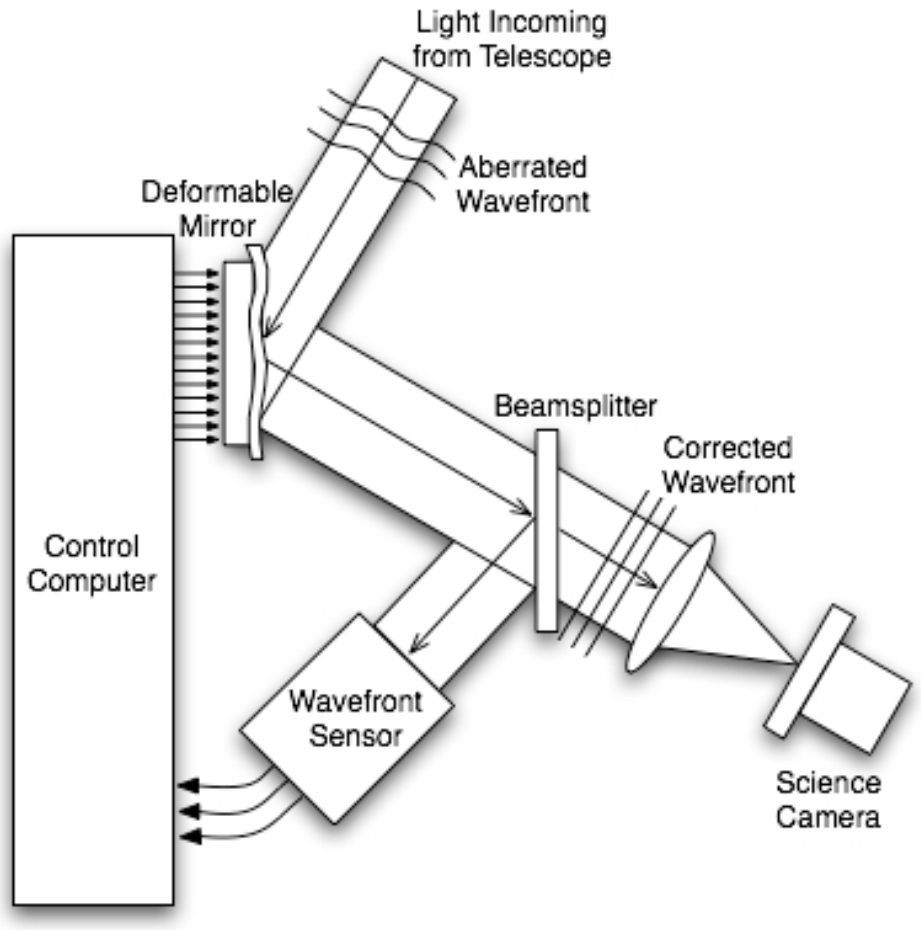


Figure 4: A conceptual diagram of a classical adaptive optics system. The wavefront sensor measures the residual wavefront error after correction by the deformable mirror, which is used by the control computer to generate new commands for the deformable mirror for the next iteration [3].

A wavefront of light that has traveled through the turbulent atmosphere and been collected by the telescope is labeled the *aberrated wavefront*. The light is reflected off a *deformable mirror* (DM), which will be explained momentarily, and hits the *beamsplitter*, transmitting a portion of the light to the science camera and reflecting the rest toward the *wavefront sensor* (WFS). The WFS is responsible for measuring the *aberrated wavefront*. A Shack-Hartmann WFS contains an optic with a grid of lenses called a lenslet array, which splits the wavefront into discrete sampled regions called

subapertures, the light passing through each lenslet focusing onto a unique region of a CCD or other discretized light detector. The displacement of the focal spots within these (usually square) regions, gives the *x* and *y* wavefront slopes (first derivatives) at each of these subapertures, as shown in Figure 5.

The WFS spots are not perfect points of light; ideally they are round spots and the *x* and *y* centre coordinates of the spots are taken as the spot displacements. In reality these spots are never perfectly circular, because of detector noise and optical system effects, thus a calculation of weighted centre of mass with respect to pixel intensity is performed to locate the spot centroids. WFS detector noise error, σ_{WFS}, is one source of residual AO system wavefront errors (WFEs), to be summarized later. The centroid computation is typically done by a fast processing unit, often a digital signal processor, and the resultant slope vectors are then used to generate control signals for the DM, which can modify its shape by moving a grid of tiny actuators below its flexible reflective surface in a manner such as to re-flatten the wavefront upon reflection. A cartoon of this DM wavefront correction is shown in Figure 6.

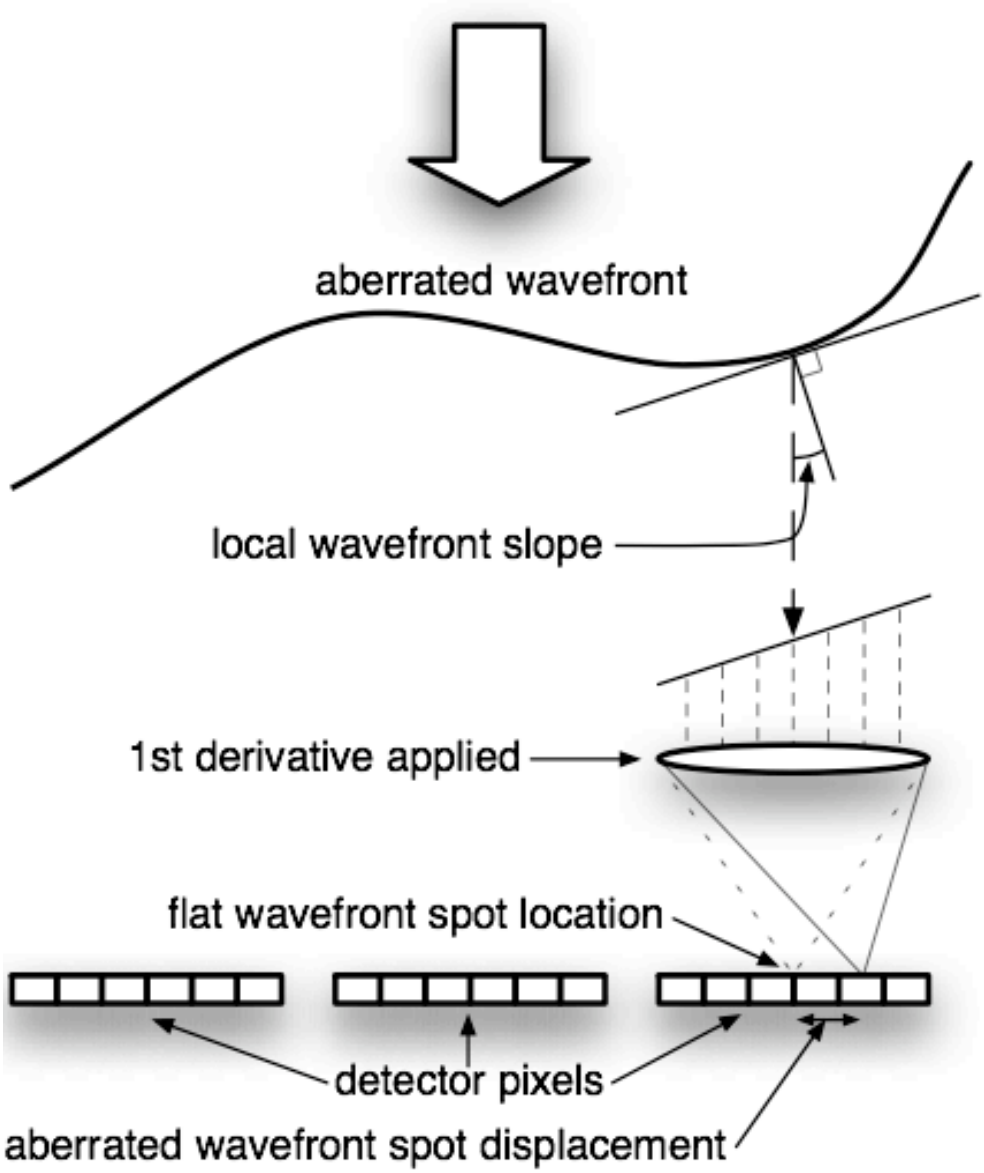


Figure 5: A 2-dimensional wavefront (for illustrative purposes) of light falling on the telescope aperture with phase aberrations from atmospheric turbulence is discretized into subapertures by a lenslet array. Focal spot positions within square pixel regions on the CCD detector are measured and related back to their local wavefront slopes. This configuration is known as a Shack-Hartmann wavefront sensor.

The new DM commands can be generated using a simple integrator control algorithm. Referring to Figure 7, it can be envisioned then how subsequent iterations of this closed control loop will produce well-corrected wavefronts for imaging at the science camera. This sequence of operations is run at a bandwidth high enough such that the change in the

atmospheric turbulence profile is relatively small between iterations, such that the system is able to 'keep up' with the evolving atmosphere within an acceptable margin of error. But since the control system cannot respond *instantaneously* to the atmospheric turbulence effects, there is a *temporal error*, $\sigma_{temporal}$. The error due to the DM not being able to perfectly take on the shape of the measured turbulence is the *fitting error*, $\sigma_{fitting}$.

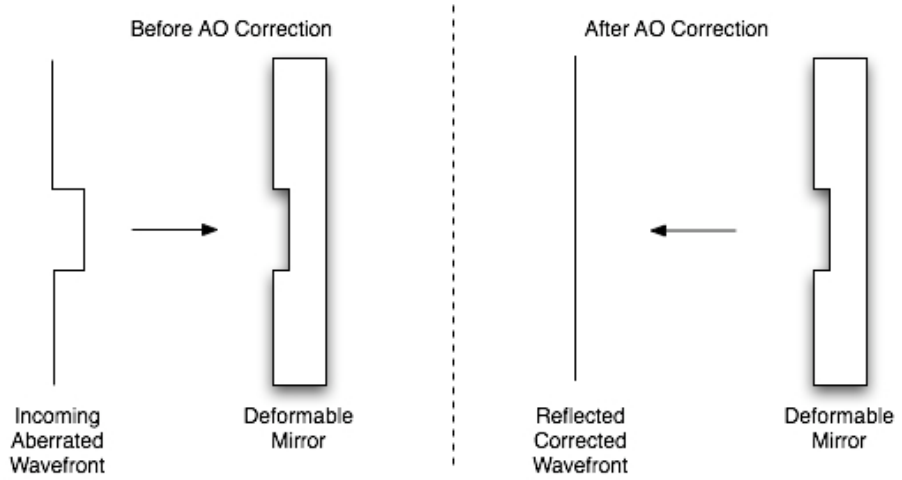


Figure 6: A basic illustration of how phase delay is corrected by a deformable mirror [4].

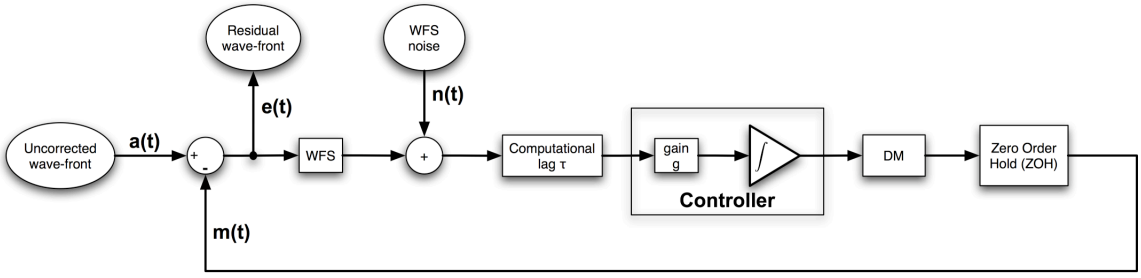


Figure 7: A classical AO closed loop control scheme employing an integrator (J.P. Véran).

In order to do wavefront sensing, either the science target needs to be bright and concentrated enough to provide enough light for both the WFS and the science camera (after reflection and transmission losses at the beamsplitter), or another light source within the field of view (FOV) is used. These so-called 'guide stars' can be real stars in

the sky, called a natural guide star (NGS), or if no such suitable NGS exists in a given patch of sky (called the *isoplanatic patch*, which has an associated *isoplanatic error*, $\sigma_{isoplanatic}$, to be explained in the next section) around the science target, an 'artificial' laser guide star (LGS) can be used. Many 10 m class telescopes produce LGSs by projecting a laser beam of sodium light (λ = 589nm) into the sky to excite a layer of sodium ions in the mesosphere between 90 - 100 km, creating a star-like spot for the WFS to guide on.

The largest sources of WFE that have been introduced that limit the performance of a classical AO system can be summarized by:

$$\sigma^2_{system} = \sigma^2_{WFS} + \sigma^2_{fitting} + \sigma^2_{temporal} + \sigma^2_{isoplanatic} + \sigma^2_{other} \quad (4)$$

where σ^2_{system} is the total AO system error, σ^2_{WFS} is the WFS noise error, $\sigma^2_{fitting}$ is the DM fitting error, $\sigma^2_{temporal}$ is the temporal error, and $\sigma^2_{isoplanatic}$ is the isoplanatic error (to be discussed in Section 1.3.1). There exist many other smaller sources of WFE for all types of AO systems, which at this time we group into a catch-all error term, σ^2_{other} [5].

1.3 Wide Field Adaptive Optics

Classical AO has some limitations that have restricted its astronomical applications. One particular challenge has been extending good AO correction across a larger FOV. This is desirable for two main reasons: sky coverage, and extended- or multiple-object imaging. The number and distribution of stars in the night sky bright enough to be used as NGSs in a classical AO system is referred to as sky coverage. Only ~5% sky coverage

is achievable with the current suite of classical AO systems on modern observatories, leaving many interesting science targets outside the realm of AO correction because of the degradation of AO correction as one moves radially away from the guide star. This challenging effect is known as *isoplanatism* and sets the upper limit on the angular distance a guide star can be located from the science target, and thus the area of good correction, known as the *isoplanatic patch*.

1.3.1 Isoplanatism

Most single stars are considered point sources of light, and thus the region of good correction need not be large in order to image these targets, provided that the guide star is close enough, or the science target itself has enough light that it can be used as a guide star. However, in the cases where the guide star is a far enough angular distance from the science target, or the science target is an extended object (ie. a galaxy or nebula), or there are multiple science targets (like in a star cluster), the required AO corrected FOV is large. WFSs in classical AO measure the wavefront phase aberrations of an entire column of atmosphere the diameter of the telescope primary mirror, with a central axis along the line of sight from the telescope to the guide star. The atmosphere has a varying turbulence profile throughout this column, thus the phase aberration contributions of different layers of atmosphere differ. Typically, the largest contributions to these aberrations occur at the ground, followed by the 10 - 15 km altitude range (the altitude of the jet stream). Figure 9 shows the relative turbulence strength for a range of altitudes as measured at the site of the Gemini South observatory on Cerro Tololo, Chile.

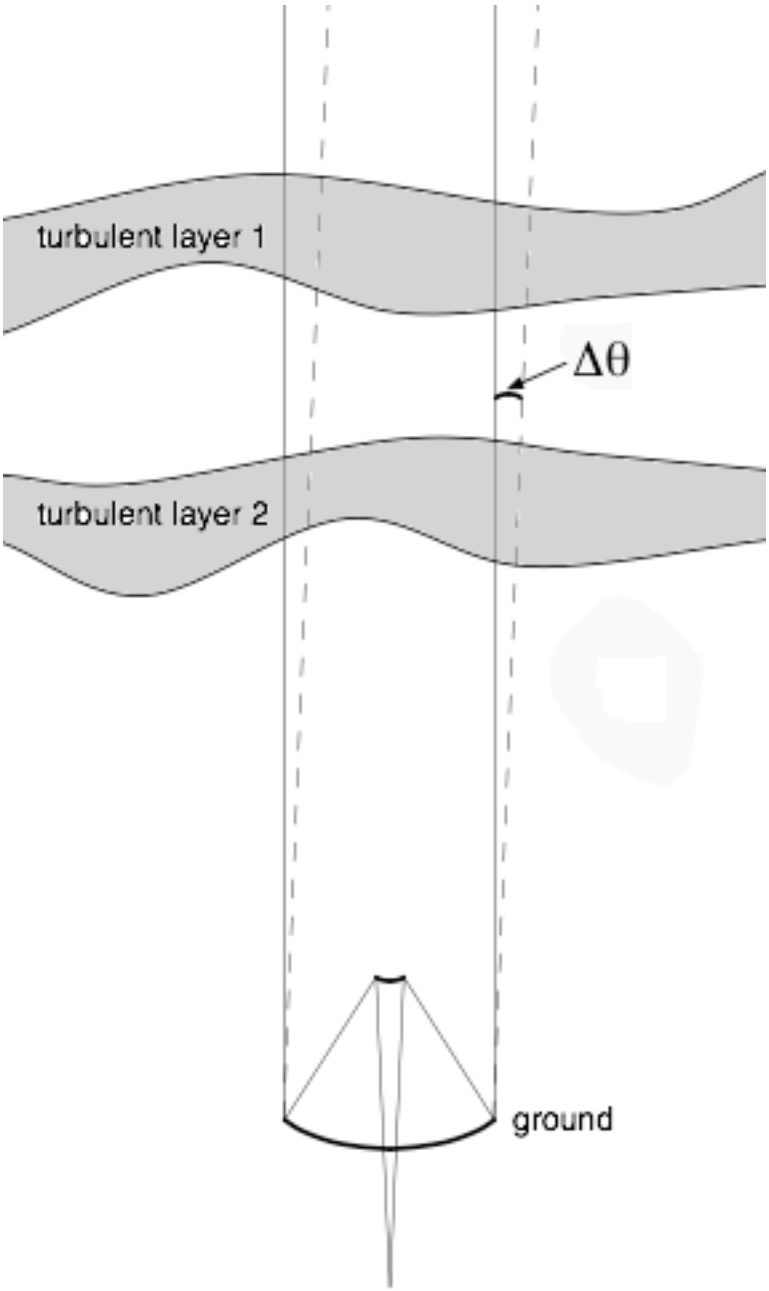


Figure 8: An illustration of the effect of isoplanatism. Light from an off-axis guide star travels through the atmospheric layers at a slightly different path, thus a slightly different column of atmosphere perturbs its wavefront than that of the on-axis science target.

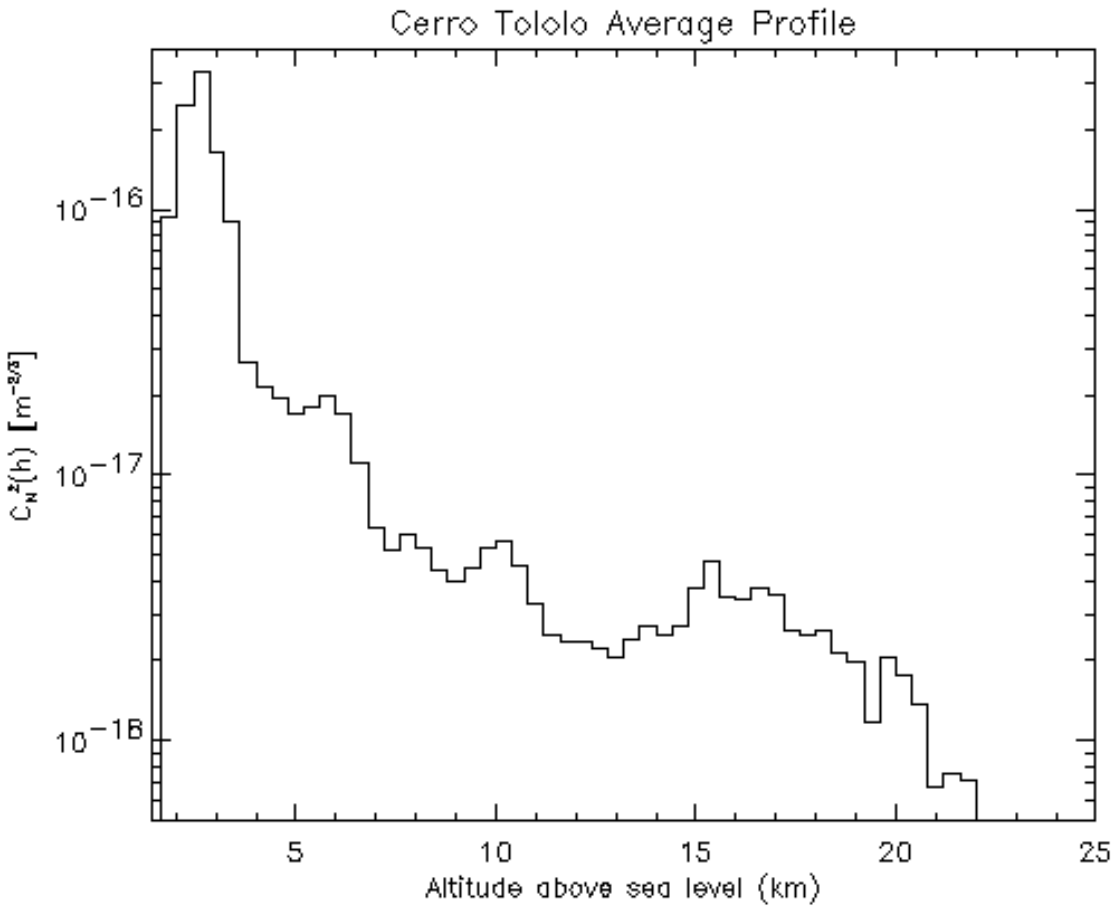


Figure 9: The average C_N^2 profile, a measure of the relative turbulence strength across a range of atmospheric layers, at the site of the Gemini South observatory [2].

It is common that an AO system needs to use an off-axis guide star brighter than the science target to provide enough light for the WFS to make wavefront measurements at the short exposure times driven by the control loop bandwidth requirements. However, the further the guide star is off-axis, the less representative the wavefront measurements acquired from it are of the atmospheric turbulence effects on the light from the science target. This is because the light from the guide star travels through a different column of atmosphere to the telescope than that of the science target, as shown in Figure 8. An example of an image with and without isoplanatism is shown in Figure 9. The degree of overlap between the two atmospheric columns is related to the off-axis angle of the guide

star. The *isoplanatic angle* is commonly defined as being the angle the guide star is off-axis at which the Strehl ratio of the AO system has fallen by 50% compared to its value centred at the guide star [6]. The angular area in the sky subtended by the isoplanatic angle around the axis from the telescope to the guide star is commonly referred to as the *isoplanatic patch*. The wavefront measurement error made by the WFS due to the guide star being off-axis is thus called the *isoplanatic error*, $\sigma_{isoplanatic}$. A LGS can be placed on or close to the science target in the sky to combat isoplanatism, however, this approach suffers from the problem of tip-tilt determination, which requires that a (albeit fainter) NGS still be available near the science target to measure the low order wavefront tip-tilt term. It also suffers from a similar problem to isoplanatism called *focal anisoplanatism*, or the *cone effect*.

1.3.2 The Cone Effect

The cone effect, or focal anisoplanatism, is another effect related to unequal turbulence profiles (as seen by the telescope) between the science target and the LGS. In this case, it is due to the finite altitude of the LGS spot, causing two complications: 1) the light emanating from the LGS downward to the telescope only probes a volume of atmospheric turbulence as high as 90-100 km and of a conical shape, not a column as in the case for an infinite source, and 2) the light from the LGS comes to focus at a different distance behind the telescope than the science target. Figure 11 illustrates this effect; it can also be appreciated from the figure how the cone effect gets worse with increasing telescope diameter [5, 6].

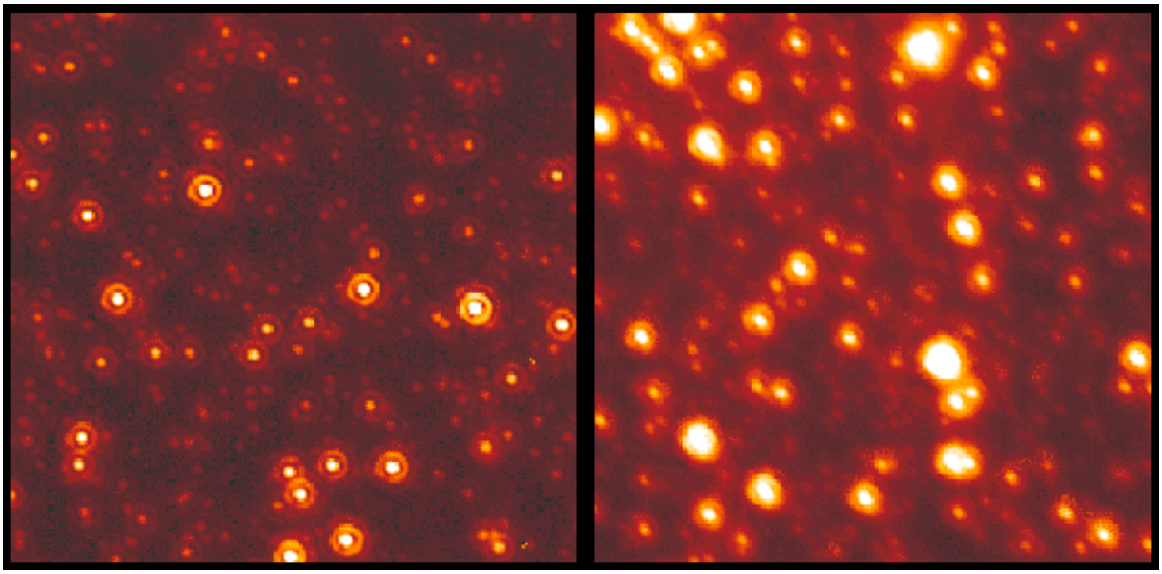


Figure 10: AO images from the Canada-France-Hawaii Telescope with the PUEO adaptive optics system and KIR infrared camera illustrating isoplanatic error. The two 7 arcsecond square images are actually part of one larger image. The region at left is very close to the guide star, and the one on the right is 30 arcseconds away [6].

1.3.3 Atmospheric Tomography

One approach that can address the isoplanatic error, cone effect, and sky coverage problems simultaneously is to apply *atmospheric tomography*, where a three-dimensional model of a large cylindrical turbulent volume above the telescope is constructed by probing overlapping conical volumes with light from multiple guide stars. This allows for good correction across a FOV much larger than the isoplanatic patch.

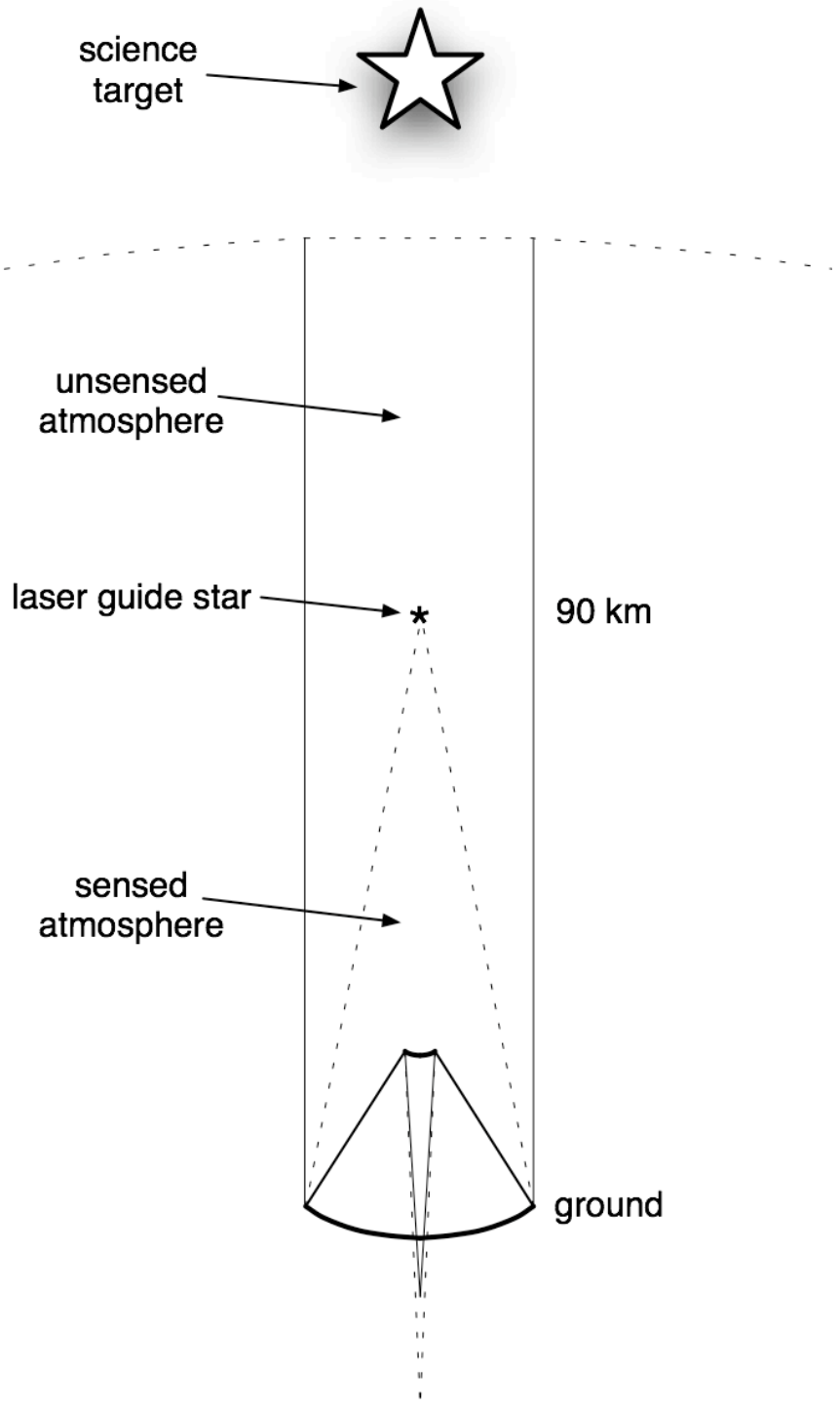


Figure 11: The cone effect (focal anisoplanatism). Light from a laser guide star (dashed) at a finite altitude probes a conical volume to the telescope, only sampling this portion of the atmosphere and leaving out the column above and around it which is traversed by the light of the science target. As well, light from a finite altitude comes to focus at a different distance behind the telescope than light coming from infinity.

1.3.4 Multi-Conjugate Adaptive Optics

Multi-Conjugate Adaptive Optics (MCAO) promises to provide diffraction-limited image quality across a FOV of 1 - 2 arcminutes by using atmospheric tomography. This will be achieved using multiple guide stars – LGSs in most cases – across the FOV to probe overlapping volumes of atmosphere, and multiple DMs optically conjugated to the layers of atmosphere known to contribute the most wavefront error, as shown in Figure 12. In the case where multiple LGSs are used, their cones of light overlap to synthesize a continuous column of atmosphere, thus mitigating the cone effect. The tomographic reconstruction of the 3D index of refraction of this column [7] gives a well-represented model of the turbulence up to ~90 km, thus mitigating isoplanatic error as well. Tip-tilt determination is still an issue, however, suitable NGSs become much easier to find with

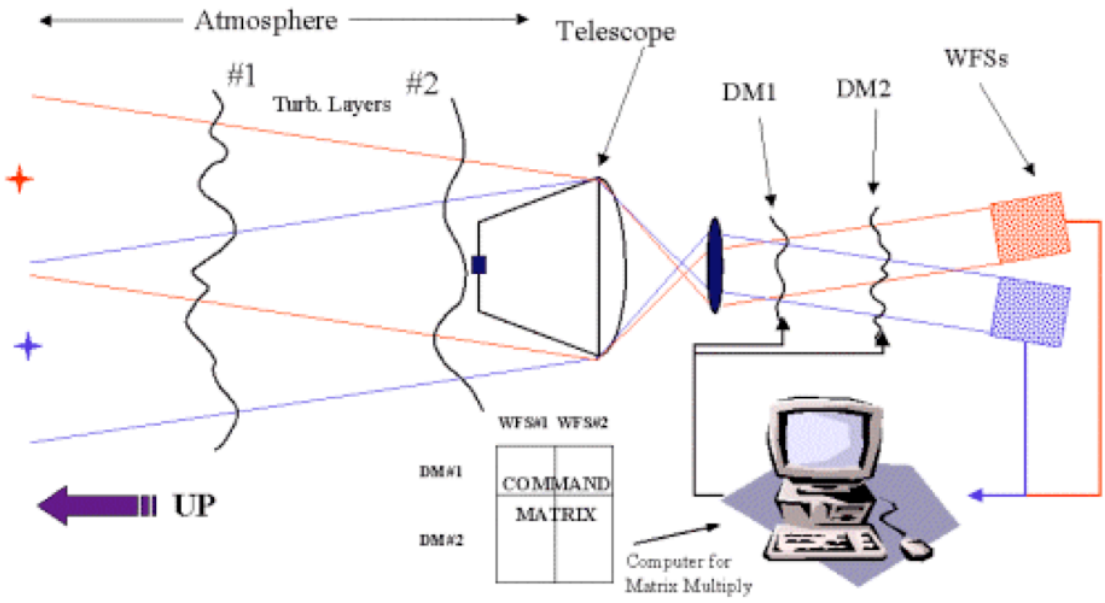


Figure 12: A Multi-Conjugate Adaptive Optics (MCAO) concept diagram. The red and blue stars are NGSs and/or LGSs. Their angled columns of WFS-probed atmosphere overlap, from which the tomography of the column directly above the telescope can be reconstructed by a multiplication of the Command Matrix by the centroids from the WFSs. DM1 is conjugated to the ground layer (just above the telescope aperture) and DM2 is conjugated to another turbulent layer at altitude, providing the best corrections for the turbulence at these layers and an averaged correction for the volumes in between [6].

the considerably larger corrected FOV provided by the LGS ‘constellation’, leading to an increase in sky coverage [6].

MCAO does suffer from *generalized fitting error*, stemming from having a discrete number of DMs, each optically conjugate to a discrete turbulent layer. Ideally, one would have an infinite number of DMs conjugated to an infinite number of layers. Since this is not possible, the correction between these conjugated layers is limited as the system is forced to apply averaging of the large turbulent volumes between the conjugated layers.

Gemini Facility MCAO System

There are a small number of first generation MCAO systems being tested on 10 m class telescopes, and future MCAO systems for ELTs in development. The Gemini MCAO system is nearing completion on the 8 m Gemini South telescope, on Cerro Tololo, Chile. It will have 5 LGSs and 3 DMs conjugate to 0, 4.5 and 9 km and will provide uniform correction over a 1 square arcminute field, producing diffraction-limited images inside this region. At least 3 NGSs are needed to give the 6 degrees of freedom needed to constrain the plate scale and first few orders of image distortion [39], however, the magnitude limit is quite faint (down to magnitude 19), so high values of sky coverage can be attained (about 15% at the galactic pole and over 70% at 30º galactic latitude). Figure 13 shows a simulated uncorrected (no AO) large FOV, and a comparison of simulated corrections over this same FOV between classical AO and Gemini MCAO [6].

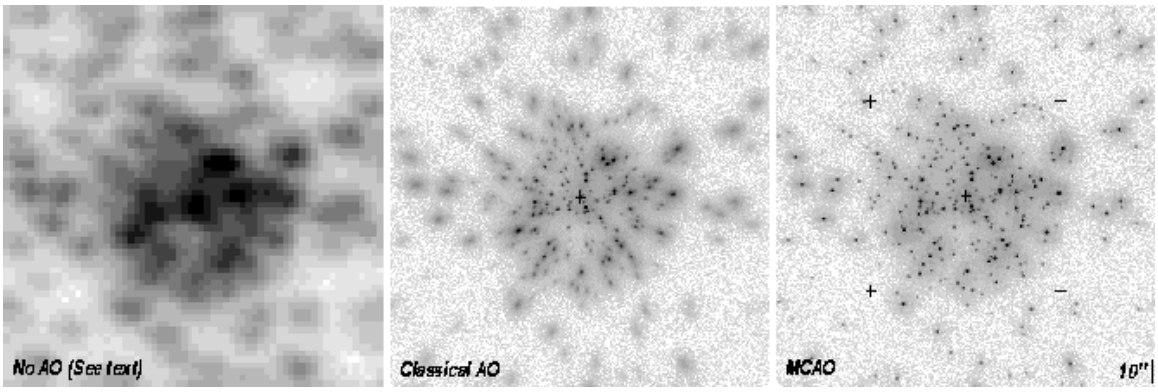


Figure 13: Left: a simulated non-AO-corrected image of many science targets spanning a large FOV. All objects are equally blurred by simulated turbulence. Centre: the same FOV with simulated classical AO correction, the guide star indicated by the '+' symbol. Image resolution from correction is clearly best near the guide star, and degrades radially outwardly. Right: the same FOV with simulated Gemini MCAO correction. Guide stars are indicated by the '+' and '-' symbols. Correction is uniform within the entire 1 arc minute squared FOV [6].

Multi-Conjugate AO Demonstrator (MAD)

The Multi-Conjugate AO Demonstrator (MAD) is a prototype MCAO instrument under test on an 8 m telescope at the European Southern Observatory's Very Large Telescope, on Cerro Paranal, Chile. There are two DMs conjugate to 0 and 8.5 km providing correction over a 2 arcminute FOV. MAD has no LGSs but uses rare groups of relatively bright (brighter than magnitude 14) tightly-clustered NGSs called asterisms for the WFSs to guide on just to demonstrate the MCAO concept. It is equipped with two different WFS systems for comparison: 1) three Shack-Hartmann WFSs moveable within the FOV, allowing selection of three suitable guide stars to demonstrate what is called 'star oriented' MCAO reconstruction, and 2) a 'layer oriented' Multi-Pyramid WFS, capable of sensing up to 8 NGSs and imaging the turbulence at the two altitudes conjugate to the DMs, to demonstrate layer oriented MCAO reconstruction. MAD was the first to demonstrate wide FOV correction with MCAO on March 25, 2007. A

comparison between a non-corrected image of the globular cluster Omega Centauri and one corrected by MAD using the star oriented reconstruction mode is shown in Figure 14 [8].

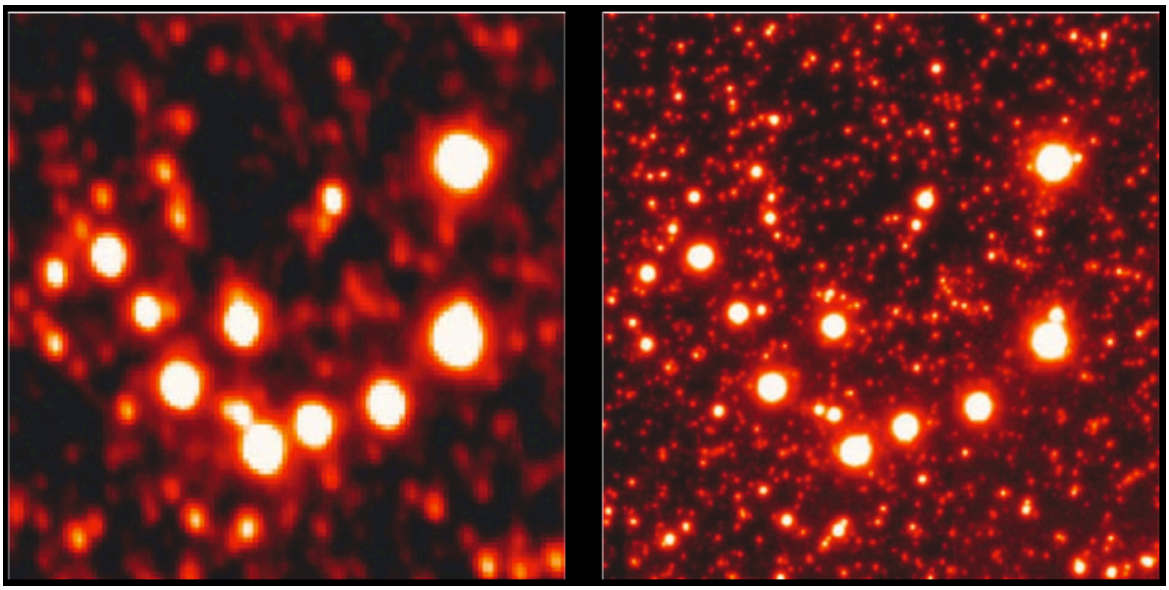


Figure 14: Left: A 20 x 20 arcsecond non-corrected image of the center of the Omega Centauri globular cluster. Right: The same region, located at the centre of the 2 arcminute corrected FOV provided by MAD using star oriented tomographic reconstruction [8].

1.3.5 The Narrow-Field Infrared Adaptive Optics System (NFIRAOS)

The Narrow-Field Infrared Adaptive Optics System (NFIRAOS), a second-generation MCAO system, is under design at the Herzberg Institute of Astrophysics and will be the facility AO instrument for the Thirty Meter Telescope (TMT). MCAO will deliver a large well-corrected FOV to the various science instruments planned for the telescope. In its first incarnation, it will use three NGSs, six LGSs, and two DMs conjugated to the ground layer and 12 km altitude to provide diffraction-limited imaging across a 10 arcsecond diameter FOV [9].

1.3.6 Multi-Object Adaptive Optics

The other diffraction-limited wide field AO instrument approach, one that avoids the DM-limited nature of MCAO, is Multi-Object Adaptive Optics (MOAO). Instead of correcting an extended science FOV as in MCAO, MOAO instead will provide localized correction around a number (5 - 40) of selected science objects spread around the FOV, as illustrated in Figure 15. This allows for a larger accessible FOV than MCAO, up to 10 x 10 arcminutes, whereas the uniform AO correction provided by current MCAO systems in development is limited to a FOV of 0.5 - 2 arcminutes. The multi-object capability will yield a large multiplex advantage over both classical and MCAO systems and will enable extragalactic studies otherwise very costly to implement with MCAO [10, 11].

The first proposed MOAO instrument study was conducted for FALCON (Fiber optics spectrograph with Adaptive optics on Large fields to Correct at Optical and Near-infrared), for the 8 m Very Large Telescope on Cerro Paranal, Chile [10]. Another MOAO instrument concept that has a similar goal to FALCON – to provide AO correction for multiple objects for high-resolution spectroscopy – is the Near-Infrared Multi-Object Spectrograph (IRMOS), which we will use to illustrate some of the common features of MOAO in the next section.

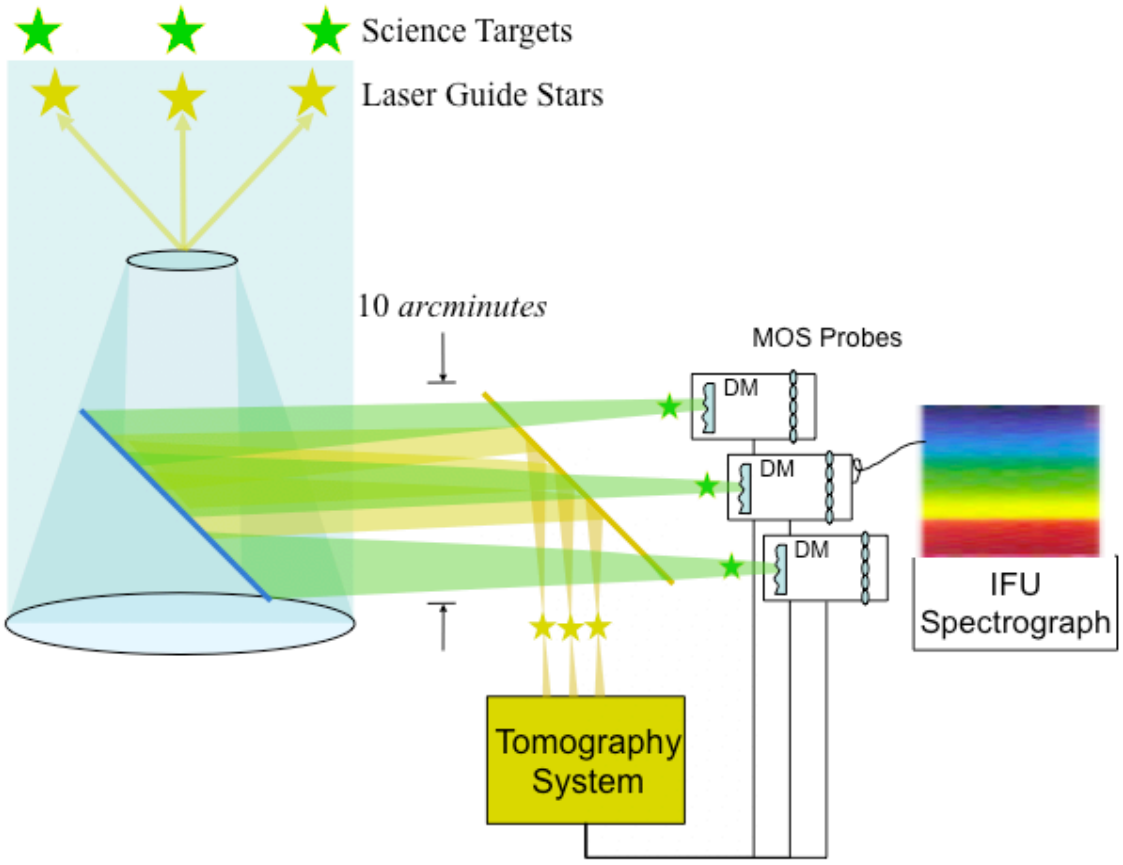


Figure 15: Multi-Object Adaptive Optics (MOAO) conceptual diagram. LGSs (indicated by yellow stars) are used for atmospheric tomography, as in MCAO. Light from science targets (in green) is picked off by the Multi-Object Spectrograph (MOS) probes, each with its own DM. The tomography system works in open loop; the LGS WFSs sense the entire turbulence. The tomographic reconstruction is projected onto the MOS probe DMs, and the corrected image is focused onto an array of fiber optics and transmitted to its own Integral Field Unit (IFU) spectrograph (D. Gavel).

Near-Infrared Multi-Object Spectrograph (IRMOS)

As an example of what a MOAO instrument may look like, we consider the University of Florida/HIA design for the Near-Infrared Multi-Object Spectrograph (IRMOS) [11, 37], a second generation instrument planned for the Thirty Meter Telescope (TMT). IRMOS will contain ~20 Multi-Object Spectrograph (MOS) probe arms, *each* containing its own Atmospheric Dispersion Corrector (ADC), tip-tilt mirror, two DMs forming a

woofer-tweeter pair[2], feeding an integral field unit (IFU) spectrograph (Figure 16). While the details of science target pick-offs, DM configurations, and spectrographs vary widely, all MOAO instruments share a common design element in their WFSs. The WFSs all sense the atmospheric turbulence independent of the DM correction. In the case of IRMOS, 8 LGS WFSs and 6 NGS tip-tilt focus WFSs are employed to sense the total, *not residual*, atmospheric turbulence. The Real-Time Computer (RTC) takes these measurements and creates a tomographic representation of the atmosphere. Then the optimal DM shape commands are generated and applied *individually for each of the ~20 science targets*. This scheme of measuring the full error signal and applying the full correction at each iteration without feedback is called open loop control (see Figure 17). As a general indication of the expected level of performance for a MOAO system, TMT called for a requirement that 50% of the J-band (λ = 1.13 μm) PSF energy be enclosed within a 50 milliarcsecond spatial pixel (spaxel). This demands a very small high order WFE budget for IRMOS.

Before MOAO systems are incorporated into instruments for 8 to 30 m telescopes, however, there are several risks that need to be retired. Many elements of an MOAO system, such as the use of atmospheric tomography, MEMS mirrors, and woofer-tweeter control have all been demonstrated to work in different lab settings and are included in advanced instrument concepts. Open loop control, however, is perhaps the greatest risk

---

[2] The woofer-tweeter arrangement is a technique to get more DM stroke by reflecting the light off two DMs in series, and also allows for the DMs to be optimised for correcting low-order, high stroke (woofer) and high-order, low stroke (tweeter) spatial aberrations [12].

to MOAO, mainly because it is the biggest unknown. Open loop control introduces unique requirements on the AO system:

- the WFS needs to have a high dynamic range, as it senses the full uncorrected atmospheric turbulence
- DM hysteresis and non-linearity need to be well-understood and the their effect mitigated
- alignment and calibration become more challenging. Non-common path (NCP) errors of open loop AO systems – due to the DM and other optics being downstream of the WFS and thus not being seen by it – may degrade image quality when small problems arise.

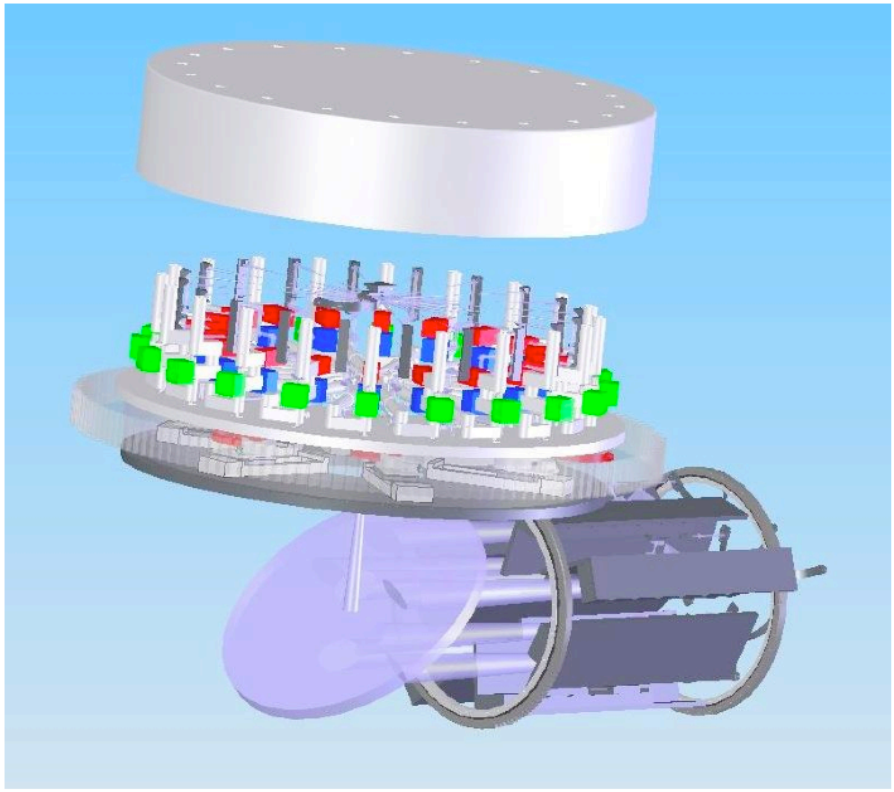


Figure 16: A model of all the MOAO components of IRMOS., the 20 MOS probe arms with integrated WFSs and woofer-tweeter DM pairs are arranged radially around the centre hole, the FOV of the telescope to be probed.

While the very first open loop AO experiment was in 1991 by Primmerman et al., who used so-called 'go to' adaptive optics to make corrections and take science images immediately following pulses from a laser guide star that had a low duty cycle [40], there has been increased interest lately in lab and on-sky experiments involving open loop control [13, 14, 15]. Some open loop AO control has recently been demonstrated in the lab and on-sky using MEMS DMs and LGSs [13, 14, 15].

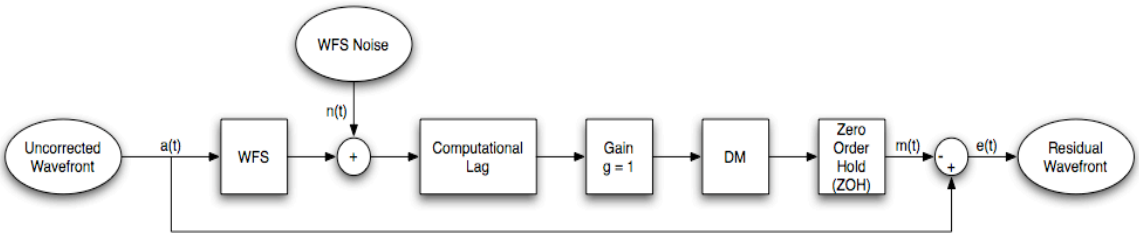


Figure 17: In the most basic implementation of an open loop control architecture, the slopes measured by the WFS are directly converted into DM commands using the reconstructor and a gain of 1. Unlike in a closed loop AO system, no integrator is required [16].

1.4 Scope of Thesis

My thesis work was conducted primarily at the Herzberg Institute of Astrophysics (HIA), a National Research Council of Canada laboratory that is at the forefront of astronomical instrumentation development in Canada. HIA is highly respected in the international astronomical community, having a long history of building quality cutting edge instruments for optical, infrared, and radio astronomy. The adaptive optics group is made up of engineers and scientists with a wealth of experience in this very specialised field, having completed a number of successful projects, in recent time most notably the ALTitude-conjugate Adaptive optics for the InfraRed (ALTAIR) for the Gemini North

Observatory. The knowledge and experience of the adaptive optics group was an invaluable resource for me throughout the thesis.

My work can be roughly categorized into two main areas: 1) high performance computing for improving adaptive optics simulation turnaround time, and 2) high performance computing for real-time control of a prototype wide field open loop adaptive optics instrument, along with a wide range of support activities for its design, integration, and test.

The first item – covered in Chapter 2 – refers to the work I did to significantly improve the performance of the state of the art AO simulation package the Linear Adaptive Optics Simulator (LAOS) [17]. LAOS is the only simulator to date that is capable of modeling 30 m class ELTs with integrated adaptive optics systems on a single computer. This work spanned from April 2006 to July 2007 and consisted of software reverse engineering, software design, programming, and rigorous testing. These improvements – incorporated into the current release of LAOS – offer a 2.5 to 3 times speedup (and potentially more, depending on the case) to instrument scientists and engineers in the Thirty Meter Telescope consortium (and perhaps others) in simulating and evaluating the feasibility and performance of a wide range of ELT configurations with integrated WFAO.

The latter item – covered in Chapter 3 – accounts for the bulk of my thesis work (from May 2006 to June 2008). In partnership with Dr. David Andersen of the Herzberg

Institute of Astrophysics (HIA), my work consisted mainly of designing and building the real-time control system (both hardware and software) for an ambitious cutting edge WFAO testbed, the Victoria Open Loop Testbed (VOLT). The objective of VOLT was to demonstrate open loop control of an AO system on-sky with a single on-axis natural guide star, to retire the risk of open loop control for MOAO. VOLT was the first of its kind (although there was a similar project –the VIsible Light LAser Guidestar Experimental System (VILLAGES) – by the University of California Santa Cruz in development and test concurrently [14]) and presented a number of unique technical challenges related to the open loop system architecture and control scheme that had not been faced before. My duties included the following:

- specifying, designing, and building the real-time control computer systems, including integrating the various specialised interface boards
- configuring the Linux operating systems of the control computers (including installating and configuring hardware drivers) for real-time performance
- resolving hardware and software conflicts, and determining system resource allocation for optimal performance
- configuring, characterizing, testing, and integrating CCD and CMOS cameras and custom deformable mirror electronics
- assisting in VOLT simulation, rapid testbench prototype build-up, and system calibration and characterization using the UVic AO Library (see Section 3.6.1)

- assisting in optomechanical component installation, optical alignment, and instrument integration in the lab and at the telescope
- assisting in data reduction and analysis for system and component characterization
- telescope operation and observation on-sky with VOLT.

Chapter 2
High Performance Adaptive Optics Simulations

Simulating the AO performance of ELTs required new tools. Older Monte Carlo end-to-end AO simulation tools required too much memory and were too slow to run simulations of these large telescopes. Analytic AO modeling codes do not include effects such as the cone effect, which become increasingly important for large apertures. The first tool capable of Monte Carlo performance simulation – on a single computer – of an ELT with integrated AO is the Linear Adaptive Optics Simulator (LAOS), a set of MATLAB scripts written by Luc Gilles and Brent Ellerbroek of the Thirty Meter Telescope [17].

2.1 Linear Adaptive Optics Simulator (LAOS)

LAOS employs minimum variance wavefront reconstruction, implemented with sparse matrix techniques for efficiency, to provide end-to-end telescope simulation with integrated AO. All AO components and phenomena are based on linear models and constrained by analytical first-order performance estimates for 8 m class telescopes. The core of the simulator uses one of two options for wavefront reconstruction from geometrical or physical optics Shack-Hartmann WFS measurements: a multigrid preconditioned conjugate gradient (MG-PCG) algorithm, or a sparse Cholesky solver [17].

LAOS is an end-to-end AO simulator implemented in MATLAB that can incorporate the model of any user-defined telescope within a range of types and sizes, bounded by the computational speed and memory constraints of the system it is running on and/or what the user considers an acceptable amount of time required to run the simulation. Large-

scale simulations such as those for instruments for MCAO on the Thirty Meter Telescope can easily require run times on the order of a week or more. As part of my M.A.Sc. work I improved the performance of LAOS to make these large scale simulations more feasible.

2.1.1 Identifying the Bottleneck

To begin speeding up LAOS, it was necessary to generate a performance profile to identify where the simulator spends most of its time, ie. the bottleneck(s). MATLAB conveniently has a module called the Profiler which does exactly this. The Profiler is started just before running a LAOS simulation and after completion produces an extensive report of how much time was spent in each function as well as a heirarchy tracing the parent/child function calling relationships, which can be conveniently saved into a set of interlinking HTML files for offline analysis. Once the bottleneck(s) has been identified, it can be determined if there is a reasonable solution promising substantial LAOS performance improvements, by way of significantly speeding up execution at the bottleneck(s). Figure 18 shows the partial Profiler report of a 400 iteration LAOS simulation run. The particular test case I used was the NFIRAOS facility AO instrument; the model consisted of a 30 m telescope with 2 DMs, 6 atmospheric turbulence layers, also called phase screens (PSs), and 9 WFSs [18, 9], using a conventional weighted centre-of-mass centroiding algorithm.

Profile Summary

Generated 26-Jun-2006 14:40:08 using cpu time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
accphi	86064	139506.080 s	117744.160 s	
sub2ind	416046	22184.560 s	22184.560 s	
ifft2	27043680	13952.610 s	13952.610 s	
Perfievl	7200	11267.970 s	11267.970 s	
Prop	19116	150734.270 s	11228.190 s	
genpowfs_ints	400	160343.560 s	10800.900 s	
fft2	13893432	9738.070 s	9738.070 s	
mkpi	6943116	29711.770 s	9680.950 s	
fftshift	81869442	8935.420 s	8935.420 s	
spbs	97314	3775.550 s	3775.550 s	

Figure 18: MATLAB Profiler results showing the top ten time consuming functions in the LAOS simulation [19].

When interpreting the Profiler results it is most useful to look at the *Self Time* of each function. As explained in the MATLAB Profiler report, "*Self Time* is the time spent in a function excluding the time spent in its child functions. *Self Time* also includes overhead resulting from the process of profiling." It is apparent here that *accphi*, the function responsible for calculating and accumulating contributions by the atmospheric PSs, DMs, and optical surfaces to the phase perturbations *phi* on the incoming wavefront, has the most *Self Time* by a large margin. I determined by a separate performance test (which allows us to get a more quantitative assessment by avoiding the inherent overhead of the Profiler) that the time spent in *accphi* accounted for approximately 76.7% of the total simulation time. It should be noted that *sub2ind* is a child function called almost

exclusively by *accphi*, and thus nearly all the time spent inside *sub2ind* is included in this 76.7% figure. From this analysis I decided that the way to make the largest immediate gains in speed for LAOS would be to speed up the bottleneck *accphi* as much as reasonably possible.

2.1.2 C MEX Implementation

The existing LAOS MATLAB function accphi.m is written in a concise manner and is logically relatively easy to follow, and no real performance improvements could be envisaged by attempting to further optimise this code. As the M language is, single lines of code often encapsulate large iterative looping operations and on-the-fly creation (and resizing) of arrays. This is where efficient C coding can make great gains over MATLAB, by minimising the number of loops needed (ie. performing multiple operations rather than just one within a given loop wherever possible) and closely managing the amount of memory being used (and controlling exactly how much will be dynamically allocated, and when). C was chosen because it is supported by MATLAB's external interface MEX, which can link MATLAB scripts to compiled C code. The C MEX library provides an API for this interface, allowing compiled MEX functions to be called by any MATLAB script. See Appendix B for details on system configuration and C MEX code compilation, and Appendix C for a source code listing.

Single-Threaded accphi_C.c

I first decided to emulate accphi.m – in how it is called and passed / returns parameters – with a single-threaded C version accphi_C.c. This would mean having to make very minimal changes in the M code of the parent functions calling *accphi*, namely Prop.m and genwfsopd.m, requiring simply that calls to *accphi(....)* be replaced by calls to

accphi_C(...). Following is a summary of some of the optimisation choices made in coding accphi_C.c:

- Multidimensional arrays are dynamically allocated as contiguous 1-dimensional arrays of the exact type and size needed. Where MATLAB would always allocate space for double precision floating point numbers regardless of the context of their usage, appropriate data types with smaller footprints in memory have been used. Using 1-dimensional arrays makes looping and array indexing faster.
- Many iterative operations are grouped together (and sometimes reordered, where such reordering will not change the final numerical result) to be executed inside a minimal number of loops, rather than having multiple loops each executing a single iterative operation.
- The MATLAB *find* function is replaced by conditional statements within loops, which avoids some redundant instances of looping through entire arrays simply to compute sets of indices for the arrays. Consequently, considerable memory usage savings were made by avoiding the allocation of these temporary arrays of indices.
- The *sub2ind* step has been removed as a child function call and has been explicitly coded into one line of C code in both instances where it is needed inside *accphi* (thus significantly tackling this secondary bottleneck, as it was reported to be by the Profiler).
- Dynamically allocated memory that was found to be unused after a certain point in the algorithm was reused where possible, to avoid growing the

working set by allocating more memory (specifically, the local copy of the *ploc* input array gets overwritten, its space being used for *xploc* and *yploc*).

Using the same NFIRAOS test case as mentioned earlier, and using some MATLAB timing functions (ie. *cputime*, *tic* and *toc*), performance comparisons were made for accphi.m vs. accphi_C.c and are summarised in Table 1. Being that the size of the working set for *accphi* varies greatly during a simulation, average execution times for *accphi* over 100 LAOS simulation steps were calculated. The average execution time for one call to accphi.m (taken from a sample of 23184 calls in a 100 iteration LAOS run) was 1.87 seconds, and the average execution time for one call to accphi_C.c (taken from a sample of 21264 calls in a 100 iteration LAOS run) was 0.260 seconds, a speed-up of 7.2 times. Total LAOS simulation time was 56562 seconds (~15.7 hours) for the accphi.m implementation and 22341 seconds (~6.2 hours) for the accphi_C.c implementation, a speed-up of about 2.5 times. This corresponds to a reduction from the original 76.7% of total LAOS simulation time spent in *accphi* to 24.7% of that original time. This result reflects the performance of single-threaded accphi_C.c using optimising compiler flags [20] for our Sun Fire V40z machine with quad dual-core AMD Opteron 875 2.2GHz CPUs with 1Mb L2 cache per core and 32GB RAM [21], running 64-bit Red Hat Linux. Non-optimised compilation suffers a 6% penalty in accphi_C.c execution time in comparison (average execution time for one call to accphi_C.c with non-optimised compilation is 0.276 seconds).

	accphi.m	accphi_C.c	Speed-up
Total LAOS simulation time (100 iteration run)	56562 s (~15.7 h)	22341 s (~6.2 h)	2.5 X
Number of calls to accphi	23184	21264	N/A
Total time spent in accphi	43432 s (76.7%)	5527 s (24.7%)	N/A
Average time per accphi call	1.87 s	0.260 s	7.2 X

Table 1: Performance comparisons of accphi.m and accphi_C.c for the NFIRAOS test case – a 30 m telescope with 2 DMs, 6 atmospheric turbulence layers (phase screens), and 9 WFSs.

Multi-Threaded accphi_C_SMP.c

Multiprocessor (SMP) systems can offer significant speed-ups beyond that possible with a single processor system, if the computational problem can be broken into pieces that can be processed in parallel (ie. they have no data dependencies, such that the result of one step does not depend – or have to wait – on the result of another). Although MATLAB does not directly support multithreading, it will 'allow' it when implemented within a MEX context (MATLAB is unaware that multithreading is going on inside the MEX function, so it works, provided that no non-reentrant code is called within that function).

There are further performance gains to be made for LAOS by processing all PS, optical surface, or DM instances in parallel. It is not possible to process all three in parallel, as it would violate the serial nature of the LAOS simulation – and more fundamentally, it would violate the structure of AO feedback loops, as the DM and tip-tilt mirror (TTM) compensations are generated in response to *phi* perturbations produced by the PSs and optical surfaces. This new parallel version of accphi_C.c I developed to exploit SMP

architectures is called accphi_C_SMP.c. The implementation of accphi_C_SMP.c can be summarised as follows (for detail on the coding changes made to to LAOS to support parallel processing, refer to Appendix C):

- Working set data for each layer is parsed and indexed by the main thread and packaged with other parameters specific to that layer in preparation for child thread creation, one thread per layer-specific task.
- These layer-specific tasks are then sorted for execution in descending order according to the size of their respective working sets. This results in beginning processing of the largest working set first, which generally minimises the total time to complete all tasks.
- The main thread then creates one child thread per task, which are essentially modified instances of accphi_C.c, up to a user-defined maximum number of concurrent threads (a LAOS parameter). Remaining threads are then spawned to take the place (and processor) of preceding threads as each completes. It should be noted that accphi_C_SMP.c does not detect the number of available processors on a system, but leaves it up to the user to decide on (and set the parameter for) how many threads per processor may be assigned.
- Each thread calculates its own layer-specific contribution to the phase *phi*, and adds it to a global *phi* array accessible to all threads. When all threads have completed, this array is passed back to the MATLAB LAOS calling function.

A performance evaluation was run for the same NFIRAOS case as above, demonstrating a further speed-up from 2.5 times for accphi_C.c to 3.0 times for accphi_C_SMP.c, as shown in Table 2.

	Total LAOS simulation time (100 iteration run)	**Speed-up**
accphi.m	56562 s (~15.7 h)	N/A
accphi_C.c	22341 s (~6.2 h)	2.5 X
accphi_C_SMP.c	19967 s (~5.5 h)	3.0 X

Table 2: LAOS performance comparisons for the same NFIRAOS case, between the original implementation with MATLAB accphi.m, that using single-threaded accphi_C.c, and that using multi-threaded accphi_C_SMP.c.

For larger LAOS working sets, a further relative speed-up could potentially be seen, as will be outlined in the next section.

2.2 LAOS MOAO Simulation

To extend this to the case of simulating IRMOS MOAO would mean a larger tomographic reconstruction problem, as the 5 arcminute FOV correction means probing a larger volume of atmosphere than that for the 2 arcminute FOV of the NFIRAOS MCAO case, and thus a longer simulation time than NFIRAOS. However, because the fraction of time spent in accphi is greater for this problem, MOAO simulation with a single field position (ie. a single MOS probe) would receive a higher relative benefit from the use of the single-threaded accphi_C.c than the NFIRAOS case, seeing a 5.1 times speed-up over the MATLAB accphi.m LAOS implementation, versus 2.5 times for NFIRAOS.

With the multithreading capabilities of accphi_C_SMP.c, however, MOAO simulations with multiple field positions (ie. multiple MOS probes) would benefit greatly, as phase screen determination after reflection off the DM surfaces on the MOS probes could be parallelized – one processor per MOS probe – limited only by the number of processors available.

2.3 Summary of LAOS Performance Enhancements

The LAOS software package was implemented as a set of MATLAB scripts and functions. MATLAB has great advantages in its ease of coding and relatively short time required to produce working code, but can lag considerably in execution speed when compared to languages like C, especially when using programming constructs such as large iterative loops. Also, because of its emphasis on user-friendliness, which necessitates its 'behind-the-scenes' memory management, MATLAB memory usage is usually not optimised for performance. For LAOS, it was determined that significant performance improvements could be achieved by replacing the bottlenecks of the simulator with C MEX implementations, where the use of loops and memory could be explicitly controlled and optimised by a combination of efficient programming and a 'smart' C compiler. Significant performance improvements were realised with this approach, first with a single-threaded implementation which produced a 2.5 times overall speed-up for the NFIRAOS MCAO test case, and then with a multi-threaded implementation – taking advantage of the lack of data dependency between atmospheric phase screen and optical surface layers – which pushed the speed-up to 3.0 times for this same test case. The advantage of this parallelization of LAOS will become more apparent with larger problems, especially MOAO.

Chapter 3
The Victoria Open Loop Testbed

3.1 Introduction

The Victoria Open Loop Testbed (VOLT) serves as a demonstration of open loop control, both on-sky at the Dominion Astrophysical Observatory's 1.2 m telescope and in the lab, to facilitate the future development of MOAO. Our goal was to demonstrate open loop control with a simple on-axis natural guide star testbed. As discussed in Section 1.3.6, open loop control is one of the biggest unknowns, and therefore the biggest risks, in building a MOAO instrument. The challenges we face in building an open loop AO system stem from alignment and calibration issues, DM non-linearities, and the large dynamic range of the WFSs.

3.2 Optical Alignment and Calibration Issues

Open loop control presents significant difficulties beyond that of a closed loop system. Small non-common path (NCP) errors can be incorporated into the total error and compensated for by a closed loop controller, whereas an open loop system like VOLT has no such benefit. In a classical AO instrument, the only NCP error is the light passing through the beamsplitter to the science camera (recall Figure 4), which is not seen by the controller. In an open loop AO instrument the DM itself is in the NCP, since it is placed downstream of the WFS, as illustrated in Figure 19. In VOLT, this drives the need for the optical alignment tolerances between the open loop WFS and DM to be high – and a more difficult procedure to achieve these tolerances in the laboratory – and the need for a reliable calibration procedure that can be performed before observing on-sky (see Section 3.6.2).

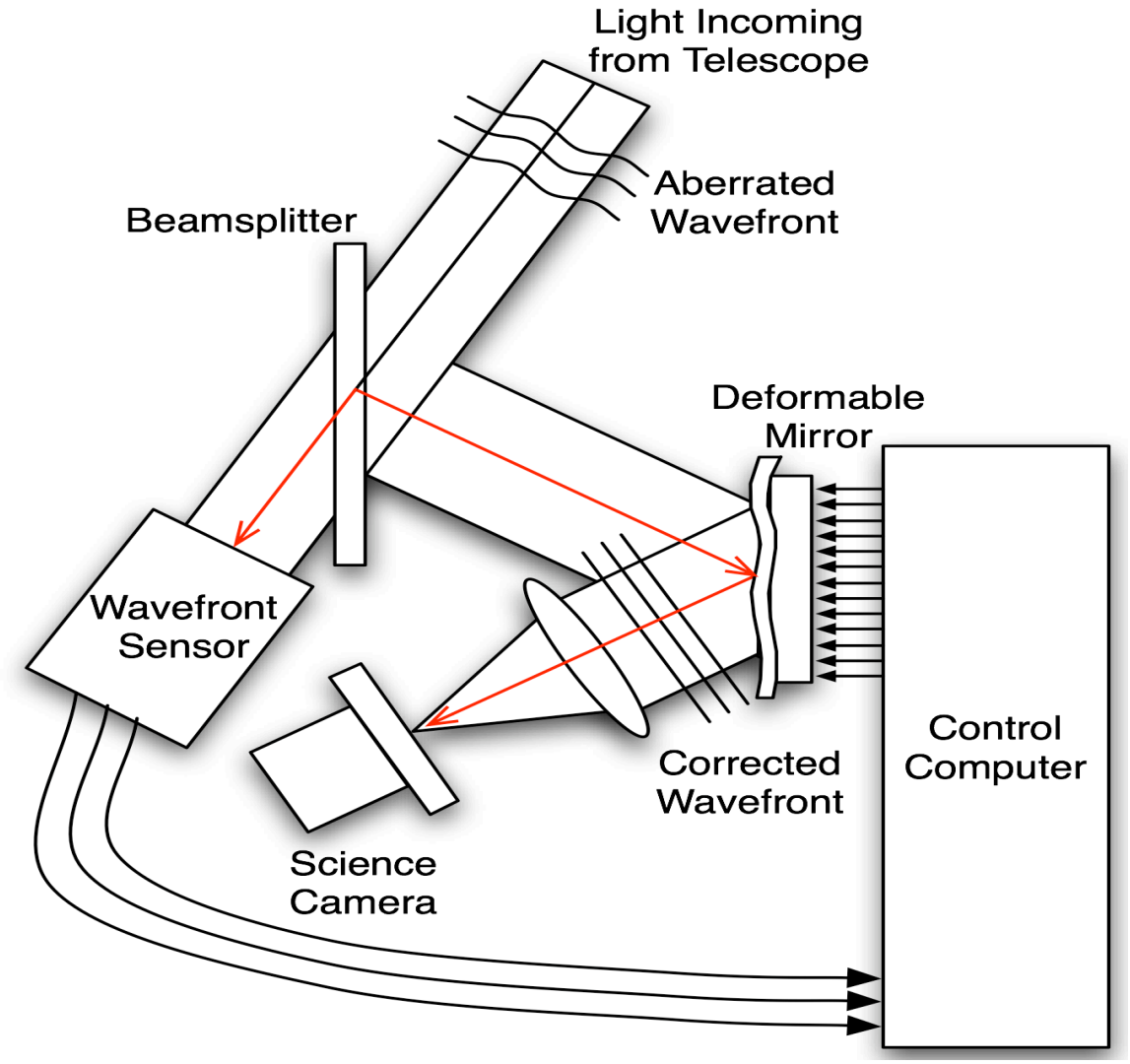


Figure 19: Conceptual diagram of a basic open loop adaptive optics system. The wavefront sensor measures the entire aberrated wavefront before ('upstream from') correction by the deformable mirror. These wavefront measurements are used to generate new commands for the deformable mirror, however, they do not account for any aberrations introduced in the non-common path (NCP), shown in red.

3.3 DM Hystereses and Non-linearities

Another aspect of having the DM in the non-common path with respect to the wavefront sensor optics is that any non-regularities in the temporal performance of the DM contribute fully to the WFE budget. Since an open loop control system does not receive the benefit of residual error term feedback after AO correction, the WFS only

sees the full error due to atmospheric turbulence and nothing of the applied correction downstream. The commanded DM shapes therefore have to be very precise. Thus, it was critical to the success of VOLT to fully characterize the hysteresis and non-linearity properties of the Laboratoire Astrophysique de l'Observatoire de Grenoble (LAOG) ALPAO DM52 Deformable Mirror (see Section 3.5.2 – Deformable Mirror Subsystem).

3.4 High Dynamic Range WFS

Already mentioned, the open loop WFS measures the full atmospheric turbulence instead of the residual after correction as is done in a classical AO system. This means that the WFS must have a larger dynamic measurement range, as the Shack-Hartmann spots will tend to move further from the centre of their subaperture pixel regions on the detector. An undersized subaperture FOV means risking light from spots (or even entire spots) spilling over into adjacent subapertures, adversely affecting the wavefront measurement. Thus, an open loop WFS must have larger subaperture FOVs to ensure the spots stay within their intended subapertures. The precision of these measurements still needs to be as good as would be required in a closed loop AO system, meaning the WFS plate scale must be maintained, just the number of pixels across a subaperture must be increased such that the subaperture size fits the expected observing conditions of the site. The larger required FOV for the open loop WFS is taken into account in the optical design of VOLT (see Section 3.5.2).

3.5 Basics of VOLT design

Many of the basic design choices in VOLT were determined based on the availability of the Laboratoire Astrophysique de l'Observatoire de Grenoble (LAOG) ALPAO DM52 deformable mirror at HIA. The DM52 has 52 actuators inscribed into a circle within an 8

x 8 grid (the DM52 is described in more detail in Section 3.5.2 – Deformable Mirror Subsystem). With poor atmospheric seeing typical to the site, every actuator of the DM would be used to maximise the degrees of freedom available for correction.

This then drove the WFS layout to be 7 x 7 subapertures, each measuring a 17 x 17 cm partition of the telescope aperture, with each subaperture bounded by DM actuators on all 4 corners, as in Figure 20. This configuration is known as the Fried alignment and is optimised so the WFS subapertures detect the wavefront slopes (derivatives) between the actuators, as opposed to being placed 'on top' of the actuators within the pupil plane, where the measured slopes would be zero (as the top of the actuator is flat). Since the DM surface is smooth with pseudo-sinusoidal shapes on it – where the actuators are located at the peaks and valleys – this would lead to the WFS sampling only these peaks and valleys and thus making a measurement of only the amplitude and not the shapes (slopes) themselves. Locating the actuators at the zero crossover points – the points with the greatest slope – between the expected wavefront peaks and valleys allows for an accurate planar approximation of the 2D slopes of these pseudo-sinusoids. The 17 x 17 cm subaperture size corresponds to a diffraction-limited FWHM of ~1 arcsecond in the red. Therefore, according to convention, a pixel size of at most ~0.5 arcseconds is needed to sample the PSF in very good conditions.

No site testing has been conducted at DAO, so we relied on anecdotal evidence that the seeing is poor (as bad as 4 arcseconds, or $r_0 \approx 2.5$ cm), with an average FWHM of ~2

arcseconds ($r_0 \approx 5$cm). We estimate the required FOV (dynamic range) of the open loop WFS based on the Noll (1976) equation for one-axis turbulent tilt:

$$\sigma^2 = 0.4509\left(\frac{D}{r_0}\right)^{5/3} \qquad (5)$$

in radians squared of phase error. To convert to wavefront error, we multiply by $\lambda/2\pi$. Since 1 radian of tilt produces an excursion of ± 2 radians at the edge of the pupil, we obtain the resulting image motion (in radians) by multiplying by $4/D$. The atmospheric induced one-axis image motion is therefore

$$\sigma_{jitter} = \sqrt{0.4509\left(D/r_0\right)^{5/3}} \times 2 \times \lambda/\pi \times 206265 \times 1000 \text{ milliarcseconds RMS} \qquad (6)$$

for $r_0 = 5$ cm at $\lambda = 500$ nm, $\sigma_{jitter} = 0.6$ arcseconds. To accommodate 3σ of jitter, and a 3 arcsecond spot, we require a FOV of 2 * 3 * 0.6 + 3 ≈ 6.6 arcseconds. We therefore placed a requirement on the open loop WFS FOV to be at least 7 arcseconds.

While the AO correction would be better at wavelengths longer than 1 μm [3], we chose to use I-band ($\lambda = 900$ nm) for the science wavelength because the science imaging camera we already had on hand at HIA, the SBIG ST-8I, still had good responsivity at these wavelengths (see Section 3.5.2) [23]. We explored the possibility of using a near-infrared science camera, but it was judged to be considerably more difficult to integrate into VOLT than the SBIG, so we decided to avoid this unnecessary risk.

---

[3] Given that $r_0 \propto \lambda^{6/5}$, at longer wavelengths the time a cell of air of size r_0 takes to transit the pupil is longer. This allows for a relaxation on the required wavefront sampling rate, and thus a drop in the required control loop bandwidth. In addition, for a given WFE measured in nm, Strehl ratio is higher at longer wavelengths.

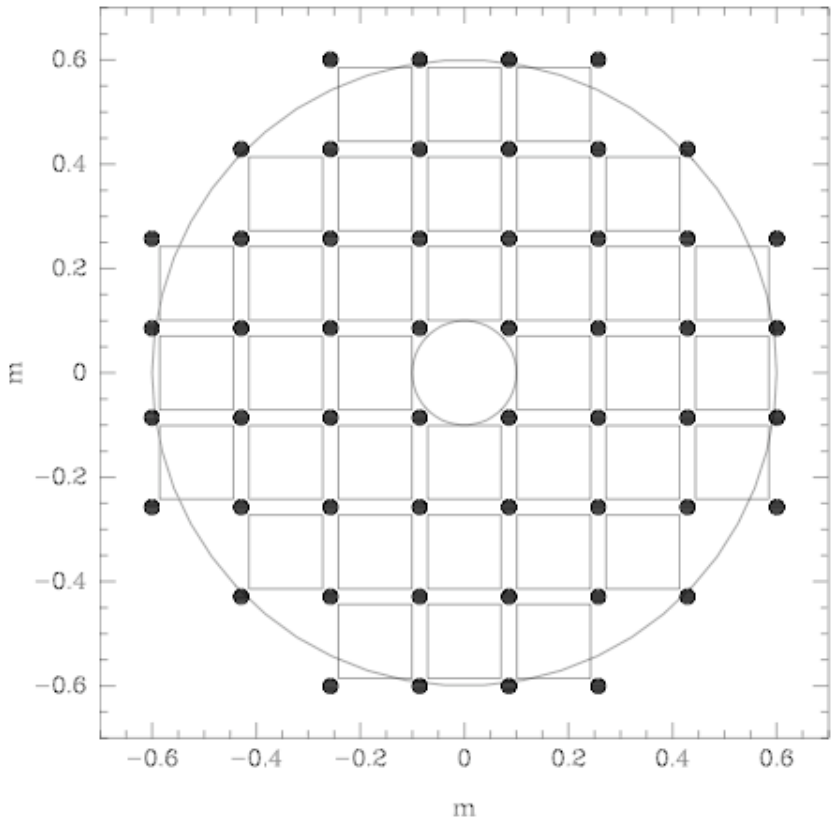


Figure 20: Relative geometry of actuators (black circles) and subapertures (squares) scaled to the entrance pupil of the telescope (primary mirror). The outer circle is the primary mirror obstruction, defining the edge of the pupil; the inner circle is the central secondary mirror obstruction, blocking light from reaching the central subaperture.

3.5.1 Simulation

A critical step in designing and building a modern astronomical instrument is to conduct thorough simulations of its operational stability and performance. This step in the VOLT design process needed to be rigorous to ensure that the expectations of the open loop AO system could be met with the technologies and budget available to us. Simulations of the VOLT system were run using three different AO simulation software packages – LAOS, PAOLA, and CAOS – and their results provided a confidence check.

Code for Adaptive Optics Systems (CAOS)

CAOS is an IDL-based AO simulator with a modular 'AO building block' graphical user interface (see Figure 21) which makes simulated system build-up easy and intuitive. CAOS can simulate atmospheric turbulence effects, wavefront sensing, and DM correction, and includes a sophisticated physical modeling package. CAOS interprets the graphical AO system layout as defined by the user and from it automatically generates IDL simulation code. This is an advantage, as it separates the user from the low-level coding details inherent to many simulations and allows more time to be spent addressing AO optical and control issues rather than simulation implementation details. But should the user need to customize the model, CAOS allows for modification of the code and/or inclusion of user-written modules (Carbillet et al., 2004), [22].

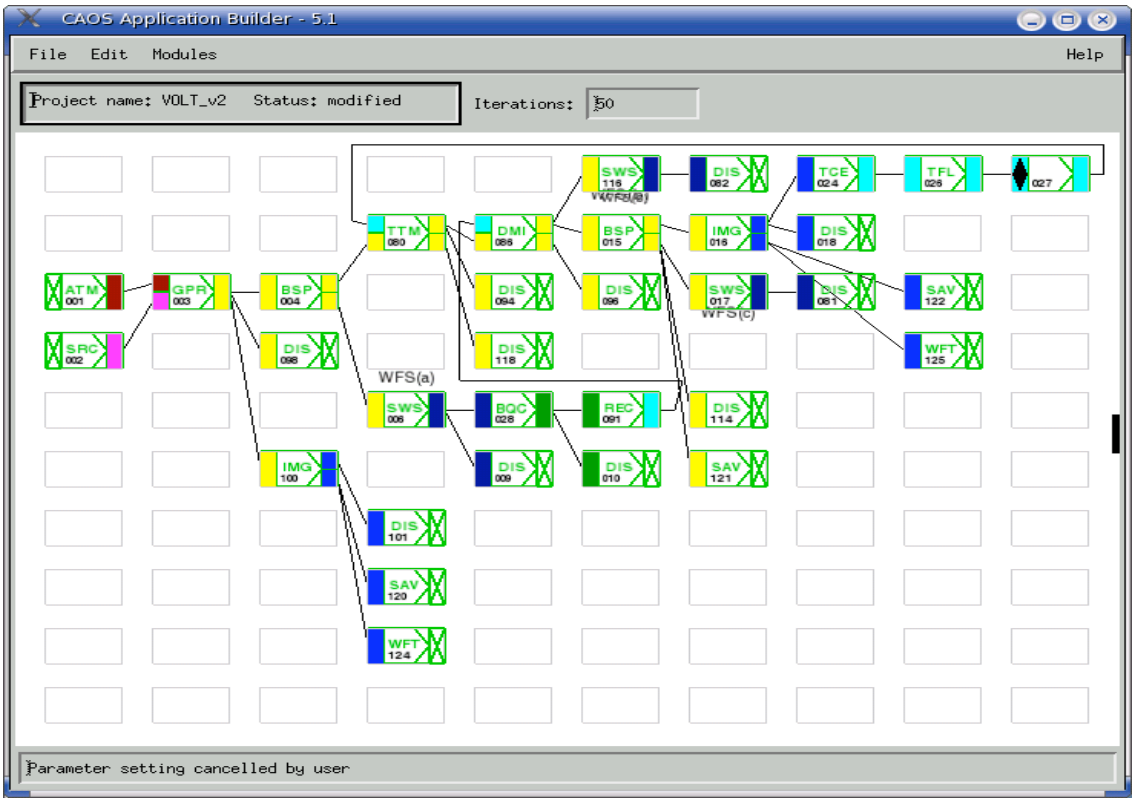


Figure 21: The VOLT open loop AO system simulation structure as built in CAOS [22].

Performance of Adaptive Optics for Large Apertures (PAOLA)

PAOLA (Jolissaint, Véran 2002; Jolissaint, Véran, Conan 2006) was developed at HIA. It simulates the effect that AO correction has on the turbulent phase power spectrum and solves for the power spectrum of the residual phase after correction, from which the science image PSF can be derived. Recalling Equation 4, PAOLA can incorporate the effects of WFS noise and spatial aliasing (σ_{WFS}), DM fitting error ($\sigma_{fitting}$), control loop temporal error ($\sigma_{temporal}$), and isoplanatic error ($\sigma_{isoplanatic}$) – all the large sources of WFE.

The analytical approach of PAOLA is computationally efficient, meaning quick simulation turnaround time. However, it is limited to simulating the first order effects on AO correction only. But this fast first-order exploration of the AO system parameter space can be very useful in narrowing the list of possible solutions to the few that warrant deeper analysis with a more powerful tool like LAOS (see Section 2.1) [23].

Simulation Parameters and Results

The following atmospheric assumptions about the DAO 1.2 m telescope site were used in the LAOS and PAOLA simulations of VOLT:

- atmospheric seeing range corresponding to r_0 values between 2.5 cm (poor) and 10 cm (excellent for DAO)
- 20 m/s wind velocity

The results of the CAOS, PAOLA, and LAOS simulations agreed very closely [22, 23]. All showed that WFE would be dominated by temporal error if operating below control loop bandwidths of 1 kHz, and calculated an RMS WFE of ~140 nm with a 39% Strehl ratio at r_0 = 5 cm and λ = 900 nm (I-band) at this minimum 1 kHz bandwidth for a bright star (negligible WFS noise). Dropping the bandwidth to 500 Hz or 200 Hz leads to a large predicted increase in WFE to 217 nm and 400 nm, respectively. At 1 kHz, even in seeing conditions as bad as r_0 = 3 cm, the simulations predicted VOLT should be able to produce a science image PSF with a diffraction limited core with ~11% Strehl ratio in the absence of WFS noise (see Figure 22) [23].

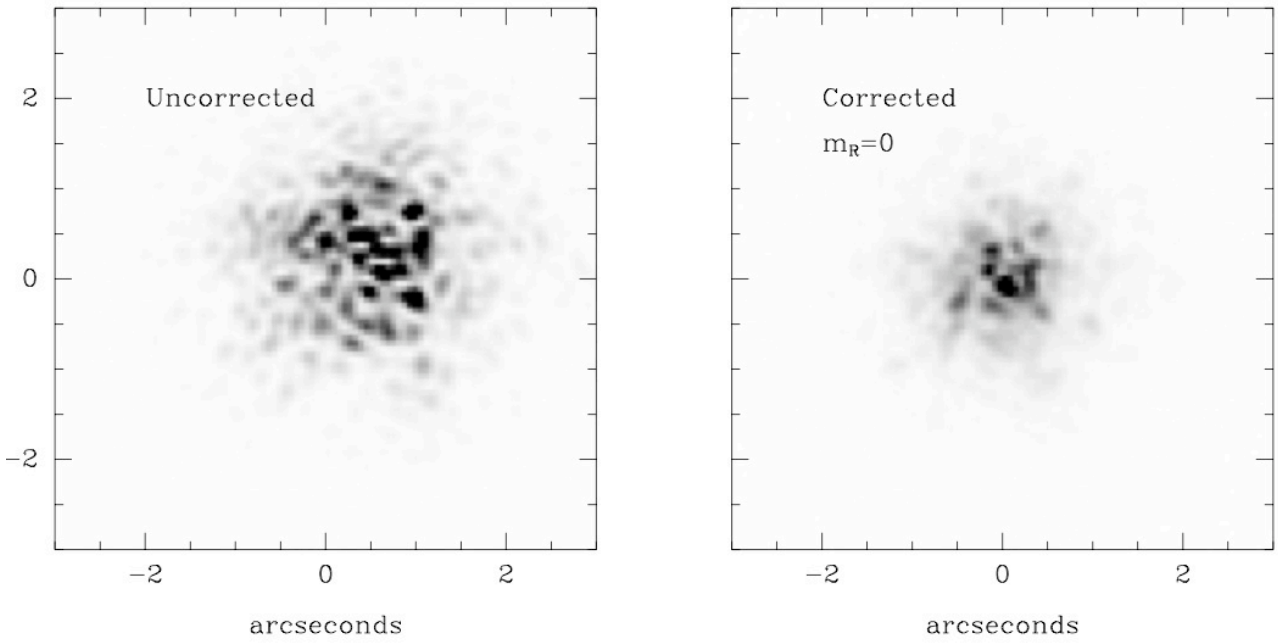


Figure 22: CAOS simulations of VOLT for a m_R = 0 star (Arcturus is close with m_R = 0.3) for r_0 = 4cm. Here WFS noise is included assuming 230 e^- of read noise for a m_R = 0 star with a system efficiency of 2.4%[4].

[4] Initial simulations assumed 40 electrons of read noise and 6% throughput. This simulation (originally for a m_R = 3 star) was scaled to match the SNR we expected on the open loop WFS based on our revised throughput and read noise estimates, described in Section 3.5.2 – Camera Subsystems.

The DM simulation then consisted of evaluating whether the ALPAO DM52 would have enough actuator stroke to correct the predicted turbulence at the DAO 1.2 m telescope site. The non tip-tilt RMS WFEs were calculated at different values within the range of atmospheric seeing for the site and fell well within the range correctable by the DM with its maximum stroke of 25 μm (see Section 3.5.2 – Deformable Mirror Subsystem), as shown in Figure 23 [23].

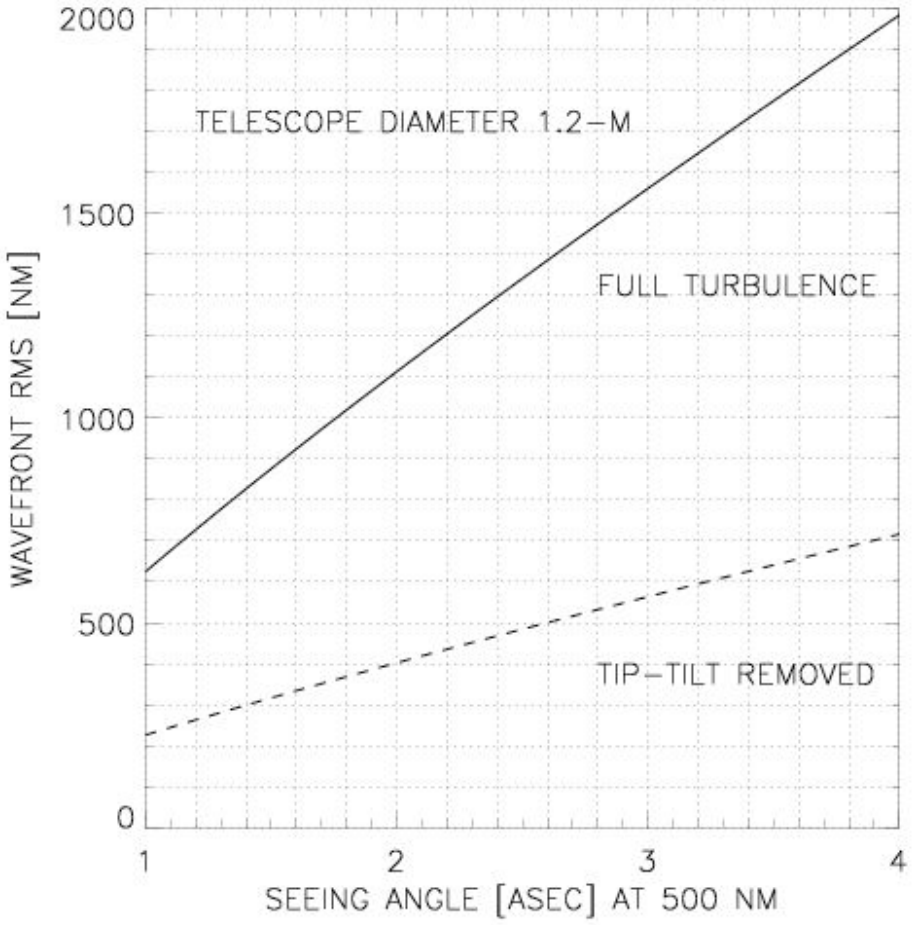


Figure 23: The RMS WFE simulated for the DAO 1.2 m telescope site. WFE as a function of atmospheric seeing, including tip-tilt error, is shown with a solid line; WFE with tip-tilt removed is shown with a dashed line. It is clear that tip-tilt error is the dominant source of atmospheric turbulence WFE (Jolissaint, 2007).

The ideal open loop WFS FOV and pixel scale – the ratio of angle on the sky to pixel size and spacing on the detector – also needed to be determined to provide enough dynamic range for the full turbulence WFS spots. There is a limit to the resolution at which the wavefront subapertures are sampled, above which no more performance gains are realised and the system is just reading out – and calculating centroids from – more pixel data than needed. Centroid computation and camera readout time contribute as limiting factors on control loop bandwidth. These simulations, however, did not account for any of the open loop sources of error we expected to encounter when operating VOLT. We make our best attempt to measure these open loop wavefront errors in the coming sections.

3.5.2 VOLT System Design

The VOLT system consists of passive optics and sensors as well as active hardware components with three control computers equipped with specialised interface cards and a suite of real-time software to control them. These three computers control three wavefront sensors – WFS A, WFS B, and WFS C, all 7 x 7 subaperture Shack-Hartmann type WFSs – and the ALPAO DM52 deformable mirror, with 52 actuators arranged in an 8 x 8 square grid. A fourth computer is dedicated to running the science camera. The interconnections between the active components of VOLT are shown in Figure 24; the optical layout of VOLT is depicted in Figure 25 and Figure 26. To keep costs down, we constrained the optical design of VOLT to use commercially available lenses and reuse two lenslet arrays available at HIA. A detailed description of all the passive optics can be found in [23].

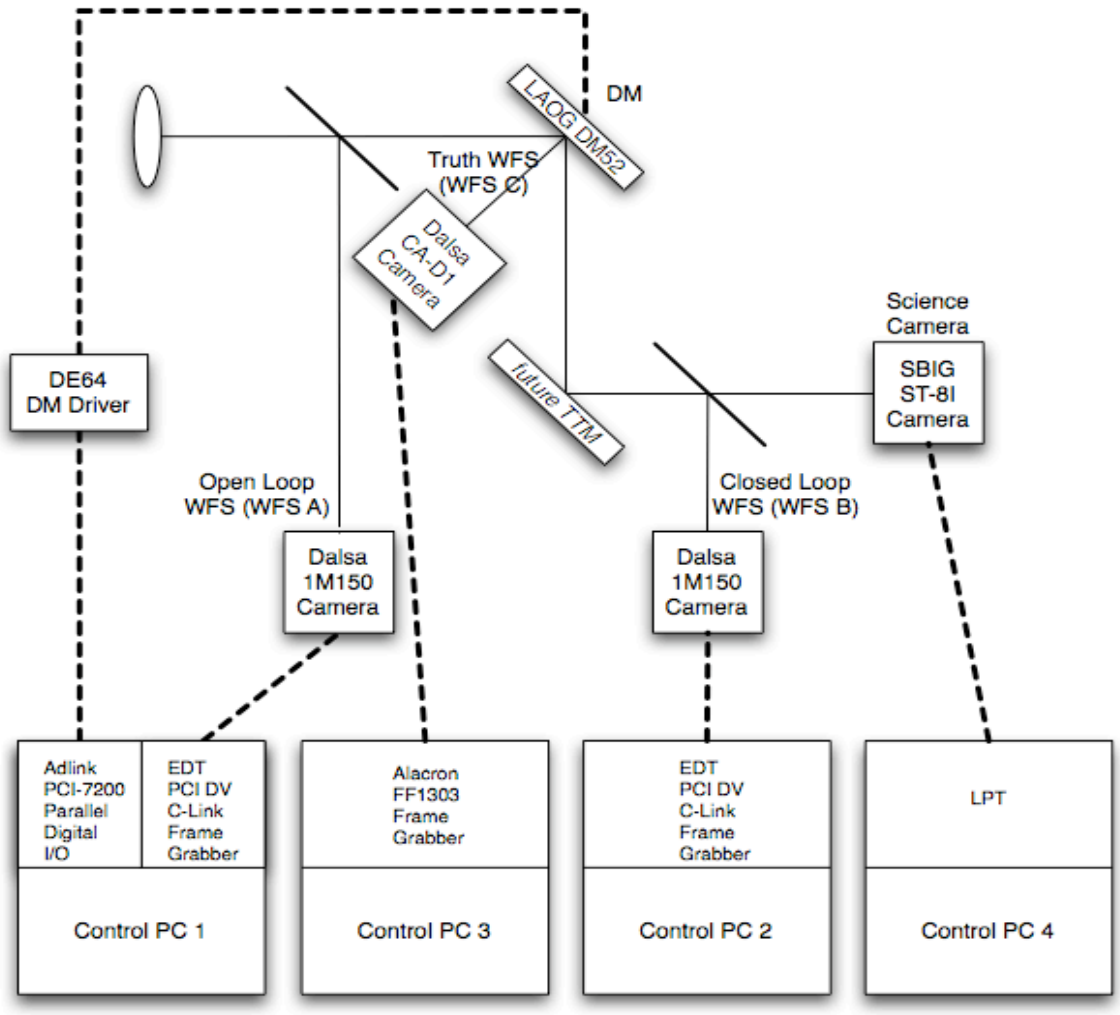


Figure 24: The VOLT system architecture. The control PCs with installed interface cards are shown with their connections to the active hardware, around a rough schematic of the optical layout showing the input at top left of a collimated beam formed by a lens just after telescope focus. The element labeled *Future TTM* refers to the possible insertion of a tip-tilt mirror – a mirror mounted on a platform that can be controlled to tip and tilt in the plane of the mirror surface at high bandwidths – in case DM stroke is an issue. A tip-tilt mirror removes the low-order / high amplitude mean global wavefront tilt Zernike term, leaving the DM to correct the remaining higher-order Zernike terms (see Appendix A).

Optical Design

The 1.2 m telescope provides a beam at an image scale of 4.88 arcsecond/mm (f/34.6) through a pinhole opening into the Coudé instrument lab, giving an entrance aperture at focus of 8 arcseconds in diameter (1.64 mm). Referring to Figure 25, a 45° mirror diverts

the beam onto the VOLT bench (this mirror is left out when VOLT is not in use to allow the beam to pass to the spectrograph, the normal operating mode instrument of the 1.2 m telescope). The beam then passes through a collimator and to a beamsplitter which directs 70% of the light to the open loop WFS A. WFS A is comprised of a number of optics and a camera: two lenses and a field stop (pinhole) re-collimate a 7.6 arcsecond FOV beam onto an square grid of 7 x 7 lenses called a lenslet array, and a final lens reimages the focal spots produced by the lenslets onto a Dalsa 1M150 CMOS detector, described in Section 3.5.2. The pixel scale of WFS A – the ratio of angle on the sky to pixel size and spacing on the detector – is 0.485 arcseconds/pixel, with 16 pixels/subaperture (7.8 arcsecond FOV). Referring to Figure 27, a sample image from WFS A is given on the left and one from WFS B (explained shortly) is on the right.

The remaining 30% of the beam is reflected off the DM, which provides the real-time correction of the wavefront. The reflective surface has a diameter of 17.6 mm and the actuator spacing is 2.5 mm. The layout and positioning of the actuators within the aperture relative to the WFS lenslets is shown in Figure 20. After the DM, the beam reflects off a stationary flat mirror (this would be the tip-tilt mirror location in a future VOLT upgrade if deemed necessary) and is split again into two beams by a beamsplitter, 96% of the light enters the residual, or 'scoring', WFS B and the remainder is directed to the science camera. WFS B can be used with the DM in classical AO closed loop operation, but WFS B is primarily used to register WFS A to the DM and to measure the residual wavefront errors in open loop operation.

The optics of WFS B serve a similar function to those of WFS A; two lenses and a field stop re-collimate a 3.9 arcsecond beam onto a lenslet array of 19 x 19 pixel subapertures and 0.15 arcsecond/pixel plate scale (3 arcsecond FOV) to create focal spots on a second Dalsa 1M150 detector, noting from Figure 27 that the WFS B spots take up a larger area within their subapertures than those of WFS A. While the WFS B spots are oversampled compared to WFS A (3 times smaller pixels in terms of arcseconds), there is little advantage in terms of centroiding accuracy because the diffraction-limited spot is close to 1 arcsecond across. The fine pixel scale of WFS B was a compromise made so we could reuse an available (and costly) lenslet array. This is because WFS B only ever sees the corrected wavefront, downstream of the DM, and thus the spot motions are confined to small moves caused by the low residual error.

The remaining light in the science camera is focused with a single lens onto a SBIG ST-8I CCD with a plate scale of 0.053 arcseconds/pixel (λ/D = 0.9 μm / 1.2 m = 0.15 arcseconds).

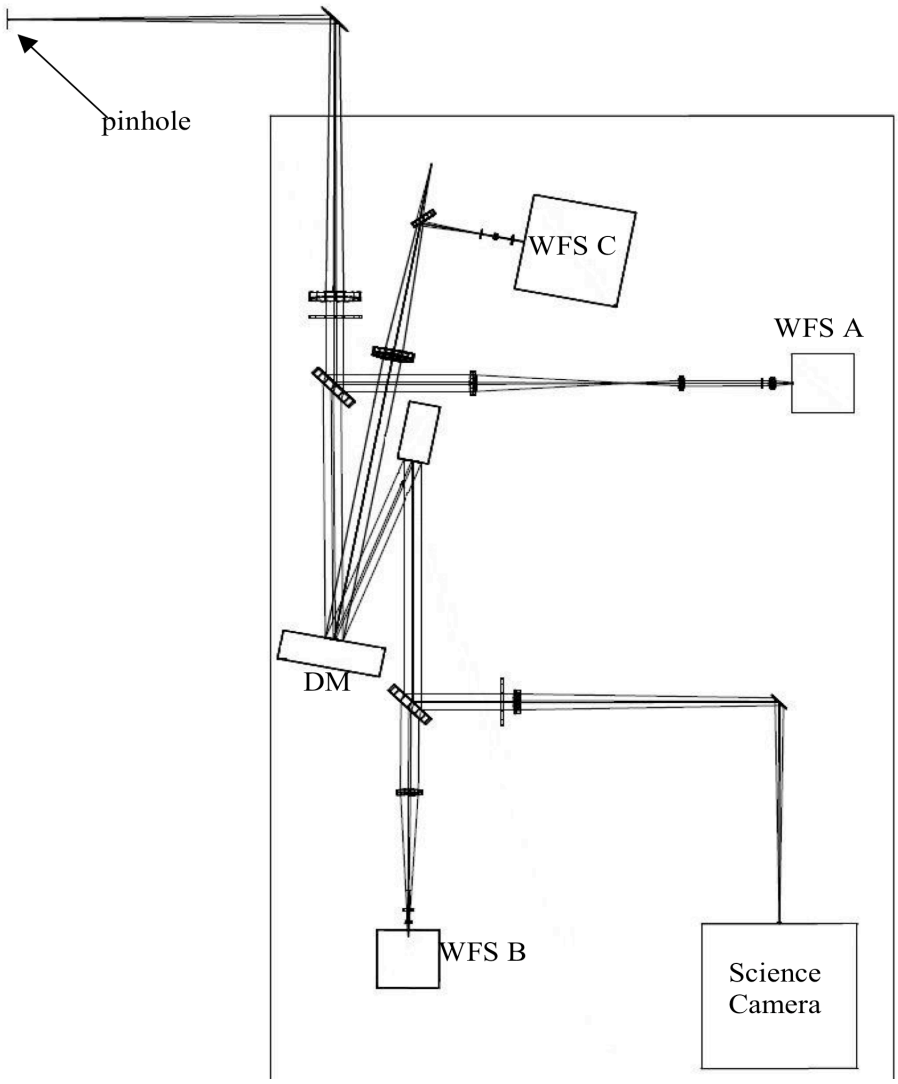


Figure 25: The VOLT optical layout. Light from the 1.2 m telescope enters the Coudé instrument lab through the pinhole at the upper left and is picked off by a 45° mirror and directed onto the VOLT bench. From there it is collimated and split by a beamsplitter between the open loop WFS A (70% of the light) and the DM. The beam reflecting off the DM then undergoes another reflection before encountering another beamsplitter that feeds the scoring WFS B (with 96% of the remaining beam) and the science camera.

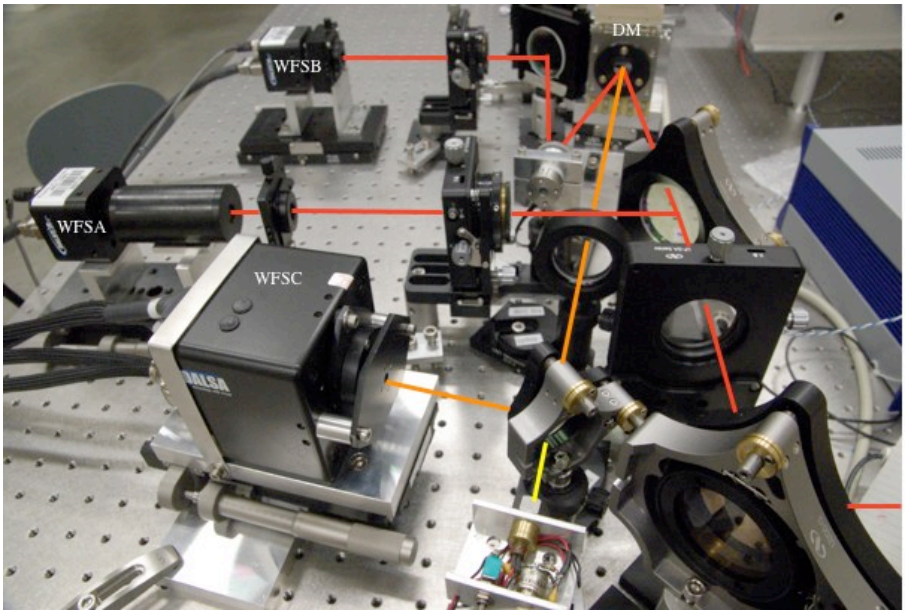


Figure 26: Photograph of VOLT in the integration laboratory, with lines illustrating the optical path. After the pick-off mirror and the collimator, a beamsplitter directs light into the open loop WFS A arm of VOLT. The light that passes through the beamsplitter encounters the DM, another fold mirror and a second beamsplitter which divides the light between the scoring WFS B and the science camera (not pictured). The truth WFS C has an independent optical path with its own light source and can be used to monitor the DM shape. The layout matched Figure 25 in the final setup in the Coudé room of the 1.2 m telescope.

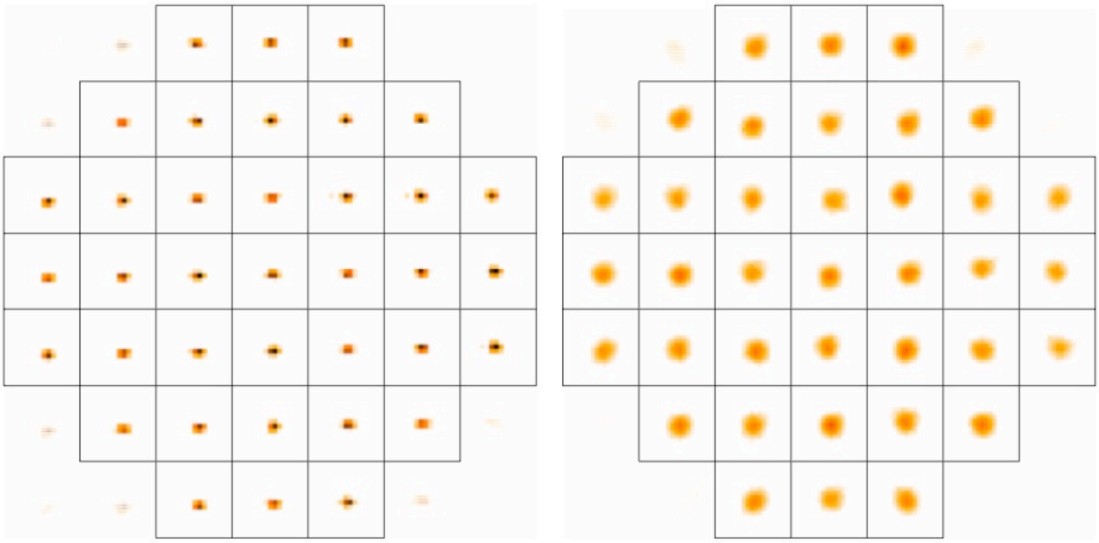


Figure 27: Sample images taken from WFS A (left) and WFS B (right) on the VOLT testbench using the calibration source. The centre subapertures are blocked by the secondary mirror when VOLT is on the telescope, and thus would have no spots as shown here. There is no central obscuration with the artificial light source in the lab.

VOLT has a third WFS, WFS C, the 'truth' WFS – so named because it sits outside of the main beam path but has a face-on view of the DM that allows it to get an independent measurement of the DM surface shape. Light from a laser diode is transmitted through a beamsplitter and collimated by a lens toward the DM, reflects off the DM back into WFS C. The light coming to focus is picked off by a beamsplitter and reflected 45° toward a recollimator and lenslet array as on WFS A and WFS B, sampling a grid of 7 x 7 subapertures of 12 pixels across and a comparable plate scale of 0.22 arcseconds/pixel (2.6 arcsecond FOV). WFS C uses a Dalsa CA-D1 CCD detector, described in the next section.

Camera Subsystems

The Dalsa 1M150 CMOS camera has a 1024 x 1024 array of 10.6 μm pixels, of which a 160 x 160 pixel region of interest (ROI) enclosing the 7 x 7 array of subapertures of WFS A or WFS B is read out at 1 byte monochrome resolution (or pixel depth). Smaller ROI readout allows running the camera at higher frame rates.

The 1M150 CMOS cameras have a relatively low pixel fill factor of 35%. This is the measure of the fractional area of the pixels that is sensitive to light – the photodiode – which is surrounded by electronics. This of course results in a loss of sensitivity, as well it raises the question of how a low fill factor influences WFS centroiding. We produced a simple simulation of a Gaussian PSF and determined the RMS centroiding error (measured relative to FWHM) resulting from the random registration of the PSF to the integral pixel array. For our centroiding calculations, a threshold of 50% the peak intensity was applied, and a simple weighted centroid was calculated (see Section 3.5.3

for centroiding algorithm details). As shown in Figure 28, it appears that the accuracy to which centroids can be calculated from a Gaussian is related to the size of the active pixel area, not the overall size of the pixel. Therefore, it was determined that a lower fill factor would not adversely affect the VOLT centroiding algorithm [23].

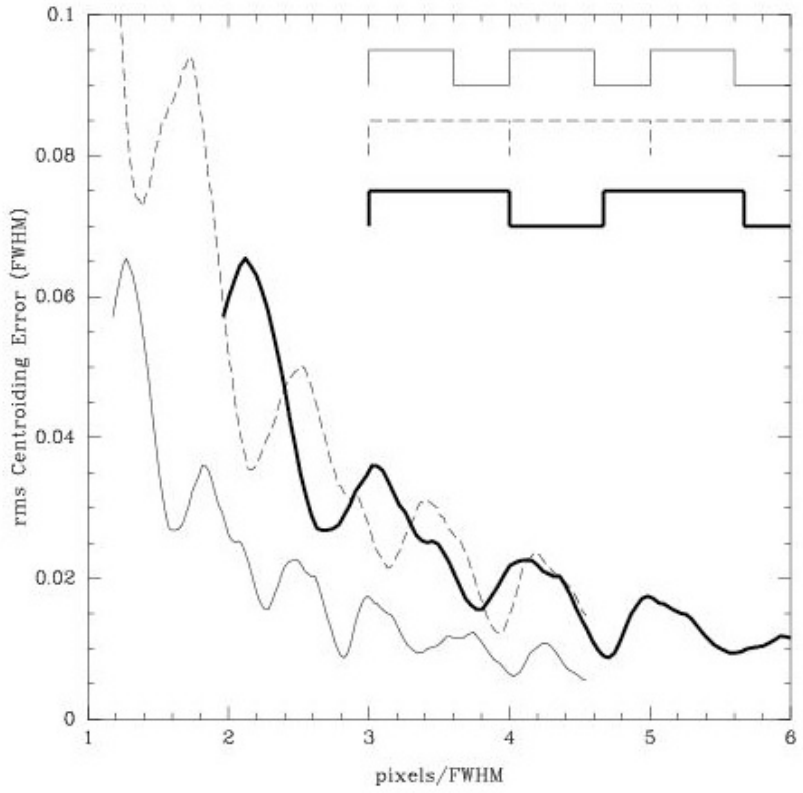


Figure 28: The solid thin line shows the centroiding error as a function of numbers of pixels across the FWHM for pixels with a 35% fill factor. The dashed line shows the same relation if 100% fill factor pixels are considered. Finally, the heavy solid line shows the centroiding error that results if 35% fill factor pixels with the same active area as the 100% fill factor pixels are used. The agreement between the heavy solid line and the dashed line shows that it is the active area of the pixels that is important – not the space between pixels. The bumpy shape of the curves is a direct result of the thresholding function (see Section 3.5.3) [23].

While the fill factor did not adversely affect VOLT performance per se, it did contribute substantially to the overall low system throughput. The throughput of our

optical system – from the telescope primary mirror to the WFSs and science camera – was originally assumed to be better than what we found when VOLT was integrated into the DAO 1.2 m telescope. The reflectances of the telescope optics degrade over time with unavoidable dust particle deposition; the choice of reflective surface type (typically silver or aluminum) plays a role as well. Losses from optics within the VOLT system were also significant, particularly from the beamsplitters. This was improved through switching in new beamsplitters optimized to allow 40% more light to WFS A and 15% more light to WFS B, resulting in a reduction by 20 times to the light reaching the science camera (which just means longer exposures need to be taken). These factors combined to significantly affect the number of photons reaching WFS A and B, and thus the sensitivity of our centroiding (especially on WFS B as it is further downstream in the system and suffers losses from additional light reflections and transmissions compared to WFS A; see Figure 25). The total light losses are summarised in Table 3.

Photon Losses	**Transmission (%)**
Atmosphere	80
Telescope	23
VOLT optics up to WFS A	50
WFS camera pixel fill factor	35
Quantum Efficiency (QE)	75
Total System Throughput (%)	2.4

Table 3: The combined telescope and VOLT optical system throughput is calculated to be 2.4% by multiplying the fractional light losses of the system optics together [24].

In addition to a low system throughput, the read noise of our 1M150 cameras was higher than expected. We measured the read noise using a simple setup including a laser diode, an integrating sphere, a photo counting diode and the 1M150 camera. A red (λ = 630 nm) laser diode, normally used as our calibration source, fed light into an integrating sphere. We placed either the sensitive photon counting diode or the 1M150 camera directly at the output of the integrating sphere. Using the photo diode we measured a flux of 0.38 mW/cm^2. 1M150 camera frames were taken at a number of different exposure times. The mean and RMS light levels were calculated for all exposures, from which we found that the digital data number (DN) counts were well-fit by the simple linear relation $DN = 17.5 + 0.78t$ where t is time in milliseconds. Subtracting the 17.5 DN offset, we could then relate the number of DN to the number of photons using the flux of the photo diode which we in turn convert to electrons using a detector QE of 70% [24][5]. We then determined the read noise of 230 e^- by assuming the RMS light level (in electrons) would be proportional to the square root of the mean light level (in electrons)[6] plus the read noise squared (Figure 29).

The control and data transfer interface for the 1M150 is the industry standard CameraLink – common among many modern CCD and CMOS cameras – which includes lines for RS232 serial communication, that is also compatible with older 9-pin computer serial ports (with appropriate adaptor cabling), for camera configuration via direct setting of hardware registers. The protocol, as described in Table 4 uses a set of low-level

---

[5]The ratio of DN to flux was also consistent with the quoted sensitivity in the 1M150 manual [24].

[6]This assumes Poisson statistics.

hexadecimal commands that indicate which register to read or write to and the values to write. These commands are relatively easy for a programmer to generate [25] and code into the initialisation routines of the control system, or to issue via a command line serial communication utility application [24]. A non-exhaustive list of configuration parameters and associated serial commands used in setting up the 1M150 camera of WFS A is given in Table 5 (there is a large number of registers and low-level settings to deal with that need not be included here, but can be reviewed in [24]).

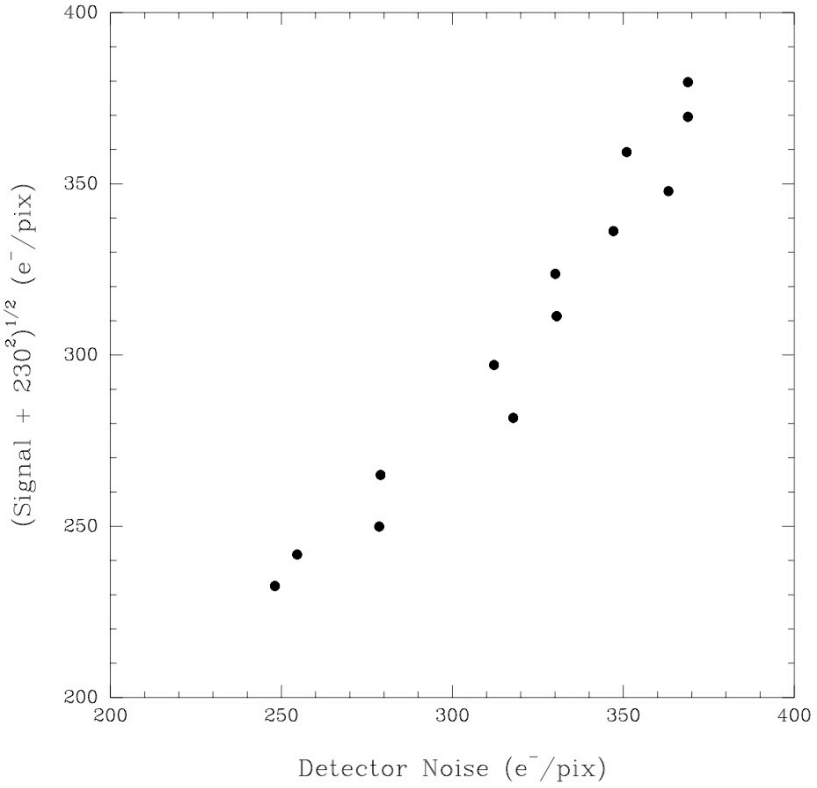


Figure 29: The signal with read noise (added in quadrature, however, the signal itself is not squared as it conforms to a Poisson distribution) as it relates to detector noise for different exposure times.

	Data Bits							
Command to camera controller	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
WRITE Address	0	1	A5	A4	A3	A2	A1	A0
WRITE Data Low Nibble	1	0	x	x	D3	D2	D1	D0
WRITE Data High Nibble	1	1	x	x	D7	D6	D5	D4
READ Data of Addresses	0	0	A5	A4	A3	A2	A1	A0

x: don't care
Ai: Address bits
Dj: Data bits

Table 4: Dalsa 1M150 camera register RS232 serial communication protocol [24].

Camera Parameter	**Default Value**	**RS232 Hexadecimal Code**
Exposure Time	1.2 ms *	`51 8x Cx 50 8x Cx 4F 8x Cx`
Line Preload, Edge Triggered	Enabled, Enabled	`4D 8A C2`
Line Pause	8	`60 88 C0`
ROI X_0 pixel	408 †	`59 81 C0 58 88 C9`
ROI Y_0 pixel	386 †	`5B 81 C0 5A 82 C8`
ROI X_1 pixel	567 †	`5D 82 C0 5C 87 C3`
ROI Y_1 pixel	545 †	`5F 82 C0 5E 81 C2`

*, x: exposure time depends on star magnitude and optical system throughput
†: ROI coordinates may vary with optical alignment drift over time, but size is fixed at 160 x 160

Table 5: The 1M150 camera configuration for WFS A.

A framegrabber is used to interface each 1M150 with its host computer, also conforming to the CameraLink protocol. Originally the VOLT Control PCs were equipped with three Alacron FF1303 framegrabbers to interface with all three WFSs. The FF1303 has a peak data rate of 132 MB/sec with Direct Memory Access (DMA) – the ability to write/read directly to/from the memory of the host PC without requiring CPU involvement – transfer over a 33 MHz PCI bus, easily satisfying the data rate requirement for VOLT at 1000 frames per second, calculated as follows:

160 pixels × 160 pixels × 1 byte × 1000 frames/sec = 25.6 MB/second

The FF1303 I/O interface can be configured by programmable FPGAs to suit a particular interface or camera, and there is also a programmable 200 MHz Nexperia (TriMedia) processor on-board with 32 MB of RAM [26]. This makes it capable of carrying out some WFS data preprocessing. VOLT was to exploit this feature by doing onboard WFS centroiding, allowing for the offloading of a significant amount of the computational burden from the host CPU and a vast reduction in the amount of data to be pushed across the PCI bus from the framegrabber to the host computer (only the x and y WFS slope pairs for each subaperture would need to be passed to the host every cycle instead of an the whole camera ROI frame). Unfortunately, the FF1303 framegrabbers proved to be extremely problematic and time consuming to integrate into VOLT and were unreliable during operation to the point that it was decided replacement framegrabbers needed to be found. The FF1303 would often get stuck in a state where the frame buffer would get locked – preventing new incoming image data from being written – and its image data skewed, resulting in garbled images being sent to the centroiding algorithm that, of course, caused the VOLT control system to fail. However, the error of zero measured illumination across a lenslet, as these bad frames would often be interpreted, was programmed to be caught by the real-time controller before allowing new errant DM commands to be generated, but could not be counted on to catch every such case, as even garbled data can be put through a centroiding algorithm and produce what appear to be reasonable results. Also, the FF1303 and drivers did not live up to their advertised Linux compatibility. Months were spent trying to get correct driver / Linux kernel version compatibility matches. The necessary support – configuration files

and expertise – for our Dalsa 1M150 cameras was claimed to exist, but in fact did not, and the needed configuration files were only produced after a process of mutual debugging lasting weeks. The replacement for the FF1303 was the Engineering Design Team (EDT) PCI-DV C-Link framegrabber. Although the EDT framegrabber did not offer onboard WFS preprocessing capability, it did more than meet the data throughput requirements with a theoretical peak bandwidth of 220 MB/second with DMA transfer (we measured 190 MB/second with our hardware), but did require an upgrade of Control PC 1 and 2 to motherboards that were equipped with 66 MHz PCI buses. Additionally, the EDT framegrabber came with well-supported Linux drivers and open source libraries and well-documented sample code, as well as very helpful setup and diagnostic utility applications. These were valuable assets during software development, especially under the tight time constraints that came with integrating this totally new hardware into the VOLT system shortly before the final (and successful) on-sky test.

WFS C uses a Dalsa CA-D1 CCD camera with 128 x 128 resolution and 16 µm pixel pitch and can satisfy the required 800 frames/second rate with 2 x 2 on-chip binning [27], a technique which synthesizes 'super pixels' by grouping the collected charge of four mutually adjacent pixels together. This results in a higher signal-to-noise ratio (SNR) for the binned super pixel than that for any single pixel, as the binned pixels are read out at the same time, resulting in the read noise being applied only once:

$$SNR_{NoBinning} = \frac{PQ_E t}{\sqrt{PQ_E t + Dt + N_R{}^2}} \qquad (7)$$

where P is the incident photon flux (photons/pixel/second), Q_E the CCD quantum efficiency, t the exposure time (seconds), D the dark current (electrons/pixel/second), and N_R the read noise (electrons rms/pixel).

$$SNR_{Binning} = \frac{MPQ_E t}{\sqrt{MPQ_E t + MDt + N_R{}^2}} \tag{8}$$

where M is the number of pixels in the binned subaperture [28].

The CA-D1 is connected to its host computer by a RS422 interface provided by a slightly different version of the FF1303 framegrabber, which has not given us the reliability problems of the CameraLink versions of that board, so it was not replaced.

The SBIG ST-8I used as the science camera has a detector array of 1530 x 1020 pixels, of which a 152 x 152 subregion corresponding to an 8 arcsecond FOV is read to measure the science image after AO correction. The ST-8I is a relatively old and slow camera commonly used by amateur astronomers and is connected to its host computer via the parallel printer port. However, it is adequate for our purposes as it is not required to have a high readout rate, is responsive in the I-band (λ = 900 nm) and only needs to take infrequent exposures on the order of seconds to gather enough light to demonstrate the AO image correction.

Deformable Mirror Subsystem

As mentioned, the deformable mirror employed in VOLT is the Laboratoire Astrophysique de l'Observatoire de Grenoble (LAOG) DM52. Its 52 voice coil actuators are arranged inside a 17.6 mm diameter circle inscribed within an 8 x 8 square grid (ie. there are no corner actuators; recall Figure 20), evenly spaced 2.5 mm apart, underneath a thin flexible membrane with a reflective silver coating (see Figure 30). Characteristics that make the DM52 suitable for open loop operation are its relatively large actuator stroke of 25 μm and integrated tip-tilt (a functional normally left to a separate tip-tilt mirror), very smooth (if broad) influence functions, and high linearity with no hysteresis.

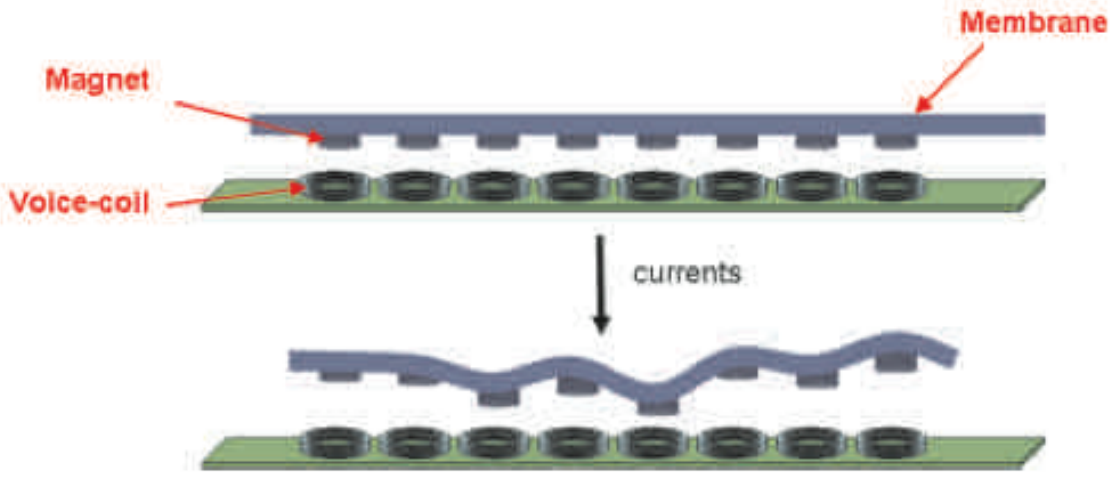


Figure 30: A cartoon illustration of the ALPAO DM52 voice coil actuator deformable mirror architecture [29].

The linearity, or 'go-to' characteristics, of the DM refers to its ability to repeatably send any given position command to an actuator (a voltage within its safe operating range) and have it move with high accuracy to that desired position. Voice coil actuators, the type used in the DM52, are well-suited where a high degree of linearity is needed, contrasted to the majority of conventional DMs that use piezoelectric stacks for actuators

and have a significant amount of inherent hysteresis and thus require closed loop feedback in order to converge. A WYKO interferometer [30] was used to provide highly accurate optical measurements of the DM surface shape in the lab. The DM actuator linearity, as a conversion from amplifier volts to microns of stroke, was then determined for all actuators, as shown in Figure 31. This also allowed us to map the influence function of each actuator, which refers to the effect that moving a particular actuator (pushing or pulling against the tension of the flexible membrane) has on the overall shape of the DM, as illustrated in Figure 32.

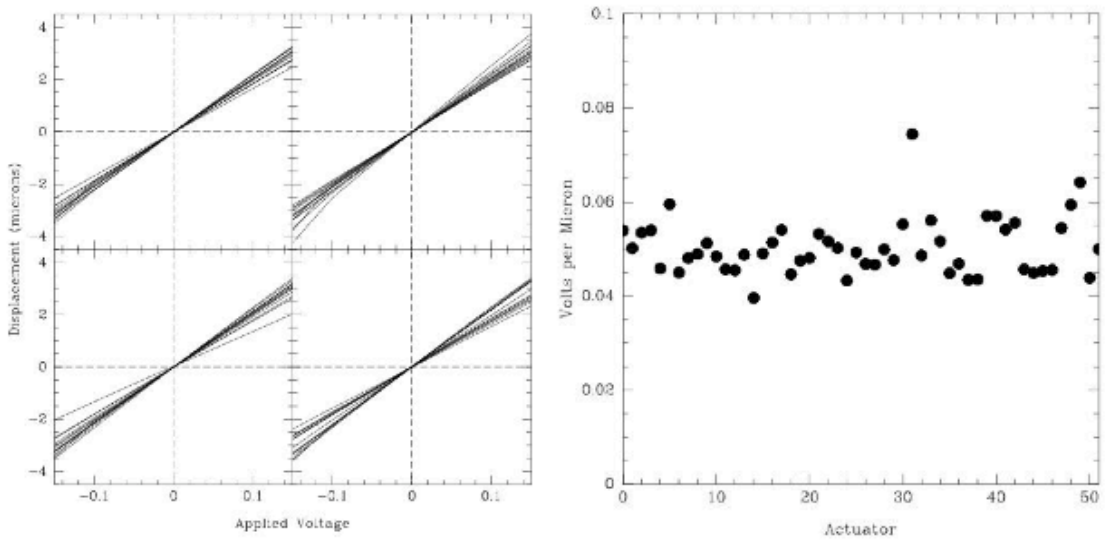


Figure 31: Left: Plots of the linear volts-to-micron relationships for the 52 actuators on our ALPAO DM52. Right: The mean volts-to-micron relationships for each of the 52 actuators.

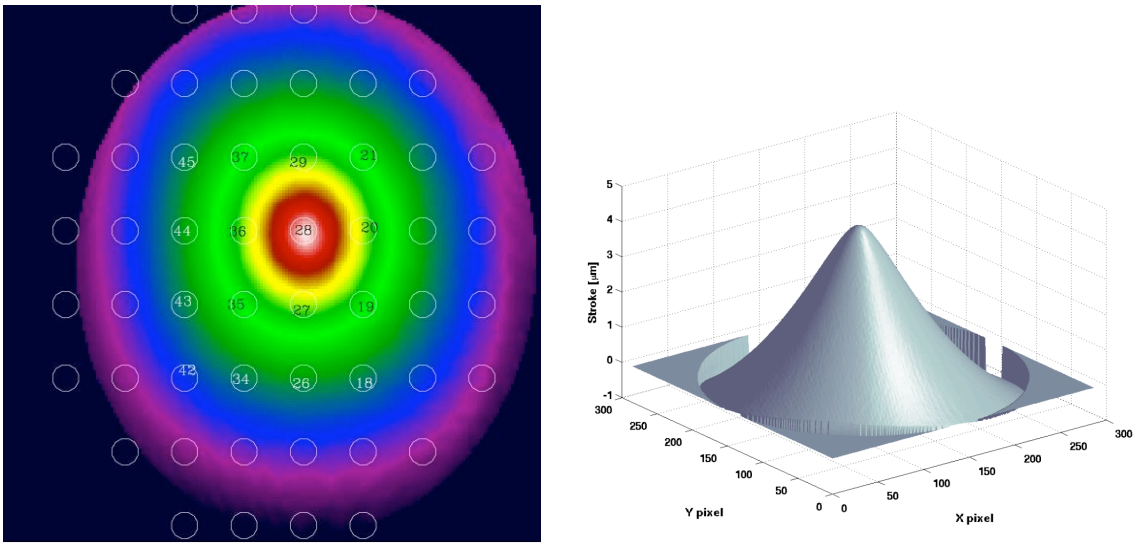


Figure 32: Left: The influence function for one DM actuator (#28). The colours show the relative height of the DM surface outward from the actuator, of the same form as that shown on the right (C. Blain 2007).

After mapping the influence functions, simulated atmospheric turbulence aberration maps[7] (phase screens) were projected onto the basis of these influence functions to generate the optimum shape the DM should take on in order to correct for such turbulence. The DM was then commanded to take on this shape, and the remaining difference between the optimum shape and that measured by the WYKO interferometer showed the open loop error, σ^2_{OL}, – a combination of non-linearities and hysteresis – of the DM to be quite small at ~10 nm RMS, making up one, practically negligible, component of σ^2_{other} in the WFE budget for VOLT, as described in Equation 4 (see Figure 33 and Figure 34) [16].

[7] Simulations of the turbulent phases screens assumed an r_0 = 5cm and a 1.2 m aperture, conditions similar to what we expected on the DAO 1.2 m telescope.

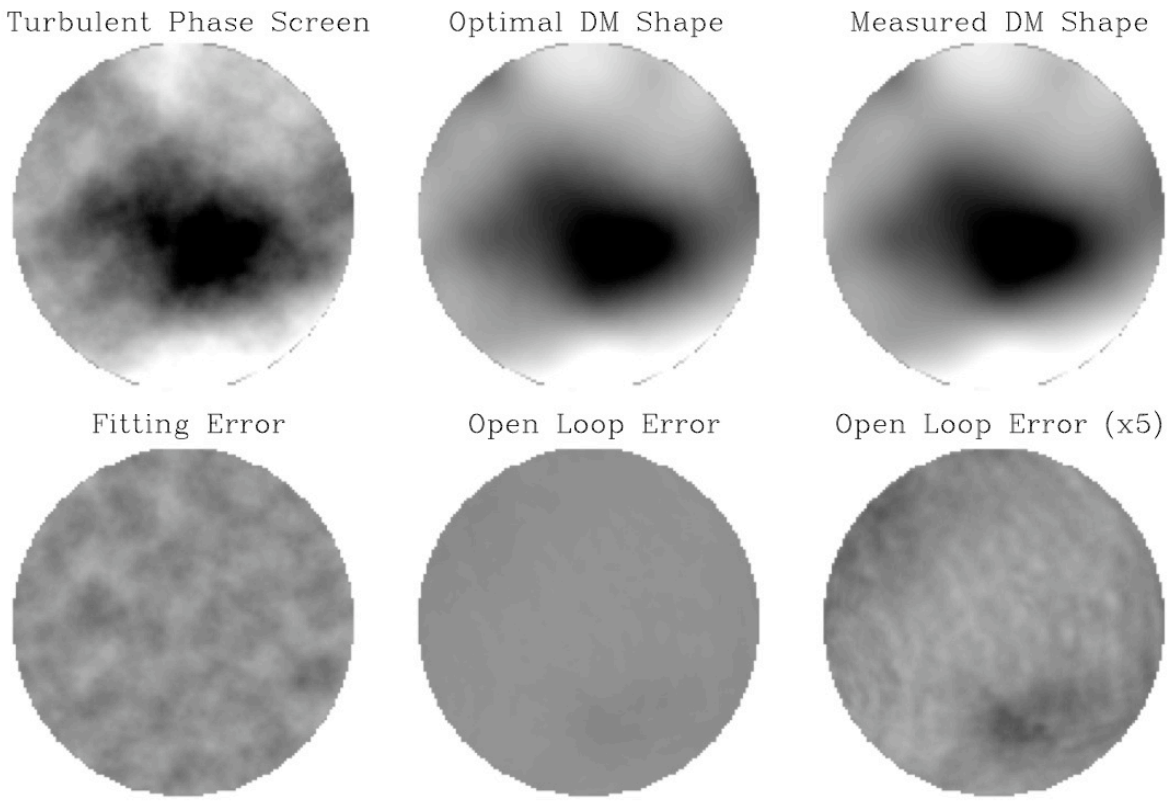


Figure 33: Top Left: Simulated phase screen corresponding to the typical observing conditions and diameter of the telescope (1.2 m). Top center: By projecting the ALPAO DM52 influence functions onto the simulated phase screen, we could determine the optimal shape of the DM and generate the appropriate DM commands. Top Right: After applying the appropriate voltages to the DM, we measured the shape of the DM with the interferometer. Bottom Left: The difference between the phase screen and the optimal DM shape is the fitting error, $\sigma^2_{fitting}$. Bottom Center and Bottom Right: The difference between the optimal and measured DM shapes is the open loop error. For the ALPAO DM52, it is roughly five times smaller than the fitting error [16].

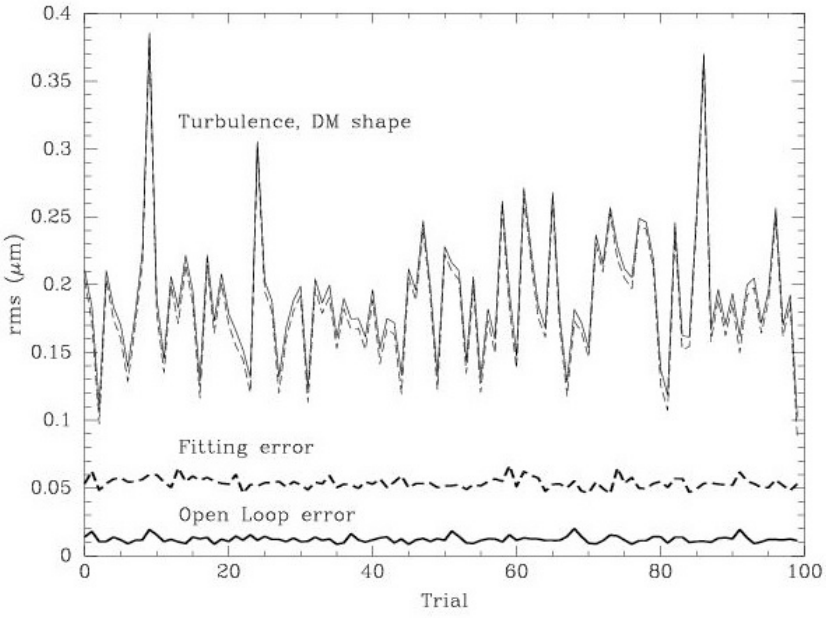


Figure 34: Plot of the standard deviation of the wavefront variations of the simulated phase screens (thin solid line) and the DM shape (thin dashed line) for 100 different realizations of $D/r_0 = 25$ turbulence. For each realization, we also calculated the fitting error, $\sigma^2_{fitting}$ (~50 nm; thick dashed line) and the open loop error (just ~10 nm; thick solid line).

We suspected that the DM might oscillate with a significant amplitude over a long duration with each shape update, adding to the VOLT wavefront error. Tests were run in the lab using an artificial light source in which the DM was commanded to take on a series of changing shapes of tip-tilt, focus, astigmatism, and coma with different amplitude steps, holding command voltages constant for at least 100 ms between steps, while WFS B recorded centroids at a high frame rate (1.4 kHz) relative to the hold time (Figure 35). We show that a large step causes significant DM oscillation for at least 40 ms, probably due to the increasing elastic resistance of the flexible membrane with bigger displacements. However, a small step caused very little DM oscillation, such that it was considered a non-issue in the overall VOLT WFE budget, as moves between control loop

iterations will be typically be small (if the system is operating fast enough to keep up with the evolving atmospheric turbulence, of course).

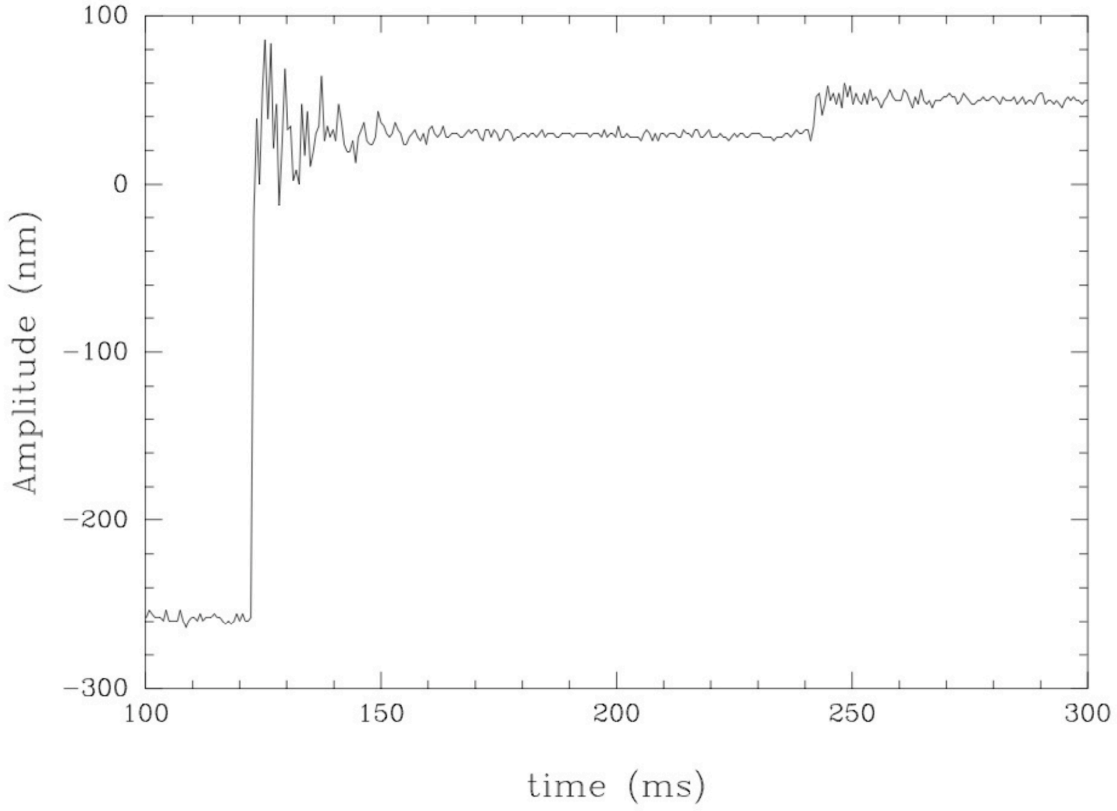


Figure 35: Plot of measured DM oscillation for two stepped shape changes. The amplitude in nanometers on the vertical axis is the displacement of a selected DM actuator computed from WFS B centroid measurements. The first step has the DM jumping from a negative focus shape (a radially symmetric depression with its lowest value at the centre of the mirror surface) to an equal and opposite positive focus shape. Large shape changes can cause high amplitude oscillations for extended periods, but small changes – as is typical from one control loop iteration to the next – do not cause significant oscillation.

However, we worried that as the open loop correction was initiated, the DM would be commanded to go from a flat starting position to full atmospheric turbulence compensation in one jump, and would cause such oscillation. This was ameliorated using a suggestion by J.P Véran that VOLT start off with a low gain value and ramp up over a

number of iterations (on the order of 30 - 50 ms) to the final open loop gain of 1, causing a gradual increase in actuator heights over many loop iterations.

The actuator command voltage range of the DM52 is ±0.5V [32]. There exist 64 amplifiers, even though only 52 are connected to real actuators[8]. The DM52 is driven by the LAOG DE64 drive electronics, a 64-channel amplifier commanded by a parallel digital interface, of the same protocol and cabling interface as a PC printer port, which makes rapid testbench setup easy. The technical specifications of the DE64 are listed in Table 6. [31].

DE64 Technical Specifications		
definition	**Value**	**unit**
Number of channels	64	-
Max output range [1]	±2	V
Max output current [2]	±1	A
DAC resolution	14	bits
Output accuracy	<5	mV
Output settling time	<10	µs
Analog bandwidth	>100	kHz
Channel update delay [3]	<2	µs
Channel update rate	>150	kCh/s
Frame rate [4]	>2500	fps

[1] This value can be set internally from ±0.1V to ±4V with a potentiometer
[2] All channels are protected against shortcuts and overheating. A thermal switch on the PCB board will also power off all amplifiers above 80°C.
[3] Delay between the last data byte and the end of DAC conversion for any channel
[4] Frame rate for 52 active channels (ie. frequency at which the set of 52 active amplifiers can be updated)

Table 6: The DE64 drive electronics technical specifications [31].

[8] If those amplifiers without actuators are not explicitly set to 0V, they can inadvertently affect the voltages of the other amplifiers as they are on the same circuit [32].

A regular PC printer port was not fast enough to satisfy the 1 kHz mirror update bandwidth which was the goal for VOLT, so a high speed ADLink PCI-7200 digital I/O board was used to send DM commands from the control computer to the DE64 [33, 34]. The DE64 command protocol reserves two single byte control codes for powering the amplifier on and off and actuator voltage commands come in successive byte triplets. Each triplet begins with an address byte code indicating which actuator is to be updated, followed by two bytes that make up the high and low bytes of the digital value corresponding to the voltage to be assigned to that actuator, as shown in Table 7.

	Data Bits							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
byte 1	1	0	A5	A4	A3	A2	A1	A0
byte 2	0	D13	D12	D11	D10	D9	D8	D7
byte 3	0	D6	D5	D4	D3	D2	D1	D0

Ai: Address bits
Dj: Data bits
Reserved codes:

- 0xD5: amplifiers power on
- 0xEA: amplifiers power off

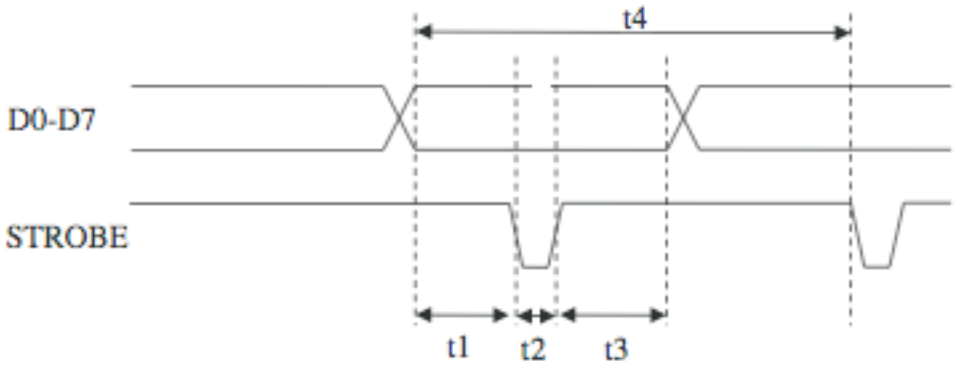
Table 7: The DE64 command protocol. It should be noted that the reserved codes do not form bit patterns of valid addresses, and thus cannot be confused for the beginning address byte of a command triplet [31].

The relation between the combined 14-bit digitized voltage codes and actual amplifier voltages is the following:

$$voltage = 2 \times \left(\frac{DAC_{code}}{8192} - 1 \right) \tag{9}$$

[31]

There is another layer to the parallel communication interface, the hardware handshaking required to latch the parallel digital line values into the DE64. This is accomplished by use of an alternating strobe signal on a separate dedicated line, as described in Figure 36 [31].



t1	>100 ns	delay between valid D0-D7 and STROBE negative edge
t2	>100 ns	width of STROBE negative pulse
t3	>100 ns	delay STROBE positive edge and D0-D7 invalid
t4	> 2 µs	delay between 2 consecutive STROBE pulses

D0-D7 refer here to the 8 parallel data lines, used for transmitting both address and data bits as described in Table 7.

Figure 36: The DE64 communications hardware handshaking protocol. Timing values are conservative; shorter values may be used with signals with a better SNR [31].

The maximum frequency at which the DE64 will accept commands without error was found experimentally with our hardware to be approximately 1.4 MHz. We ran VOLT with the PCI-7200 write frequency set at 1.2 MHz, 14% below the maximum, to provide a safety margin because data transmission errors were found sometimes to result in errant DE64 amplifier voltages, which can potentially damage the DM if they fall outside its safe operating range. Referring to Figure 36, we can assume the maximum channel

update delay to be 2 µs, thus, at 1.2 MHz write speed the maximum total time to carry out one complete update of all DM actuator positions is the following:

$$0.8\ \mu s \times 3\ \text{bytes} \times 52\ \text{amplifiers} + 2\ \mu s \approx 125\ \mu s$$

The 2 µs added at the end is the channel update delay applied after the last amplifier voltage command has been read.

We found that running the control loop faster than 800 Hz resulted in a degradation of dynamic turbulence correction due to the ALPAO DM52, and thus this set the upper limit on our VOLT system bandwidth.

Control Computers

As described in Section 3.5.1, our goal for the VOLT control bandwidth was 1 kHz, which drove the system requirements for the control computers and their specialised interface hardware. This meant that on each control loop iteration the system must be able to complete the following tasks within the deadline of 1 ms:

- capture a completed WFS frame
- compute WFS spot centroids
- perform centroid-to-DM-commands reconstruction
- issue new DM commands

Open loop control in VOLT is handled by Control PC 1 (see Figure 24), a 1.6 GHz dual processor desktop server PC with 2 GB of RAM, of which 200 MB are reserved in upper memory for framegrabber DMA buffers for incoming camera data from the WFS. Control PC 2 is an identical desktop server PC with the same memory configuration used

for framegrabber buffers and is responsible for recording the result of the open loop correction with WFS B. Control PC 3 is a 1.4 GHz single processor desktop PC with 385 MB of RAM, of which 85 MB are reserved in upper memory for framegrabber buffers and is responsible for recording the DM surface shape. Science camera image capture is handled by Control PC 4, a 1.5 GHz machine running Windows XP. Control PCs 1 through 3 are running Linux with kernel versions that are compatible with their unique set of interface boards (see Section 3.5.2 – Hardware Issues). Budget constraints precluded purchasing a commercial real-time operating system for VOLT, and considerable time was spent attempting to use the freely-distributed real-time version of Linux called the Real Time Application Interface (RTAI), but support for our interface hardware was lacking. In the end, we employed a modified pseudo-real-time scheme from the UVic Woofer-Tweeter Adaptive Optics test bench [12], which was to run Linux in a 'stripped-down' mode where all non-essential operating system services are disabled (including the graphical user interface; just a command prompt is needed), leaving very few contenders to demand system resources and cause pre-emption while running the VOLT experiment, as will be explained in more detail in Section 3.5.3 [35].

Hardware Issues

There were considerably challenging hardware issues to resolve, especially on Control PC 1, where the ADLink and original FF1303 boards were very particular about kernel versions and settings. Each board had a list of kernels it would run on, between which there were only three matches, of which only one actually turned out to work. Multiprocessor functionality was available for these kernel versions, which would have

sped up processing in the control loop, but the ADLink board would not run with this option enabled, despite the ADLink claim that it was supported. The EDT replacements for the FF1303 boards were not nearly as particular on kernel versions, but required that we upgrade the motherboards of Control PC 1 and 2 from desktop PC motherboards to server motherboards with faster PCI bus speed (66 MHz vs. 33 MHz). As well, there were hardware compatibility issues between the two interface boards to resolve in Control PC 1, as they competed for the same interrupt request (IRQ) lines. IRQs can be happily shared between devices in a desktop PC environment, but IRQ sharing means large performance hits in a real-time environment. This could be manually resolved in the BIOS of the original desktop PC motherboard that was used in Control PC 1, but the replacement server motherboards did not provide that level of control to the user. There were fortunately five PCI slots to choose from, and an arrangement of the ADLink and EDT boards was found that would ensure mutually exclusive IRQ assignments. However, there was still an IRQ conflict issue between the ADLink board and the SCSI controller of the motherboard, even though it had (supposedly) been disabled in the BIOS. This manifested itself by causing the ADLink board to intermittently fail when writing to the parallel interface, which was confirmed by monitoring the number of recorded interrupt signal events on that IRQ jump by large numbers at the same time a failed write was encountered. Since it was low-level motherboard hardware generating this interrupt signal – even after being disabled in the BIOS and its drivers prevented from being loaded – there was no known way to suppress this effect. Instead, a software solution was found. A C macro was written that, when the intermittent interrupt signal caused a failure, prompted an immediate retry of the ADLink parallel interface write

operation until a successful write was achieved. This was almost always successful upon the first retry, thus the total time to update the DM was not significantly affected.

3.5.3 Real-Time Controller

The real-time controller for VOLT consists of the high performance hardware described above and the *real-time pipeline*, software that is responsible for the time sequencing of all the tasks of the control system within timing deadlines such that it can preserve the required control loop bandwidth of VOLT. It generally consists of a real-time operating system and a set of customized software that decomposes the higher-level tasks into low-level hardware and software implementation, optimised for speed. In a true real-time operating system, precise task timing is guaranteed by direct control over virtually all system resources and a strict task-priority-based scheduling policy (ie. a task can only be preempted – put on hold – by another if that one is of higher priority), or disabling task preemption altogether.

As we were without a true real-time operating system for VOLT as mentioned, a low-cost pseudo-real-time solution was found. With most Linux operating system services disabled, we could build a pseudo-real-time system that had no actual task scheduler, but counted on the infrequency of other higher-priority operating system tasks demanding system resources (most notably the CPU) and thus preempting the real-time pipeline 'task' (as it is viewed by the operating system). Many modern processors allow direct read access to their CPU cycle count clock, a free running non-preemptible 32-bit hardware clock, via the Real Time Stamp Counter (RTDSC) assembler instruction made

available through a C macro. Once calibrated to actual time elapsed, this feature allowed us to implement real-time task timing and sequencing, but could make no assurance that task deadlines would be met, as there is no way in a standard Linux kernel to prevent task preemption by the operating system [35]. But these preemptions were infrequent enough for most of our short VOLT experiment cycles that they did not seriously affect the stability or performance of the controller. Extensive testing showed that the software control loop missed deadlines very rarely, exhibiting a standard deviation in loop iteration time of just a few percent, quite tolerable for our experiment. In the rare cases where operating system tasks generated a relatively large number of preemptions over an experiment cycle, driving the standard deviation up, we could just re-run the experiment.

Real-Time Pipeline Timing

As illustrated in Figure 37, a pipeline loop iteration on Control PC 1 begins with the DMA transfer of a completed WFS A frame from the camera to the host via the EDT framegrabber, benchmarked at 175 μs. Once the frame is transferred, the WFS A camera is triggered to begin exposure of the next frame and centroid computation is performed on the frame just captured, taking another 300 μs. Computed centroids are then used to generate new DM commands by a reconstructor-matrix-by-centroids-vector multiplication taking 345 μs. Lastly, the commands are dispatched to the ADLink digital output board to carry out the high-speed write of DM actuator voltage values to the DE64 amplifier, which has a 65 μs software function call return overhead included (ie. after 65 μs the pipeline can continue processing or calling other functions, while the PCI-7200 finishes the digital output to the DE64), recalling that a full 52 actuator update when writing to the DE64 at 1.2 MHz spans 125 μs. The remainder of the loop iteration is idle

time spent waiting for the next WFS A camera exposure to complete. The required exposure time for a given light source is determined experimentally by measuring the lenslet spot SNR on the WFS A subapertures. The task timing and ordering in Figure 37 are optimal for open loop control, assuming that none of the tasks can be made to run faster (for example, using better algorithms and/or parallel processing in the centroiding and DM command generation phases) than what is shown, as it ensures the DM shape update happens as soon as possible once new commands have been generated. As seen from Figure 37, the lower limit on loop time is 945 μs, and thus the upper limit on the control bandwidth would be 1058 Hz, meeting the VOLT control loop bandwidth specification of 1 kHz, except that the 800 Hz bandwidth limitation of the DM is lower and thus sets this limit. In practice, as will be elaborated in Section 3.7, WFS A exposures longer than 1 ms were required anyway for even the brightest stars available, due to the relatively low optical throughput of the 1.2 m telescope and VOLT optical system, thus also pushing the control loop bandwidth below 1 kHz.

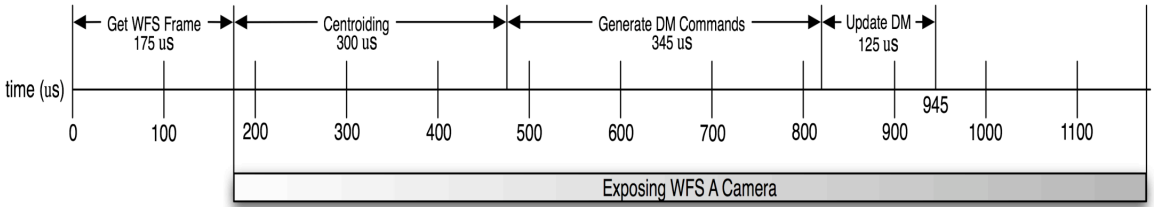


Figure 37: The real-time task timing diagram for a single iteration of the open-loop real-time pipeline. The fastest possible loop time is 945 μs, giving a maximum control loop bandwidth of 1058 Hz. Here we assume a 1 ms WFS A exposure time, giving a control loop bandwidth in this case of about 850 Hz.

Control PCs 2 and 3 run nearly identical instances of the real-time pipeline to that on Control PC 1, except that DM command generation and dispatching is disabled (as they

have no DMs), passively taking WFS B and WFS C centroid measurements, respectively, as in Figure 38.

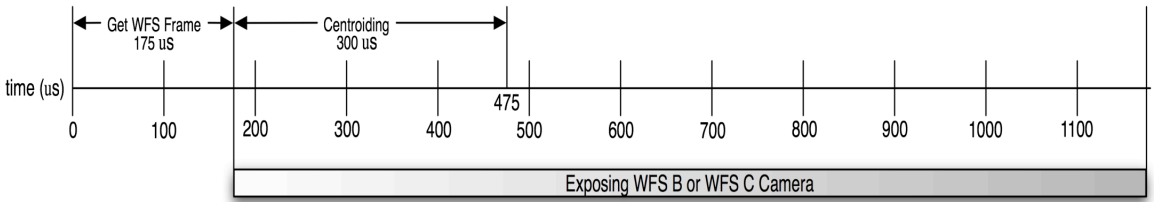


Figure 38: The real-time task timing diagram for a single iteration of the real-time pipeline for recording 'scoring' wavefront data with WFS B and DM 'truth' shape data with WFS C. The fastest possible loop time is 475 µs, giving a maximum control loop bandwidth of 2100 Hz. Here we assume a 1 ms WFS exposure time, giving a control loop bandwidth in this case of about 850 Hz.

It should be noted that the task ordering in the real-time pipeline when running in classical closed loop mode with WFS B is optimal from a control loop delay perspective if it follows that depicted in Figure 39. This is because 60 µs of the 125 µs DM update phase can be applied in parallel with the WFS B frame transfer, due to the 65 µs software function call return overhead of dispatching the DM commands via the PCI-7200. The 125 µs DM update phase could also possibly be cut down, by pushing the write speed to the DE64 close to the maximum failsafe speed, somewhere around 1.4 MHz. This task reordering also avoids updating the DM shape at the same time WFS B is taking its next exposure – recall the WFS is downstream of the DM in a closed loop AO system – thus introducing noise into the centroiding computation. The possible effects of this case have not been quantified, as it was not a concern for open loop operation, the primary mode of VOLT.

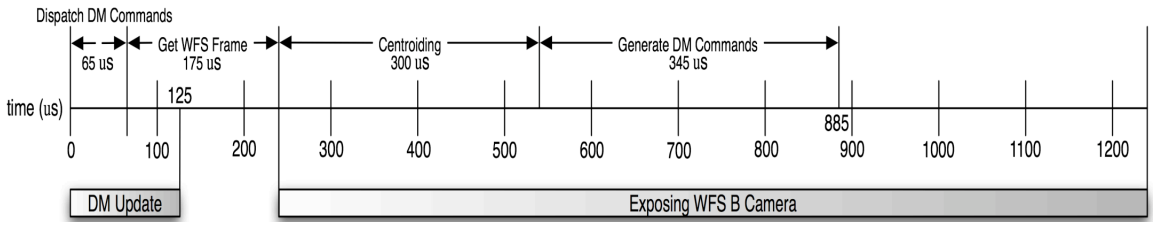


Figure 39: The real-time task timing diagram for a single iteration of the real-time pipeline optimised for closed loop operation with WFS B. The fastest possible loop time is 885 μs, giving a maximum control loop bandwidth of 1130 Hz. Here we assume a 1 ms WFS B exposure time, giving a control loop bandwidth in this case of about 806 Hz.

Real-Time Pipeline Software Architecture

The UVic Woofer-Tweeter Adaptive Optics Testbench [12] real-time pipeline software, designed and implemented by Aaron Hilton in 2005 [35], was used as a basis from which to build the VOLT real-time pipeline. The VOLT real-time pipeline is comprised of high-level object-oriented C++ code and MATLAB functions compiled into a custom type of linkable C libraries called modules (MATLAB-to-C compilation is only available up until MATLAB version 6.5) which act like regular C/C++ functions. VOLT AO system parameters are conveniently created and stored on disk by utilities provided by the UVic AO Library (discussed in Section 3.6.1). These parameters are then loaded by the VOLT real-time pipeline at startup and selected output telemetry is saved to the same file at the end of an experiment. The VOLT real-time pipeline is shown in Unified Modeling Language (UML) notation in Figure 40; low-level hardware drivers and libraries for the framegrabbers and ADLink board are considered implementation details and are not shown.

Using the object-oriented approach of C++, it makes sense conceptually to organise the real-time pipeline according to the key active hardware and software components of the VOLT control loop. The *Camera* and *MirrorDriver* object classes represent any WFSs and DMs in the system (*MirrorDriver* actually has empty member functions that provide a generic framework to interface with a tip-tilt mirror driver board and control the tip-tilt platform). The MATLAB modules are not actually an object, but are like regular functions which are called by the pipeline main program, `main()`. The functionality of loading AO instrument parameters for the setup of the real-time pipeline before the experiment starts, and the saving of system telemetry after the experiment ends, is provided by the MATLAB modules `loadAOVar()` and `saveAOVar()`. There also exist *processStats* and *ErrorReporter* objects, which are additional software objects that deal with real-time performance statistics generation and real-time deadline watchdog timing, respectively. What follows is a description of the C++ class objects, explaining key variables (attributes) and member functions (operations). For clarity, we will generally refer to the open loop pipeline running on Control PC 1, which is of the most interest as it represents a control scheme unique to VOLT, in the following subsections. For more detail refer to the complete source code listing in Appendix D.

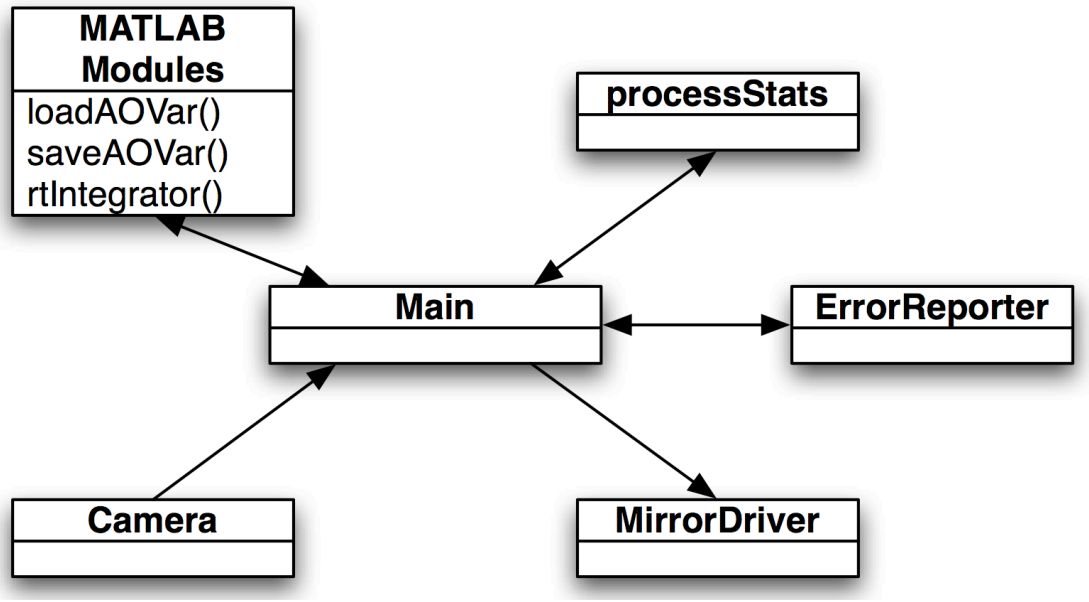


Figure 40: VOLT real-time pipeline software modules and objects in UML notation. The general directions of data / command flow is indicated by the arrows (ie. there is data sent from the Pipeline module to the Camera upon initialisation, but during real-time operation the Pipeline only consumes data produced by the Camera module).

Camera

The object class referred to as *Camera* is a generic term representing either of two camera object classes we have built for VOLT – the Dalsa 1M150 camera has the object class *WFS_1M150* and the Dalsa CA-D1 camera has the object class *WFS_CAD1*. The two cameras have different enough hardware and software interfaces that it makes sense to have separate classes to represent them. For our purposes here, we will refer only to the pipeline running on Control PC 1, the open loop control pipeline implementation, and thus all references to *Camera* are synonymous to the *WFS_1M150* object class (Appendix D contains all pipeline code variations).

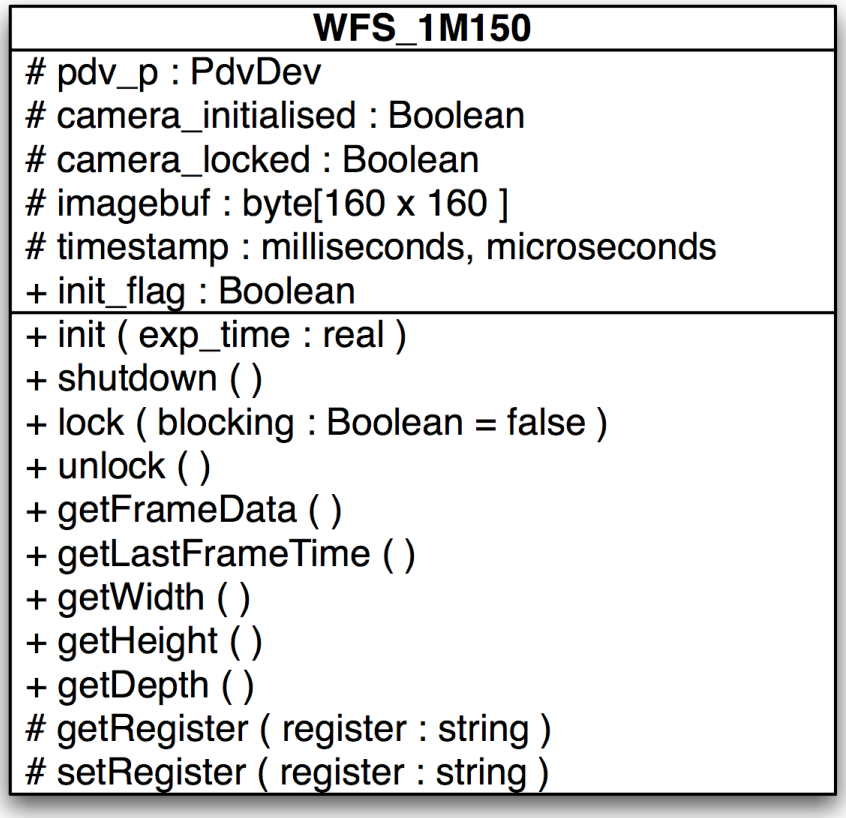


Figure 41: The UML representation of the *WFS_1M150* (Camera) object class. Public and protected attributes or operations are denoted by the '+' or '#' symbol, respectively.

The UML diagram of Figure 41 is structured with the class object name at top, beneath it are what are known as attributes of the object. Variables, constants, or embedded objects of another class can all be attributes. At bottom are the operations of the object, those functions that any instance of the object is capable of performing on itself or for an external caller. The symbols + and # indicate whether an attribute is publicly accessible for reading/writing or if an operation can be invoked by an external process. The attributes of the *WFS_1M150* object are all protected (ie. they can only be viewed or modified by a class member operation) with the exception of `init_flag`. As is considered good practice in object-oriented software design, the implementation details

of controlling the EDT framegrabber (via its custom libraries that provide an interface layer to the low-level device driver) are hidden from calling processes, allowing for a very modular approach to the software design, and thus ease of interchangeability between similar class objects, like the *WFS_1M150* and *WFS_CAD1* in this case.

WFS_1M150 object attributes:

`pdv_p`

An instance of the EDT custom C structure PdvDev that is registered by the EDT library call pdv_open( ). It encapsulates the unique (and access-protected) parameters that allow control and reading of the state of one framegrabber device (in the open loop pipeline only one instance of the *WFS_1M150* object is created, and thus only one instance of the PdvDev structure).

`camera_initialised`

A Boolean flag indicating to operations within a given *WFS_1M150* object whether the camera has been successfully initialised, or can be set by those operations given the ability to do so (ie. `init()`).

`camera_locked`

A Boolean flag that provides protection for operations within a given *WFS_1M150* object pertaining to whether the camera subsystem (framegrabber and camera) has been locked for exclusive access by another operation, or is available to be locked. This prevents race conditions, where a programming error could lead to multiple operations called in quick succession all vying for access to the same hardware resource, which can lead to unpredictable results or system deadlock.

`imagebuf[]`

A pointer to a vector in memory of 8-bit image data (in C/C++ we use the `unsigned char` datatype) captured by the framegrabber from the camera.

`timestamp`

The time the most recent frame transfer from the camera to the framegrabber was completed is written to this attribute.

`init_flag`

A flag which provides a means for the pipeline to indicate to the *WFS_1M150* object whether the EDT framegrabber and Dalsa 1M150 camera need to be initialised or not (it is mostly for time saving, as the process of setting up the camera by the serial interface takes on the order of 10 seconds, and these settings will often not need to be changed from one run to the next).

WFS_1M150 object operations:

`init ( exp_time : double )`

Opens, or registers, the EDT framegrabber device and class attribute `pdv_p` and configures the 1M150 camera via the serial interface and starts the first frame capture.

`shutdown ( )`

Closes the EDT framegrabber device and class attribute `pdv_p`.

`lock ( blocking : Boolean )`

Locks access to the camera data accessor operations `getFrameData()` and `getLastFrameTime()` and sets the `camera_locked` flag to true. If the input parameter `blocking` is set to true, `lock()` will wait (or block) on a new frame to arrive before setting `camera_locked` to true. This is the normal use of `lock()` in the pipeline, which always precedes a call to `getFrameData()`, thus we ensure a new frame is acquired on successive calls and 'stale' frame data cannot be mistakenly read.

`unlock( )`

Unlocks access to the camera data accessor operations by clearing the `camera_locked` flag (ie. set to false).

`getFrameData( )`

Returns the pointer to the locked frame data `imagebuf` and triggers the next camera exposure to begin.

`getLastFrameTime( )`

Returns a pointer to `timestamp`, which was updated on the last call to `lock()`.

`getWidth( ) / getHeight( ) / getDepth( )`

These are accessor operations used to query the dimensions of the image data coming from the camera, as set up by the EDT card initialisation application initcam on system boot. The dimensions are image width and height (in pixels) and depth (in bits per pixel).

`getRegister( register : string ) / setRegister( register : string )`

These protected operations are modified inclusions from the EDT camera interface library and provide a means to read and write the state of configuration registers on the camera, such as those which determine the ROI or exposure time. Since they are protected, they can only be called from other WFS_1M150 member functions (ie. `init()`) and not from an external caller.

MirrorDriver

The object class *MirrorDriver* encapsulates the attributes and operations necessary to incorporate the ALPAO DM52 into the VOLT system (as well as the framework for a generic tip-tilt mirror, as mentioned before, which we will not address here).

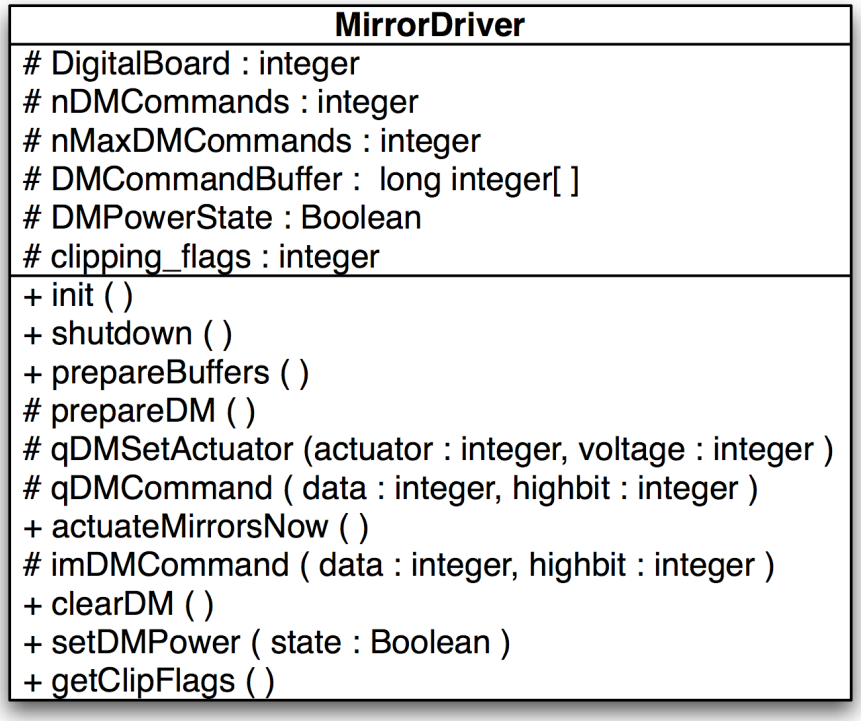


Figure 42: The UML representation of the *MirrorDriver* object class. Public and protected attributes or operations are denoted by the '+' or '#' symbol, respectively.

MirrorDriver object attributes:

`DigitalBoard`

A unique integer identifier generated by the ADLink registration routine `Register_Card()` which is used by other routines in that library to differentiate between multiple Adlink cards which may be installed in the same system.

`nDMCommands`

An integer that keeps count of the number of digital control codes to be written out over the ADLink parallel interface to the DE64 drive electronics.

`nMaxDMCommands`

An integer that is assigned upon initialisation with the maximum possible number of digital control codes that can be written to the DE64 on one complete 52 actuator position update cycle. This is used to guard against errant control codes being sent.

`DMCommandBuffer[]`

An array of unsigned long integers of `nMaxDMCommands` number of elements. It is loaded with the digital control codes which are written out to the DM for each complete 52 actuator position update.

`DMPowerState`

A Boolean that reflects the current DM power status, indicating whether the DE64 has its amplifiers powered on or off.

`clipping_flags`

An integer that is read or written to in a bitwise manner, the first four bits that are set or reset to represent clipping conditions for the DM and a possible tip-tilt mirror. Clipping refers to the condition where the command generated for any DM actuator (or the tip-tilt stage) is outside of the allowable physical range for the hardware, either below the minimum or above the maximum allowed value. The first two bits are assigned to the high and low clipping conditions for the DM, and the last two bits for the high and low clipping conditions for the tip-tilt mirror.

MirrorDriver object operations:

`init ( )`

Opens, or registers, an instance of the PCI-7200 board with the ADLink library call Register_Card, configures it for asynchronous output (see `actuateMirrorsNow()`), and allocates `DMCommandBuffer`.

`shutdown ( )`

Clears the DM actuators (see `clearDM()`), powers the DE64 amplifiers off (see `setDMPower()`), frees the allocated memory of `DMCommandBuffer`, and releases the instance of the ADLink PCI-7200 board, `DigitalBoard`.

```
prepareBuffers( )
prepareDM( )
qDMSetActuator( actuator : integer, voltage : integer)
qDMCommand( data : integer, highbit : integer)
```

The combination of these four operations fills `DMCommandBuffer` with digital control codes for the next full DM update. `prepareBuffers()` is a generic buffer preparation routine, which just calls `prepareDM()`, which in turn calls `qDMSetActuator()`, which in turn calls `qDMCommand( )` (an equivalent string of calls would be made as well for a tip-tilt mirror by `prepareBuffers()`, if applicable). `qDMSetActuator()` first does a safety check that the voltage value being requested for a given actuator falls within the allowable range – if is not, it sets the appropriate flag in `clipping_flags` and modifies the voltage value to the clipped value (maximum or minimum allowable voltage) – and then forms the command triplet that is the command protocol for updating an actuator position on the DE64, as explained earlier. This leaves `qDMCommand()` to perform the final stage of reforming them for hardware handshaking over the ADLink digital interface. This can be broken into three steps for clarity: 1) mapping the control code data to the correct parallel digital output pins of the ADLink PCI-7200 by bit-shifting the command triplets, as each bit of a 32-bit long integer to be written out is mapped to an actual output pin in hardware (there are 32 output pins), 2) a threefold copy of the command triplets, and 3) append the alternating high-low-high hardware handshaking strobe signal to the three copies of each command in each triplet, (recall that the strobe uses another pin on the output of the PCI-7200). At this point, `DMCommandBuffer` is full and ready to be written out over the PCI-7200.

```
actuateMirrorsNow( )
```

After the above four operations have been carried out, the correctly formed series of commands that fill `DMCommandBuffer` are sent to the PCI-7200 using the library call `DO_ContWritePort()` to begin writing them out over the parallel interface in series asynchronously (ie. `DO_ContWritePort()` returns ‘immediately’ (actually after 65 μs, as explained in Section 3.5.3), leaving the PCI-7200 to finish writing out the entire

contents of `DMCommandBuffer` in the background while the pipeline moves on to the next step).

`imDMCommand( data : int, highbit : int )`

Immediately writes one command, `data` with `highbit` (to indicate if it is an address or not) to the DE64, duplicating it three times and applying strobe signals as in `qDMCommand()`, then sending the three codes for asynchronous output to the PCI-7200.

`clearDM( )`

Set all DE64 amplifier voltages to zero. This is accomplished by forming the command triplet for one actuator address and voltage (zero), passing each element of the triplet in series to `imDMCommand()`, and repeating for all remaining actuators.

`setDMPower( state : Boolean )`

Power the DM up or down, according to the Boolean value of `state`. In the case where `state` equals true, `clearDM()` is called to zero all the amplifier presets (amplifier voltage values for each actuator can be set and stored even when the amplifiers are off, for when they are powered back on), followed by sending the reserved *on* control code via `imDMCommand()`. In the case where `state = FALSE`, the *off* control code is simply sent via `imDMCommand()` to turn the DE64 amplifiers off. Lastly, `DMPowerState` is updated to reflect the new power state of the DM.

`getClipFlags( )`

Simply returns the current state of `clipping_flags`.

Main

The pipeline main program, `main()`, is not actually an object itself but is the module that combines the other modules and objects together into a working system, implementing the iterative open loop AO controller algorithm and consuming from, processing, and passing data to the class object instances of *WFS_1M150* and *MirrorDriver*, respectively.

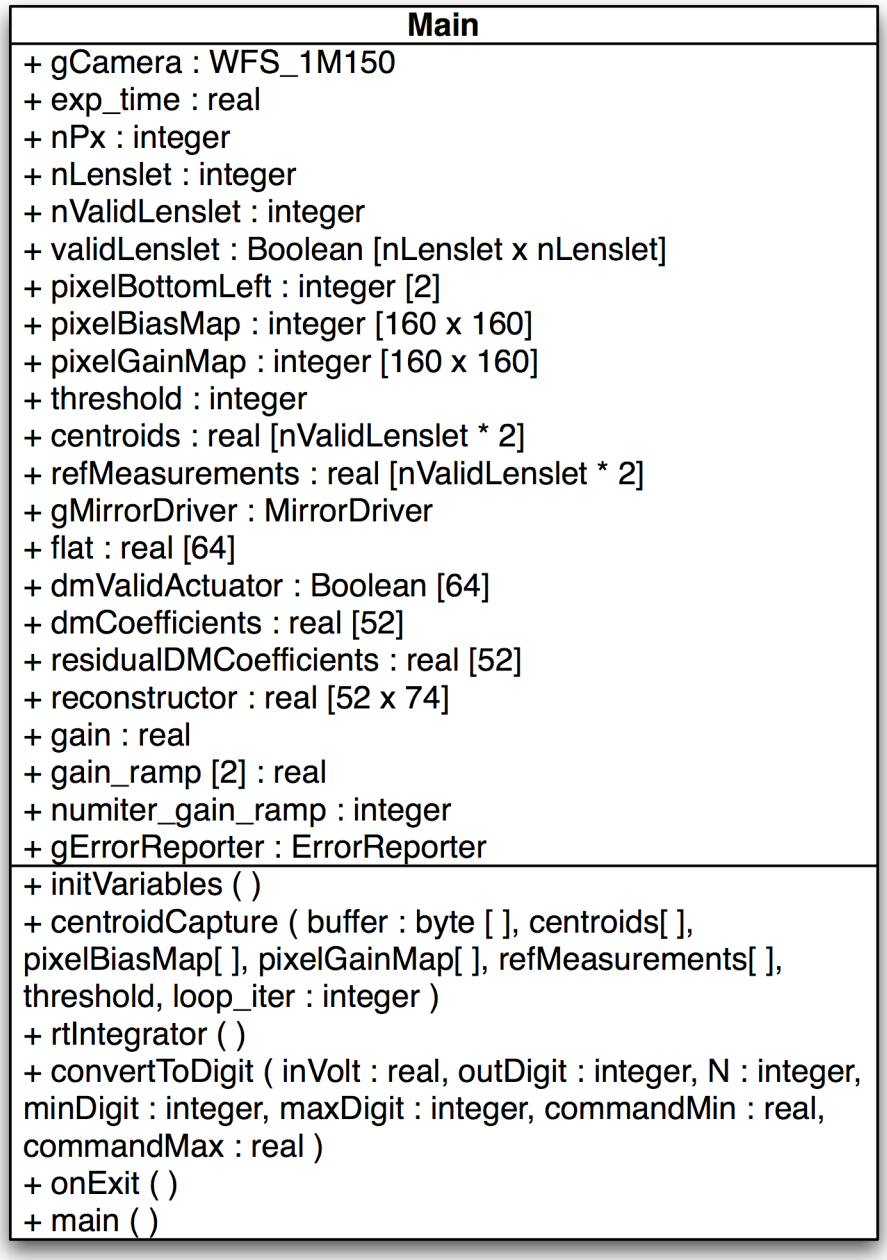


Figure 43: The UML representation of the pipeline `main()` program.

Main program parameters and global variables:

`gCamera`

The instance of the *WFS_1M150* object representing WFS A (or WFS B on Control PC 2).

`nPx`

An integer representing the number of pixels across a row or down a column within a WFS lenslet (ie. each lenslet is a square of `nPx` x `nPx` pixels) and is loaded into `main()` by `loadAOVar()`. `nPx` has a value of 16 for WFS A and 19 for WFS B.

`nLenslet`

An integer representing the number of lenslets (or subapertures) across a row or down a column of a square grid WFS, ignoring the clipping of corner subapertures that comes with inscribing a circular aperture within a square. `nLenslet` has a value of 7 for both WFS A and WFS B.

`nValidLenslet`

An integer representing the number of lenslets (or subapertures) that are within the inscribed circle of the aperture and are considered valid for wavefront processing. When VOLT is running on-sky, there is a central obscuration in the pupil caused by the telescope's secondary mirror that blocks light from reaching the central subaperture of both WFS A and B, leaving 36 valid lenslets, as shown in Figure 44. When running in the lab with an artificial light source there is no central obscuration so in this case `nValidLenslet` is 37.

`validLenslet[]`

A 7 x 7 logical matrix mapping of all lenslets. Entries with value 1 indicate a subaperture that is to be included in the wavefront processing. Those with 0 are ignored.

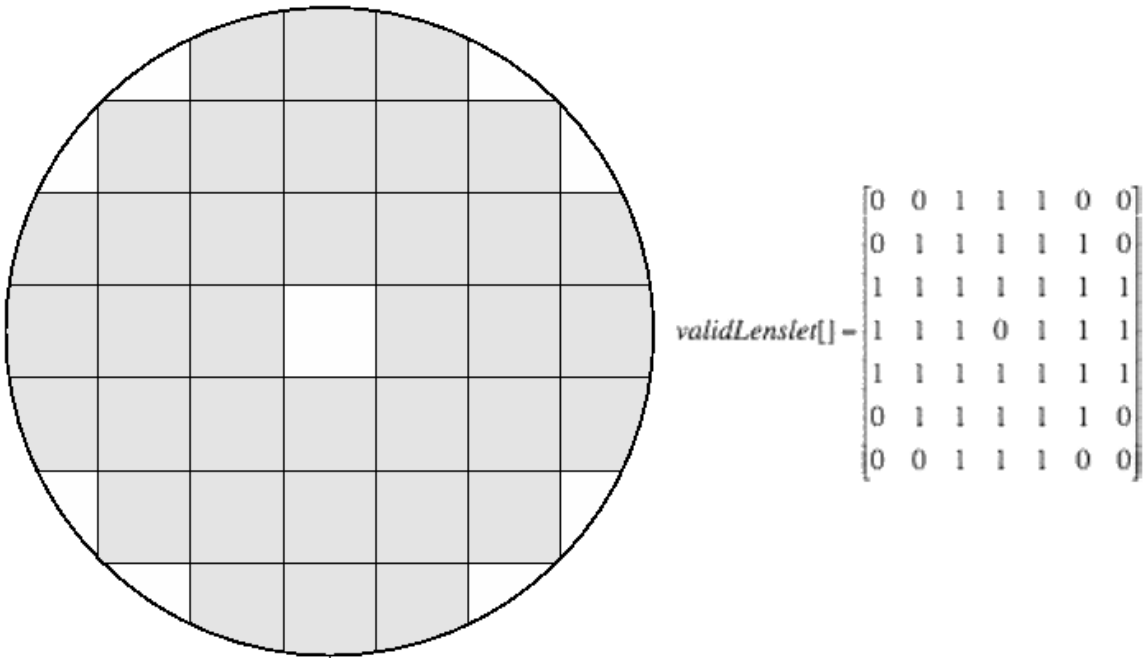


Figure 44: Left: The portion of the WFS illuminated by the beam with a central obscuration from the secondary mirror of the telescope. Valid lenslets / subapertures are shaded, represented by ones in the `validLenslet[]` matrix at right.

`pixelBottomLeft[]`

The *x* and *y* pixel offsets within the WFS camera ROI (160 x 160 for WFS A and B) that locate the bottom left corner of the grid of subapertures, which `centroidCapture()` uses as a starting point from where to begin centroiding.

`centroids[]`

The *x* and *y* lenslet spot centre of mass coordinate pairs, computed by `centroidCapture()` (to be explained shortly).

`pixelBiasMap[]`

An image the size of the ROI filled with pixel values that represent their individual data number biases (CMOS and CCD pixels can have a non-zero digital value offset), recorded during calibration. It should be noted that these calibrated values are negated when forming `pixelBiasMap` to make their subtraction from wavefront measurements

inside `centroidCapture()` faster, since an addition requires fewer operations in software than a subtraction.

`pixelGainMap[]`

This is just a 160 x 160 matrix of ones, indicating that all CMOS pixels of the Dalsa 1M150 cameras exhibit the same gain characteristics.

`threshold`

A pixel value, determined experimentally for a given camera, ROI, and exposure time, that is chosen to filter out pixels with low counts, which are considered noise and adversely affect the sensitivity of the centroiding function.

`refMeasurements[]`

The centroid pixel offsets calibrated for the WFS with a flat wavefront. Ideally `refMeasurements` would be all zeros, but misalignments, non perfect image scaling or rotation, accuracy reduction by digitally sampling an image, bad pixels, and various other effects cause these to have non-zero values.

`gMirrorDriver`

The instance of *MirrorDriver* that represents the DM.

`dmValidActuator[]`

An 8 x 8 logical matrix mapping of all the DM actuators. All 52 entries with a 1 represent the real actuators, those with 0 represent the amplifiers not connected to actuators. `dmValidActuator` is used by `rtIntegrator( )` to select only valid actuators to generate new commands for the DM on each iteration (to be explained shortly). In Figure 20, these are denoted by black circles.

$$dmValidActuator[] = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \end{bmatrix}$$

`flat[]`

A vector with 64 voltage values for all DM52 amplifiers (those not representing valid actuators are set to zero) that have been calibrated as the current set of actuator displacements needed to synthesize a flat DM surface shape (see Section 3.6.2 – Flattening the DM). `flat[]` is the basis onto which DM actuator commands generated in the pipeline are superposed before being written to the DM via the DE64.

`reconstructor[]`

The 52 x 72 matrix of real numbers that translates computed open loop centroid values to DM actuator commands.

`gain`

The open loop gain factor applied to newly generated DM commands in `rtIntegrator()`. `gain` can be a constant value, passed in on the command line by the user, or a changing value according to the following two parameters `gain_ramp[]` and `numiter_gain_ramp`.

`gain_ramp[]`

A 2-element parameter that is passed into `main()` from the command line specifying if the open loop gain should ramp up from an initial value (the first element) to a final value (the second element) over a number of control loop iterations. This is a useful feature for for maintaining open loop control loop stability during the first few iterations

by avoiding large moves with the DM on any single step (see Section 3.5.2 – Deformable Mirror Subsystem).

`numiter_gain_ramp`

If `gain_ramp[]` has been set, the user must also specify over how many control loop iterations the ramping must be applied using the integer parameter `numiter_gain_ramp`.

`residualDMCoefficients[]`

A vector of DM voltages generated by multiplying the `reconstructor[]` by the `centroids[]`, elaborated below in the description for `rtIntegrator()`. The 'residual' in `residualDMCoefficients` is a relic from the classical closed loop AO case, where the WFS would be measuring the corrected wavefront downstream from the DM, and thus measuring the residual error. In the case of VOLT, WFS A is measuring the entire atmospheric turbulence term without correction, so the name is not so fitting.

`dmCoefficients[]`

This is the vector of new DM voltages, `residualDMCoefficients`, multiplied by the current value of `gain`. This is what is passed to `gMirror` to be set on the DM.

`gErrorReporter`

The instance of the *ErrorReporter* class, used to monitor the pipeline loop timing and report if deadlines are exceeded.

Main program functions:

```
centroidCapture( buffer : byte[], centroids[],
     pixelBiasMap[], pixelGainMap[], refMeasurements[],
     threshold, loop_iter : integer)
```

The key step of WFS data processing, computing the WFS spot centroids, is carried out by a call to `centroidCapture()` after receiving a frame from `gCamera`, the *WFS_1M150* instance that represents WFS A. The centroiding algorithm used in the

VOLT experiment is a weighted centre of mass computation with pixel bias removal and thresholding. This is applied to all valid subapertures to generate *x* and *y* centroid pairs, which are then subtracted from a constant term flat wavefront reference taken during WFS calibration, `refMeasurements` (see Figure 45). The classical mathematical centre of mass formulae are shown in Equations 10 and 11 without thresholding. Thresholding means ignoring (ie. setting to zero) all pixels with intensities (data number counts) less than a given threshold value. This improves the SNR of the centroid computation, throwing out the noise floor of the camera pixels (its implementation in source code can be found in centroidCapture.cpp, in Appendix D).

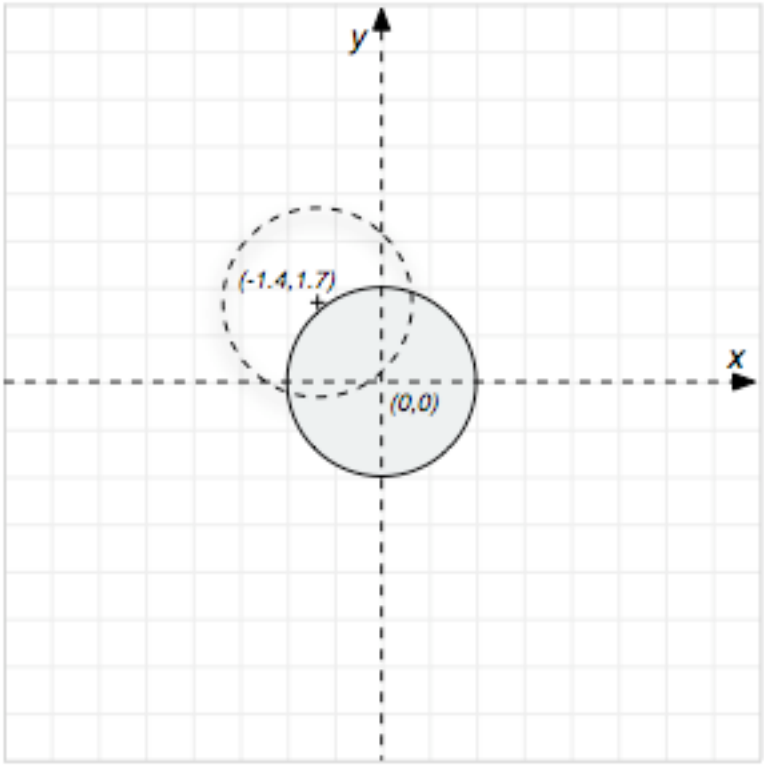


Figure 45: A 16 x 16 pixel Shack-Hartmann subaperture (as is used for WFS A) showing the idealised spot centroid case for a flat wavefront (grey circle), and that for a given calibrated offset, or reference measurement, of *(-1.4,1.7)* pixels (dashed circle). Centroiding computations subtract the unique reference measurement from each calculated subaperture centroid *(x,y)* pair (both with or without thresholding applied).

$$x_{centroid} = \frac{\sum\sum xI(x,y)}{\sum\sum I(x,y)} \tag{10}$$

$$y_{centroid} = \frac{\sum\sum yI(x,y)}{\sum\sum I(x,y)} \tag{11}$$

where *x* and *y* are the Cartesian pixel coordinates within the subaperture, as laid out in Figure 45, and *I(x,y)* is the measured pixel intensity at these coordinates.

`rtIntegrator( )`

The MATLAB module `rtIntegrator()` is included here because it acts as a data processing function for `main()`, just like `centroidCapture()`. `rtIntegrator()` only actually acts as an integrator when the system is running in closed loop with WFS B (see below). The name *rtIntegrator* is a relic from the classical AO control scheme of the UVic pipeline; in open loop `rtIntegrator()` provides the centroid-to-DM-commands reconstruction step, a matrix-by-vector multiplication, without feedback:

```
residualDMCoefficients[52x1] =
               reconstructor[52x72] * centroids[72x1]

dmCoefficients[52x1] = gain * residualDMCoefficients[52x1]
```

And for the closed loop mode, used for system calibration or running as a classic AO instrument using WFS B, feedback of `dmCoefficients[]` is used:

```
residualDMCoefficients[52x1] =
               reconstructor[52x72] * centroids[72x1]

dmCoefficients[52x1] = dmCoefficients[52x1]
               + gain * residualDMCoefficients[52x1]
```

`centroidCapture()` and `rtIntegrator()` together represent the combined *computational lag* and *gain* stages in the open loop control scheme as modeled in Figure 17. It was much simpler to implement in a MATLAB coding environment and then translate into compiled C code than to directly program the matrix math in C/C++, although it can be done and might result in some worthwhile time savings on this step.

Real-Time Pipeline Operation Summary

Now that all the real-time pipeline objects and member functions have been described, it follows to give a condensed summary of a typical VOLT pipeline execution, describing the command line launch, pipeline software initialisation and real-time control loop (with each loop iteration starting with a WFS measurement and ending with a DM update, as illustrated in Figure 37).

Typical command line launch:

```
> rtpipeline -i 30000 –e 1.2 –g 0.1 1 –r 200 –w 1
```

where the command line options have the following usages:

```
-i <iterations>
   set the number of AO loop iterations
-e <exposure time>
   set the camera exposure time in milliseconds
-g <initial gain  final gain>
   set gain for loop (open or closed loop).  final gain
   is only used (set) in conjunction with -r option
```

```
-r <num ramp iterations>
    set gain to ramp up from initial gain to final gain
    over num ramp iterations
-w <0/1>
    set value to 0 to skip WFS camera exposure time & ROI
    initialisation phase
```

Pseudocode real-time pipeline execution summary:

```
initVariables(); — load AO system parameters from file
gCamera = new WFS_1M150(); — create WFS A object
gCamera->init(exposure_time); — initialise the WFS A camera
gMirrorDriver = new MirrorDriver(); — create DM object
gMirrorDriver->init(); — initialise the DM
gMirrorDriver->setDMPower( true ); — power up DM actuators at 0 V
gMirrorDriver->prepareBuffers( flat_DM_commands );
gMirrorDriver->actuateMirrorsNow(); — flatten the DM
gCamera->getFrameData(); — start exposing first WFS A frame

for(loopIteration = 1 to loopIteration = iterations)
    {
    gCamera->lock( true ); — lock access to WFS A camera; wait for new frame
    camera_frame = gCamera->getFrameData(); — get WFS A frame
                                              and start exposing next frame
    centroidCapture( camera_frame, resultant centroids
                     address, pixelBiasMap, pixelGainMap,
                     refMeasurements, threshold,
                     loopIteration); — calculate WFS A centroids
    gCamera->unlock();— release WFS A camera access
    rtIntegrator(); — get DM commands via centroids-by-reconstructor multiply
    gMirrorDriver->prepareBuffers( DM commands );
    gMirrorDriver->actuateMirrorsNow(); — update DM
    }

saveAOVar(centroids); — save all WFS A centroids to file
saveAOVar(DM Commands); — save all DM commands to file
```

3.6 Laboratory Testing

3.6.1 UVic AO Library MATLAB Simulator and Controller

The UVic AO Library, written by R. Conan, consists of a set of custom MATLAB scripts that enable rapid setup and simulation of an AO system in software, and non-real-time control of AO hardware.

The UVic AO Library also provides an easy way to quickly build up AO system variables and parameters through a well-developed object-oriented structure that allows for seamless interchangeability between simulated AO system objects and real hardware (ie. WFSs, DMs, and science camera) or phenomena (ie. turbulence, read noise, and NCP error) they represent. The modular nature of the library allows for a high degree of reconfigurability and exploration of the AO system parameter space, and simulated and real hardware can be mixed together to form a working system. This was extremely helpful for optical alignment in VOLT, allowing direct alignment quality evaluation by numerical comparison between simulated aspects of the system and those observed by the cameras on the real instrument.

Where support for a particular hardware component does not already exist, the openly available UVic AO Library MATLAB source code can be modified to suit (along with the very useful MATLAB feature C MEX, allowing C function compilation and linking to MATLAB scripts).

For VOLT the UVic AO Library was invaluable, as it provided the quick system simulation, alignment, and control setup mentioned (see Appendix E for sample code); particularly useful were its utilities for generating the interaction matrix (to be explained shortly) and reconstructor. It provided a seamless way to store AO system parameters onto disk for loading into the real-time pipeline at run time, and for storing system telemetry to disk after an experiment. The MATLAB data file *aoVariables.mat* forms this link between the UVic AO Library VOLT system model and the real-time pipeline,

containing the following pipeline input parameters (for a review of their meaning refer to Section 3.5.3 – Real-Time Pipeline Software Architecture):

- exp time
- nPx
- nLenslet
- nValidLenslet
- validLenslet
- pixelBottomLeft
- pixelBiasMap
- pixelGainMap
- threshold
- refMeasurements
- dmValidActuator
- reconstructor

The telemetry that is most commonly recorded and written to *aoVariables.mat* at the end of a VOLT experiment are:

- centroids
- residualDMCoefficients

and the real-time loop cycle length and performance statistics (in RTDSC clock cycles, or tics, as in Section 3.5.3):

- tics_per_cycle
- min_tics
- max_tics
- mean_tics
- rms_tics

3.6.2 Optical alignment

The following subsections describe three key alignment tasks that address issues unique to an open loop AO scheme, and were critically important to the success of VOLT. Optical alignment of the VOLT instrument – conducted by Dr. David Andersen with my assistance – was performed using a combination of the UVic AO Library software tools mentioned above and manual alignment tools. The manual alignment tools included artificial light sources (lasers and LEDs), flat mirrors, pupil masks, phase screens, and a small telescope – with eyepiece and adjustable focus – which allowed conjugation to any optical surface along the beam path of the system to check alignments and evaluate AO correction.

Flattening the DM

Before the DM can be used to correct for the turbulence measured on WFS A, it needs to be as flat as possible so as to minimise introduced (and unsensed) NCP error in the science image. A perfectly flat DM means an unaltered wavefront upon reflection, which should be the reference shape from which to start the experiment. The voltage offsets of

all actuators then need to be determined to provide a default set of DM commands to take on the flat shape at startup.

This was first accomplished in the HIA integration lab before the DM was integrated into VOLT using the WYKO interferometer [30]. An interferometric image of the DM surface shape at zero volts was taken and the tip/tilt and piston terms removed (refer to Appendix A for more information on these terms). This modified shape was then projected onto the DM influence functions (as described in Section 3.5.2 – Deformable Mirror Subsystem) to represent this shape in terms of DM commands. It then followed to simply apply the negative of this set of commands to drive the DM this best flat estimation, which was then measured with the interferometer to be flat with 30 nm RMS.

After the DM was integrated into VOLT, it was much more difficult to move it to be measured with the WYKO interferometer (which was too big to be integrated onto the optical table itself), so another method of determining the reference DM flat had to be used, to mitigate the effect of the quality of the flat degrading over time. This was accomplished using the 'truth' WFS C. First, the DM was commanded to take on the shape of the last known good flat, perhaps one determined from the interferometer (even though the DM had likely drifted, it was still a good starting point). The artificial source of WFS C was used to illuminate the DM and the WFS spots observed. The DM position was then adjusted so that the mean WFS C spots were centred within their respective subapertures. The DM was then replaced with a regular flat mirror and its position

adjusted so as to centre the WFS C spots again. A reference measurement of the WFS C centroids was then taken and the flat mirror was switched out for the DM.

We built interaction matrices to provide a conversion from DM actuator voltages to WFS *x* and *y* centroid offsets. This is generated by pushing and pulling each actuator (±0.04 V actuator voltages) in turn and measuring *x* and *y* WFS centroid offsets, forming a matrix with structure like that in Figure 46. Then, the inverse of this interaction matrix gives us the reconstructor for running the DM in closed loop off the 'truth' WFS C – a direct conversion from measured *x* and *y* WFS C centroid offsets to DM commands.

To apply the flat, the reference measurements taken earlier from the flat mirror are multiplied by the WFS C reconstructor to command the DM to the new (hopefully better) flat.

Aligning WFS B

WFS B needed to be aligned, or registered, to the DM so that its subapertures were sampling the correct area of the pupil, between the DM actuator locations, as was illustrated in Figure 20. Using an LED artificial light source, this was done by poking a single actuator up at a relatively high stroke such that the surrounding lenslet spots on WFS B were quite noticeably skewed. Ideally, there should be four evenly elongated WFS spots pointing inward at a common point (the actuator location). If the pattern is different, we translated the WFS B lenslet array to achieve the desired geometry, with the WFS B subapertures centred between the DM actuators.

The WFS B interaction matrix also needed to be generated as well as its inverse, the reconstructor for running the DM in closed loop off WFS B (the classical AO configuration).

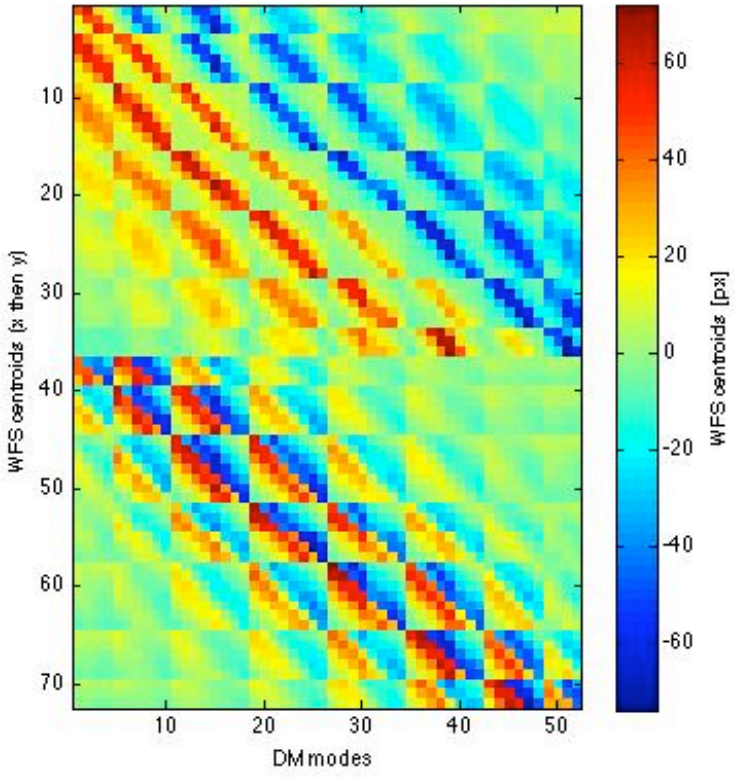


Figure 46: The WFS B to DM interaction matrix, describing the conversion to WFS B centroids from DM actuator voltages. WFS B *x* and *y* centroids are shown on the vertical axis (*x* centroids from 1 to 36; *y* centroids from 37 to 72) and DM actuator numbers along the horizontal axis. Note the symmetry of the interaction matrix indicates that the geometry is achieved.

Aligning WFS A to the DM

After we established that the ALPAO DM was well suited for use in an open loop AO system, we integrated the DM into the VOLT system. Key to the operation of any open loop AO system is the ability to register the open loop WFS A to the DM. To accomplish

this – and since WFS A cannot see the DM – we have to register WFS A to WFS B, using the alignment described in the last step. Thus, we align both WFSs to the optical beam so they are seeing exactly the same wavefront region and orientation on corresponding lenslets. We begin this process by placing masks just after the VOLT collimator which illuminate only a few subapertures on WFS B. We found the best technique was to have holes in the mask roughly the size of a subaperture (2.5 mm) and then position the mask so that light passing through this hole illuminates 4 subapertures on WFS B. We treat the fraction of light in each subaperture analogously to a quad-cell. By adjusting the WFS A optics and achieving the same fractional split of light in the four corresponding subapertures on both WFS A and WFS B, we find that we achieve a suitable alignment (see Figure 47).

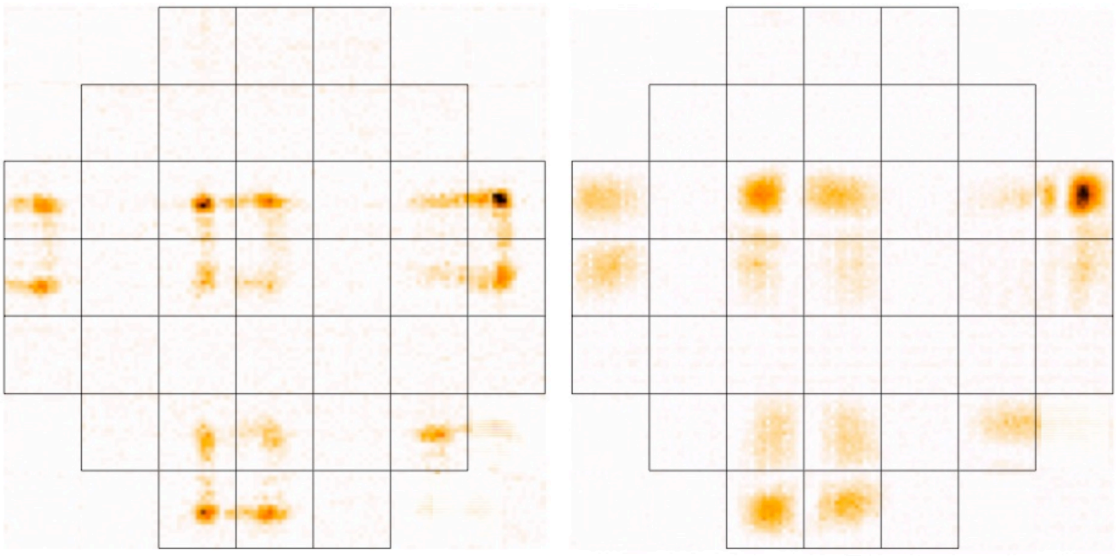


Figure 47: WFS A to DM registration by matching illumination patterns between WFS A (left) and WFS B (right). A mask is placed in the collimated beam upstream of both WFSs, producing non-uniform spot illumination patterns. The WFS A optics are then adjusted so that its illumination pattern matches that of WFS B.

Once the two WFSs appear well-registered, we confirm and quantify the registration by placing phase screens[9] in the beam just after the collimator. By comparing the centroids measured on both WFS A and B, we can determine the relative plate scales of the two WFSs[10]. The WFS A reconstructor is then created by scaling the WFS B reconstructor by the ratio of the plate scales, its form shown in Figure 48.

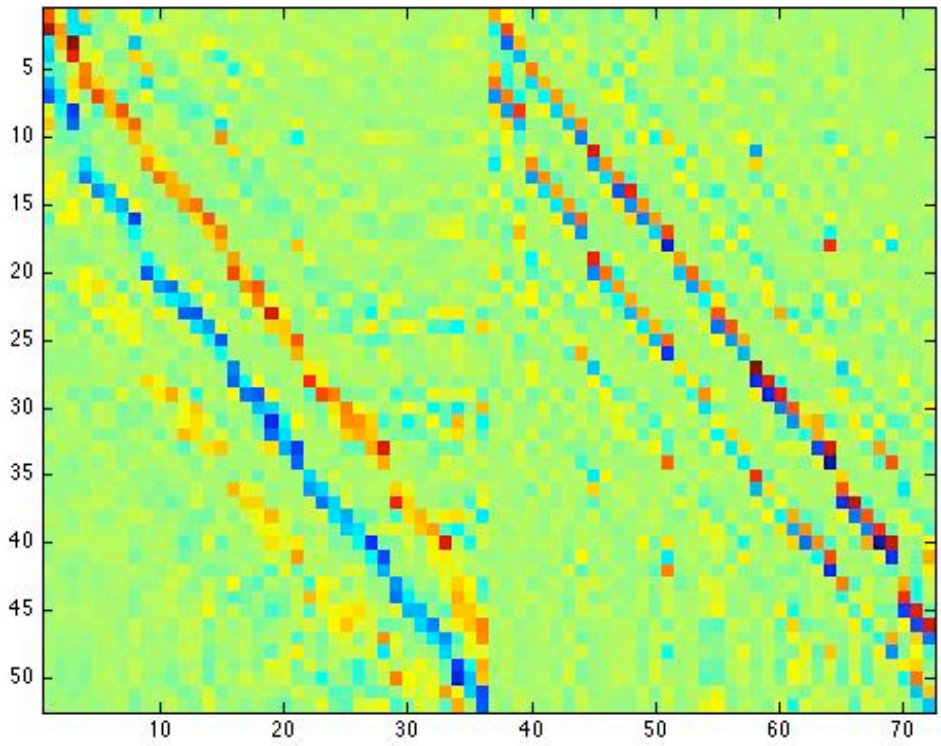


Figure 48: The WFS A reconstructor shown colour-coded to illustrate the 52 x 72 element matrix structure. In open loop operation, the reconstructor is multiplied by a vector of 72 centroid measurements (*x* and *y* pairs for all 36 subapertures) to generate 52 new DM actuator commands.

---

[9] We use low cost plastic CD cases that do a reasonable job at simulating static atmospheric turbulence (approximating the Kolmogorov power spectrum; see Appendix A)

[10] We measure the absolute plate scale of WFS A by rotating a plate of known thickness placed in the diverging beam before the collimator through a fixed angle which produces a known offset in arcseconds.

We measured the RMS scatter in the relation between the WFS A and B plate scales at just 0.06 arcseconds (Figure 49), which corresponds to a registration wavefront error of 70 nm. This WFS A to DM registration error is the VOLT non-common path error, σ^2_{NCP}, and makes up part of σ^2_{other} in our total VOLT open loop WFE budget.

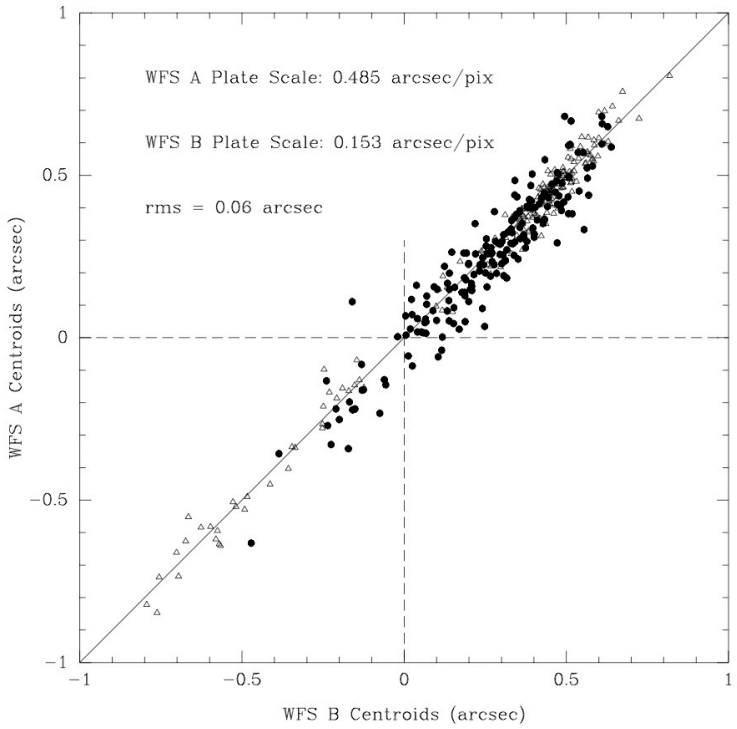


Figure 49: Vertical (filled circles) and Horizontal (open triangles) centroids measured on WFS A and WFS B. The standard deviation of the centroids around a unity slope is 0.06 arcseconds, which translates into a registration error of just 70 nm.

Once WFS A and WFS B are aligned and we have built the WFS A reconstructor, VOLT can apply open loop corrections. We initially tested that VOLT is working by correcting aberrations from static phase screens in single steps. After establishing that the resultant corrected images are still diffraction-limited, we can align VOLT to the telescope and prepare to observe.

3.7 VOLT Observations

VOLT was designed to be both a lab and on-sky experiment. Once we proved VOLT worked in the lab, it could be mounted on the DAO 1.2 m telescope. The DAO 1.2 m telescope is configured to bring light to what is called a Coudé focus, a room located below and to the side of the telescope away from the moving telescope and dome. This is very convenient for installing a test instrument like VOLT because there is no requirement to design the instrument for changing gravity angles (as is the case for the Cassegrain configuration, in which instruments are mounted off the back of the moving telescope), which adds considerable complexity and cost.

The VOLT test bench is located in the Coudé room very close to where the telescope light passes through a pinhole in the wall, using a mirror cantilevered off the edge of the bench to divert light to VOLT rather than to the spectrograph, the regular observing mode of the 1.2 m telescope. The VOLT electronics (Control PCs and associated peripherals, DE64 amplifier, and WFS camera power supplies) are located immediately in front of the test bench to accommodate the 3 m CameraLink cables to the WFS cameras and the cable from the DM to the DE64. Ideally, one would want all heat-generating electronics to be in another room away from the optics to prevent the rising warm air from introducing turbulence in the beam on the bench, but that would have meant more difficulty during integration and testing (plus there was not enough free space in the 1.2 m control room to reasonably fit in our hardware). It was very convenient to be able to have real time feedback from the cameras via the computer monitors while performing alignment on the bench.

3.7.1 September 2007 Run

The first attempted on-sky VOLT observation was in September, 2007. VOLT was moved from the integration and test area of HIA and up the hill to the DAO 1.2m telescope and installed in the Coudé room. A laser was mounted on the VOLT bench, and the beam aligned to be level, parallel to the table. This was used to back-propagate light through VOLT and off the 45° fold mirror to align the instrument as a unit with the Coudé feed pinhole and telescope. This first alignment of VOLT in-situ was challenging, causing delays to running the instrument on-sky. We were still experiencing problems with the Alacron FF1303 framegrabbers (as mentioned in Section 3.5.2) at this time. We did manage to take some wavefront measurements of the very bright star Vega (m_R = 0.0), and discovered that the SNR was much lower than expected. Although conditions were not ideal as it was a bit hazy, this low SNR prompted us to revisit our estimates of the system throughput and the 1M150 detector readnoise (discussed in Section 3.5.2).

3.7.2 January 2008 Run

The observing conditions of the January 2008 run were good, but we were plagued by persistent technical problems with the FF1303 framegrabbers and then by failure of the DE64 DM driver electronics (which was later traced to a faulty FPGA and replaced by LAOG). At the time of this run, we decided to abandon our efforts to get the FF1303 framegrabbers working reliably, and placed an order for the EDT framegrabbers. Nevertheless, WFS A open loop measurements were taken for Capella (m_R = 0.08) and the power spectral density (PSD) of the measured turbulence was consistent with the Kolmogorov spectrum (see Appendix A). The low SNR was still a major concern,

prompting us to make a change in the VOLT beamsplitters to boost the amount of light to the WFSs.

3.7.3 April 2008 Run

New beamsplitters were installed in VOLT to improve the throughput to WFS A and B: the beamsplitter before WFS A went from a 50/50 percentage split to 70/30, and the second beamsplitter, before WFS B, went from a 50/50 to a 96/4 split. The new EDT framegrabbers and new control computers were still being integrated leading up to the on-sky run, leaving too many technical issues yet to solve, however the run would have been lost anyway due to bad weather.

3.7.4 May 2008 Run

On May 22, 2008, we had our first success applying an open loop correction on-sky. During setup we ensured that VOLT was receiving unobstructed science target light using WFS A. We took a long exposure (50 - 100 ms) on WFS A, and aligned the telescope to VOLT by adjusting the pointing and focus to center all the spots in the appropriate subapertures.

The night was clear, but the seeing was poor; we measured the FWHM of Arcturus (m_R = 0.3) to be 2.5 arcseconds in I-band which corresponds to an r_0 = 4 cm at 500 nm. This was done by taking a series of 2 second science camera exposures using a 1/100th neutral density filter; 5 exposures were co-added to produce the image at left in Figure 50. After aligning Arcturus on WFS A and WFS B, we were able to operate VOLT in open loop

with a sampling frequency of 750 Hz on WFS A and observed significant sharpening of the PSF, as shown at right in Figure 50, and in Figure 51.

The sampling frequency of WFS B was substantially lower. Less than half the light that arrives at WFS A is available at WFS B due to the 70/30 beamsplitter. Furthermore, the light is spread out over 10 times more pixels on WFS B[11]. Therefore we were only able to use a sampling frequency of 50 Hz for WFS B. Even at these sampling frequencies, we were operating in a very low SNR regime with the weighted centre of mass centroiding algorithm we were using. The presence of substantial pattern noise on the 1M150 detectors forced us to raise our threshold a factor of 6 greater than the read noise. The peak flux above that threshold was just a factor of 3 - 5 greater, so we were introducing substantial WFS noise. Despite these problems, we still were able to measure an on-sky Rejection Transfer Function (RTF), the most important metric for evaluating VOLT performance.

[11] On chip binning is not available on CMOS detectors, so there is a significant penalty to over-sampling the PSF when read-noise limited.

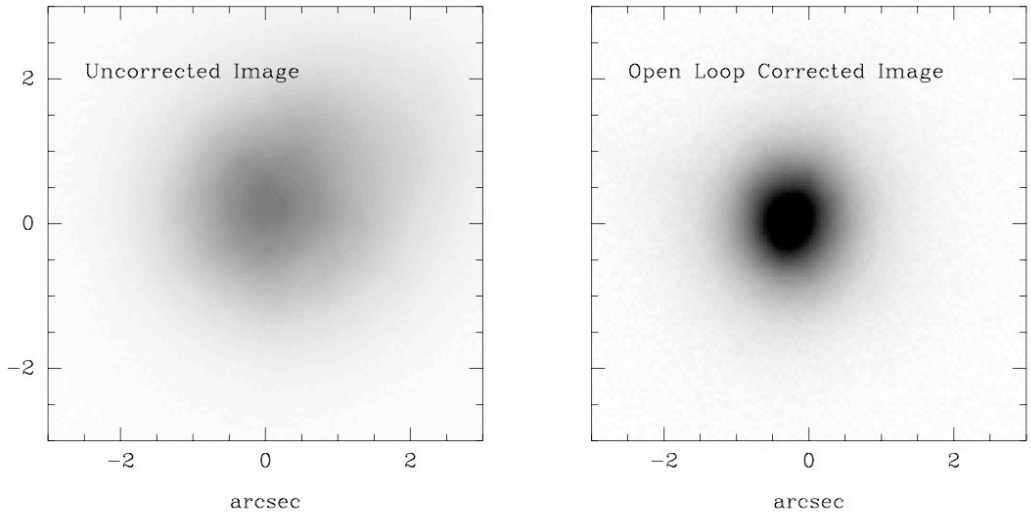


Figure 50: VOLT images of Arcturus from May 22, 2008. In the left panel, the I-band FWHM of the uncorrected image is 2.5 arcseconds, which corresponds to r_0 = 4 cm at a wavelength of 500 nm. With the open loop wavefront sensor taking frames at 750 Hz, we obtained significant image correction, with the FWHM dropping to 0.5 arcseconds. Both exposures were 20 s, and both images have the same log stretch, demonstrating the factor of 5 increase in the peak flux.

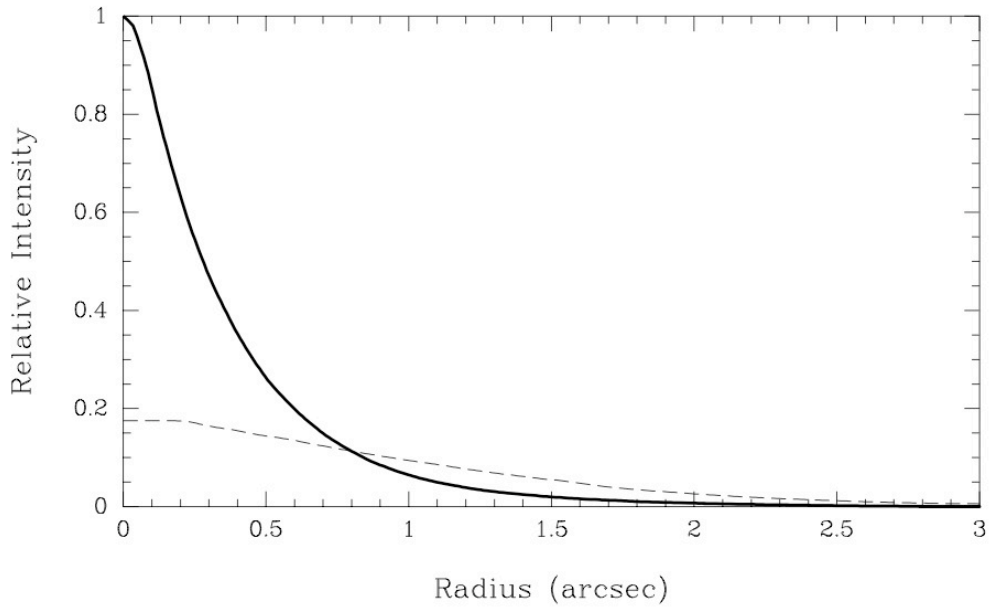


Figure 51: Radial profiles of the two images of Arcturus shown in Figure 50. The uncorrected image is shown with a dashed line, and the open loop corrected image with a heavy solid line. These profiles show the significant reduction in FWHM with the open loop correction applied (2.5 arcseconds to 0.5 arcseconds FWHM), and the factor of 5 increase in the peak flux. The level of correction observed is consistent with our lab measurements of the WFS noise error, which dominates the VOLT error budget.

3.5 Rejection Transfer Function

The key performance metric of VOLT is the Rejection Transfer Function (RTF). The RTF of a control system is the transfer function that describes how much the system suppresses or amplifies noise, as a function of frequency.

While observing with VOLT, we store all centroids measured for WFS A and WFS B. In post processing, we project these measurements onto low order Zernike polynomials (see Appendix A) in order to measure the power spectral density (PSD) of the turbulence and that of the residual after correction (see Figure 52). The power spectrum of the open loop measurements from WFS A show that the power of the measured turbulence was proportional to $f^{2.64}$, where f is the frequency, as expected from theory ($f^{8/3}$) (see Appendix A). At low temporal frequencies, we see that the residual PSD measured by WFS B is substantially lower. We are limited to measuring the VOLT correction to 25 Hz because the low SNR of spots measured on WFS B drive the need for longer exposure times, and thus lower sampling frequencies. If we take the ratio of these PSDs, we construct the open loop RTF measured on-sky (Figure 53). Shown with the on-sky RTF is the lab-measured RTF. For the lab measurement, we used only measurements from WFS B. We took two time series of centroids with WFS B, with and without the open loop correction being applied. We found, however, that we can not take these time series while the system runs only on WFS noise alone, as WFS A has substantially higher WFS noise than WFS B at high SNR (which is a product of the much finer WFS B pixel scale). Therefore, we need to inject noise into the system that is substantially higher than the WFS A noise floor. This is done by blowing hot air from a heat gun through the beam

during the measurements. Figure 53 shows that at low frequencies, the lab RTF measurement begins to level off, due to the higher noise floor of WFS A being translated into a decrease in perceived performance by WFS B. The on-sky RTF measurement is not subject to this effect at frequencies above ~5 Hz because the amount of atmospheric turbulence is significantly greater than the turbulence source used for the lab measurements. The lab-measured RTF shows that at 750 Hz, we achieve some correction at frequencies less than 70 Hz. The peak of the RTF is close to 6 db and 187.5 Hz ($F_s/4$), the expected overshoot of an open loop system with a simple controller (Figure 17). The slope of the RTF is 17 db of rejection per decade in frequency, which is consistent with an open loop control system that has a delay slightly longer than one frame (Hampton 2008). By examing Figure 52, we see though that the RTF below 5 Hz will flatten out. This is a feature also seen from VILLAGES (Gavel, 2008), but it is not yet understood why open loop RTFs display this peculiar behavior at low frequencies.

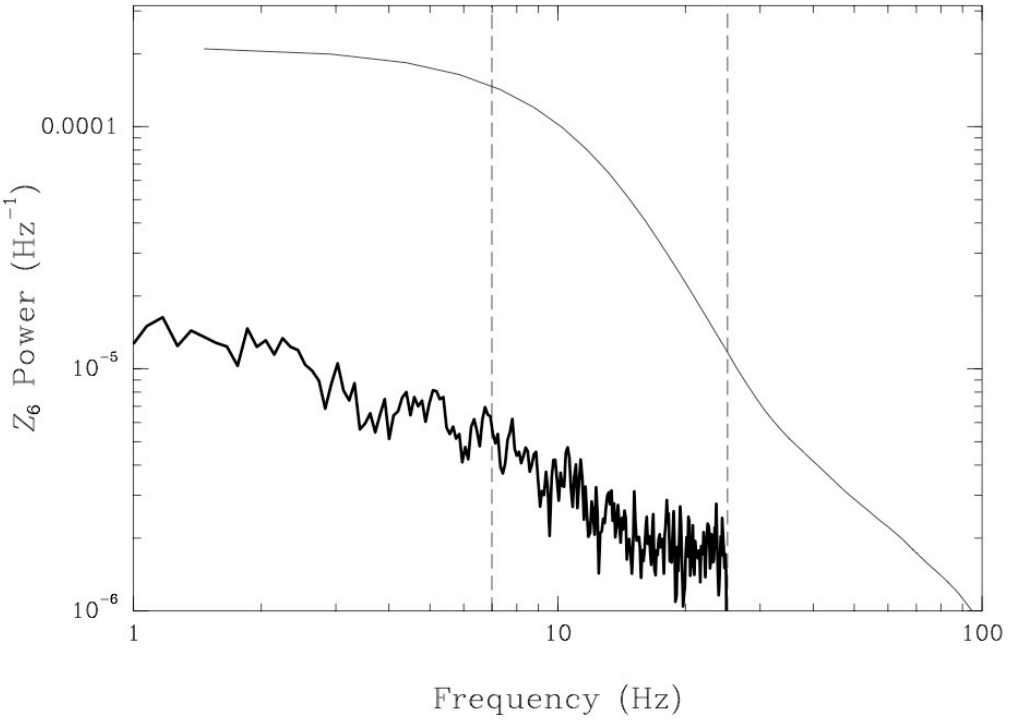


Figure 52: Relative power spectral densities of the atmosphere (measured by projecting the open loop WFS A centroids onto the sixth Zernike polynomial; thin solid line) and the open loop corrected wavefront (measured by projecting the open loop WFS B centroids onto the sixth Zernike polynomial; thick solid line). The maximum frequency to which we can measure the WFS B PSD is 25 Hz, a limitation set by the SNR of the WFS spots on WFS B. The WFS A atmospheric PSD between the two dashed lines is proportional to $f^{2.64}$, which is very close to $f^{8/3}$ expected for Kolmogorov turbulence. The difference between the two curves shows that we are obtaining a significant open loop correction.

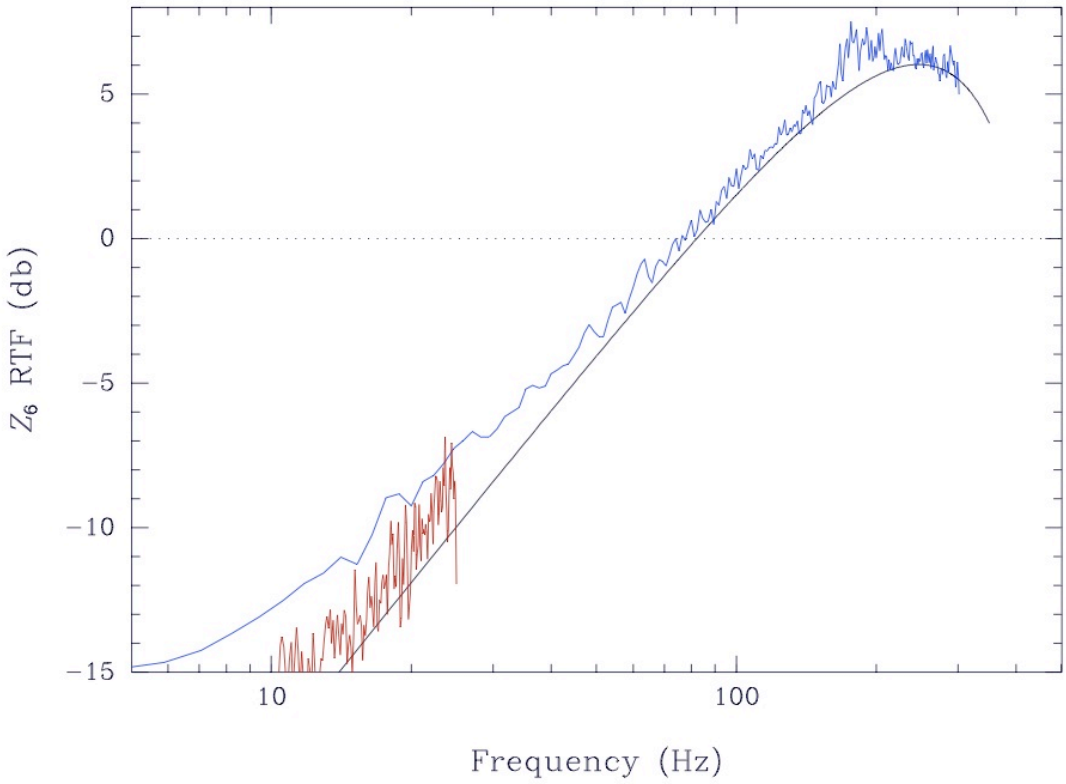


Figure 53: The VOLT rejection transfer function (RTF) for the sixth Zernike Polynomial, theoretical (black line), measured in the lab (blue line) and on-sky from observations of Arcturus (red line). By taking the ratio of the two PSDs between the dashed lines shown in Figure 52, we are able to measure the on-sky RTF of VOLT operating in open loop at 750 Hz.

3.6 VOLT Wavefront Error Budget

By using our LED calibration light source to match the relatively low light-levels of Arcturus observed by VOLT, we were able to estimate that the contribution to the VOLT error budget from WFS A noise was between 250 nm and 500 nm (we adopt 300 nm here). The next greatest VOLT wavefront error term also had little to do with open loop operation; recall that the PAOLA simulations indicated that we could expect another 200 nm of temporal wavefront error from control loop lag. The 70 nm of non-common path error due to the misalignment of WFS A to WFS B and the DM is relatively

inconsequential in comparison. The quadrature sum of all these terms, plus the 50 nm of fitting error and 10 nm of open loop 'go-to' error as in Figure 34 is then calculated as such:

$$\begin{aligned}\sigma_{VOLT} &= \sqrt{\sigma_{VOLT}^2} = \sqrt{\sigma_{WFS}^2 + \sigma_{temporal}^2 + \sigma_{NCP}^2 + \sigma_{fitting}^2 + \sigma_{OL}^2} \\ &\approx \sqrt{300^2 + 200^2 + 70^2 + 50^2 + 10^2} \\ &\approx 370nm\end{aligned} \quad (12)$$

This estimate of the VOLT total wavefront error agrees qualitatively with the 0.5 arcsecond FWHM observed VOLT performance (Figure 50 and Figure 51) and earlier CAOS simulations (Figure 22).

3.7 Summary

The goal of VOLT was to demonstrate open loop control on-sky and in the lab using a simple, on-axis natural guide star. The major components of the design are an ALPAO 52 actuator DM, an imaging CCD with a pixel scale sufficient to well sample a diffraction-limited image from the DAO 1.2m telescope, and three wavefront sensors tasked to sense: 1) the open-loop atmospheric turbulence (WFS A), 2) the residual wavefront error after the WFS A slopes are used to generate DM commands (WFS B), and 3) the DM shape using an independent light source (WFS C).

I was responsible for building the real-time control system for VOLT, involving a mix of hardware assembly, integration, and troubleshooting, and software tasks at various levels – from low-level hardware driver installation and configuration, to operating system optimisation, to C/C++ real-time pipeline development, to MATLAB interfacing

and simulation. I assisted Dr. David Andersen in optomechanical component installation, optical alignment, and VOLT integration in the lab and at the telescope, and in data reduction and analysis for system and component characterization. During our on-sky experiments we worked as a team to operate the 1.2 m telescope and VOLT together and achieved a successful open loop AO demonstration, thus retiring the greatest risk in the way of developing Multi-Object Adaptive Optics systems for ELTs.

We were able to demonstrate open loop control on-sky on May 22, 2008. While observing Arcturus ($m_R = 0.3$), we were able to obtain a substantial improvement in image quality (from 2.5 arcseconds FWHM to 0.5 arcseconds FWHM) while operating VOLT at a frame rate of 750 Hz using WFS A, consistent with simulations. We estimate that the residual wavefront error after open loop correction was ~400 nm, which is consistent with our measurements of WFS A noise (~300 nm), lag (200 nm), registration (70 nm), fitting (50 nm) and open loop 'go-to' (10 nm) wavefront errors (the 370 nm stated earlier is approximate given the large uncertainty in the WFS A noise). We projected the centroids measured from WFS A and WFS B onto Zernike polynomials, and were able to show that the open loop WFS power spectrum of turbulence has a slope proportional to $f^{2.64}$, very close to the expected slope of $f^{8/3}$. Using the residual wavefront measurements of the scoring WFS B that were taken simultaneously, we were able to measure an on-sky RTF up to 25 Hz. This section of RTF is in good agreement with the lab measured RTF, a very good result. Overall, the lab-measured RTF meets our expectations based on analysis of the simple open loop controller.

VOLT is one of only two on-sky open loop AO systems. The VIsible Light LAser Guidestar Experimental System (VILLAGES) [13, 14] ran in open loop on-sky within months of VOLT and measured a similar RTF, including the low frequency flattening seen with VOLT. However, VOLT has advantages over VILLAGES for pursuing further study of open loop control. VILLAGES is being tested on the 1 m Nickel Telescope at Lick Observatory at Mt. Hamilton, California. It is a Cassegrain instrument and hangs off the back of the telescope, and thus experiences the complications of a changing gravity vector and telescope vibrations. The primary objective of VILLAGES is not to demonstrate open loop control, but rather to demonstrate a number of different AO proofs of concept, including the feasibility of MEMS mirrors for astronomical AO, the use of pulsed lasers for efficient generation of LGSs with a minimum of laser power, up-link AO correction for the lasers, and visible light AO. VOLT's single purpose is to explore open loop control, and we are in a position to make several advances in this direction.

Initially, we had hoped to achieve diffraction-limited performance for bright stars with VOLT, but after measuring the low telescope throughput (including 5 aluminum mirrors) and the high 230 e^- read noise of the 1M150 WFS cameras, our expectations of the initial VOLT performance were lowered. However, significant open loop correction was achieved with VOLT, and diffraction-limited performance is likely attainable with lowering the lag and WFS noise wavefront errors, and we will be able to sense how changes in DM to WFS alignment (and perhaps other open loop contributions to the wavefront error budget) affect the Strehl ratio. We are looking at a number of factors which should improve VOLT's performance:

- The aluminum mirrors in the DAO 1.2 m optical train will be replaced in summer 2008 by silver mirrors along with a re-coated primary, which should substantially increase the telescope throughput.
- We are considering new open loop control schemes which could be used to filter the WFS noise.
- We have started looking at using a Gaussian convolution of our subapertures to increase the signal while suppressing particularly the detector pattern noise[12].
- Lower read noise cameras with higher sensitivity could be purchased; the cameras used in VOLT were far from optimal; their selection was largely driven by budget.
- A faster deformable mirror that will allow higher control loop bandwidths.
- If a multi-processor (SMP) Linux kernel version could be found or modified that would satisfy the specialised interface boards, the maximum bandwidth of the control loop could be increased (provided a suitable DM exists) from the time savings that could be made by parallelizing the WFS centroiding and reconstructor multiplication steps.

Despite the obvious room for improvement in various areas of the VOLT design, we achieved our objective – a successful demonstration of open loop AO – and at a very

---

[12] Similar in form to weighted centroiding [36], we have performed static tests in the lab implementing this approach, and at low SNR we have been able to reduce the WFS noise from ~400 nm to 270 nm. We expect this change should not significantly increase the frame delay or wavefront error due to computational lag. It will particularly benefit WFS B, which has very large spots and considerably less light than WFS A. Using this method, we will be able to use WFS B to score the open loop performance at much higher temporal frequencies.

reasonable cost (~$40,000 CAD, excluding the cost of the ALPAO DM52 and electronics which were already available).

Chapter 4
Conclusions

Multi-Object Adaptive Optics will provide localized correction around a number (5 - 40) of selected science objects spread around the FOV, offering a larger accessible FOV than MCAO – up to 10 x 10 arcminute – compared to the 0.5 - 2 arcminute limitation of current MCAO systems and those under development. MOAO does pose significant challenges that need to be retired before being implemented on 8 to 30 m telescopes, the most significant being open loop control. Open loop control introduces unique requirements on the AO system: 1) The WFS needs to have a high dynamic range, as it senses the full uncorrected atmospheric turbulence, 2) DM hysteresis and non-linearity need to be well-understood and the their effect mitigated, 3) Alignment and calibration become more challenging, and 4) Non-common path (NCP) errors of open loop AO systems – due to the DM and other optics being downstream of the WFS and thus not being seen by it – may degrade image quality when small problems arise.

The first tool capable of Monte Carlo performance simulation of an ELT with integrated AO – able to run on a single computer – is the Linear Adaptive Optics Simulator (LAOS). LAOS is an end-to-end AO simulator and can incorporate the model of any user-defined telescope within a range of types and sizes, bounded by the computational speed and memory constraints of the system it is running on and/or what the user considers an acceptable amount of time required to run the simulation. Large-scale simulations such as those for MCAO instruments for the Thirty Meter Telescope can easily require run times on the order of a week or more. The MATLAB C MEX library provides the ability to write custom C functions that are compiled so they can be

called by any MATLAB script. Significant performance improvements were realised with this approach, first with a single-threaded implementation which produced a 2.5 times overall speed-up for the NFIRAOS MCAO test case, and then with a multi-threaded implementation – taking advantage of the lack of data dependency between phase screen layers – which pushed the speed-up to 3.0 times for this same test case. The advantage of this parallelization of LAOS will become more apparent with larger problems, especially MOAO.

The goal of VOLT was to demonstrate open loop control on-sky and in the lab using a simple on-axis natural guide star, and thus help pave the way for future MOAO instrument development. This was achieved on May 22, 2008 while observing Arcturus ($m_R = 0.3$). We were able to obtain a substantial improvement in image quality (from 2.5 arcseconds FWHM to 0.5 arcseconds FWHM) while operating VOLT at a frame rate of 750 Hz using WFS A, consistent with simulations. We estimate that the residual wavefront error after open loop correction was ~400 nm. Our measured open loop WFS power spectrum of the atmospheric turbulence has a slope proportional to $f^{2.64}$, very close to the expected slope of $f^{8/3}$. We were able to measure an on-sky RTF up to 25 Hz in good agreement with the lab measured RTF, matching the characteristics of a simple open loop controller.

We had originally hoped to achieve diffraction-limited performance for bright stars with VOLT, but low telescope throughput and high WFS read noise – effects that can be mitigated with better optics and a higher budget – made our expectations more modest.

Significant open loop correction was achieved with VOLT, and our new goal is to lower the lag and WFS noise wavefront error terms sufficiently to achieve diffraction-limited performance. We are looking at a number of factors which should allow this, including 1) new telescope optical coatings, 2) new open loop control schemes to improve WFS noise filtering, 3) Gaussian convolution for improved centroiding noise rejection, 4) wavefront sensor cameras with lower read noise and higher sensitivity, 5) a deformable mirror with a higher bandwidth limit, and 5) improved real-time processing performance with parallelized centroiding and reconstruction phases, to allow higher control loop bandwidths and thus a faster response to the evolving atmospheric turbulence that causes image blurring. We would also like to better understand the open loop rejection transfer function we and others have observed, in particular the flattening at low frequencies that diverges from theoretical predictions.

To summarise, the thesis work I have described here is valuable in two key ways: 1) Large-scale simulations of Extremely Large Telescopes, along with the variety of adaptive optics systems proposed for them, have become significantly more feasible to run with LAOS, reducing simulation times by factors of 5 or more, and 2) The knowledge and experience gained from designing, prototyping, integrating, and in-lab and on-sky testing of the Victoria Open Loop Testbed, conducted primarily by Dr. David Andersen and I, will serve as a valuable proof of concept and reference for building open loop adaptive optics systems, which should pave the way for the development of Multi-Object Adaptive Optics instruments for future Extremely Large Telescopes.

Bibliography

[1] Phys134: Astro Fundamentals, Liverpool John Moores University, Physics Department website:www.astro.ljmu.ac.uk/scopes.html

[2] Gemini Observatory Adaptive Optics Site Characterisation website: http://www.gemini.edu/sciops/instruments/adaptiveOptics/Seeing.html

[3] C. Max, “Introduction to Adaptive Optics and its History”, NSF Center for Adaptive Optics, University of California at Santa Cruz, DOE Lawrence Livermore National Laboratory, American Astronomical Society 197th Meeting

[4] Lawrence Livermore National Laboratory and NSF Center for Adaptive Optics

[5] Robert K. Tyson, Benjamin W. Frazier, "Field Guide to Adaptive Optics", SPIE Press, 2004.

[6] Francois Rigaut and Jean-Rene Roy, Editors Gemini Observatory, "The Science Case for the Multi-Conjugate Adaptive Optics System on the Gemini South Telescope", Version 2.0 PRT-AO-G0107. October 29, 2001.

[7] F. Rigaut, B. Ellerbroek, R. Flicker, "Principles, Limitations and Performance of Multi-Conjugate Adaptive Optics", SPIE Vol. 4007, 2000

[8] E. Marchetti, R. Brast, B. Delabre et al, "On-sky Testing of the Multi-Conjugate Adaptive Optics Demonstrator", The Messenger, Vol. 129, p. 8-13, September 2007

[9] G. Herriot, P. Hickson, B. Ellerbroek et al, "NFIRAOS: TMT narrow field, near-infrared facility adaptive optics", Proc. SPIE Vol. 6272, 2006

[10] F. Assemat, E. Gendron, F. Hammer, "The FALCON concept: multi-object adaptive optics and atmospheric tomography for integral field spectroscopy. Principles and performances on an 8 meter telescope.", February 5, 2008

[11] Eikenberry. S., et al., “IRMOS: The near-infrared multi-object spectrograph for TMT,” SPIE 6269, 62695W (2006).

[12] R. Conan, P. Hampton, "Thirty Meter Telescope -- Woofer-Tweeter system model", Issue 1.0, UVic AOLab-WT-M2, University of Victoria, September 14, 2005

[13] Ammons, S.M., Gavel, D.T., Reinig, M.R., Dillon, D.R., Morzinski, K.M., “On-sky demonstrations of open-loop Shack-Hartmann wavefront sensing with villages,” Proc. SPIE 7015 (2008)

[14] Gavel, D.T., et al. “Visible light laser guidestar experimental system (Villages): on-sky tests of new technologies for visible wavelength all-sky coverage adaptive optics systems,” Proc. SPIE 7015 (2008)

[15] Myers, Richard M., 2008

[16] D. Andersen, M. Fischer, et al. “VOLT: The Victoria Open Loop Testbed”, SPIE (2008)

[17] L. Gilles, B. Ellerbroek, “LAOS: Linear Adaptive Optics Simulator”, 2006

[18] Herriot, G. et al. “NFIRAOS – TMT’s initial adaptive optics system”, Proc. SPIE 7015 (2008)

[19] The MathWorks, Inc., 1994-2006

[20] Advanced Micro Devices, “Compiler Usage Guidelines for 64-Bit Operating Systems on AMD64 Platforms”, 32035, Rev. 3.18, June 2005.

[21] Sun Microsystems Inc., “Sun Fire V40z Server datasheet”, 2006.

[22] K. Bandara, D. Andersen, "Modeling the performance of Victoria Open Loop Test bed (VOLT) using Code for Adaptive Optics Systems (CAOS)", v1.0, May 29, 2007

[23] Murray Fletcher, David Andersen, Mike Fischer, Laurent Jolissaint, Jean-Pierre Véran, "VOLT WFS Modeling and Optical Layout v1.0", November 7, 2006

[24] DALSA, “1M28, 1M75, and 1M150 Camera User’s Manual”, 03-32-00525 rev 05 (2004)

[25] A. Azouz, "Development of a rapid prototyping environment for an adaptive optic bench", December 14, 2007

[26] Alacron Inc., “FastFrame 1303 PCI Board” datasheet

[27] DALSA, “CA-D1 Camera User’s Manual”, 03-32-00176 (2001)

[28] Nikon MicroscopyU website: www.microscopyu.com, Fellers, K. Vogt, M. Davidson, "CCD Signal to Noise Ratio", National High Magnetic Field Laboratory, The Florida State University

[29] ALPAO, "DM52 Deformable Mirror product datasheet", www.alpao.fr

[30] WYKO Corporation, "WYKO Optical Testing Operator's Guide", PN 908-079, v.1.6, May 1995

[31] ALPAO, "DE64 64 Channels Drive Electronics User's Manual", ALPAO-MAN-DE64-001, August 29, 2005

[32] ALPAO, "DM52 52 Actuator Magnetic Deformable Mirror User's Manual", ALPAO-MAN-DM52-001, Issue 1.0, August 29, 2005

[33] ADLink Technology Inc., "PCIS-DASK Data Acquisition Software Development Kit For NuDAQ PCI-bus Cards, User's Guide", Rev. 4.1.2, March 7, 2006

[34] ALPAO, "DM64 Fast Interface User's Manual", ALPAO-MAN-DE64-002, Issue 1.0, August 29, 2005

[35] A. Hilton, "Pipeline Design Document", June 21, 2005

[36] Nicole, M., Fusco, T., Rousset, Michau, V., "Improvement of Shack – Hartmann wave-front sensor measurement for extreme adaptive optics", Opt. Let. 29, 2743 (2004)

[37] D. Andersen, S. Eikenberry et al. "The MOAO System of the IRMOS Near-Infrared Multi-Object Spectrograph for TMT", 2006

[38] Review of Optometry Online website: www.revoptom.com

[39] Herriot, G., "TMT Astrometry Colloquium", TMT.AOS.PRE.07.035.DRF01, TMT Observatory Corporation, September 28, 2007

[40] Primmerman, C.A., Murphy, D.V., Page, D.A., Zollars, B.G., Barclay, H.T., "Compensation of atmospheric optical distortion using a synthetic beacon," Nature, 353, 141 (1991)

Appendix A
Zernike Polynomials and Kolmogorov Turbulence

A useful way to represent wavefront phase aberrations is to use the Zernike infinite polynomial series, where radial (index *n*) and azimuthal (index *m*) polynomials are reduced – in the case of a circular aperture where the radius at its boundary is normalized to unity – to the forms shown in Table A1. The Zernike 'order' refers to whichever is larger, *m* or *n*. The graphical representations in Figure A1 illustrate the shapes of the first six Zernike orders [5].

	n = 0	**n = 1**	**n = 2**	**n = 3**	**n = 4**
m = 0	1		$2r^2-1$		$6r^4-6r^2-1$
m = 1		r		$3r^3-2r$	
m = 2			r^2		$4r^4-3r^2$
m = 3				r^3	
m = 4					r^4

Table A1: Radial Zernike terms up to the 4^{th} order [5].

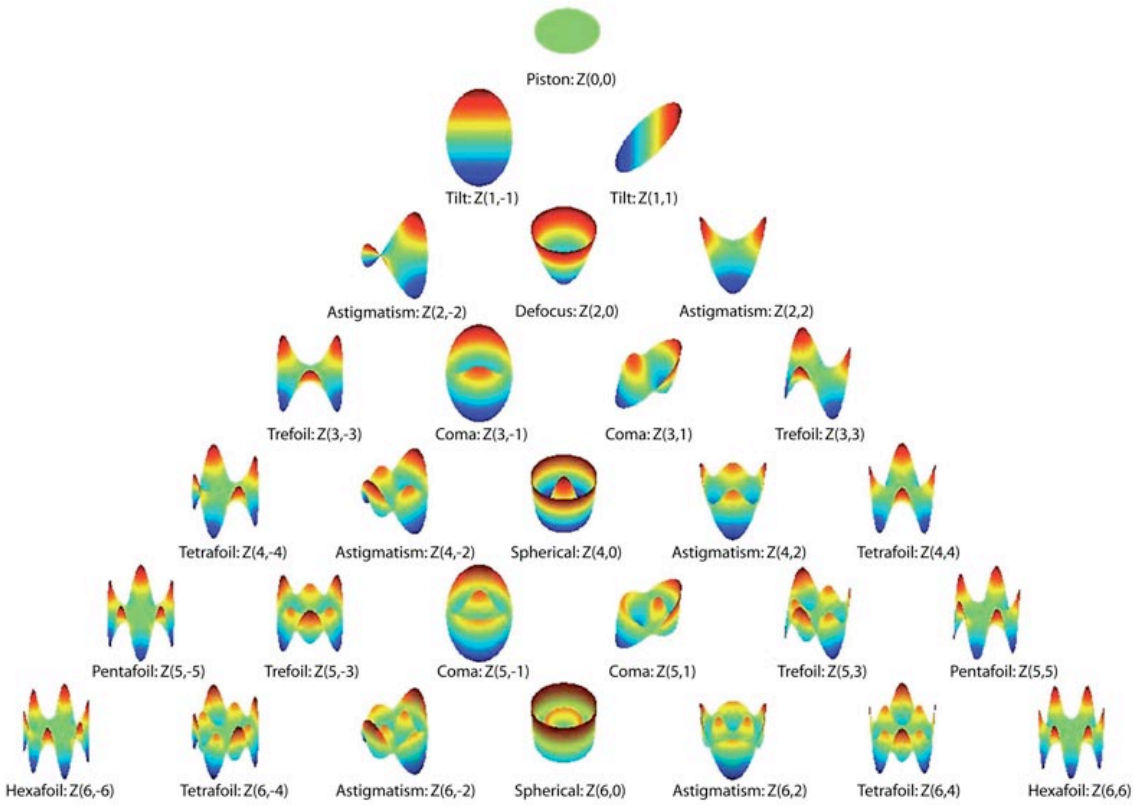


Figure A1: Graphical representation of Zernike terms up to the 6th order [38]

Referring to Figure A2, the temporal power spectrum is related to phase aberration from atmospheric turbulence by the Kolmogorov spectrum, which show that for low frequencies up to the crossover frequency f_c the phase spectrum follows a -2/3 power law. If the tilt Zernike order is removed, the phase spectrum follows a +4/3 power law. The crossover frequency is estimated as follows:

$$f_c \approx 0.7\frac{v}{D}$$

where v is the turbulence-weighted wind velocity and D the telescope diameter.

Above the crossover frequency, the phase spectrum follows a -8/3 power law.

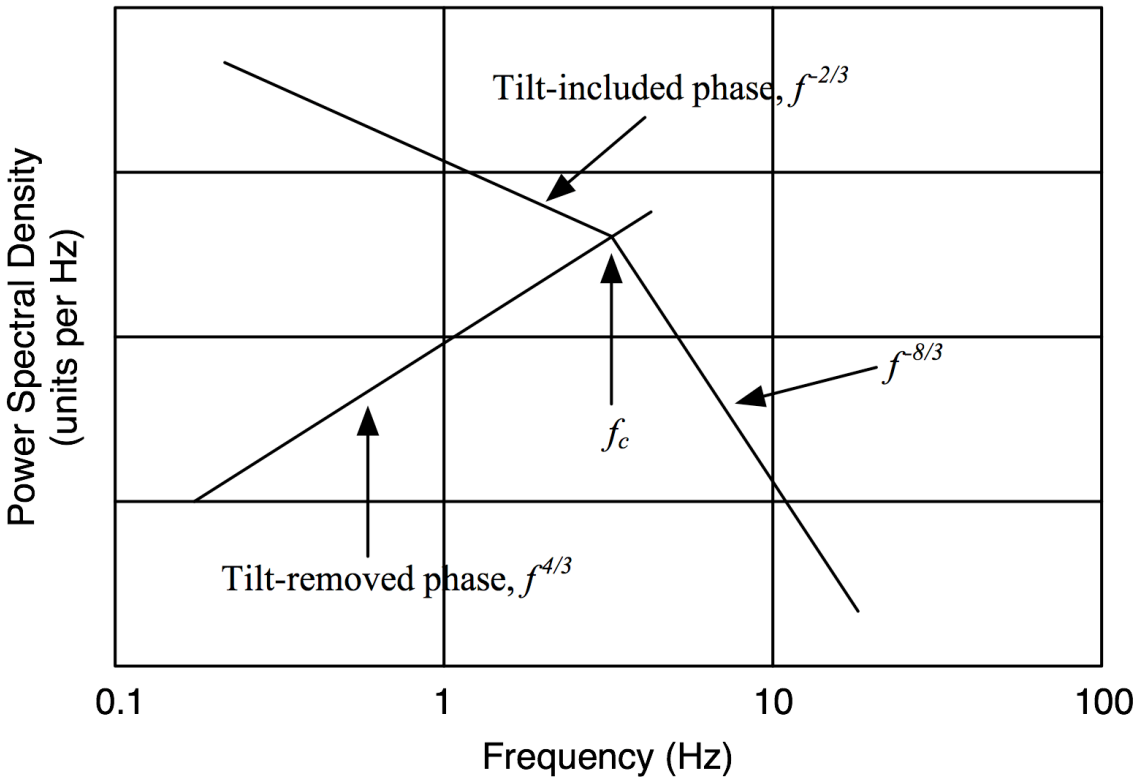


Figure A2: An approximation of the Kolmogorov atmospheric turbulence spectrum [5]

Appendix B:
Performance Enhancements to the Linear Adaptive Optics Simulator

Performance Enhancements to the Linear Adaptive Optics Simulator

Michael Fischer
November 1, 2006

Herzberg Institute of Astrophysics
National Research Council Canada
mike.fischer@nrc.ca

This paper will describe the changes made to the Linear Adaptive Optics Simulator (LAOS) software package in response to the desire to considerably reduce the time necessary to run a large scale adaptive optics simulation. Preliminary results show speed-ups of up to 3 times.

1. Introduction

The LAOS software package is currently implemented as a set of MATLAB scripts and functions. MATLAB has great advantages in its ease of coding and relatively short time required to produce working code, but can lag considerably in execution speed when compared to languages like C, especially when using programming constructs such as large iterative loops. Also, because of its emphasis on user-friendliness which necessitates its 'behind-the-scenes' memory management, MATLAB memory usage is usually not optimised for performance. In the context of LAOS, it was determined that significant performance improvements could be achieved by replacing the bottleneck(s) of the simulator with C implementations, where the use of loops and memory could be explicitly controlled and optimised by a combination of efficient programming and a 'smart' C compiler. C was chosen because it is supported by MATLAB's external interface MEX, which can link MATLAB scripts to compiled C code (or Fortran). The MEX C library (mex.h) provides an API for this interface, allowing compiled MEX functions to be called by any MATLAB script.

2. Identifying the Bottleneck

To begin speeding up LAOS, it was necessary to generate a performance profile to identify where the simulator spends most of its time, ie. the bottleneck(s). MATLAB conveniently has a module called the Profiler which does exactly this. The Profiler is started just before running a LAOS simulation and after completion produces an extensive report of how much time was spent in each function as well as a heirarchy tracing the parent/child function calling relationships, which can be conveniently saved into a set of interlinking HTML files for offline analysis. Once the bottleneck(s) has been identified, it can be determined if there is a reasonable solution promising substantial LAOS performance improvements, by way of significantly speeding up execution at the bottleneck(s). Figure 1 shows the partial Profiler report of a 400 iteration LAOS simulation run. The particular test case was for a 30-m telescope with 2 deformable mirrors (DMs), 6 atmospheric phase screen (PS) layers, no optical surfaces, and 9 physical optics wavefront sensors (WFS), using a conventional centroiding algorithm [1] (path: ~/LAOSdir/DRIVER_DIR/30M_2TT_1TTF/NOISE_FREE/CENT_noelong_POLC_POWFS1)

Profile Summary
Generated 26–Jun–2006 14:40:08 using cpu time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
accphi	86064	139506.080 s	117744.160 s	
sub2ind	416046	22184.560 s	22184.560 s	
ifft2	27043680	13952.610 s	13952.610 s	
Perfievl	7200	11267.970 s	11267.970 s	
Prop	19116	150734.270 s	11228.190 s	
genpowfs_ints	400	160343.560 s	10800.900 s	
fft2	13893432	9738.070 s	9738.070 s	
mkpi	6943116	29711.770 s	9680.950 s	
fftshift	81869442	8935.420 s	8935.420 s	
spbs	97314	3775.550 s	3775.550 s	

Figure 1: MATLAB Profiler results showing the top ten time consuming functions in the LAOS simulation [3].

When interpreting the Profiler results it is most useful to look at the *Self Time* of each function. As explained in the MATLAB Profiler report, "*Self Time* is the time spent in a function excluding the time spent in its child functions. *Self Time* also includes overhead resulting from the process of profiling." It is apparent here that *accphi*, the function responsible for calculating and accumulating contributions by the atmospheric PSs, DMs, and optical surfaces to the phase perturbations *phi* on the incoming wavefront, has the most *Self Time* by a large margin. It was determined by a separate performance test (which allows us to get a more quantitative assessment by avoiding the inherent overhead of the Profiler) that the time spent in *accphi* accounted for approximately 76.7% of the total simulation time. It should be noted that *sub2ind* is a child function called almost exclusively by *accphi*, and thus nearly all the time spent inside *sub2ind* is included in this 76.7% figure. From this analysis it was decided that the way to make the largest immediate gains in speed for LAOS would be to speed up the bottleneck *accphi* as much as reasonably possible.

3. C MEX Implementation

The existing LAOS MATLAB function accphi.m is written in a concise manner and is logically relatively easy to follow, and no real performance improvements could be envisaged by attempting to further optimise this code. As the M language is, single lines of code often encapsulate large iterative looping operations and on-the-fly creation (and resizing) of arrays. This is where efficient C coding can make great gains over MATLAB, by minimising the number of loops needed (ie. performing multiple operations rather than just one within a given loop wherever possible) and closely managing the amount of memory being used (and controlling exactly how much will be dynamically allocated, and when).

To compile C source code under MATLAB, the path to the C libraries installed with MATLAB may need to be added to the file $matlabroot/bin/$arch/librarypath.txt (where $matlabroot is the MATLAB install directory and $arch refers to the particular architecture of the machine). In our case for a 64-bit Linux system we add the following lines:

$*matlabroot*/bin/glnxa64
$*matlabroot*/sys/os/glnxa64

MATLAB allows you to specify the C compiler to be used (we used GNU's gcc) by running

```
>> mex -setup
```

once and following the instructions, creating the file of compiler options mexopts.sh, located inside the MATLAB preferences directory ~/.matlab/*release*/mexopts.sh. You can modify mexopts.sh, changing the options within the section pertaining to your machine architecture and operating system. An example of these options as were used for the 64-bit Linux machine used during development is shown below:

~/.matlab/R2006a/mexopts.sh

```
#----------------------------------------------------------------
            ;;
        glnxa64)
#----------------------------------------------------------------
            RPATH="-Wl,-rpath-link -L$TMW_ROOT/bin/$Arch"

            CC='gcc'
            CFLAGS='-fPIC -fno-omit-frame-pointer -ansi -D_GNU_SOURCE -pthread -fexceptions'

        #Add AMD 64-bit Opteron architecture-specific optimisation flags #to above line later:
        #-O3 -ffast-math -funroll-all-loops -fpeel-loops -ftracer
        #-funswitch-loops

            CLIBS="$RPATH $MLIBS -lm -lstdc++"
            COPTIMFLAGS='-O -DNDEBUG'
            CDEBUGFLAGS='-g'
        #
            CXX='g++'
            CXXFLAGS='-fPIC -fno-omit-frame-pointer -ansi -D_GNU_SOURCE -pthread'
            CXXLIBS="$RPATH $MLIBS -lm"
            CXXOPTIMFLAGS='-O -DNDEBUG'
            CXXDEBUGFLAGS='-g'
```

[3]

Then to compile the source code file.c we use the MATLAB *mex* function as follows:

```
>> mex file1.c
```

This will produce a linkable binary with an extension specific to your operating system, in the case of our 64-bit Linux machine it would be file1.mexa64. According to the documentation, MATLAB 7.1 and 7.2 (R2006a, the most current release) were built with gcc 3.4.5 and it recommends using this or an older version for compilation under MEX. Release 3.4.6 was used in development for this project, although the most current release 4.1.1 has also been tested and appears to work fine, but if you experience errors it might be safer to try using a gcc compiler from the 3.4.x family. It is also advisable to use a MATLAB release of 7.2 or higher, as this was used in development and some differences were noted with compatibility with release 7.1.

3.1 Single-Threaded accphi_C.c

It was first decided to emulate accphi.m – in how it is called and passed / returns parameters – with a single-threaded C version accphi_C.c. This would mean having to make very minimal

changes in the M code of the parent functions calling *accphi*, namely Prop.m and genwfsopd.m, requiring simply that calls to *accphi(....)* be replaced by calls to *accphi_C(...)*

Following is a summary of some of the optimisation choices made in coding accphi_C.c:

- Multidimensional arrays are dynamically allocated as contiguous 1-dimensional arrays of the exact type and size needed. Where MATLAB would always allocate space for double precision floating point numbers regardless of the context of their usage, appropriate data types with smaller footprints in memory have been used. Using 1-dimensional arrays makes looping and array indexing faster.
- Many iterative operations are grouped together (and sometimes reordered, where such reordering will not change the final numerical result) to be executed inside a minimal number of loops, rather than having multiple loops each executing a single iterative operation.
- The MATLAB *find* function is replaced by conditional statements within loops, which avoids some redundant instances of looping through entire arrays simply to compute sets of indices for the arrays. Consequently, considerable memory usage savings were made by avoiding the allocation of these temporary arrays of indices.
- The *sub2ind* step has been removed as a child function call and has been explicitly coded into one line of C code in both instances where it is needed inside *accphi* (thus significantly tackling this secondary bottleneck, as it was reported to be by the Profiler).
- Dynamically allocated memory that was found to be unused after a certain point in the algorithm was reused where possible, to avoid growing the working set by allocating more memory (specifically, the local copy of the *ploc* input array gets overwritten, its space being used for *xploc* and *yploc*).

Using the same test case as mentioned earlier, and using some MATLAB timing functions (ie. cputime, tic and toc), performance comparisons were made for accphi.m vs. accphi_C.c and are summarised in Table 1. Being that the size of the working set for *accphi* varies greatly during a simulation, average execution times for *accphi* over 100 LAOS simulation steps were calculated. The average execution time for one call to accphi.m (taken from a sample of 23184 calls in a 100 iteration LAOS run) was 1.87 seconds, and the average execution time for one call to accphi_C.c (taken from a sample of 21264 calls in a 100 iteration LAOS run) was 0.260 seconds, a speed-up of 7.2 times. Total LAOS simulation time was 56562 seconds (~15.7 hours) for the accphi.m implementation and 22341 seconds (~6.2 hours) for the accphi_C.c implementation, a speed-up of about 2.5 times. This corresponds to a reduction from the original 76.7% of total LAOS simulation time spent in *accphi* to 24.7% of that original time. This result reflects the performance of single-threaded accphi_C.c using optimising compiler flags [4] for our Sun Fire V40z machine with quad dual-core AMD Opteron 875 2.2GHz CPUs with 1Mb L2 cache per core and 32GB RAM [6], running 64-bit Red Hat Linux. Non-optimised compilation suffers a 6% penalty in accphi_C.c execution time in comparison (average execution time for one call to accphi_C.c with non-optimised compilation is 0.276 seconds).

	accphi.m	**accphi_C.c**	**Speed-up**
Total LAOS simulation time (100 iteration run)	56562 s (~15.7 h)	22341 s (~6.2 h)	2.5 X
Number of calls to accphi	23184	21264	N/A
Total time spent in accphi	43432 s (76.7%)	5527 s (24.7%)	N/A
Average time per accphi call	1.87 s	0.260 s	7.2 X

Table 1: Performance comparisons of accphi.m and accphi_C.c.

3.2 Multi-Threaded accphi_C_SMP.c

Multiprocessor (SMP) systems can offer significant speed-ups beyond that possible with a single processor system, if the computational problem can be broken into pieces that can be processed in parallel (ie. they have no data dependencies, such that the result of one step does not depend on the result of another). Although MATLAB does not directly support multithreading, it will 'allow' it when implemented within a MEX context (MATLAB is unaware that multithreading is going on inside the MEX function, so it works, provided that no non-reentrant code is called within that function).

There are further performance gains to be made for LAOS by processing all PS, optical surface, or DM instances in parallel. It is not possible to process all three in parallel, as it would violate the serial nature of the LAOS simulation (and more fundamentally, it would violate the structure of AO feedback loops, as the DM and TTM compensations are generated in response to *phi* perturbations produced by the PSs and optical surfaces). This new parallel version of accphi_C.c which has been developed is called accphi_C_SMP.c.

Changes made to to LAOS to support parallel processing were the following:

- An additional parameter was created in the LAOS parms structure, parms.max_threads, in ~/*LAOSdir*/STORED_GEOMS/*geom*/aper_dm_tomo_wfs/Setup_aper_dm_tomo_wfs.m (where *LAOSdir* and *geom* refer to your LAOS installation directory and the particular stored geometry of your simulation). Manually setting this integer value defines the maximum allowed number of concurrent threads which can be run. Commonly, this would be set to equal the number of processors available on a given system, but performance may differ from system to system, and the optimal setting for parms.max_threads is best determined by testing. This parameter can also serve to limit the number of processors which a LAOS simulation is allowed to use, which could be useful to avoid 'hogging' too many processors on shared SMP systems.

- PS, optical surface, and DM *phi* contributions are registered by a call to accphi_C_SMP.c in a modified Prop.m. Rather than looping through processing one PS, optical surface, or DM at a time by looping calls to accphi.m or accphi_C.c, Prop.m is modified in a way that the looped processing is removed and **all** data pertinent to **all** PSs, **all** optical surfaces, or **all** DMs is passed to accphi_C_SMP.c in a single call. Time and memory requirement savings are made, as Prop.m no longer uses the *find* function to do PS layer / optical surface / DM-specific input data parsing (it can be done faster inside accphi_C_SMP.c), nor does MATLAB have to dynamically allocate (or reallocate) memory to store the result of *find* on every iteration of the loop. This approach also saves MATLAB the overhead of the looping *accphi* function calls. It should be noted that accphi_C_SMP is not an emulation of accphi.m, as accphi_C.c was, in that Prop.m required modification and the parameter list of accphi_C_SMP has grown.

- The only other place *accphi* is called from, genwfsopd.m, was modified to make calls to accphi_C_SMP.c rather than accphi.m / accphi_C.c (this is just the case of having a single layer to be processed).

The implementation of accphi_C_SMP.c can be summarised as follows:

- Working set data for each layer is parsed and indexed by the main thread and packaged with other parameters specific to that layer in preparation for thread creation.

- These layer-specific tasks are then sorted for execution in descending order according to the size of their respective working sets. This results in the largest working set beginning

processing first, which generally minimises the total time to complete all tasks.

- The main thread then creates one child thread per task, which are essentially modified instances of accphi_C.c, up to parms.max_threads. Remaining threads are then spawned to take the place (and processor) of preceding threads as each completes. It should be noted that accphi_C_SMP.c does not detect the number of available processors on a system, but leaves it up to the user to decide how many threads per processor may be assigned (by setting parms.max_threads).

- Each thread calculates its own layer-specific contribution to the phase *phi*, and writes it to a global array accessible to all threads. When all threads have completed, this array is passed back to MATLAB.

A performance evaluation was run with the same LAOS simulation case and system architecture as detailed for accphi_C.c above, demonstrating a further speed-up from 2.5 times for accphi_C.c to 3.0 times for accphi_C_SMP.c. For larger LAOS working sets, a further relative speed-up could potentially be seen.

6. Troubleshooting

There is a potential error which may occur while using accphi_C.c or accphi_C_SMP.c in a LAOS simulation regarding which version of the shared library libgcc is installed on your system along with MATLAB. If you encounter an error like the following, you might have to create a symbolic link for libgcc_s.so.1 inside the directory $matlabroot/sys/os/$arch/ to a more current version elsewhere on your system (likely found in /lib or /lib64):

??? Invalid MEX-file '/LAOS/LAOS_CODE/sim/accphi_C.mexa64':
/opt/matlab/bin/glnxa64/../../sys/os/glnxa64/libgcc_s.so.1: version `GCC_4.2.0' not found (required by /usr/lib64/libstdc++.so.6)

References

1. L. Gilles, B. Ellerbroek, “LAOS: Linear Adaptive Optics Simulator”, 2006.
2. L. Gilles, “LAOS: Adaptive Optics Minimum Variance Wavefront Reconstructor Construction and Modeling”.
3. The MathWorks, Inc., 1994-2006.
4. Advanced Micro Devices, “Compiler Usage Guidelines for 64-Bit Operating Systems on AMD64 Platforms”, 32035, Rev. 3.18, June 2005.
5. T. Yang, http://www.cs.ucsb.edu/~tyang/, University of California Santa Barbara
6. Sun Microsystems Inc., “Sun Fire V40z Server datasheet”, 2006.

Appendix C:
LAOS Performance Modifications Source Code

```c
/* accphi_C.c (single-threaded)

   Computes contribution to a phase profile from a phase screen, dm, or surface

   May 15, 2007 -- NO AMP Pupil Parameters passed into accphi_C

   Mike Fischer - mike.fischer@nrc.ca
*/

#include <mex.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* allowable values for iflag parameter; determine which case to process */
#define AMP_FLAG 4
#define SURF_FLAG 3
#define DM_FLAG 2
#define PS_FLAG 1
#define RECON_PS_FLAG 0

/* define the positions of the parameters in the function call to accphi_C_SMP */
#define LOC_OUT_POS 0
//#define XSCL_POS 1
#define THETA_POS 1
#define H_POS 2
#define LOC_IN_POS 3
#define DXT_POS 4
#define PHI_IN_POS 5
//#define IFLAG_POS 7

/* Entry point */
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{

/* declare pointers to data of mxArray input parameters for readability */
double *ploc, *theta, h, *xloc, dxt, *x;
//double xscl;
//int iflag;

//iflag = (int) mxGetScalar(prhs[IFLAG_POS]);
ploc = mxGetPr(prhs[LOC_OUT_POS]);
//xscl = (double)mxGetScalar(prhs[XSCL_POS]);
theta = mxGetPr(prhs[THETA_POS]);
h = (double)(mxGetScalar(prhs[H_POS]));
xloc = mxGetPr(prhs[LOC_IN_POS]);
dxt = (double)(mxGetScalar(prhs[DXT_POS]));
x = mxGetPr(prhs[PHI_IN_POS]);
```



```c
double *ploc_; /* local copy of input array parameter ploc */
/* pointers to the two columns of ploc_, our local copy of ploc input parameter */
double *xploc, *yploc;
double *phi; /* pointer to MEX-allocated array return parameter for phi */

double minx; /* minimum value in xloc input array */
double ploc_temp;
double eps = 1e-12; /* tolerance on contributions to phase */

int *mapx; /* will calloc mapx array later */
int *ixloc; /* will malloc ixloc array according to size of input array xloc */
int *nploc; /* will malloc nploc array according to size of input array ploc */

int numRow_ploc, numRow_xloc; /* # of rows in ploc and xloc input arrays */
int num_ploc, num_xloc; /* # of elements in ploc and xloc input arrays */
int i; /* loop counter / index */
int dx; /* row and column dimensions of square matrix mapx */
/* xind is a single temporary int rather than an array of
indices as in the Matlab implementation */
int xind;
/* ind2 is a single temporary int rather than an array of
indices as in the Matlab implementation */
int ind2;
int ix, iy, xx, xy; /* nested loop counters / sub-indices */

int num_unident_pts = 0; /* number of unidentified (erroneous) points */
double temp_weight = 0; /* weight of unidentified point */
double max_weight = 0; /* max weight of all unidentified pointe */

int imaxx, imaxp;
double minp, maxp;


/* ------- Let's get to work ------------------------------------------------- */
/* get number of rows (column dimension) of ploc (is "np" in Matlab accphi.m) */
numRow_ploc = mxGetM(prhs[LOC_OUT_POS]);
/* total length of ploc vector (arrays stored as continuous vectors) */
num_ploc = mxGetNumberOfElements(prhs[LOC_OUT_POS]);

/* get number of rows (column dimension) of xloc */
numRow_xloc = mxGetM(prhs[LOC_IN_POS]);
/* total length n of xloc vector */
num_xloc = mxGetNumberOfElements(prhs[LOC_IN_POS]);

/* ---We'll dynamically allocate as many arrays as possible at this point, in
hope that they will be adjacent in memory, and to minimise the number of overhead
penalties we incur for calling the dynamic memory allocation operation ------ */

/* C MEX dynamic array allocation (to facilitate passing phi back to Matlab) */
```

```
/* Automatically initialised full of zeros */
plhs[0] = mxCreateDoubleMatrix(numRow_ploc, 1, mxREAL); /* mxREAL is the double data typ
e for Matlab */
phi = mxGetPr(plhs[0]);

/* Have to use MEX version of malloc, mxMalloc, for local dynamic allocation */
ploc_ = mxMalloc(num_ploc * sizeof(double)); /* allocate local copy of ploc, "ploc_" */
nploc = mxMalloc(num_ploc * sizeof(int)); /* allocate nploc (same length as ploc_) */
ixloc = mxMalloc(num_xloc * sizeof(int)); /* allocate ixloc */

/* Find minimum value inside xloc */
minx = xloc[0];
for(i=1; i < num_xloc; i++)
  {
        if(minx > xloc[i])
        {
        minx = xloc[i];
        }
  }

   for(i=0; i < num_ploc; i++)
    {
 //   ploc_[i] = ploc[i] * xscl;
 //   ploc_[i] = ploc_[i] + h * theta[i/numRow_ploc];
      ploc_[i] = ploc[i] + h * theta[i/numRow_ploc];

   if(i==0)
        {
        minp = ploc_[0];
        maxp = ploc_[0];
        }
   else
        {
        if(minp > ploc_[i])
        minp = ploc_[i];

        if(maxp < ploc_[i])
        maxp = ploc_[i];
        }
    }


while(minx > minp)
{
    minx = minx - dxt;
}

/* invert dxt so we can multiply by it rather than divide (faster) */
dxt = 1/dxt;
```



```
/* Fill in ixloc and calculate dx */
for(i=0; i < num_xloc; i++)
  {
    ixloc[i] = (int)round(((xloc[i] - minx) * dxt)) + 2;

    if(i==0)
    {
    imaxx = ixloc[0];
    }
    else if(ixloc[i] > imaxx)
    {
      imaxx = ixloc[i];
    }
  }
imaxp = (int)ceil((maxp-minx)*dxt + 2);

if(imaxx > imaxp)
    dx = imaxx + 1;
else
    dx = imaxp + 1;
/* dimensions of mapx to be dx X dx */

/* Allocate space for mapx */
mapx = mxMalloc(dx*dx*sizeof(int));

/* Initialise mapx with -1 values to differentiate valid (>=0)
indices and nonvalid (-1) indices */
for(i=0; i<(dx*dx); i++)
  {
    mapx[i] = -1;
  }

/* Compute reshaped indices and load mapx with values from ixloc */
for(i=0; i < numRow_xloc; i++)
  {
    /* "sub2ind" step */
    xind = (ixloc[numRow_xloc + i] - 1) * dx + ixloc[i] - 1;
    mapx[xind] = i;
  }

/* Compute values and load local copy of ploc, "ploc_", and nploc arrays */
for(i=0; i < num_ploc; i++)
  {

    ploc_[i] = (ploc_[i] - minx)*dxt + 2;

    nploc[i] = (int)ceil(ploc_[i]);
    ploc_[i] = nploc[i] - ploc_[i];
```

```
  } /* for */

/* ploc_ is not needed from this point onward, so we reuse its memory space */
xploc = &ploc_[0]; /* xploc points at beginning of the 1st column of ploc_ */
yploc = &ploc_[numRow_ploc]; /* yploc points at beginning of 2nd column of ploc_*/

/* Compute phi contributions */
for(i=0; i<numRow_ploc; i++)
  {
/* these two lines are equivalent to xploc=1-ploc(:,1) and yploc=1-ploc(:,2)
 by use of the pointers into ploc_ above */
    xploc[i] = 1 - xploc[i];
    yploc[i] = 1 - yploc[i];

    for(ix=-1; ix<1; ix++)
      {
    xploc[i] = 1 - xploc[i];

    for(iy=-1; iy<1; iy++)
      {
        yploc[i] = 1 - yploc[i];

        if((xploc[i] > eps) && (yploc[i] > eps))
          {
        xx = nploc[i] + ix;
        xy = nploc[numRow_ploc + i] + iy;

        /* "sub2ind" step */
        ind2 = (xy-1) * dx + xx - 1;

        ind2 = mapx[ind2];

        /* 0 is a valid index in C, so instead we treat all instances
           of -1 (initialised above) as invalid */
        if(ind2 == -1)
          {
            num_unident_pts++;
            temp_weight = xploc[ind2] * yploc[ind2];
            if(temp_weight > max_weight)
              max_weight = temp_weight;
          }
        else
          { /* register phi contribution */
            phi[i] = phi[i] + xploc[i] * yploc[i] * x[ind2];
          //phi[i] = phi[i] + xploc[i] * yploc[i];
          }
          } /* if */
      } /* for */
```



```
      } /* for */
  } /* for */

/* Report unidentified points, if any */
 if(num_unident_pts > 0)
   mexPrintf("%d unidentified points in accphi with max weight %f\n", num_unident_pts,
max_weight);

mxFree(nploc);
mxFree(ploc_);
mxFree(ixloc);
mxFree(mapx);

} /* accphi_C.c */
```

```
/* accphi_C_SMP multi-threaded ACML version

 Computes contribution to a phase profile from multiple phase screens, DMs, or surfaces.
 Uses pthreads multithreading library to process layers in parallel.

 Version 2.2 - June 12, 2007

 Mike Fischer - mike.fischer@nrc.ca


  Matlab call to accphi_C_SMP:

  accphi_C_SMP({loc_out, xscl},{pupmag,
pupmis},theta,h,loc_in,dxt,phi_in,iflag,max_threads);

    Parameter formats and types:
        loc_out:    matrix [(varies) x 2], double
        xscl:       vector [1 x num_layers], double
        pupmag:     matrix [2 x 2 x num_layers], double (pupil magnification, DM case
only)
        pupmis:     vector [1 x 2], double (pupil misregistration, DM case only)
        theta:      vector [1 x 2], double
        h:          vector [1 x num_layers], double
        loc_in:     matrix [(varies) x 3], double
        dxt:        vector [1 x num_layers], double
        phi_in:     vector [1 x (varies)]
        iflag:      scalar, integer
        max_threads:scalar, integer (user-defined max number of concurrent threads
allowed)

*/

#ifndef _REENTRANT
#error Compile with _REENTRANT defined since this uses threads.  Applying it for you...
#define _REENTRANT
#endif

#include <errno.h>
#include <mex.h>
#include <stdio.h>
#include <stdlib.h>
#include "/scr/fischerm/lib/ACML/gnu64/include/acml.h"
#include <math.h>
#include <pthread.h>
#include <semaphore.h>

/* The following is a workaround for spurious signals causing errors with sem_wait --
   in case you don't have the GNU C libraries (it is normally defined in the GNU ----
   version of unistd.h ------------------------------------------------------------- */
```



```
#ifndef TEMP_FAILURE_RETRY
#define TEMP_FAILURE_RETRY(expr) \
    ({ long int _res; \
       do _res = (long int) (expr); \
       while (_res == -1L && errno == EINTR); \
       _res; })
#endif
/* ------------------------------------------------------------------------------- */

//#define DEBUG
//#printdefine TEXT_OUT  /* COMMENT OUT TO DISABLE RUNTIME TEXT OUTPUT */
#define TEST_ARR_OUT

/* allowable values for iflag parameter; determine which case to process */
#define AMP_FLAG 4
#define SURF_FLAG 3
#define DM_FLAG 2
#define PS_FLAG 1
#define RECON_PS_FLAG 0

#define PUPPARMS

/* Define struct for passing data to threads ---------------------------------------- */
typedef struct {
  double xscl;
  double h;
  double dxt;
  int num_loc_out;      /* is num_ploc in calcphi */
  int numRow_loc_out;   /* is numRow_ploc in calcphi */
  int numRow_loc_in;    /* number of rows in loc_in */
  int num_loc_in_index; /* num_xloc_indices in calcphi; # of valid indices into
             loc_in from which to form xloc in calcphi */
  int iflag;
  double *loc_out;
#ifdef PUPPARMS
  double *pupmag;
  double *pupmis;
#endif
  double *theta;
  double *loc_in;       /* xloc in calcphi */
  double *phi_in;       /* x in calcphi */
  double *phi_out;      /* pointer to phi_out array of accumulated phi values to
                pass to plhs[0] Matlab arg */
  int *loc_in_index;    /* emulates index in Prop.m; indices into loc_in from which
                to form xloc in calcphi */
#ifdef TEST_ARR_OUT
  double *test_arr_out; /* pointer to test output array to pass to plhs[1] Matlab arg */
#endif
} thread_data_t;
```

```
typedef struct {
  pthread_t thread_id;
  thread_data_t thread_parms;
} thread_bundle_t;

thread_bundle_t *thread_bundle;
/* ------------------------------------------------------------------------------- */

int num_layers_remaining; /* to keep track of how many layers remain to be processed */

/* Create mutexes to ensure exclusive access to num_layers_remaining and phi_out ---- */
pthread_mutex_t mutex_num_layers_remaining = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex_phi = PTHREAD_MUTEX_INITIALIZER;
/* ------------------------------------------------------------------------------- */

/* Create counting semaphore to control number of concurrently running threads ------ */
sem_t semA;
/* ------------------------------------------------------------------------------- */

int num_concurrent_threads; /* allowed number of concurrently running threads */

static void * calcphi(void *arg); /* calcphi function prototype */

/* ------------------------------------------------------------------------------- */
/* ------------------------------   START_NEXT_THREAD ------------------------------ */
/* ------------------------------------------------------------------------------- */

static void * start_next_thread(void)
{
    int id, err;

    pthread_mutex_lock(&mutex_num_layers_remaining);

    id = num_layers_remaining - 1;
    num_layers_remaining--;

    pthread_mutex_unlock(&mutex_num_layers_remaining);

    /* If there are still layers to be processed, create another thread. */
    if (id >= 0)
    {
        err = pthread_create(&thread_bundle[id].thread_id, NULL, calcphi, \
                &thread_bundle[id].thread_parms);
        if (err != 0)
        {
            mexPrintf("Error %d Can't create thread for layer %d.\n", err, id+1);
            return;
        }
```



```
    }
} /* start_next_thread */

/* ------------------------------------------------------------------------------- */
/* ------------------------------   CALCPHI   ------------------------------------- */
/* ------------------------------------------------------------------------------- */

static void * calcphi(void * arg)
{
    thread_data_t * data = (thread_data_t *) arg;

    double xscl = data->xscl;
    #ifdef PUPPARMS
    double *pupmag = data->pupmag; /* pupmag is known to be an array of length 4 (2x2)
*/
    double *pupmis = data->pupmis; /* pupmis is known to be a vector of length 2 */
    #endif
    double *ploc = data->loc_out;
    double *theta = data->theta;
    double h = data->h;
    double *xloc = data->loc_in; /* xloc is actually a subset of loc_in, indexed by
                    loc_in_index */
    double dxt = data->dxt;
    double *x = data->phi_in;
    double *phi;
    double *phi_out = data->phi_out;

    #ifdef TEST_ARR_OUT
    double *test_arr_out = data->test_arr_out; /* pointer to test output array to pass t
o plhs[1] Matlab arg */
    #endif

    double *ploc_; /* our local copy of ploc input parameter */
    double *xploc, *yploc;  /* pointers to the two columns of ploc_ */
    double *ones; /* dummy vector of 1's for ACML matrix functions */

    double minx, minp = 0, maxp = 0;
    double eps = 1e-12; /* tolerance on contributions to phase */
    double theta_temp[2] = {0, 0};

    int *mapx; /* will calloc mapx array later */
    int *ixloc; /* will malloc ixloc array according to size of input array xloc */
    int *nploc; /* will malloc nploc array according to size of input array ploc */

    int num_ploc = data->num_loc_out;
    int numRow_ploc = data->numRow_loc_out; /* # of rows in ploc input array */
    int numRow_loc_in = data->numRow_loc_in; /* # of elements in xloc, when parsed
                        from loc_in */
    int num_xloc_indices = data->num_loc_in_index; /* # of rows in xloc */
```

```
    int *loc_in_index = data->loc_in_index; /* column indices with which to parse
                          loc_in to get xloc */
    int iflag = data->iflag;

    int i; /* loop counter / index */
    int imaxx = 0, imaxp = 0;
    int dx; /* row and column dimensions of square matrix mapx */

    /* xind is a single temporary int rather than an array of
    indices as in the Matlab implementation */
    int xind;

    /* ind2 is a single temporary int rather than an array of
    indices as in the Matlab implementation */
    int ind2;

    int ix, iy, xx, xy; /* nested loop counters / sub-indices */

    int num_unident_pts = 0; /* number of unidentified (erroneous) points */
    double temp_weight = 0; /* weight of unidentified point */
    double max_weight = 0; /* max weight of all unidentified points */

    /* ------------------------------------------------------------------------- */

    /* ---We'll dynamically allocate as many arrays as possible at this point, in ---
    hope that they will be adjacent in memory, and to minimise the number of overhead
    penalties we incur for calling the dynamic memory allocation operation ---------- */

    ploc_ = malloc(num_ploc * sizeof(double)); /* allocate local copy of ploc, "ploc_"*/
    nploc = malloc(num_ploc * sizeof(int)); /* allocate nploc (same length as ploc_) */
    ixloc = malloc(num_xloc_indices * 2 * sizeof(int)); /* allocate ixloc */
    phi = calloc(numRow_ploc, sizeof(double)); /* local phi results array */
    ones = mxMalloc(numRow_ploc * sizeof(double));
    for(i=0; i < numRow_ploc; i++)
        ones[i] = 1;


    /* Find minimum value inside 2-column xloc (1st 2 columns of loc_in) ------------ */
    minx = xloc[loc_in_index[0]];
    for (i=0; i < num_xloc_indices; i++)
    {
        if ((xloc[numRow_loc_in + loc_in_index[i]] < xloc[loc_in_index[i]]) && \
        (xloc[numRow_loc_in + loc_in_index[i]] < minx))
            minx = xloc[numRow_loc_in + loc_in_index[i]];
        else if ((xloc[loc_in_index[i]] < xloc[numRow_loc_in + loc_in_index[i]]) && \
         (xloc[loc_in_index[i]] < minx))
            minx = xloc[loc_in_index[i]];
    }
    /* ------------------------------------------------------------------------- */
```



```
#ifdef PUPPARMS
    if(iflag == DM_FLAG) /* DM case */
    {

   // loc_out = loc_out * xscl AND loc_out = loc_out * pupmag'
dgemm('N','N', numRow_ploc, 2, 2, xscl, ploc, numRow_ploc, pupmag, 2, 0, ploc_,
numRow_ploc);

// loc_out = loc_out + ones(nout,1) * pupmis
dgemm('N','N', numRow_ploc, 2, 1, 1, ones, numRow_ploc, pupmis, 1, 1, ploc_,
numRow_ploc);

// theta = theta * pupmag'
dgemm('N','N', 1, 2, 2, 1, theta, 1, pupmag, 2, 0, theta_temp, 1);

// ploc=ploc+h*ones(np,1)*theta;
dgemm('N','N', numRow_ploc, 2, 1, h, ones, numRow_ploc, theta_temp, 1, 1, ploc_,
numRow_ploc);

 for(i=0; i < num_ploc; i++)
        {
      // All the following is now done in the section above with ACML functions
      //   ploc_[i] = ((ploc[i % numRow_ploc] * pupmag[i/numRow_ploc] + \
      //         ploc[i % numRow_ploc + numRow_ploc] * pupmag[i/numRow_ploc + 2]) * xscl
) \
          //   + pupmis[i/numRow_ploc] \
      //         + h * (theta[0] * pupmag[i/numRow_ploc] + theta[1] * pupmag[i/numRow_pl
oc + 2]);
   if(i==0)
        {
            minp = ploc_[0];
            maxp = ploc_[0];
        }
   else
        {
            if(minp > ploc_[i])
            minp = ploc_[i];

            if(maxp < ploc_[i])
            maxp = ploc_[i];
        }
    }
}
else /* all cases other than DM */
{
#endif
 for(i=0; i < num_ploc; i++)
   {
```

```
        ploc_[i] = ploc[i]*xscl;
   }

    // ploc=ploc+h*ones(np,1)*theta;
dgemm('N','N', numRow_ploc, 2, 1, h, ones, numRow_ploc, theta, 1, 1, ploc_,
numRow_ploc);
// if(iflag==1)
   //  printf("accphi_C_SMP_ACML: made it past dgemm\n");

   for(i=0; i < num_ploc; i++)
    {
      // The following 2 lines commented because this is now done by ACML call above
      //   ploc_[i] = ploc[i] * xscl;
      //  ploc_[i] = ploc_[i] + h * theta[i/numRow_ploc];

   if(i==0)
        {
        minp = ploc_[0];
        maxp = ploc_[0];
        }
   else
        {
        if(minp > ploc_[i])
        minp = ploc_[i];

        if(maxp < ploc_[i])
        maxp = ploc_[i];
        }
    }
#ifdef PUPPARMS
}
#endif
  while(minx > minp)
  {
    minx = minx - dxt;
  }
    #ifdef DEBUG
    if(iflag==PS_FLAG)
    printf("accphi_C_SMP thread: h = %f, minx = %f, dxt = %f, maxp = %f\n", h, minx, dxt
, maxp);
    #endif
    /* invert dxt so we can multiply by it rather than divide (faster) */
    dxt = 1/dxt;

    /* Fill in ixloc and calculate dx ------------------------------------------------ */
    for (i=0; i < num_xloc_indices; i++)
    {
        ixloc[i] = (int)round(((xloc[loc_in_index[i]] - minx) * dxt)) + 2;
        ixloc[num_xloc_indices + i] = \
```



```
      (int)round(((xloc[numRow_loc_in + loc_in_index[i]] - minx) * dxt)) + 2;

        if (i==0)
        {
            imaxx = ixloc[0];
            if (ixloc[num_xloc_indices] > imaxx)
                imaxx = ixloc[num_xloc_indices]; /* ixloc[num_xloc_indices + i], where
i=0 */
        }
        else if ((ixloc[i] > imaxx) || (ixloc[num_xloc_indices + i] > imaxx))
        {
            if (ixloc[num_xloc_indices + i] > ixloc[i])
                imaxx = ixloc[num_xloc_indices + i];
            else
                imaxx = ixloc[i];
        }
    }

    imaxp = (int)ceil((maxp-minx)*dxt + 2);
    #ifdef DEBUG
    if(iflag==PS_FLAG)
    printf("accphi_C_SMP_ACML thread: imaxp = %d \n", imaxp);
    #endif

    if(imaxx > imaxp)
    dx = imaxx + 1;
    else
    dx = imaxp + 1;   /* dimensions of mapx to be dx X dx */
    #ifdef DEBUG
    if(iflag==PS_FLAG)
    printf("accphi_C_SMP_ACML thread: dx = %d\n", dx);
    #endif
    /* ------------------------------------------------------------------------------ */

    /* Allocate space for mapx */
    mapx = malloc(dx*dx*sizeof(int));

    /* Initialise mapx with -1 values to differentiate valid indices (those >=0) ----
       and nonvalid (-1) indices ------------------------------------------------- */
    for (i=0; i<(dx*dx); i++)
    {
        mapx[i] = -1;
    #ifdef TEST_ARR_OUT  // CAUTION: can only fill test_arr_out this way if we run
                         // as single-threaded, otherwise collision will occur
        test_arr_out[i] = -1;
    #endif
    }
    /* ------------------------------------------------------------------------------ */
```

```
   // if(iflag==PS_FLAG)
   //   printf("accphi_C_SMP mapx:\n");
    /* Compute reshaped indices and load mapx with values from ixloc ---------------- */
    for (i=0; i < num_xloc_indices; i++)
    {
        /* "sub2ind" step */
        xind = (ixloc[num_xloc_indices + i] - 1) * dx + ixloc[i] - 1;
        mapx[xind] = i;

        #ifdef TEST_ARR_OUT // CAUTION: can only fill test_arr_out this way if we run
                            // as single-threaded, otherwise collision will occur
        test_arr_out[xind] = i;
        #endif
       // if(iflag==PS_FLAG)
       //   mexPrintf("%d\n",mapx[xind]);
    }
    /* ------------------------------------------------------------------------------ */

    //printf("num_ploc = %d\n", num_ploc);
    /* Compute values and load local copy of ploc, "ploc_", and nploc arrays -------- */
    for (i=0; i < num_ploc; i++)
    {
        ploc_[i] = (ploc_[i] - minx)*dxt + 2;
        nploc[i] = (int)ceil(ploc_[i]);
        ploc_[i] = nploc[i] - ploc_[i];
    } /* for */

    /* ploc_ is not needed from this point onward, so we reuse its memory space */
    xploc = &ploc_[0]; /* xploc points at beginning of the 1st column of ploc_ */
    yploc = &ploc_[numRow_ploc]; /* yploc points at beginning of 2nd column of ploc_*/
    /* ------------------------------------------------------------------------------ */

    /* Compute phi contributions ---------------------------------------------------- */
    for (i=0; i < numRow_ploc; i++)
    {
        /* these two lines are equivalent to xploc=1-ploc(:,1) and yploc=1-ploc(:,2)
         by use of the pointers into ploc_ above */
        xploc[i] = 1 - xploc[i];
        yploc[i] = 1 - yploc[i];

        for (ix=-1; ix<1; ix++)
        {
            xploc[i] = 1 - xploc[i];

            for (iy=-1; iy<1; iy++)
            {
                yploc[i] = 1 - yploc[i];

                if ((xploc[i] > eps) && (yploc[i] > eps))
```



```
                {
                    xx = nploc[i] + ix;
                    xy = nploc[numRow_ploc + i] + iy;

                    /* "sub2ind" step */
                    ind2 = (xy-1) * dx + xx - 1;

                    ind2 = mapx[ind2];

              if(ind2 >= dx*dx)
                {
                  printf("accphi_C_SMP_ACML:ind2 out of bounds! ind2 = %d, i = %d\n", ind2
, i);
                  exit(0);
                }

                    /* 0 is a valid index in C, so instead we treat all instances
                       of -1 (initialised above) as invalid */
                    if (ind2 == -1)
                    {
                        num_unident_pts++;
                        temp_weight = xploc[ind2] * yploc[ind2];
                        if (temp_weight > max_weight)
                            max_weight = temp_weight;
                    }
                    else
                    { /* register phi contribution */
                        if (iflag != DM_FLAG) /* all cases except DMs */
                        {
                            phi[i] = phi[i] + xploc[i] * yploc[i] *
x[loc_in_index[ind2]];
                        }
                        else /* DM case; x input array isn't indexed with loc_in_index
*/
                            phi[i] = phi[i] + xploc[i] * yploc[i] * x[ind2];
                    }
                } /* if */
            } /* for */
        } /* for */
    } /* for */
    /* ------------------------------------------------------------------------------ */

    /* Report unidentified points, if any ------------------------------------------- */
#ifdef TEXT_OUT
       if (num_unident_pts > 0)
        printf("accphi_C_SMP: %d unidentified points in accphi with max weight %f\n", \
           num_unident_pts, max_weight);
#endif
    /* ------------------------------------------------------------------------------ */
```

```
    /* Add result from this thread into global phi_out ------------------------------ */
    pthread_mutex_lock(&mutex_phi);
    for (i=0; i < numRow_ploc; i++)
    {
        phi_out[i] = phi_out[i] + phi[i];
    }
    pthread_mutex_unlock(&mutex_phi);
    /* ---------------------------------------------------------------------------- */

    /* Free all dynamically allocated memory ---------------------------------------- */
    free(nploc);
    free(ploc_);
    free(ixloc);
    free(phi);
    free(mapx);
    /* ---------------------------------------------------------------------------- */

    /* Spawn the next thread (if layers remain to be processed) */
    start_next_thread();

    /* Post the semaphore to indicate this thread has finished and can be joined */
    sem_post(&semA);

    return;

} /* calcphi */

/* ---------------------------------------------------------------------------- */
/* ------------------------------   THR_SIZE_COMP   ------------------------------ */
/* ---------------------------------------------------------------------------- */
/* Compare problem set size between threads for sorting before spawning.  We spawn
   threads in descending order of working set size, to generally minimise overall
   processing time */

int thr_size_comp(void const *p, void const *q)
{

    if (((thread_bundle_t *)p)->thread_parms.num_loc_in_index < \
    ((thread_bundle_t *)q)->thread_parms.num_loc_in_index)
        return -1;
    else if (((thread_bundle_t *)p)->thread_parms.num_loc_in_index > \
         ((thread_bundle_t *)q)->thread_parms.num_loc_in_index)
        return 1;
    else
        return 0;

} /* thr_size_comp */
```



```
/* ---------------------------------------------------------------------------- */
/* ------------------------------   MAIN   ------------------------------------- */
/* ---------------------------------------------------------------------------- */

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
/* define the positions of the parameters in the function call to accphi_C_SMP */
#define LOC_OUT_POS 0
#define PUP_PARMS_POS 1
#define THETA_POS 2
#define H_POS 3
#define LOC_IN_POS 4
#define DXT_POS 5
#define PHI_IN_POS 6
#define IFLAG_POS 7
#define NUM_CONCURRENT_THREADS_POS 8
/* how many input parameters we have (those above) */
#define NUM_INPUT_PARMS 9

    /* Declare pointers to data of mxArray input parameters for readability */
    double *loc_out, *xscl = NULL, *theta, *h, *loc_in, *dxt;
#ifdef PUPPARMS
    double **pupmag, **pupmis;
#endif
    int iflag; /* flag to indicate whether we're processing surfaces, DMs, or PSs */

    double *phi_out; /* ptr to MEX-allocated array for Matlab return parameter for phi
*/

    /* Variables to hold numbers of elements for arrays */
    int num_layers, numCol_loc_in, num_loc_in, numRow_loc_in, num_loc_out,
numRow_loc_out;

    int **index; /* array of pointers to mxMalloc'd arrays of ints, 1 for each layer */
    int *layer_k; /* array of count values of valid # of indices per layer */
    int *temp_index; /* temporary array of valid indices for a given layer */

    int column_ind; /* index, top to bottom, within a given column */
    int i, j, k; /* loop counters */
    int err;

    #ifdef TEST_ARR_OUT
    double *test_arr_out;
    int test_arr_size;
    #endif

    if(nrhs != NUM_INPUT_PARMS)
    {
      mexPrintf("accphi_C_SMP: Incorrect number of input parameters.\n");
```

```
  return;
}

iflag = (int) mxGetScalar(prhs[IFLAG_POS]);

/* Get maximum number of threads allowed to run concurrently */
num_concurrent_threads = (int) mxGetScalar(prhs[NUM_CONCURRENT_THREADS_POS]);

/* we can tell how many PSs/DMs/surfaces to process by the number of  elements --
   in the h (height) input array, which determines how many threads to spawn ---- */

num_layers = mxGetNumberOfElements(prhs[H_POS]);
num_layers_remaining = num_layers; /* initialise num_layers_remaining */

theta = mxGetPr(prhs[THETA_POS]);
h = mxGetPr(prhs[H_POS]);
loc_in = mxGetPr(prhs[LOC_IN_POS]);

if((theta == NULL) || (h == NULL) || (loc_in == NULL))
    {
    mexPrintf("accphi_C_SMP: Incorrect format of input parameters.\n");
    return;
    }
else if(mxGetNumberOfElements(prhs[THETA_POS]) != 2)
    {
    mexPrintf("accphi_C_SMP: Incorrect format of input parameters.\n");
    mexPrintf("Wrong dimensions for theta (needs to be 1x2 vector).\n");
    return;
    }

num_loc_in = mxGetNumberOfElements(prhs[LOC_IN_POS]);
numRow_loc_in = mxGetM(prhs[LOC_IN_POS]);

/* iflag == 3 (for surfaces) means that dxt is calculated in accphi_C_SMP ------- */
/* rather than received as an input parameter ---------------------------------- */
if (iflag != SURF_FLAG)
{
  dxt = mxGetPr(prhs[DXT_POS]);
  if(dxt == NULL)
  {
    mexPrintf("accphi_C_SMP: Incorrect number or format of input parameters.\n");
    return;
  }
  if(dxt == 0)
  {
    mexPrintf("accphi_C_SMP: dxt is zero!.\n");
    return;
  }
```



```
    }
    /* ---------------------------------------------------------------------------- */

//#ifdef    DEBUG /* send text to Matlab console for debugging purposes */
//    mexPrintf("accphi_C_SMP: Number of layers = %d\n", num_layers);
//#endif

    if(mxIsCell(prhs[LOC_OUT_POS])) /* loc_out is packaged together with xscl
                                       parameter vector in a cell array */
      {
        loc_out = mxGetPr(mxGetCell(prhs[LOC_OUT_POS], 0));
        num_loc_out = mxGetNumberOfElements(mxGetCell(prhs[LOC_OUT_POS], 0));
        numRow_loc_out = mxGetM(mxGetCell(prhs[LOC_OUT_POS], 0)); /* numRow_loc_out is
also                                                                  the number of phi
elements to                                                           return to Matlab */

        xscl = mxGetPr(mxGetCell(prhs[LOC_OUT_POS], 1));
        if(xscl == NULL)
        {
        mexPrintf("accphi_C_SMP: Incorrect format of input parameters.\n");
        mexPrintf("smp_xscl parameter missing from {loc_out, smp_xscl}.\n");
        return;
        }
#ifdef PUPPARMS
       if((iflag == DM_FLAG) && (mxIsCell(prhs[PUP_PARMS_POS]))) /* check for correct
cell array                                                            input parameters for DM
case */
        {
         pupmag = mxMalloc(num_layers * sizeof(double *));
         pupmis = mxMalloc(num_layers * sizeof(double *));
         for(i=0;i<num_layers;i++)                 /* each pupmag is 4 elements */
           {                                       /* each pupmis is 2 elements */
              pupmag[i] = mxGetPr(mxGetCell(prhs[PUP_PARMS_POS], 0)) + i * 4;
              pupmis[i] = mxGetPr(mxGetCell(prhs[PUP_PARMS_POS], 1)) + i * 2;
              if((pupmag[i] == NULL) || (pupmis[i] == NULL))
              {
                mexPrintf("accphi_C_SMP: Incorrect format of input parameters.\n");
                mexPrintf("DM {pupmag, pupmis} parameter malformed.\n");
                return;
              }
           }
        }
       else if((iflag == DM_FLAG) && !(mxIsCell(prhs[PUP_PARMS_POS])))
        {
        mexPrintf("accphi_C_SMP: Incorrect format of input parameters.\n");
        mexPrintf("DM {pupmag, pupmis} parameter malformed.\n");
```

```
        return;
        }
#endif
      }
       else /* {loc_out, smp_xscl} not passed in correctly */
      {
         mexPrintf("accphi_C_SMP: Incorrect format of input parameters.\n");
         mexPrintf("{loc_out, smp_xscl} parameter malformed.\n");
         return;
      }

    /* Only need to set num_concurrent_threads to the number of layers provided ----- */
    if (num_concurrent_threads > num_layers)
        num_concurrent_threads = num_layers;
    /* ------------------------------------------------------------------------------ */

    /* Initialise semaphore for controlling # of currently running threads ---------- */
    if (sem_init(&semA, 0, 0) != 0)
    {
        mexPrintf("accphi_C_SMP: ERROR: can't initialise semaphore.\n");
        return;
    }
    /* ------------------------------------------------------------------------------ */

    thread_bundle = (thread_bundle_t *)mxMalloc(num_layers * sizeof(thread_bundle_t));
    index = mxMalloc(num_layers * sizeof(int *));
    /* count of valid loc_in indices for a given layer */
    layer_k = mxMalloc(num_layers * sizeof(int));
    /* allocate temporary index array */
    temp_index = mxMalloc(numRow_loc_in * sizeof(int));

    /* C MEX dynamic array allocation (to facilitate passing phi back to Matlab) */
    plhs[0] = mxCreateDoubleMatrix(numRow_loc_out, 1, mxREAL); /* mxREAL is the double
                                  data type for Matlab */
    /* accumulated phi output directed to first LHS Matlab arg */
    phi_out = mxGetPr(plhs[0]);

#ifdef TEST_ARR_OUT
test_arr_size = 65536;
plhs[1] = mxCreateDoubleMatrix(test_arr_size, 1, mxREAL);
test_arr_out = mxGetPr(plhs[1]);
#endif

    /* ########### SORT & ASSEMBLE LAYER-SPECIFIC DATA FOR PASSING TO THREADS ####### */
    for (i=0; i < num_layers; i++)
    {
        layer_k[i] = 0;
        column_ind = 0;
```



```
        /* Scan down 3rd column of loc_in to find values equal to h of this layer; -- */
        /* record indices of these elements ---------------------------------------- */
        for (j=(numRow_loc_in*2); j < (numRow_loc_in*3); j++)
        {
            if (loc_in[j] == h[i])
            {/* index of corresponding value in 1st column of loc_in */
                temp_index[layer_k[i]] = column_ind;
                layer_k[i]++;
            }
            column_ind++; /* increment index within the 3rd column */
        } /* for */
        /* ------------------------------------------------------------------------- */

    /* allocate index array for this layer */
        index[i] = mxMalloc(layer_k[i] * sizeof(int));

    /* fill index array for this layer with valid indices ---------------------- */
        for (k=0; k < layer_k[i]; k++)
        {
            index[i][k] = temp_index[k];
        }
        /* ------------------------------------------------------------------------- */

        /* Package parameters for passing to threads ------------------------------- */
        thread_bundle[i].thread_parms.loc_out = loc_out;
        thread_bundle[i].thread_parms.xscl = xscl[i];
        thread_bundle[i].thread_parms.num_loc_out = num_loc_out;
        thread_bundle[i].thread_parms.numRow_loc_out = numRow_loc_out;
        thread_bundle[i].thread_parms.theta = theta;
        thread_bundle[i].thread_parms.loc_in = loc_in;
        thread_bundle[i].thread_parms.h = h[i];
        thread_bundle[i].thread_parms.numRow_loc_in = numRow_loc_in;
        thread_bundle[i].thread_parms.loc_in_index = &index[i][0];
        thread_bundle[i].thread_parms.num_loc_in_index = layer_k[i];
        thread_bundle[i].thread_parms.phi_out = phi_out;
        thread_bundle[i].thread_parms.iflag = iflag;
        #ifdef TEST_ARR_OUT
        thread_bundle[i].thread_parms.test_arr_out = test_arr_out;
        #endif

/*  if (xscl == NULL)
      {
        thread_bundle[i].thread_parms.xscl = 1;
        mexPrintf("Forced NULL xscl to 1 for thread (genwfsopd.m call to accphi_C_SMP).
\n");
      }
    else
      {
        thread_bundle[i].thread_parms.xscl = xscl[i];
```

```
      }
 */
    if (iflag == DM_FLAG) /* for the case of DM */
      {
#ifdef PUPPARMS
        thread_bundle[i].thread_parms.pupmag = pupmag[i];
        thread_bundle[i].thread_parms.pupmis = pupmis[i];
#endif
        thread_bundle[i].thread_parms.phi_in = mxGetPr(mxGetCell(prhs[PHI_IN_POS], i));
      }
    else /* otherwise */
        thread_bundle[i].thread_parms.phi_in = mxGetPr(prhs[PHI_IN_POS]);
    if (iflag == SURF_FLAG) /* for the case of a surface */
        thread_bundle[i].thread_parms.dxt = loc_in[index[i][1]] - loc_in[index[i][0]];
    else /* otherwise */
        thread_bundle[i].thread_parms.dxt = dxt[i];
    } /* for */
        /* ------------------------------------------------------------------------ */
    /* ######################################################################## */


    /* Sort threads in increasing order of working set size */
    qsort(&thread_bundle[0], num_layers, sizeof(thread_bundle_t), thr_size_comp);

    /* Call start_next_thread to spawn first num_concurrent_threads threads --------- */
    for (i=0; i < num_concurrent_threads; i++)
    {
        start_next_thread();
    }
    /* ------------------------------------------------------------------------ */

    /* Join on returning threads ---------------------------------------------- */
    for (i = num_layers - 1; i >= 0; i--)
    {
        if (TEMP_FAILURE_RETRY(sem_wait(&semA)) != 0)
        {
            mexPrintf("accphi_C_SMP: ERROR: sem_wait failed.\n");
            return;
        }

        if (pthread_join(thread_bundle[i].thread_id,NULL) != 0)
        {
            mexPrintf("accphi_C_SMP: ERROR: pthread_join failed.\n");
            return;
        }
    }

//#ifdef TEXT_OUT
    mexPrintf("accphi_C_SMP: All threads terminated.\n");
```



```
//#endif
    /* ------------------------------------------------------------------------ */

    /* Free all dynamically allocated memory and semaphores ------------------------ */
    mxFree(temp_index);
    mxFree(thread_bundle);
    mxFree(index);
    mxFree(layer_k);

    sem_destroy(&semA);
    /* ------------------------------------------------------------------------ */
    mexPrintf("accphi_C_SMP: memory freed.  Done\n");
} /* accphi_C_SMP.c */
```

Appendix D:
VOLT Real-Time Pipeline Source Code Listing

```
/*****************************************************************************
 Adaptive Optics Realtime Pipeline
 Developed by:
   Owen Mead-Robins - Matlab and Pipeline build and integration
   and
   Aaron Hilton     - Design and Framegrabber/Mirror IO drivers

   Built in the Summer of 2005

   Thanks to:
   Rodolphe Conan - Invaluable guidance and technological support.
   Peter Hampton  - Super cool controller designs.
   Brian Wallace  - The guy with all the facts, and optics wiz.
   Colin Bradley  - One of the best project directors around!

 Modified for use in Victoria Open Loop Testbed (VOLT) by Mike Fischer, 2007-8

 Heartbeat:

   The adaptive optics pipeline should operate as fast as possible while
   keeping a regular rhythm for scientific analysis.  This pipeline uses
   the concept of a Heartbeat interval, which keeps the mirror control
   operations occurring at precisely the same latency on each iteration.

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

// Standard Includes
#include <stdio.h>
#include <iostream>
#include <iomanip>
#include <cstdio>

// Matlab specific libraries
#include "matlab.h"
#include "rtIntegrator.h"
#include "shackProcessing.h"
#include "loadAOVar.h"
#include "saveAOVar.h"
#include "meshgrid.h"
#include "squeeze.h"
#include "flipud.h"
#include "repmat.h"

// Normally
#define WFS
#define CENTROIDING
#define MIRRORS
//#define TIP_TILT
#define PROCESS_STATS
```



```
// To enforce a regular loop time:
//#define HEARTBEAT

// To get a full ROI test frame back in aoVariables:
//#define G_FRAME
// To run shape test for DM ringing:
//#define DM_SHAPE_TEST
#ifdef DM_SHAPE_TEST
#define SHAPE_CHANGE_INTERVAL 200
#include "dm_shape_gen.h"
#else

// Uncomment the following line to include the MATLAB real-time
// integrator in the loop (you need libintegrator.so for both
// open- or closed-loop mode; these modes are selected inside rtIntegrator.m)
#define MLFINTEGRATOR
// Also have to define OPEN_LOOP for open-loop control. If commented out, the
// system will run in closed-loop (provided you made the appropriate changes
// inside rtIntegrator.m)
#define OPEN_LOOP
#endif

// C++ Class object includes -- order of inclusion matters!
#include "MirrorDriver.h"
#include "WFS_1M150_EDT.h"
#include "processStats.h"
#include "ErrorReporter.h"

#include <curses.h>

// Allows for grabbing of cpu ticks from processor for pipeline timing
#define rdtsc(t) asm __volatile__ ("rdtsc" : "=A" (t))

// Experiment length -- normally set as a command line option
long long NUM_CPU_TIMEDIFFS = 100; // default

double exp_time = 10, frame_time = 15;
bool ramping_gain = false, manual_gain_set = false;
double temp_gain[2] = {0.1, 0.1}, *gain_ramp;
unsigned int numiter_gain_ramp = 1; // default

// Include the centroid processing function.
void centroidCapture( unsigned char *buffer, double *centroids,
                      double *pixelBiasMap, double *pixelGainMap,
                      double *refMeasurements, double *threshold, int loop_iter);

// Shack-Hartmann WFS global variables
#ifdef WFS
    #ifdef G_FRAME
```

```
mxArray *gFrame = NULL;
    #endif
mxArray *nLenslet = NULL;
mxArray *nValidLenslet = NULL;
mxArray *validLenslet = NULL;
mxArray *pixelBottomLeft = NULL;
mxArray *pixelBiasMap = NULL;
mxArray *pixelGainMap = NULL;
mxArray *threshold = NULL;
mxArray *refMeasurements = NULL;
mxArray *nPx = NULL;
mxArray *exp_time_out = NULL;
#endif

// Performance statistics global variables
mxArray *tics_per_cycle = NULL;
mxArray *min_tics = NULL;
mxArray *max_tics = NULL;
mxArray *mean_tics = NULL;
mxArray *rms_tics = NULL;

#ifdef MIRRORS
// Integrator global variables
mxArray *dmValidActuator = NULL;
mxArray *reconstructor = NULL;
mxArray *dmCoefficients = NULL;
mxArray *residualDMCoefficients = NULL;
mxArray *gain = NULL;
mxArray *gain_out = NULL;
mxArray *numiter_gain_ramp_out = NULL;

// Mirror coefficients
//mxArray *residualTTCoefficients = NULL;
//mxArray *ttCoef = NULL;
//mxArray *ttCoef0 = NULL;
mxArray *DMCoef = NULL;
mxArray *DMCoef0 = NULL;
mxArray *flat = NULL;
mxArray *flat_digit = NULL;

    #ifdef DM_SHAPE_TEST
mxArray *dm_shape_slopes = NULL;
mxArray *kdm_shape_iteration = NULL;
mxArray *kscale_index = NULL;
int NUM_SCALE_FACTORS = 4;
    #endif

// Declare an instance of MirrorDriver for the DM
MirrorDriver *gMirrorDriver = 0;
```



```
#endif

#ifdef WFS
// Declare an instance of WFS_1M150 for the WFS
WFS_1M150 *gCamera = 0;
mxArray *centroids = NULL;
#endif

// Pipeline loop global MATLAB variables
mxArray *kIterations = NULL;
mxArray *outputs = NULL;

// Declare an instance of the ErrorReporter
ErrorReporter *gErrorReporter = 0;

// Utility routine for DM analog volt to digit conversion
void convertToDigit(double *inVolt, double *outDigit, int N,
            double minDigit, double maxDigit,
              double commandMin, double commandMax) {
  for (int k=0; k<N; k++)
  {
    outDigit[k] = (maxDigit-minDigit)*(-inVolt[k]-commandMin)/(commandMax-commandMin) +
minDigit;
  }
}

void onExit(void);

// Pipeline variables initialisation
void initVariables() {
    kIterations = mxCreateDoubleMatrix(1,1,mxREAL);
    double data[1] = {1};
    memcpy(mxGetPr(kIterations), data, sizeof(double));

// Read in the variables from the Matlab variable file aoVariables.mat
#ifdef WFS
    #ifdef G_FRAME
    //gFrame= mlfLoadAOVar(mxCreateString("gFrame" ));
    #endif
    nLenslet      = mlfLoadAOVar(mxCreateString("nLenslet" ));
    nValidLenslet      = mlfLoadAOVar(mxCreateString("nValidLenslet" ));
    validLenslet      = mlfLoadAOVar(mxCreateString("validLenslet" ));
    threshold      = mlfLoadAOVar(mxCreateString("threshold" ));
    pixelBottomLeft      = mlfLoadAOVar(mxCreateString("pixelBottomLeft" ));
    pixelBiasMap      = mlfLoadAOVar(mxCreateString("pixelBiasMap" ));
    pixelGainMap      = mlfLoadAOVar(mxCreateString("pixelGainMap" ));
    refMeasurements= mlfLoadAOVar(mxCreateString("refMeasurements" ));
    nPx      = mlfLoadAOVar(mxCreateString("nPx" ));
```

```
    #ifdef MIRRORS
    dmValidActuator      = mlfLoadAOVar(mxCreateString("dmValidActuator" ));
    reconstructor      = mlfLoadAOVar(mxCreateString("reconstructor" ));
    #endif
#endif

#ifdef MIRRORS
    //ttCoef0      = mlfLoadAOVar(mxCreateString("ttCoef0" ));
    DMCoef0      = mlfLoadAOVar(mxCreateString("DMCoef0" ));
    flat      = mlfLoadAOVar(mxCreateString("flat" ));
    #ifdef DM_SHAPE_TEST
    dm_shape_slopes = mlfLoadAOVar(mxCreateString("dm_shape_slopes" ));
    kdm_shape_iteration = mxCreateDoubleMatrix(1,1,mxREAL);
    double data1[1] = {1};
    memcpy(mxGetPr(kdm_shape_iteration), data1, sizeof(double));
    kscale_index = mxCreateDoubleMatrix(1,1,mxREAL);
    double data2[1] = {0};
    memcpy(mxGetPr(kscale_index), data2, sizeof(double));
    #endif
#endif

    double *curMxArray;

#ifdef WFS
    #ifdef G_FRAME
    gFrame = mxCreateDoubleMatrix( CAMERA_1M150_RESX, CAMERA_1M150_RESY, mxREAL);
    #endif
    nPx = mclInitializeDoubleVector(mxGetM(nPx), mxGetN(nPx), mxGetPr(nPx) );
    refMeasurements = mclInitializeDoubleVector(mxGetM(refMeasurements),
        mxGetN(refMeasurements), mxGetPr(refMeasurements));
    nLenslet = mclInitializeDoubleVector(mxGetM(nLenslet), mxGetN(nLenslet),
    mxGetPr(nLenslet) );
    nValidLenslet = mclInitializeDoubleVector(mxGetM(nValidLenslet),
        mxGetN(nValidLenslet), mxGetPr(nValidLenslet));
    pixelBottomLeft = mclInitializeDoubleVector(mxGetM(pixelBottomLeft),
        mxGetN(pixelBottomLeft), mxGetPr(pixelBottomLeft));
    validLenslet = mclInitializeDoubleVector(mxGetM(validLenslet),
        mxGetN(validLenslet), mxGetPr(validLenslet) );
    pixelBiasMap = mclInitializeDoubleVector(mxGetM(pixelBiasMap),
        mxGetN(pixelBiasMap), mxGetPr(pixelBiasMap) );
    pixelGainMap = mclInitializeDoubleVector(mxGetM(pixelGainMap),
        mxGetN(pixelGainMap), mxGetPr(pixelGainMap) );
    threshold = mclInitializeDoubleVector(mxGetM(threshold),
        mxGetN(threshold), mxGetPr(threshold) );
    centroids = mxCreateDoubleMatrix( NVALIDLENSLET*2, NUM_CPU_TIMEDIFFS, mxREAL);
    centroids = mclInitializeDoubleVector( mxGetM(centroids) , mxGetN(centroids) ,
mxGetPr(centroids) );
#endif
```



```
    curMxArray = mxGetPr(kIterations);
    kIterations = mclInitializeDoubleVector(mxGetM(kIterations), mxGetN(kIterations),
                            curMxArray);

#ifdef MIRRORS
        curMxArray = mxGetPr(dmValidActuator);
    dmValidActuator = mclInitializeDoubleVector(mxGetM(dmValidActuator),
mxGetN(dmValidActuator), curMxArray);

        curMxArray = mxGetPr(reconstructor);
    reconstructor = mclInitializeDoubleVector(mxGetM(reconstructor),
mxGetN(reconstructor), curMxArray);

    residualDMCoefficients      = mxCreateDoubleMatrix( mxGetM(reconstructor),
NUM_CPU_TIMEDIFFS, mxREAL);
        curMxArray = mxGetPr(residualDMCoefficients);
    residualDMCoefficients = mclInitializeDoubleVector( mxGetM(residualDMCoefficients),
mxGetN(residualDMCoefficients), curMxArray);

    dmCoefficients = mxCreateDoubleMatrix( mxGetM(reconstructor), 1, mxREAL );
    dmCoefficients = mclInitializeDoubleVector(
              mxGetM(dmCoefficients), mxGetN(dmCoefficients), mxGetPr(dmCoefficients) );

    //TTCoefficients = mxCreateDoubleMatrix( 2, 1, mxREAL );
    //TTCoefficients = mclInitializeDoubleVector(
    //            mxGetM(TTCoefficients), mxGetN(TTCoefficients), mxGetPr(TTCoefficients
) );

    DMCoefficients = mxCreateDoubleMatrix( NDMCOMMANDS , 1, mxREAL );
    DMCoefficients = mclInitializeDoubleVector(
              mxGetM(DMCoefficients), mxGetN(DMCoefficients), mxGetPr(DMCoefficients) );

    //ttCoef0 = mclInitializeDoubleVector( mxGetM(ttCoef0), mxGetN(ttCoef0), mxGetPr(ttC
oef0) );
    //ttCoef  = mxCreateDoubleMatrix( mxGetM(ttCoef0), mxGetN(ttCoef0), mxREAL);
    //ttCoef  = mclInitializeDoubleVector( mxGetM(ttCoef0), mxGetN(ttCoef0) , mxGetPr(tt
Coef) );

        curMxArray = mxGetPr(DMCoef0);
    DMCoef0 = mclInitializeDoubleVector(mxGetM(DMCoef0), mxGetN(DMCoef0),curMxArray );
    DMCoef  = mxCreateDoubleMatrix( mxGetM(DMCoef0), mxGetN(DMCoef0), mxREAL);
    DMCoef  = mclInitializeDoubleVector( mxGetM(DMCoef0), mxGetN(DMCoef0) ,
mxGetPr(DMCoef) );

    curMxArray = mxGetPr(flat);
    flat = mclInitializeDoubleVector(mxGetM(flat), mxGetN(flat),curMxArray );

    flat_digit = mxCreateDoubleMatrix( mxGetM(flat), mxGetN(flat), mxREAL);
    flat_digit = mclInitializeDoubleVector(mxGetM(flat), mxGetN(flat), mxGetPr(flat_digi
```

```
t) );

    #ifdef DM_SHAPE_TEST
    curMxArray = mxGetPr(kdm_shape_iteration);
     kdm_shape_iteration = mclInitializeDoubleVector(mxGetM(kdm_shape_iteration),
mxGetN(kdm_shape_iteration), curMxArray);

    curMxArray = mxGetPr(kscale_index);
     kscale_index = mclInitializeDoubleVector(mxGetM(kscale_index), mxGetN(kscale_index)
, curMxArray);

    curMxArray = mxGetPr(dm_shape_slopes);
    dm_shape_slopes = mclInitializeDoubleVector(mxGetM(dm_shape_slopes),
mxGetN(dm_shape_slopes),curMxArray );
    #endif
#endif
}

// Terminate MCR and library function and free resources.
void onExit(void)
{

#ifdef WFS
  if( gCamera )
{
    delete gCamera;
}
#endif

#ifdef MIRRORS
  if( gMirrorDriver )
    {
      delete gMirrorDriver;
    }
#endif

  if( gErrorReporter )
    {
      delete gErrorReporter;
    }

  TerminateModule_meshgrid();
  TerminateModule_squeeze();
  TerminateModule_flipud();
  TerminateModule_repmat();
  TerminateModule_shackProcessing();
#ifdef MLFINTEGRATOR
  TerminateModule_rtIntegrator();
#endif
```



```
  TerminateModule_saveAOVar();
  TerminateModule_loadAOVar();
#ifdef DM_SHAPE_TEST
  TerminateModule_dm_shape_gen();
#endif
}

// Main
int main(int argc, char* argv[])
{
    char dummy1;
    int camera_init = 1;

    // Pointer to WFS camera frame buffer
    unsigned char *camera_frame;
    unsigned int *frame_time_ptr;
    double *centroid_ptr;

    // Performance counters.
    long long cpuHeartBeatInterval = 1670000; // default 1 ms (for mach274 computer)
    bool heartbeat_wait = false;
    long long cpuBeforeLoopStartTime;
    long long cpuStartTime;   // Beginning of loop iteration.
    long long cpuEndTime;     // Start + heartbeat interval.
    long long cpuCurrentTime; // Tmp, current time.
    long long cpuTimeDiff;    // Tmp, time difference calculations.

    if( argc < 3 )
    {
        printf( "usage: integratorLoop\n");
        printf( "  -i <iterations>      set number of AO loop iterations (default=100).
\n" );
        printf( "  -e <exposure time>       set camera exposure time (default=10ms).\n"
);
        printf( "  -g <initial gain  final gain>        set gain for integrator loop if
running closed-loop. (default=0.1).\n * final gain is only used in conjunction with -r
option, thus is not set if using a constant gain.\n");
        printf( "  -r <num ramp iterations>     set gain to ramp up from [initial gain]
to [final gain] over num ramp iterations (default=20).\n" );
        printf( "  -w <0/1>     set value to 0 to skip WFS camera exposure time & ROI
initialisation phase.  \n" );
        exit(1);
    }

    printf( "starting pipeline...\n" );

    // Initialise the matlab modules (required modules made by matlab compiler)
    InitializeModule_loadAOVar();
    InitializeModule_saveAOVar();
```

```
#ifdef MLFINTEGRATOR
    InitializeModule_rtIntegrator();
#endif
    InitializeModule_shackProcessing();
#ifdef DM_SHAPE_TEST
    InitializeModule_dm_shape_gen();
#endif
    // Initialise non-core matlab functions
    InitializeModule_meshgrid();
    InitializeModule_squeeze();
    InitializeModule_flipud();
    InitializeModule_repmat();

    // Make sure we clean up properly
    atexit(onExit);

    --argc;
    ++argv;
    while (argc && ((argv[0][0] == '-') || (argv[0][0] == '/')))
    {
        switch (argv[0][1])
        {
            case 'g':       /* gain */
                ++argv;
                --argc;
                manual_gain_set = true;
                temp_gain[0] = atof(argv[0]);
                if(argc > 0)
                    ++argv;
                if(strtof(argv[0],NULL))
                {
                    temp_gain[1] = atof(argv[0]);
                    ramping_gain = true;
                }
                else
                {
                    --argv;
                    ramping_gain = false;
                }
                if(argc > 7)
                    ++argv;
                break;

            case 'r':   /* # of iterations to ramp gain up over */
                ++argv;
                --argc;
                numiter_gain_ramp = atoi(argv[0]);
                if(argc > 7)
                    ++argv;
```



```
                break;

            case 'e':       /* exposure time */
                ++argv;
                --argc;
                exp_time = atof(argv[0]);
                if(argc > 7)
                    ++argv;
                break;

            case 't':       /* loop time */
                ++argv;
                --argc;
                if(atoll(argv[0]) == 0)
                {
                    heartbeat_wait = false;
                    printf("loop time = as fast as possible\n");
                }
                else
                {
                    heartbeat_wait = true;
                    cpuHeartBeatInterval = atoll(argv[0]);
                    printf("loop time = %lld\n",cpuHeartBeatInterval);
                }
                if(argc > 7)
                    ++argv;
                break;

            case 'i':       /* # loop iterations */
                ++argv;
                --argc;
                NUM_CPU_TIMEDIFFS = atoll(argv[0]);
                printf("loop iterations  = %lld\n",NUM_CPU_TIMEDIFFS);
                if(argc > 7)
                    ++argv;
                break;

            case 'w':       /* WFS init skip */
                ++argv;
                --argc;
                camera_init = atoi(argv[0]);
                if(camera_init == 0)
                    printf("WFS exposure time & ROI initialisation skipped.\n");
                if(argc > 7)
                    ++argv;
                break;
        }
    }
```

```
    printf("exposure time = %f\n",exp_time);

    // Perform all the messy loading of Matlab information
    initVariables();
    double *loopIteration = mxGetPr(kIterations);

    gain_out = mclInitializeDoubleVector(1, 2, temp_gain);
    outputs =  mlfSaveAOVar(gain_out, mxCreateString("gain" ));
    // Initialise gain to the first value of temp_gain.  It might be overwritten in the
loop if we are ramping
    gain = mclInitializeDoubleVector(1, 1, &temp_gain[0]);

    if(manual_gain_set == true)
    {
        if(ramping_gain == true)
        {
            printf("integrator gain will ramp from %f to %f over %d
iterations\n",temp_gain[0],temp_gain[1],numiter_gain_ramp);
            gain_ramp = new double[numiter_gain_ramp];
            float step_size = (temp_gain[1] - temp_gain[0]) / (numiter_gain_ramp - 1);
            for(int i = 0; i<numiter_gain_ramp; i++)
            {
                gain_ramp[i] = temp_gain[0] + i*step_size;
            }
        }
        else
        {
            printf("integrator gain = %f\n",temp_gain[0]);
        }
    }
    else
    {
        temp_gain[0] = 1;
        temp_gain[1] = 0;
        printf("integrator gain = %f\n",temp_gain[0]);
    }

#ifdef G_FRAME
printf( "WARNING: G_FRAME set.  Returning a full WFS frame to aoVariables on last
iteration (RUNS SLOW).\n" );
#endif

#ifdef WFS
    gCamera = new WFS_1M150();

    if (!camera_init)
        gCamera->init_flag = false;

    if( gCamera->init(exp_time,frame_time) != true )
```



```
    {
        printf( "ERROR: Camera WFS did not initialize.\n" );
        exit(1);
    }

#endif

#ifdef MIRRORS
    gMirrorDriver = new MirrorDriver();
    if( gMirrorDriver == 0 ) {
        printf( "Failed to allocate Mirror Driver.\n" );
        exit(1);
    }
    #ifdef DM_SHAPE_TEST
    double *dm_shape_iteration = mxGetPr(kdm_shape_iteration);
    double *scale_index = mxGetPr(kscale_index);
    #endif
#endif

    gErrorReporter = new ErrorReporter();
    if( gErrorReporter == 0 ) {
        printf("Failed to allocate Error Reporter.\n" );
        exit(1);
    }

    try
    {

#ifdef MIRRORS
    gMirrorDriver->init();
    gMirrorDriver->setDMPower(true);

    #ifdef MIRROR_DEBUG
    for(int i = 0; i< 64; i++)
    printf("flat[%d]=%f\n",i,mxGetPr(flat)[i]);
    #endif

    #ifdef TIP_TILT
      convertToDigit( mxGetPr(ttCoef0), mxGetPr(ttCoef),mxGetN(ttCoef0),
              32767,65535,0,10);
    #endif
      convertToDigit( mxGetPr(DMCoef0), mxGetPr(DMCoef),mxGetM(DMCoef0)*mxGetN(DMCoef0),
              6144,10240,-0.5,0.5);

      convertToDigit( mxGetPr(flat), mxGetPr(flat_digit),mxGetM(flat)*mxGetN(flat),
            6144,10240,-0.5,0.5);
    #ifdef MIRROR_DEBUG
    for(int i = 0; i< 64; i++)
    printf("flat_digit[%d]=%f\n",i,mxGetPr(flat_digit)[i]);
```

```
    #endif

    #ifdef TIP_TILT
      gMirrorDriver->prepareBuffers( mxGetPr(ttCoef), mxGetPr(DMCoef));
    #else
      gMirrorDriver->prepareBuffers( NULL, mxGetPr(flat_digit));
      //gMirrorDriver->prepareBuffers( NULL, mxGetPr(DMCoef));
    #endif

      gMirrorDriver->actuateMirrorsNow();
      printf("===>>DM FLATTENED\n");
//    printf("gErrorReporter...\n");
      gErrorReporter->init();
//    printf("resetStats...\n");
      resetStats();
#endif

      printf("Press a key and Enter to start the loop ...\n");
      scanf("%s",&dummy1);
      printf("GO \n");

      rdtsc(cpuBeforeLoopStartTime);

      // Just a dummy frame to trigger first image capture
      gCamera->getFrameData();

        for( ; *loopIteration <= NUM_CPU_TIMEDIFFS; (*loopIteration)++)
        {
            rdtsc(cpuStartTime);

#ifdef WFS
            gCamera->lock( true );
#endif
/*
            rdtsc( cpuCurrentTime );
            cpuTimeDiff = cpuCurrentTime - cpuStartTime;
            printf("Time diff. [lock]  =%d\n",cpuTimeDiff);
            rdtsc(cpuStartTime);
*/
#ifdef MIRRORS
//          Drive the mirrors to their target positions.
            // MDF - January 21, 2008 -- commented out and moved to end of loop, to issu
e movement command immediately after buffers are ready
            //gMirrorDriver->actuateMirrorsNow();
#endif
/*
            rdtsc( cpuCurrentTime );
            cpuTimeDiff = cpuCurrentTime - cpuStartTime;
            printf("Time diff. [mirrors 1]    = %d\n",cpuTimeDiff);
```



```
            rdtsc(cpuStartTime);
*/
#ifdef WFS
// Grab the most recent camera frame from the ring buffer
            camera_frame = gCamera->getFrameData();
#endif
/*
            rdtsc( cpuCurrentTime );
            cpuTimeDiff = cpuCurrentTime - cpuStartTime;
            printf("Time diff. [getFrameData]    = %d\n",cpuTimeDiff);
            rdtsc(cpuStartTime);
*/
/*          rdtsc( cpuCurrentTime );
            cpuTimeDiff = cpuCurrentTime - cpuBeforeLoopStartTime;
            printf("Absolute time diff. [getFrameData]    = %d\n",cpuTimeDiff);
*/

#ifdef WFS
//frame_time_ptr = gCamera->getLastFrameTime();
//printf("last frame timestamp: %lld sec, %lld
microsec\n",frame_time_ptr[0],frame_time_ptr[1]);
    #ifdef G_FRAME  // gFrame output test mode writes camera frames to aoVariables
            if(*loopIteration >= NUM_CPU_TIMEDIFFS - 1) // only write on last iteration
            {
                double *pgFrame = mxGetPr(gFrame);

                for(int i=0; i<CAMERA_1M150_FRAME_SIZE; i++)
                    pgFrame[i]=(double)camera_frame[i];
            }
    #endif

            centroid_ptr = mxGetPr(centroids);
            centroid_ptr += (int)((*loopIteration)-1)*NVALIDLENSLET*2;

            centroidCapture( camera_frame, centroid_ptr,
                        mxGetPr(pixelBiasMap), mxGetPr(pixelGainMap),
                        mxGetPr(refMeasurements), mxGetPr(threshold),
(int)((*loopIteration)-1));

            gCamera->unlock();
#endif
/*
            rdtsc( cpuCurrentTime );
            cpuTimeDiff = cpuCurrentTime - cpuStartTime;
            printf("Time diff. [centroid]   = %d\n",cpuTimeDiff);
            rdtsc(cpuStartTime);
*/
/*
            rdtsc( cpuCurrentTime );
```

```
            cpuTimeDiff = cpuCurrentTime - cpuBeforeLoopStartTime;
            printf("Absolute time diff. [centroid]    = %d\n",cpuTimeDiff);
*/
#ifdef MLFINTEGRATOR
            if(ramping_gain == true)
            {
                if(*loopIteration <= numiter_gain_ramp)
                {
                mxGetPr(gain)[0] = gain_ramp[(int)*loopIteration-1];
                }
                else
                {
                    ramping_gain == false; // to stop checking after the ramp is done
                }
            }

            // Run the integrator
            mlfRtIntegrator();
#endif
/*
            rdtsc( cpuCurrentTime );
            cpuTimeDiff = cpuCurrentTime - cpuStartTime;
            printf("Time diff. [mlfRtIntegrator]   = %d\n",cpuTimeDiff);
            rdtsc(cpuStartTime);
*/
/*
            rdtsc( cpuCurrentTime );
            cpuTimeDiff = cpuCurrentTime - cpuBeforeLoopStartTime;
            printf("Absolute time diff. [mlfRtIntegrator]    = %d\n",cpuTimeDiff);
*/
//printf("loopIteration %d\n",(int)*loopIteration);
#ifdef MIRRORS
    #ifdef DM_SHAPE_TEST
            if(!((int)*loopIteration % SHAPE_CHANGE_INTERVAL))
            {
                (*scale_index)++;
//printf("DM_SHAPE_TEST scale index: %f\n",*scale_index);
                if((int)(*scale_index) > NUM_SCALE_FACTORS)
                {
                    (*dm_shape_iteration)++;
                    *scale_index = 1;
                }
                printf("Sending shape %d, scale index %d at loopIteration %d...\n",
(int)*dm_shape_iteration,(int)*scale_index, (int)*loopIteration);

                //double *dm_shape_slope_ptr = mxGetPr(dm_shape_slopes);

            mlfDm_shape_gen();
```



```
            //  convertToDigit( shape_matrix_ptr,
mxGetPr(DMCoef),mxGetM(DMCoef0)*mxGetN(DMCoef0),
            //          6144,10240,-0.5,0.5);

                gMirrorDriver->prepareBuffers( mxGetPr(ttCoef), mxGetPr(DMCoef));
            }

    #else
            // Prepare mirror control DMA Buffers.
            gMirrorDriver->prepareBuffers( mxGetPr(ttCoef), mxGetPr(DMCoef));
    #endif
            // Use the following if you just want a flat on the DM at every iteration
(for testing)
            //  gMirrorDriver->prepareBuffers( mxGetPr(ttCoef), mxGetPr(flat_digit));

            // MDF - Jan 21, 2008 -- moved mirror movement command to here instead of at
start of loop
            gMirrorDriver->actuateMirrorsNow();
#endif
/*
            rdtsc( cpuCurrentTime );
            cpuTimeDiff = cpuCurrentTime - cpuBeforeLoopStartTime;
            printf("Absolute time diff. [mirror command dispatch]    =
%d\n",cpuTimeDiff);
*/
            // Get current time for loop timing statistics
            rdtsc( cpuCurrentTime );
            cpuTimeDiff = cpuCurrentTime - cpuStartTime;

/*          printf("Time diff. [mirror command dispatch] = %d\n",cpuTimeDiff);
            //rdtsc(cpuStartTime);
*/

            // Ensure we have not exceeded the heartbeat
            //  if( cpuTimeDiff > cpuHeartBeatInterval ) {
            //      gErrorReporter->storeHeartBeatError(*loopIteration, cpuTimeDiff);
            //  }
#ifdef PROCESS_STATS
            if((*loopIteration > 1) && (*loopIteration < NUM_CPU_TIMEDIFFS))
                processStats(cpuTimeDiff);
#endif

            if(heartbeat_wait)
            {
                // Pooling wait on HeartBeatInterval
                do
                {
                    rdtsc( cpuCurrentTime );
                    cpuTimeDiff = cpuCurrentTime - cpuStartTime;
```

```
                } while( cpuTimeDiff < cpuHeartBeatInterval );
            }
            //rdtsc(cpuEndTime);
        }

        printf("Press a key and Enter to end the loop ...\n");
        scanf("%s",&dummy1);

#ifdef WFS
        printf("Saving WFS centroids\n");
        outputs =  mlfSaveAOVar( centroids, mxCreateString("WFS_centroids" ));

#endif

#ifdef PROCESS_STATS
        outputStats();
#endif
        // MDF -- the following uncommented for testing
        //gErrorReporter->reportHeartBeatError(cpuHeartBeatInterval);

#ifdef MIRRORS
        outputs =
          mlfSaveAOVar(residualDMCoefficients, mxCreateString("residualDMCoefficients"
));
#endif

#ifdef G_FRAME
        outputs =  mlfSaveAOVar( gFrame, mxCreateString("gFrame" ));
#endif

        exp_time_out = mclInitializeDoubleVector(1, 1, &exp_time);
        outputs =  mlfSaveAOVar(exp_time_out, mxCreateString("exp_time" ));
#ifdef MIRRORS
        double n_i_gain_ramp = numiter_gain_ramp;
        numiter_gain_ramp_out = mclInitializeDoubleVector(1, 1, &n_i_gain_ramp);
        outputs =  mlfSaveAOVar(numiter_gain_ramp_out, mxCreateString("numiter_gain_ramp
" ));
#endif
        (*loopIteration)--;
        outputs =  mlfSaveAOVar(kIterations, mxCreateString("num_loop_iterations" ));

        if(heartbeat_wait == false)
        {
            double t_p_c = 0;
            tics_per_cycle = mclInitializeDoubleVector(1, 1, &t_p_c);
            outputs =  mlfSaveAOVar(tics_per_cycle, mxCreateString("tics_per_cycle" ));
        }
        else
        {
```



```
            double t_p_c = cpuHeartBeatInterval;
            tics_per_cycle = mclInitializeDoubleVector(1, 1, &t_p_c);
            outputs =  mlfSaveAOVar(tics_per_cycle, mxCreateString("tics_per_cycle" ));
        }

    }
    catch (...)
    {
        std::cerr << "Unexpected error thrown" << std::endl;
        return -1;
    }

    return 0;
}
```

```
#ifndef __WFS_1M150_EDT_H_
#define __WFS_1M150_EDT_H_

#include "edtinc.h"

//--- 1M150 Specifications ---
//#define CAMERA_1M150_RESX 232
//#define CAMERA_1M150_RESY 232
#define CAMERA_1M150_RESX 160
#define CAMERA_1M150_RESY 160
#define CAMERA_1M150_FRAME_SIZE (CAMERA_1M150_RESX*CAMERA_1M150_RESY)
#define REG_TIME_INCREMENT_MSEC 40000

// Define the following two if using DMA
#define USE_DMA

// --- Centroiding algorithm specifications ---
// -- WFSA
#define XOFFSETPX 0
#define YOFFSETPX 19
// -- WFSB
//#define XOFFSETPX 13
//#define YOFFSETPX 15

// Number of pixels across one row (or down one column) of a subaperture
// -- WFSA
#define NPX 16
// -- WFSB
//#define NPX 19

#define NUM_1M150_LENSLETS 7

// If you're on-sky, you have a central obscuration from the secondary mirror
#define CENTRAL_OBSCURATION
#ifdef CENTRAL_OBSCURATION
#define NVALIDLENSLET 36
#else
#define NVALIDLENSLET 37
#endif

//--- EDT framegrabber serial camera interface constants ---
#define BUFSIZE 512
#define READ_EXPOSURE_TIME_HEX "11 10 F"
#define WRITE_EXPOSURE_TIME_HEX_TEMPLATE "51 8X CX 50 8X CX 4F 8X CX"
#define READ_FRAME_TIME_HEX "17 16 15"
#define WRITE_FRAME_TIME_HEX_TEMPLATE "57 8X CX 56 8X CX 55 8X CX"
//#define ROI_FREE_PLACEMENT "4D 88 C0"
#define LINE_PRELOAD_AND_CAMERA_FREERUN "4D 88 C2"
#define LINE_PRELOAD_AND_CAMERA_EDGE_TRIGGERED "4D 8A C2"
```



```
#define EXSYNC "4C 81 C4"
// For line pause of 40
//#define LINE_PAUSE "60 80 C4"
// For line pause of 8
#define LINE_PAUSE "60 88 C0"
// The following hex codes are for ROI of 160x160 for WFSA
// (X0,Y0) = 408,386 & (X1,Y1) = 567,545
#define X0 "59 81 C0 58 88 C9"
#define Y0 "5B 81 C0 5A 82 C8"
#define X1 "5D 82 C0 5C 87 C3"
#define Y1 "5F 82 C0 5E 81 C2"

// For gaussian centroiding scheme
static double gauss[NPX * NPX] =
{
0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000 , 0.0000, 0.0000, 0.0000,
0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000 , 0.0000, 0.0000, 0.0000
, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0001 , 0.0000, 0.0000, 0.0000
, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000, 0.0001, 0.0003, 0.0009, 0.0013 , 0.0009, 0.0003, 0.0001
, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0001, 0.0006, 0.0036, 0.0103, 0.0146 , 0.0103, 0.0036, 0.0006
, 0.0001, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0003, 0.0036, 0.0207, 0.0586, 0.0829 , 0.0586, 0.0207, 0.0036
, 0.0003, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0009, 0.0103, 0.0586, 0.1660, 0.2349 , 0.1660, 0.0586, 0.0103
, 0.0009, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0001, 0.0013, 0.0146, 0.0829, 0.2349, 0.3325 , 0.2349, 0.0829, 0.0146
, 0.0013, 0.0001, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0009, 0.0103, 0.0586, 0.1660, 0.2349 , 0.1660, 0.0586, 0.0103
, 0.0009, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0003, 0.0036, 0.0207, 0.0586, 0.0829 , 0.0586, 0.0207, 0.0036
, 0.0003, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0001, 0.0006, 0.0036, 0.0103, 0.0146 , 0.0103, 0.0036, 0.0006
, 0.0001, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000, 0.0001, 0.0003, 0.0009, 0.0013 , 0.0009, 0.0003, 0.0001
, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0001 , 0.0000, 0.0000, 0.0000
, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000 , 0.0000, 0.0000, 0.0000
, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000 , 0.0000, 0.0000, 0.0000
, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000 , 0.0000, 0.0000, 0.0000
, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000

};
```

```
class WFS_1M150
{
public:
    WFS_1M150();
    ~WFS_1M150() { shutdown(); }

    // Flag set by pipeline to indicate whether a framegrabber and camera init are
needed
    bool init_flag;

    /**
     * Initialize the camera firmware.
     */
    bool init(double exposure_time_msec, double frame_time_msec);

    /**
     * Close the EDT device.
     */
    void shutdown();

    /**
     * Data dimension accessor methods:
     *   getWidth, getHeight, getDepth, getPitch
     */
    int getWidth();
    int getHeight();
    int getDepth();
    int getPitch();

    /**
     * Lock the most recent frame recieved from camera.
     * @param blocking Wait until next most recent frame arrives.
     * @returns True if frame was successfully locked.
     */
    bool lock( bool blocking = false );

    /**
     * Unlock newest frame.
     */
    void unlock();

//--- Locked Frame accessor methods, only work when frame is successfully locked.

    /**
     * Get the pointer to the locked frame data.
     */
    unsigned char *getFrameData();
```



```
    /**
     * Get the timestamp of our most recently locked frame.
     */
    unsigned int *getLastFrameTime();

protected:
    // EDT PDV framegrabber device instance object type
    PdvDev *pdv_p;
    // Boolean flag to indicate camera initialisation state
    bool camera_initialised;
    // Boolean flag to ensure exclusive access to the camera
    bool camera_locked;
    // Pointer to buffer containing the most recently acquired frame
    unsigned char *imagebuf;
    // Timestamp of last frame acquired by EDT card: timestamp[ms, us]
    unsigned int timestamp[2];
    // Utilities for serial communication with the camera
    bool CameraSerialWriteRead(char *argv, char *output);
    bool setRegister(double time_msec, char *cmdTemplate);
    double getRegister(char *cmd);
};

#endif // __WFS_1M150_H_
```

```
/**
 * Wavefront Sensor Interface
 * Initialize the EDT board
 * and gather frame data.
 */

#include <stdint.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

#include "WFS_1M150_EDT.h"

#include <limits.h>

#define gVerbosityLevel 0

#define SUCCESS 0
#define FAILURE -1

//#define TRIGGERED
//#define PW_TRIG
//#define FREERUN
#ifdef FREERUN
    // Number of buffers in DMA ringbuffer
    #define NUM_CAMERA_BUFFERS 4
#else
    #define NUM_CAMERA_BUFFERS 2
#endif

// If you want to write out to aoVariables whether FREERUN is set or not:
#define WRITE_TO_AOVARS
#ifdef WRITE_TO_AOVARS
#include "matlab.h"
#include "saveAOVar.h"
mxArray *freerun_or_triggered = NULL;
mxArray *wfs_output = NULL;
#endif

// To query the camera to check that exposure/frame times are set correctly
//#define CAMERA_QUERY

unsigned int mcl;

WFS_1M150::WFS_1M150()
{
    camera_initialised = false;
    camera_locked = false;
```



```
    pdv_p = NULL;
    imagebuf = NULL;
    init_flag = true;
}

/**
 * Initialize camera hardware, setup frame buffers, and start the camera capturing
frames.
 */
bool WFS_1M150::init(double exposure_time_msec, double frame_time_msec)
{

    int     unit = 0, channel = 0, i;
    char    devname[128];
    char    errstr[64];
    char serialOut[BUFSIZE];
    int dmasize = 0;

    /*
    * open the interface
    *
    * EDT_INTERFACE is defined in edtdef.h (included via edtinc.h).
    * With one EDT PCI DV board in the chassis it's addressed as
    * unit 0; the first channel (bottom connector) on multi-channel
    * boards such as the PCI DV C-Link, is channel 0
    *
    */

    printf("initialising camera and framegrabber (do not interrupt)...\n");
    if((pdv_p = pdv_open(EDT_INTERFACE, unit))==NULL)
    //if ((pdv_p = pdv_open_channel(EDT_INTERFACE, unit, channel)) == NULL)
    {
        sprintf(errstr, "pdv_open_channel(%s%d_%d)", devname, unit, channel);
        pdv_perror(errstr);
        return FALSE;
    }

    if(init_flag)
    {
    // Set ROI according to X0, Y0, X1, Y1 in WFS_1M150_EDT.h
    CameraSerialWriteRead(LINE_PAUSE,serialOut);
    CameraSerialWriteRead(X0,serialOut);
    CameraSerialWriteRead(Y0,serialOut);
    CameraSerialWriteRead(X1,serialOut);
    CameraSerialWriteRead(Y1,serialOut);
#ifdef FREERUN
    CameraSerialWriteRead(LINE_PRELOAD_AND_CAMERA_FREERUN,serialOut);
#endif
#ifdef TRIGGERED
```

```
    CameraSerialWriteRead(EXSYNC,serialOut);
    CameraSerialWriteRead(LINE_PRELOAD_AND_CAMERA_EDGE_TRIGGERED,serialOut);
#else // Pseudo-freerun mode
    CameraSerialWriteRead(LINE_PRELOAD_AND_CAMERA_FREERUN,serialOut);
#endif

    if (gVerbosityLevel > 0)
        printf("Setting the frame time to %.4fms...\n",frame_time_msec);

    if ( setRegister(frame_time_msec,WRITE_FRAME_TIME_HEX_TEMPLATE) == false)
        {
            printf("ERROR: Changing the frame time failed!\n");
            camera_initialised = false;
            return false;
        }
    else
        {
            if (gVerbosityLevel > 0)
                printf("Setting the exposure time to  %.4fms...\n",exposure_time_msec);

            if ( setRegister(exposure_time_msec,WRITE_EXPOSURE_TIME_HEX_TEMPLATE) ==
false)
            {
                printf("ERROR: Changing the exposure time failed!\n");
                camera_initialised = false;
                return false;
            }
        }

#ifdef CAMERA_QUERY
printf("Exposure time setting on camera: %.4fms\n",getRegister(READ_EXPOSURE_TIME_HEX));
printf("Frame time setting on camera: %.4fms\n",getRegister(READ_FRAME_TIME_HEX));
#endif
    }

    camera_initialised = true;
    camera_locked = false;

#ifdef WRITE_TO_AOVARS
#ifdef FREERUN
double freerun = 1;
freerun_or_triggered = mclInitializeDoubleVector(1, 1, &freerun);
wfs_output =  mlfSaveAOVar(freerun_or_triggered, mxCreateString("freerun_or_triggered"
));
#else
double freerun = 0;
freerun_or_triggered = mclInitializeDoubleVector(1, 1, &freerun);
wfs_output =  mlfSaveAOVar(freerun_or_triggered, mxCreateString("freerun_or_triggered"
));
```



```
#endif
#endif

#ifdef USE_DMA
/* Construct a ring buffer */
    if (pdv_multibuf(pdv_p, NUM_CAMERA_BUFFERS) == -1)
    {
        char    msg[64];

        sprintf(msg, "pdv_multibuf(0x%x, %d)", pdv_p, NUM_CAMERA_BUFFERS);
        pdv_perror(msg);
        return (0);
    }
    dmasize = pdv_get_dmasize(pdv_p);
    printf("dma size = %d\n",dmasize);

    // 4 second timeout on image acquisition should be really obvious to the user
    pdv_set_timeout(pdv_p, 4000);

    #ifdef FREERUN
    //printf("calling pdv_start_images...\n");
    pdv_start_images(pdv_p, 0); // 0 means freerun
    #endif
    #ifdef TRIGGERED
    // Triggered exposure mode.

        #ifdef PW_TRIG  // Pulse-width triggered
    // Want CC3 to be used as PRIN (exposure control) signal, so
    // PDV_MODE_CNTL = 0100 0000 = 0x40 -- this can be set in .cfg file too
    //edt_reg_write(pdv_p, PDV_MODE_CNTL,0x40);

    mcl = edt_reg_read(pdv_p, PDV_MODE_CNTL); // save PDV_MODE_CNTL settings
    // Every pdv_start_image call will hold PRIN high for (exposure_time_msec*1000)
microsecs
    //pdv_set_exposure_mcl(pdv_p, (int)(exposure_time_msec*1000));
    camera_locked = true; // need this for the first dummy frame to get past lock()
        #else
    // Assert EXSYNC
    //edt_reg_write(pdv_p, PDV_MODE_CNTL,0x10);
/*  edt_reg_write(pdv_p, PDV_MODE_CNTL,0);

    printf("init: PDV_MODE_CNTL: %x \n",edt_reg_read(pdv_p, PDV_MODE_CNTL));

    mcl = edt_reg_read(pdv_p, PDV_MODE_CNTL) & 0xf0;

    pdv_start_image(pdv_p); // Start the first image capture

    // manually open shutter (EXSYNC line CC1) to start first image
    edt_reg_write(pdv_p, PDV_MODE_CNTL, mcl | 0x1);
```

```
*/
        #endif
    #else  // Pseudo-freerun
    pdv_start_image(pdv_p); // Start the first image capture
    #endif
#else
    // Start the first image capture
    pdv_start_image(pdv_p);
#endif

        return true;
}

/**
 * Data dimension accessor methods:
 *   getWidth, getHeight, getDepth, getPitch
 */
int WFS_1M150::getWidth() { return pdv_get_width(pdv_p); }
int WFS_1M150::getHeight() { return pdv_get_height(pdv_p); }
int WFS_1M150::getDepth() { return pdv_get_depth(pdv_p); }
int WFS_1M150::getPitch() { return pdv_get_pitch(pdv_p); }

/**
 * EDT card serial communication routine (via CameraLink)
 */
bool WFS_1M150::CameraSerialWriteRead(char *argv, char *output)
    {
    int     argc = 1;
    int     unit = 0;
    int     hexin = 1;
    int     inter = 0;
    int     verbose = 0;
    int     do_wait = 0;
    int     timeout = 0;
    int     foiunit = -1;
    int     channel = 0;
    int     readonly = 0;
    int     baud = 0;
    int     i;
    int     ret;
    int     nbytes;
    int     length;
    u_char  hbuf[BUFSIZE];
    char    tmpbuf[BUFSIZE];
    char    getbuf[BUFSIZE];
    char    *ibuf_p;
    u_char  lastbyte, waitc;
    char    buf[BUFSIZE];
    char    bs[32][3];
```



```
    bool success = true;

// Fails here when compiled with USE_DMA.  Why???
//  if (timeout < 1)
        timeout = pdv_p->dd_p->serial_timeout;

//timeout = pdv_get_serial_timeout(pdv_p);
//pdv_p->dd_p->serial_timeout = 1000;
//printf("serial timeout %d\n", timeout);

    //if (verbose)
    //  printf("serial timeout %d\n", timeout);
//printf("foiunit = %d\n",foiunit);
    if (foiunit >= 0)
    {
        /* edt_foi_autoconfig(ed) ; */
        edt_set_foiunit(pdv_p, foiunit);
        edt_set_rci_dma(pdv_p, foiunit, 0);
    }

    if (inter)
    {
        ibuf_p = getbuf;
        printf("\nEnter command (Ctrl-C to quit)\n\n");
    }
    else
    {
        ibuf_p = argv;
    }

    if (baud)
    {
        pdv_set_baud(pdv_p, baud);
    }

    /* flush any junk */
    (void) pdv_serial_read(pdv_p, buf, BUFSIZE);

    do
    {
        if (inter)
        {
            printf("> ");
            fgets(ibuf_p,BUFSIZE-1,stdin);
        }

        if (!readonly)
        {
            if (hexin)
```

```
            {
                nbytes = sscanf(ibuf_p, "%s %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s
%s",
                                bs[0], bs[1], bs[2], bs[3], bs[4], bs[5], bs[6], bs[7],
                                bs[8], bs[9], bs[10], bs[11], bs[12], bs[13], bs[14],
bs[15]);

                for (i = 0; i < nbytes; i++)
                {
                    if (strlen(bs[i]) > 2)
                    {
                        printf("hex string format error\n");
                        break;
                    }
                    hbuf[i] = (u_char) (strtoul(bs[i], NULL, 16) & 0xff);
                }

                pdv_serial_binary_command(pdv_p, (char *) hbuf, nbytes);
                /* edt_msleep(10000); */
            }
        }

            //pdv_serial_wait(pdv_p, timeout, 64);
            pdv_serial_wait(pdv_p, 1000, 64);


            if (do_wait)
            {
                printf("return to read response: ");
                getchar();
            }

        /*
                * read and print the response. Could be large or small so loop
                * no more characters (OR waitchar seen, if any). Format output
                * for 1) ASCII, 2) HEX, or 3) Pulnix STX/ETX format, as
                * appropriate
        */
            do
            {
                ret = pdv_serial_read(pdv_p, buf, BUFSIZE);
                if (verbose)
                    printf("read returned %d\n", ret);

                if (*buf)
                    lastbyte = (u_char)buf[strlen(buf)-1];

                if (ret != 0)
                {
```



```
                    buf[ret + 1] = 0;
                    if (hexin)
                    {
                        int     i;
                        char tmpBuf[BUFSIZE];

                        if (ret)
                        {
//              printf("resp <");
                            for (i = 0; i < ret; i++) {
                                sprintf(output+2*i,"%02x",(u_char) buf[i]);
//            printf("%s\n",tmpBuf);
//            if (strcmp(tmpBuf,"06")!=0) {
//              //printf("ERROR: Exposure time failed!\n");
//              success = false;
//            }
/*              printf("%s%02x", i ? " " : "", (u_char) buf[i]); */
                            }
//              printf(">\n");
                        }
                    }
                    else        /* simple ASCII */
//          print_ascii_string(buf);
                        printf("WARNING: Cannot read camera register!\n Check if the cam
era is powered on and connected");
                    length += ret;
            //  success = false;
                }

                if (pdv_p->devid == PDVFOI_ID)
                {
 printf("here\n");

                    ret = pdv_serial_wait(pdv_p, 500, 0);
                }
                else if (pdv_get_waitchar(pdv_p, &waitc) && (lastbyte == waitc))
                {
                    ret = 0; /* jump out if waitchar is enabled/received */
                }
                else
                {
                ret = pdv_serial_wait(pdv_p, 500, 64);
                }
            } while (ret > 0);
    } while (inter);

return success;
}
```

```
/**
 * EDT card register writing routine (for setting exposure time)
 */
bool WFS_1M150::setRegister(double time_msec, char *cmdTemplate) {

    long nIncrement = long(REG_TIME_INCREMENT_MSEC*time_msec);
    int k, kc = 4;
    char hexIncrement[BUFSIZE], command[BUFSIZE], tmpBuf[BUFSIZE], serialOut[BUFSIZE];

    // Command template
    strcpy(command,cmdTemplate);
    sprintf(tmpBuf,"%x",nIncrement);

    // Check exposure time length
    int nHex = strlen(tmpBuf);
    if (nHex>6) {
        printf("ERROR: Two large value!!\n");
        return false;
    }

    // Pad the hex exposure time with zeros
    for (k=0;k<6-nHex;k++)
        strcpy(hexIncrement+k,"0");

        strcpy(hexIncrement+k,tmpBuf);

    // Set the command
    for (k=0 ; k<strlen(hexIncrement)/2; k++) {
        strncpy(command+kc,hexIncrement+2*k+1,1);
        strncpy(command+kc+3,hexIncrement+2*k,1);
        kc+= 9;
    }

//     printf("command=%s\n",command);
    CameraSerialWriteRead(command,serialOut);
    return ( strtol(serialOut,NULL,16) == 2147483647 ) ? true : false;
}

/**
 * EDT card register reading routine
 */
double WFS_1M150::getRegister(char *cmd) {
    char serialOut[BUFSIZE];
    CameraSerialWriteRead(cmd,serialOut);
    return strtol(serialOut,NULL,16)/REG_TIME_INCREMENT_MSEC;
}

/**
 * Close down the camera hardware.
```



```
 */
void WFS_1M150::shutdown()
{
    int rval;

    //printf("WFS_1M150_EDT::shutdown camera, and clean-up.\n" );
    rval = pdv_close(pdv_p);

    if( rval != 0 )
    {
        printf ("WFS_1M150_EDT::shutdown() pdv_close returned %i\n", rval);
    }
}


/**  FIXME: MDF -- need to do more to support EDT blocking
 * lock
 */
bool WFS_1M150::lock( bool blocking )
{
    if( blocking )
    {
        if( gVerbosityLevel >= 2 )
            printf( "WFS_1M150::lock() waiting for frame.\n" );

#ifdef USE_DMA
   // Wait for next available image. Address of imagebuf acquired in getFrameData()
   imagebuf = pdv_wait_image_timed(pdv_p, timestamp);
#endif

    }

    camera_locked = true;
    return true;
}

/** FIXME: MDF -- need to do more to support EDT blocking
 * unlock
 * Clear camera_locked flag to disable all the locked frame accessor methods.
 */
void WFS_1M150::unlock()
{
    if( gVerbosityLevel >= 2 )
        printf( "WFS_1M150::unlock()\n" );

    camera_locked = false;
}

//--- Locked Frame accessor methods, only work when frame is successfully locked.
```

```
/**
 * Get the locked frame data, unsigned char[].
 */
unsigned char *WFS_1M150::getFrameData()
{
    if( camera_locked == false )
        return 0;

    /*
    * acquire an image
    */

#ifdef USE_DMA

    #ifdef TRIGGERED
/*printf("getFrameData about to drop EXSYNC. PDV_MODE_CNTL reg: %x\n",edt_reg_read(pdv_p
, PDV_MODE_CNTL));
    edt_reg_write(pdv_p, PDV_MODE_CNTL, mcl);
printf("getFrameData waiting for image. PDV_MODE_CNTL reg: %x\n",edt_reg_read(pdv_p,
PDV_MODE_CNTL));
*/
    #endif
    // The following moved to lock() for now
    //imagebuf = pdv_wait_image_timed(pdv_p, timestamp);

    #ifdef FREERUN
    pdv_start_image(pdv_p);
    #endif
    #ifdef TRIGGERED
        #ifdef PW_TRIG
        // assert EXSYNC (line CC1) to reset camera readout
            //edt_reg_write(pdv_p, PDV_MODE_CNTL, mcl | 0x01);
        // assert PRIN (line CC3) to start exposure
        //edt_reg_write(pdv_p, PDV_MODE_CNTL, mcl | 0x04);

        pdv_start_image(pdv_p);
        // deassert EXSYNC (line CC1)
                //edt_reg_write(pdv_p, PDV_MODE_CNTL, mcl);
                edt_reg_write(pdv_p, PDV_MODE_CNTL, mcl & 0xFE);

        #else

        // manually open shutter (EXSYNC line CC1) to start next image
        edt_reg_write(pdv_p, PDV_MODE_CNTL, mcl | 0x1);
        printf("getFrameData got image. Reassert EXSYNC to start new image.  PDV_MODE_CN
TL reg: %x\n",edt_reg_read(pdv_p, PDV_MODE_CNTL));
        pdv_start_image(pdv_p);
        #endif
```



```
    #else // Pseudo-freerun mode
    pdv_start_image(pdv_p);
    #endif
#else // No DMA
    pdv_start_image(pdv_p);
#endif
    return imagebuf;
}

/**
 * Get the integration time of our most recently locked frame.
 */
unsigned int *WFS_1M150::getLastFrameTime()
{
    if( camera_locked == false )
        return 0;

    return timestamp;
}
```

```
/*
 * Rodolphe's wonderful spot centroiding calculation. (it's FAST!)[er then matlab]
 * March 7, 2006 (Tuesday)
 */
#include <stdio.h>
#include "WFS_1M150_EDT.h"

//#define rdtsc(t) asm __volatile__ ("rdtsc" : "=A" (t))

// Uncomment the following if you are applying thresholding in the centroiding algorithm
#define THRESHOLDING

//#define MATLAB_DEBUG
#ifdef MATLAB_DEBUG  // for partial_frame output, use MEX
#include "matlab.h"
#include "loadAOVar.h"
#include "saveAOVar.h"
mxArray *pixel_buf = NULL;
mxArray *out = NULL;
#endif

#define PRINT_WARNINGS

#ifdef CENTRAL_OBSCURATION
static double validLenslet[NUM_1M150_LENSLETS * NUM_1M150_LENSLETS] =
{
    0,0,1,1,1,0,0,
    0,1,1,1,1,1,0,
    1,1,1,1,1,1,1,
    1,1,1,0,1,1,1,
    1,1,1,1,1,1,1,
    0,1,1,1,1,1,0,
    0,0,1,1,1,0,0
};
#else
    static double validLenslet[NUM_1M150_LENSLETS * NUM_1M150_LENSLETS] =
{
    0,0,1,1,1,0,0,
    0,1,1,1,1,1,0,
    1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,
    0,1,1,1,1,1,0,
    0,0,1,1,1,0,0
};
#endif

static double gDoubleTranslationBuffer[256] =
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
```



```
 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36,
 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48,
 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72,
 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96,
 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108,
 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120,
 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132,
 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144,
 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156,
 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168,
 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180,
 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192,
 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204,
 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216,
 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228,
 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240,
 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252,
 253, 254, 255
};

static int gVerbosityLevel = 0;

void cleanup();

void centroidCapture( unsigned char *buffer, double *centroids,
              double *pixelBiasMap, double *pixelGainMap,
              double *refMeasurements, double *threshold, int loop_iter)
{
    int iLenslet, jLenslet, iPx, jPx, k1, k2, kLenslet, count=0, kSumPx=0, flag=1, i, j;
    double sumPx=0.0, xSumPx=0.0,ySumPx=0.0, px, pxThresholded;

#ifdef MATLAB_DEBUG
    pixel_buf = mxCreateDoubleMatrix(CAMERA_1M150_RESY, CAMERA_1M150_RESX, mxREAL);
    pixel_buf = mclInitializeDoubleVector( mxGetM(pixel_buf), mxGetN(pixel_buf),
mxGetPr(pixel_buf));
#endif

    // MDF - the following nested loop could be parallelised with dual-processors
    for ( jLenslet=0; jLenslet<NUM_1M150_LENSLETS; jLenslet++)
    {
        for ( iLenslet=0; iLenslet<NUM_1M150_LENSLETS; iLenslet++)
        {
            kLenslet = jLenslet*NUM_1M150_LENSLETS + iLenslet;
            if (validLenslet[kLenslet])
            {
                sumPx=0.0;
```

```
                xSumPx=0.0;
                ySumPx=0.0;
                for (jPx =0; jPx<NPX; jPx++)
                {
                    for (iPx =0; iPx<NPX; iPx++)
                    {
                        // NOTE: All this inner-loop computation is extremely expensive.
                        //  (yep, lots of room for improvement)

                        // TODO: Changing all variables (except output 'centroids'
vector)
                        //  from double to float will allow SIMD operations to be used h
ere.

                        // TODO: Compute starting offsets outside iPx and jPx loops for
k1 and k2,
                        // only increment in inner loop.
                        i  = XOFFSETPX + iLenslet*NPX + iPx;
                        j  = YOFFSETPX + jLenslet*NPX + jPx;
                        k1 = i*CAMERA_1M150_RESY + j;
                        k2 = i*CAMERA_1M150_RESX +  j;

                        // NOTE:
                        //  pixelBiasMap is negated in aoVariables so subtract is implic
it, allows us to add instead.
                        //    (relevent for SIMD MADD instruction)
                        //  pixelGainMap is reciprocated so we multiply instead of expen
sive divide.
                        //    (multiply and add instruction can be combined into one MAD
D instruction)

                        px = ( gDoubleTranslationBuffer[buffer[k1]] + pixelBiasMap[k1]
);

#ifndef THRESHOLDING // Don't Apply thresholding
                        if( px<=0 )
                            px = 0;
    #ifdef MATLAB_DEBUG
                    //if (kLenslet == __)
                        mxGetPr(pixel_buf)[k1] = px;
    #endif
                        sumPx  += px;
                        xSumPx += jPx*px;
                        ySumPx += iPx*px;
#else // Apply thresholding
                        pxThresholded = px - *threshold;
                        if( pxThresholded<=0 )
                            pxThresholded=0;
    #ifdef MATLAB_DEBUG
```



```
                    if (kLenslet == 23)
                        mxGetPr(pixel_buf)[k1] = pxThresholded + 30;
                    else
                        mxGetPr(pixel_buf)[k1] = pxThresholded;
    #endif
                        sumPx  += pxThresholded;
                        xSumPx += jPx*pxThresholded;
                        ySumPx += iPx*pxThresholded;
#endif
                    }
                }

                if (sumPx == 0)
                {
                    kSumPx++;
                    centroids[count] = 0;//refMeasurements[count];
                    centroids[count + NVALIDLENSLET] = 0;//refMeasurements[count +
NVALIDLENSLET];

                    if(loop_iter)  // anything greater than 0
                                        {
                                                // If no light in subaperture, take
centroids from last loop iteration to account for intermittent drop-out.
                                                centroids[count] = centroids[count-
NVALIDLENSLET*2];
                                                centroids[count + NVALIDLENSLET] =
centroids[count-NVALIDLENSLET];
                                        }
                                        // to guard against segmentation faults, in case
the lenslets in the first frame are empty:
                                        else
                                        {
                                                centroids[count] = 0;
                                                centroids[count + NVALIDLENSLET] = 0;
                                        }
                }
                else
                {
                    centroids[count] = refMeasurements[count] - xSumPx/sumPx;
                    centroids[count + NVALIDLENSLET] = refMeasurements[count +
NVALIDLENSLET] - ySumPx/sumPx;
                }
                count++;
            } //end valid lenslet
        } // for jLenslet
    } // for iLenslet

#ifdef PRINT_WARNINGS
    if (kSumPx>0)
```

```
      printf("WARNING: %d lenslets with all pixel at 0\n",kSumPx);
#endif

#ifdef MATLAB_DEBUG
    printf("Saving centroiding window...\n");
    out =  mlfSaveAOVar( pixel_buf, mxCreateString("centroiding_window"));
#endif

    return;
}
```

```
/**
 MirrorDriver
 University of Victoria
 LACIR - Adaptive Optics
 Primary Author: Aaron Hilton  (otri@uvic.ca)
 Adaptations for Victoria Open Loop Testbed: Mike Fischer (mike.fischer@nrc.ca)
 Last update: December 2007
**/

#ifndef __MIRRORDRIVER_H_
#define __MIRRORDRIVER_H_

#include "dask.h"
//#include "conio.h"

#define TT_DIGITAL_MAX 0xffff
#define TT_DIGITAL_MIN 0x7fff

#define NDMCOMMANDS 52

class MirrorDriver
{
public:
    MirrorDriver();

    ~MirrorDriver() { shutdown(); }

    bool init();

    void shutdown();

    /**
     * Prepare mirror control DMA Buffers.
     * @param tiptilt [2x1] matrix.
     * @param DM [8x8] matrix.
     * @returns true if all signals are in range.
     *          false if anything clipped.
     */
    bool prepareBuffers( double *tiptilt, double *DM );

    /**
     * Drive the mirrors to their target positions.
     */
    void actuateMirrorsNow();

    /**
     * Get the clipping flags.
     */
    int getClipFlags();
```



```
    /**
     * Clipping flags.
     * H represents High clipping, or value exceeded Maximum.
     * L represents Low clipping, or value was below Minimum.
     */
    static const int CLIP_TT_H = 1<<0;
    static const int CLIP_TT_L = 1<<1;
    static const int CLIP_DM_H = 1<<2;
    static const int CLIP_DM_L = 1<<3;

    /**
     * Get the clip limit.
     * @param clipflag One of the above clip flags.
     */
    double getClipLimit( int clipflag );

    /**
     * Clear the tiptilt mirror to Zero voltage state (for power-off)
     */
    void clearTipTilt();

    /**
     * Clear the DM mirror.
     */
    void clearDM();

    /**
     * Immediately set the power state of the DM driver electronics.
     */
    void setDMPower( bool state );

    /**
     * Analog subsystem voltage to digital conversion.
     *
     * Analog output conversion is:
     * -10v == 0
     *   0v == 0x7fff
     *  10v == 0xffff
     */
    unsigned short AOVolts2Digital( double v );

protected:
    // MDF May 23 -- AnalogBoard is Advantech PCI-1720U
    //               DigitalBoard is Adlink PCI-7200
    int AnalogBoard, DigitalBoard;

    //============ TIPTILT =============
    /**
```

```
     * Prepare the tiptilt command buffer from the
     * 2x1 tiptilt vector.
     * FIXME: Change tiptilt input from volts to pure digital form!!
     */
    void prepareTipTilt( double *data );

    unsigned short TipTiltOutput[2];

    //============ DM =============
    /**
     * Prepare the DM command buffer from the
     * 8x8 DM actuator matrix.
     */
    void prepareDM( double *data );

    /**
    * Queue output to a single DM actuator.
    * @param actuator Any DM actuator from 0 to 63
    * @param voltage Any voltage value from 0 to 0x4000 (0x2000 is mid stroke)
    * WARNING: Avoid setting actuator voltages above 0x2800 or below 0x1800
    * NOTE: The above requirement is currently enforced.
    */
    void qDMSetActuator( int actuator, int voltage );

    /**
     * Queue DM command
     * @param data Data to write out to the DM.
     * @param highbit High bit which controls address and data sequence.
     */
    void qDMCommand( int data, int highbit );

    /**
     * Immediately send DM command.
     * @param data Data to write out to the DM.
     * @param highbit High bit which controls address and data sequence.
     */
    void imDMCommand( int data, int highbit );

    //============ DM COMMAND BUFFER =============

    unsigned int nDMCommands, nMaxDMCommands;
    unsigned long *DMCommandBuffer;

    // Set the clipping flag whenever clipping occurs.
    int clipping_flags;

    //--------- Hardware state ----------
    bool DMPowerState;
```



```
};

#endif //__MIRRORDRIVER_H_
```

```
/**
 MirrorDriver
 University of Victoria
 LACIR - Adaptive Optics
 Primary Author: Aaron Hilton  (otri@uvic.ca)
 Adapted by Mike Fischer for use in VOLT real-time control pipeline (mike.fischer@nrc.
ca)

 Last update: April 2, 2008

**/

#include "MirrorDriver.h"

#include <stdio.h>
#include <stdint.h>
#include <memory.h>

#include <sys/types.h>
#include <unistd.h>
#include <math.h>

#define HAVE_HARDWARE
//#define USE_DMA
//#define USER_PAUSE
//#define SIM_DM

// Uncomment the following to use CN2 connector on front mounting plate of PCI-7200
card,
// otherwise the CN1 connector on interior PCB surface will be used
#define CN2

#ifdef USER_PAUSE
char dummy1;
#endif

// 1.2 MHz DM command write frequency
#define ADLINK_WRITE_FREQ 1200000

#ifdef HAVE_HARDWARE
#include "conio.h"
#include <time.h>
#include <sys/time.h>
#endif // HAVE_HARDWARE

// Uncomment the following to enable tip-tilt mirror hardware
//#define TIP_TILT

// Defines for PCI-7200 card and ALPAO DE64 drive electronics
```



```
#define MASK_DAC_ADDR   0x3F
#define MASK_DAC_DATA  0x7F
#define CODE_POWER_ON   0x00D5
#define CODE_POWER_OFF  0x00EA

// Set the verbosity level of debug output.
//#define DEBUG_MATLAB
//#define DEBUG_BUFFERS
#define DEBUG_LEVEL 0
//#define FCN_ENTRY_PRINT

#if DEBUG_MATLAB
    #include "mex.h"
    #define printf mexPrintf
#endif

// DM strobe line is inverted, so everything except
// the DM commands must hold the strobe line high.
#ifdef CN2
    #define INV_STROBE (1<<8)
#else
    #define INV_STROBE (1<<24)
#endif

#ifdef TIPTILT
// Tip-Tilt channels are oddly placed with the new 96 Analog Output board.
// zero-based channel numbers
#define TT_CHANNEL_X 57
#define TT_CHANNEL_Y 58
#endif

//************
// Constrain the digital outputs to the mirrors to these ranges:

// General AnalogOutput tip-tilt voltage limits.
#define AO_VOLTS_MAX 10
#define AO_VOLTS_MIN -10

// Limit tiptilt voltage to 0v (0x7fff) and 10v (0xffff)
// *** Negative voltage WILL KILL the tiptilt drive electronics ***
#define TT_DIGITAL_MAX 0xffff
#define TT_DIGITAL_MIN 0x7fff

// Limit DM voltage to 0.5v (0x2800) and -0.5v (0x1800)
#define DM_DIGITAL_MAX 0x27ff
#define DM_DIGITAL_MIN 0x1800

// Writes to the PCI-7200 sporadically fail due to spurious interrupt generation on
// the IRQ assigned to the board on our new server machines, despite the conflicting
```

```
// hardware device (SCSI controller) being disabled at various levels.  The workaround
// is to have an automatic retry upon a write failure.
#ifndef TEMP_ADLINK_FAILURE_RETRY
#define TEMP_ADLINK_FAILURE_RETRY(expr) \
    ({ long int _res; \
       do _res = (long int) (expr); \
       while (_res < 0 ); \
       _res; })
#endif

//*************
// Mirror Maps

// DM Translation table.  Same as "map" in alpao_upload
// MDF May 16 -- this is the correct translation table for the Alpao mirror
static int sDMTranslate[64] =
{
    16, 17, 47, 62, 28, 35, 19, 26,
    32, 57, 55, 37, 63, 39, 53, 38,
    41, 44, 34, 33, 52, 46, 50, 59,
    42, 45, 36, 22, 61, 56, 54, 60,
    24, 29, 27, 15,  8, 11,  0,  6,
    30, 25, 23,  5, 31, 13,  4,  3,
    40,  9, 12,  2, 20, 21,  7, 43,
    48, 49,  1, 10, 18, 14, 51, 58,
};

// Valid actuators that actually exist.
static int sDMIsValidActuator[64] =
{
    0,0,1,1,1,1,0,0,
    0,1,1,1,1,1,1,0,
    1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,
    0,1,1,1,1,1,1,0,
    0,0,1,1,1,1,0,0,
};

//*************
MirrorDriver::MirrorDriver()
{
    AnalogBoard = 0;
    DigitalBoard = -1; // default PCI-7200 board ID to -1, as 0 is a valid ID

    // Set tiptilt to midpoint by default.
    mTipTiltOutput[1] = mTipTiltOutput[0] = (TT_DIGITAL_MAX+TT_DIGITAL_MIN)/2;
```



```
    nDMCommands = 0;
    nMaxDMCommands = 0;
    DMCommandBuffer = 0;
}

bool MirrorDriver::init()
{
    U16 ret;

#ifdef TIP_TILT
    // MDF May 28 -- FIXME: replace commented out T-T board init code below for PCI-1720
board
    /* code deleted */

    // Set tiptilt to midpoint by default.
    mTipTiltOutput[0] = mTipTiltOutput[1] = (TT_DIGITAL_MAX+TT_DIGITAL_MIN)/2;
#endif // TIP_TILT

// Initialise PCI-7200 DM driver board
#ifdef HAVE_HARDWARE
    // Attach to PCI-7200 Digital IO Board for driving DM
    DigitalBoard = Register_Card (PCI_7200, 0);
    if (DigitalBoard < 0)
    {
      printf ("ERROR: Register_Card failed %d\n", DigitalBoard);
      return -1;
    }
    // Set for asynchronous output
    ret = DO_7200_Config (DigitalBoard, TRIG_INT_PACER, OREQ_DISABLE, OTRIG_LOW);
    if (ret != NoError)
    {
      printf ("ERROR: PCI_7200_Config failed %d\n", ret);
      return -1;
    }

#endif // HAVE_HARDWARE

    // Allocate DM buffer
    nMaxDMCommands = 2*
        8*8 // Size of matrix
        *3  // requires 3 commands per actuator.
        *3; // requires a 3 part strobe signal sequence per command.

#if (DEBUG_LEVEL > 0)
    printf( "::init() => nMaxDMCommands=%d\n", nMaxDMCommands );
#endif // DEBUG_LEVEL > 0

    nDMCommands = 0;
    DMCommandBuffer = new unsigned long[nMaxDMCommands];
```

```
    clipping_flags = 0;
    return true;
}

#define safe_delete(x) { if(x) {delete x; x=0;} }

void MirrorDriver::shutdown()
{
#ifdef TIP_TILT
    clearTipTilt();
#endif  // TIP_TILT
    clearDM();
    setDMPower(false);

#ifdef HAVE_HARDWARE
    if (DigitalBoard >= 0)
    {
        if(Release_Card (DigitalBoard))
            printf("PCI7200: Release_Card failed.\n");
    }
    DigitalBoard=-1;
#endif // HAVE_HARDWARE

    safe_delete( DMCommandBuffer );
}

/**
 * Prepare mirror control buffers.
 * @param tiptilt [2x1] matrix.
 * @param DM [8x8] matrix.
 * @returns true if all signals are in range.
 *          false if anything clipped.
 */
bool MirrorDriver::prepareBuffers( double *tiptilt, double *DM)
{
#ifdef FCN_ENTRY_PRINT
    printf ("in ::prepareBuffers()\n");
#endif

    nDMCommands = 0;
    clipping_flags = 0;

    if( tiptilt )
        prepareTipTilt( tiptilt );

    if( DM )
        prepareDM( DM );
```



```
    return (clipping_flags==0);
}

/**
 * Drive the mirrors to their target positions.
 */
void MirrorDriver::actuateMirrorsNow()
{
    int i, ret;
    BOOLEAN last_write_done = FALSE;
    U32 data_written_cnt;

#ifdef FCN_ENTRY_PRINT
    printf ("in ::actuateMirrorsNow()\n");
#endif

// TipTilt mirror
#ifdef TIP_TILT
    #ifdef HAVE_HARDWARE
    // MDF May 28 -- FIXME: need to replace the following with code for PCI-1720 T-T dri
ver card
    //_PdAO96WriteHold( AnalogBoard, TT_CHANNEL_X, mTipTiltOutput[0] );
    //_PdAO96WriteHold( AnalogBoard, TT_CHANNEL_Y, mTipTiltOutput[1] );
    #endif // TIP_TILT
#endif // HAVE_HARDWARE
    // Make sure last asynchronous write to DIO is done
    do {

        ret = DO_AsyncCheck(DigitalBoard,&last_write_done,&data_written_cnt);
        if (ret != NoError)
        {
            printf ("ERROR ::imDMCommand() DO_AsyncCheck failed here %d\n", ret);
            return;
        }

    } while(!last_write_done)


    #if (DEBUG_LEVEL > 0)
    for( int i=0; i<nDMCommands; )
    {
        printf ("MirrorDriver::actuateMirrorsNow(): #commands=%d, command for actuator %
d = %d\n", nDMCommands, (DMCommandBuffer[i] >> 16)&0x3F, ((DMCommandBuffer[i+3] >>
16)&0x7F)<<7 | (DMCommandBuffer[i+6] >> 16)&0x7F);
        i = i + 9;
    }
    #endif

    // Write values to digital output pins
```

```
    if (TEMP_ADLINK_FAILURE_RETRY( DO_ContWritePort (DigitalBoard, 0, DMCommandBuffer,
nDMCommands, 1, ADLINK_WRITE_FREQ, ASYNCH_OP)) != NoError)
        {
            printf ("ERROR ::actuateMirrorsNow() TEMP_ADLINK_FAILURE_RETRY(DO_ContWriteP
ort) failed here %d\n", ret);
            return;
        }
}

/**
 * Get the clipping flags.
 */
int MirrorDriver::getClipFlags()
{
    return clipping_flags;
}

/**
 * Get the clip limit.
 * @param clipflag One of the above clip flags.
 */
double MirrorDriver::getClipLimit( int clipflag )
{
    switch( clipflag )
    {
        case CLIP_TT_H: return (double)(TT_DIGITAL_MAX);
        case CLIP_TT_L: return (double)(TT_DIGITAL_MIN);
        case CLIP_DM_H: return (double)(DM_DIGITAL_MAX);
        case CLIP_DM_L: return (double)(DM_DIGITAL_MIN);
        default:
#if (DEBUG_LEVEL > 0)
            printf("MirrorDriver: Unknown clip flag.\n" );
#endif // DEBUG_LEVEL > 0
            return -1;
    }
}

/**
 * Clear the tiptilt mirror to Zero voltage state (for power-off)
 * FIXME: MDF -- need to insert PCI-1700 T-T board library calls here
 */
void MirrorDriver::clearTipTilt()
{
    mTipTiltOutput[0] = mTipTiltOutput[1] = TT_DIGITAL_MIN;

    /*
    _PdAO96WriteHold( AnalogBoard, TT_CHANNEL_X, mTipTiltOutput[0] );
    _PdAO96WriteHold( AnalogBoard, TT_CHANNEL_Y, mTipTiltOutput[1] );
    _PdAO96Update( AnalogBoard );
```



```
    */
    // MDF May 28 -- replace commented code above with PCI-1700 T-T board library calls

}

/**
 * Clear the DM mirror state.
 */
void MirrorDriver::clearDM()
{
// Clear all channels to zero volts output (0x2000).
    int channelValue = 0x2000;

#ifdef FCN_ENTRY_PRINT
    printf( "in ::clearDM()\n" );
#endif

    for( int channel=0; channel<64; channel++ )
    {
        // Address with high bit on (starts data writing sequence).
        imDMCommand( (channel & MASK_DAC_ADDR), 1 );
        // High 7 data bits, high bit off.
        imDMCommand( ((channelValue >> 7) & MASK_DAC_DATA), 0 );
        // Low 7 data bits, high bit off.
        imDMCommand( ((channelValue >> 0) & MASK_DAC_DATA), 0 );
    }
#ifdef USER_PAUSE
    printf("clearDM(): hit a key and enter to continue... \n");
    scanf("%s",&dummy1);
#endif

}

/**
 * Immediately set the power state of the DM driver electronics.
 * The power state should be entirely managed by this driver.
 *  Commands Bytes (including high bits)
 *    0xD5 - CODE_POWER_ON - power electronics on
 *    0xEA - CODE_POWER_OFF - power electronics off
 */
void MirrorDriver::setDMPower( bool state )
{
#ifdef FCN_ENTRY_PRINT
    printf ("in ::setDMPower()\n");
#endif
    if( DMPowerState == state ) return;

    // On power ON state, set all actuators to zero volts.
    if( state == true )
```

```
        clearDM();

    // Send power state command.
    int command = state ? CODE_POWER_ON : CODE_POWER_OFF;

#if (DEBUG_LEVEL > 0)
    printf("MirrorDriver::setDMPower(): sending power state command = %x\n",command);
#endif

    imDMCommand( (command & 0x7F), 1 );

#ifdef USER_PAUSE
    printf("setDMPower(): hit a key and enter to continue... \n");
    scanf("%s",&dummy1);
#endif

    DMPowerState = state;
}


/**
 * TipTilt mirror voltage to digital voltage conversion.
 *
 * Analog output conversion is:
 * -10v == 0
 *   0v == 0x7fff
 *  10v == 0xffff
 */
unsigned short MirrorDriver::AOVolts2Digital( double v )
{
    if( v > AO_VOLTS_MAX )
        v = AO_VOLTS_MAX;
    else if( v < AO_VOLTS_MIN )
        v = AO_VOLTS_MIN;

    return (unsigned short)(0x7fff + (v * .1 * 0x7fff ));
}


//===== Protected Members ========

/**
 * Prepare the tiptilt command buffer from the
 * 2x1 tiptilt vector.
 */
void MirrorDriver::prepareTipTilt( double *data )
{
    // Do limit checking in here now, properly.
    for( int i=0; i<2; i++ )
```



```
    {
        // Fraction test, make sure no Voltage data is pushed through.
        int value = (int)data[i];
        double frac = (double)value - data[i];
        if( frac != 0.0 )
            printf( "MirrorDriver Warning: TipTilt data has fractional result.  Likely b
eing fed volts instead of digital data.\n" );

        if( value > TT_DIGITAL_MAX )
        {
            value = TT_DIGITAL_MAX;
            clipping_flags |= CLIP_TT_H;
        }
        else if( value < TT_DIGITAL_MIN )
        {
            value = TT_DIGITAL_MIN;
            clipping_flags |= CLIP_TT_L;
        }
        mTipTiltOutput[i] = (unsigned short)value;
//      printf( "TT%d: %f/%f\n", i, v, data[i] );
    }
}


//============ DM =============
/**
 * Prepare the DM command buffer from the
 * DM actuator vector.
 */
void MirrorDriver::prepareDM( double *data )
{
#ifdef FCN_ENTRY_PRINT
    printf ("in ::prepareDM()\n");
#endif
    nDMCommands = 0;
    for( int i=0; i<64; i++ )
    {
        qDMSetActuator( i, (int)data[i] );
    }
}

/**
* Queue output to a single DM actuator.
* @param actuator Any DM actuator from 0 to 63
* @param voltage Any voltage value from 0 to 4000 (2000 is mid stroke)
*/
void MirrorDriver::qDMSetActuator( int actuator, int voltage )
{
#ifdef FCN_ENTRY_PRINT
```

```
    printf ("in ::qDMSetActuator()\n");
#endif

    if( (actuator>=0) && (actuator < 64) && sDMIsValidActuator[actuator])
    {
        // Limit the voltage to 0.5v (0x2800) and -0.5v (0x1800)
        if( voltage > DM_DIGITAL_MAX )
        {
            printf( "qDMSetActuator: actuator: %d, voltage: %d > DM_DIGITAL_MAX...clippi
ng to 0.5V\n", actuator, voltage );
            voltage = DM_DIGITAL_MAX;
            clipping_flags |= CLIP_DM_H;
        } else if( voltage < DM_DIGITAL_MIN )
        {
            printf( "qDMSetActuator: actuator: %d, voltage: %d < DM_DIGITAL_MIN...clippi
ng to -0.5V\n", actuator, voltage );
            voltage = DM_DIGITAL_MIN;
            clipping_flags |= CLIP_DM_L;
        }

        // Strobe the actuator value (masked to keep it in range).

        // Strobe address with high bit on (starts data writing sequence).
        qDMCommand( (sDMTranslate[actuator] & MASK_DAC_ADDR), 1 );

        // Strobe high 7 data bits, high bit off.
        qDMCommand( ((voltage >> 7) & MASK_DAC_DATA), 0 );

        // Strobe low 7 data bits, high bit off.
        qDMCommand( ((voltage >> 0) & MASK_DAC_DATA), 0 );
    }
}

/**
* Queue DM command
* @param data Data to write out to the DM.
* @param highbit High bit which controls address and data sequence.
*/
void MirrorDriver::qDMCommand( int data, int highbit )
{
#ifdef FCN_ENTRY_PRINT
    printf ("in ::qDMCommand()\n");
#endif

    if( nDMCommands >= nMaxDMCommands-3 )
        return;

#ifdef CN2
// Changed bit-shift to use CN2 connector on mounting plate of PCI-7200 card
```



```
    int writtenCommand = ((data & MASK_DAC_DATA)) | (highbit<<7);
#else
// Using original CN1 connector on interior PCB surface of PCI-7200 card
    int writtenCommand = ((data & MASK_DAC_DATA)<<16) | (highbit<<23);
#endif

    DMCommandBuffer[nDMCommands] = writtenCommand | INV_STROBE;
    nDMCommands++;
    DMCommandBuffer[nDMCommands] = writtenCommand; // negative strobe
    nDMCommands++;
    DMCommandBuffer[nDMCommands] = writtenCommand | INV_STROBE;
    nDMCommands++;

}

/**
 * Immediately send DM command.
 * @param data Data to write out to the DM.
 * @param highbit High bit which controls address and data sequence.
 */
void MirrorDriver::imDMCommand( int data, int highbit )
{
    unsigned long output;
    U16 ret;
    BOOLEAN last_write_done = FALSE;
    U32 data_written_cnt;

#ifdef FCN_ENTRY_PRINT
    printf( "in ::imDMCommand()\n");
#endif

#ifdef CN2
    U32 command = ((data & MASK_DAC_DATA)) | (highbit<<7);
#else
    U32 command = ((data & MASK_DAC_DATA)<<16) | (highbit<<23);
#endif

    output = command | INV_STROBE; // hold strobe high

#if DEBUG_BUFFERS
    printf( "::imDMCommand output 1: %4x\n", output >> 16 );
#endif

// Make sure last asynchronous write to DIO is done
    do {
        ret = DO_AsyncCheck(DigitalBoard,&last_write_done,&data_written_cnt);
        if (ret != NoError)
        {
            printf ("ERROR ::imDMCommand() DO_AsyncCheck failed here %d\n", ret);
```

```
            return;
        }
    } while(!last_write_done)

#ifdef HAVE_HARDWARE
if (TEMP_ADLINK_FAILURE_RETRY( DO_ContWritePort (DigitalBoard, 0, &output, 1, 1,
ADLINK_WRITE_FREQ, ASYNCH_OP)) != NoError)
        {
            printf ("ERROR ::imDMCommand() TEMP_ADLINK_FAILURE_RETRY(DO_ContWritePort) f
ailed here 1 %d\n", ret);
            return;
        }
#endif // HAVE_HARDWARE

    output = command; // negative strobe low

#if DEBUG_BUFFERS
    printf( "::imDMCommand output 2: %4x\n", output >> 16);
#endif

// Make sure last asynchronous write to DIO is done
    do {
        ret = DO_AsyncCheck(DigitalBoard,&last_write_done,&data_written_cnt);
        if (ret != NoError)
        {
            printf ("ERROR ::imDMCommand() DO_AsyncCheck failed here %d\n", ret);
            return;
        }
    } while(!last_write_done)

#ifdef HAVE_HARDWARE
if (TEMP_ADLINK_FAILURE_RETRY( DO_ContWritePort (DigitalBoard, 0, &output, 1, 1,
ADLINK_WRITE_FREQ, ASYNCH_OP)) != NoError)
        {
            printf ("ERROR ::imDMCommand() TEMP_ADLINK_FAILURE_RETRY(DO_ContWritePort) f
ailed here 2 %d\n", ret);
            return;
        }
#endif // HAVE_HARDWARE

    output = command | INV_STROBE; // hold strobe high

#if DEBUG_BUFFERS
    printf( "::imDMCommand output 3: %4x\n", output >> 16);
#endif

// Make sure last asynchronous write to DIO is done
    do {
        ret = DO_AsyncCheck(DigitalBoard,&last_write_done,&data_written_cnt);
```



```
        if (ret != NoError)
        {
            printf ("ERROR ::imDMCommand() DO_AsyncCheck failed here %d\n", ret);
            return;
        }
    } while(!last_write_done)

#ifdef HAVE_HARDWARE
if (TEMP_ADLINK_FAILURE_RETRY( DO_ContWritePort (DigitalBoard, 0, &output, 1, 1,
ADLINK_WRITE_FREQ, ASYNCH_OP)) != NoError)
        {
            printf ("ERROR ::imDMCommand() TEMP_ADLINK_FAILURE_RETRY(DO_ContWritePort) f
ailed here 3 %d\n", ret);
            return;
        }
#endif  // HAVE_HARDWARE

}

//============ MASTER BUFFER =============

/**
 * Utility to create a bit string from an unsigned 32bit value.
 */
void makeBitString( unsigned int bits, char *string )
{
    for( int i=0; i<32; i++ )
    {
        int mask = 1<<i;
        if( (bits & mask) > 0 )
            string[31-i] = '1';
        else
            string[31-i] = '0';
    }
    string[32] = '\0';
}
```

```
function rtIntegrator()
% Matlab has a terrible tendency to copy all of its parameters in the compiled version.
% Therefore we use globals to streamline the passing of data with no copy overhead.

global kIterations;
global centroids;
global dmValidActuator;
global reconstructor;
global gain;
global dmCoefficients;
global residualDMCoefficients;
global DMCoef
global DMCoef0
%global ttCoef
%global ttCoef0

residualDMCoefficients(:,kIterations)  = reconstructor*centroids(:,kIterations);

% CLOSED-LOOP mode.  Uncomment the following line to enable (don't
% forget to disable open-loop mode at the same time)
% dmCoefficients = dmCoefficients + gain.*residualDMCoefficients(:,kIterations);

% OPEN-LOOP mode (no memory of dmCoefficients from last iteration).
% Uncomment the following line to enable (don't forget to disable closed-loop mode
% at the same time)
dmCoefficients = gain.*residualDMCoefficients(:,kIterations);

DMCoef(dmValidActuator) = DMCoef0(dmValidActuator) + dmCoefficients;
DMCoef = reshape(DMCoef,8,8);
DMCoef = DMCoef(:,end:-1:1); % fliplr(DMCoef)

DMCoef(dmValidActuator) = convertToDigit(...
      DMCoef0(dmValidActuator) + dmCoefficients,...
      6144,10240,-0.5,0.5,[-0.5,0.5] );

  % DMCoef = reshape(DMCoef,8,8)';
  DMCoef = reshape(DMCoef,8,8);
  DMCoef = DMCoef(:,end:-1:1); % fliplr(DMCoef)

%%%% Another fliplr??? check with Dave

 DMCoef = DMCoef(:,end:-1:1); % fliplr(DMCoef)


function digit = convertToDigit(a,minDigit,maxDigit,commandMin,commandMax,clip)
% converToDigit Arbitrary command unit to digital value converter
% digit = convertToDigit(command,minCommand,maxCommand,minDigit,maxDigit)
% Convert the command to the corresponding mirror digital value command
% based on the minimum and maximum of the command and of the digital values
```



```
% $Id$
%global clip_count

index = find(a<clip(1));
if ~isempty(index)
    warning('Clipping!');
    a(index) = clip(1);
    %clip_count = clip_count + 1;
end
index = find(a>clip(2));
if ~isempty(index)
    warning('Clipping!');
    a(index) = clip(2);
    %clip_count = clip_count + 1;
end
a(a>clip(2)) = clip(2);

%digit = round( (maxDigit-minDigit).*...
%    (a-commandMin)./(commandMax-commandMin) + minDigit );

% MDF - "a" is negated for reverse polarity DE64 DM amplifier (voltage commands come out
negated)
  digit = round( (maxDigit-minDigit).*...
      (-a-commandMin)./(commandMax-commandMin) + minDigit );
```

```
HOST        = linux
ADLINK      = /data/installers/pci-dask_413
EDTDIR      = /opt/EDTpdv
INCLDIR_EDT = $(EDTDIR)
LIBDIR_EDT  = $(EDTDIR)

INCLDIR     = $(INCLDIR_EDT)

LIBROOT     = /home/aoteam/repositories/matlab
MATLABROOT  = /data/matlab7

CFLAGS_EDT  = -O3 -mtune=athlon -DLINUX -I. -I$(INCLDIR) -I$(ADLINK)/include
-I$(matlab)/extern/include

CFLAGS  = $(CFLAGS_EDT)

LDFLAGS_EDT = -L. -L$(ADLINK)/lib -L$(EDTDIR) -L$(MATLABROOT)/bin/glnx86
LDTAIL_EDT  = -lpdv -lpthread -lcurses

LDFLAGS     = $(LDFLAGS_EDT)
LDTAIL      = $(LDTAIL_EDT)

LD_LIBRARY_PATH := $(LD_LIBRARY_PATH):/data/matlab6.5/bin/glnx86:/data/matlab6.
5/sys/os/glnx86:$(LIBDIR_EDT):/$(ADLINK):.

matlab=/data/matlab6.5
mcc = $(matlab)/bin/mcc

pipeline_edt: libloadAOVar.so libsaveAOVar.so libintegrator.so libshackProcessing.so
libdmshapegen.so
    g++ -o rtpipeline $(CFLAGS) $(LDFLAGS)  -lloadAOVar -lsaveAOVar -lpci_dask
-lintegrator -lshackProcessing -ldmshapegen rtpipeline.cpp processStats.cpp
ErrorReporter.cpp MirrorDriver.cpp WFS_1M150_EDT.cpp centroidCapture1M150.cpp $(LDTAIL)

pipeline_edt_convolve: libloadAOVar.so libsaveAOVar.so libintegrator.so
libshackProcessing.so libdmshapegen.so
    g++ -o rtpipeline $(CFLAGS) $(LDFLAGS)  -lloadAOVar -lsaveAOVar -lpci_dask
-lintegrator -lshackProcessing -lconvolve_lenslet rtpipeline.cpp processStats.cpp
ErrorReporter.cpp MirrorDriver.cpp WFS_1M150_EDT.cpp centroidCapture1M150_convolve.cpp
$(LDTAIL)

pipeline_edt_noDM: libloadAOVar.so libsaveAOVar.so libintegrator.so libshackProcessing.
so
    g++ -o rtpipeline $(CFLAGS) $(LDFLAGS)  -lloadAOVar -lsaveAOVar -lintegrator
-lshackProcessing rtpipeline.cpp processStats.cpp ErrorReporter.cpp  WFS_1M150_EDT.cpp
centroidCapture1M150.cpp $(LDTAIL)

dm_utilities:
    g++ -o dm_util $(CFLAGS) $(LDFLAGS) -lpci_dask MirrorDriver.cpp dm_utilities.cpp
```



```
$(LDTAIL)

libintegrator.so: rtIntegrator.m
    $(mcc) -t -W main -L C -h -T link:lib -o libintegrator rtIntegrator.m

libloadAOVar.so: loadAOVar.m
    $(mcc) -t -W main -L C -h -T link:lib -o libloadAOVar loadAOVar.m

libsaveAOVar.so: saveAOVar.m
    $(mcc) -t -W main -L C -h -T link:lib -o libsaveAOVar saveAOVar.m

libshackProcessing.so: shackProcessing.m
    $(mcc) -t -W main -L C -h -T link:lib -o libshackProcessing -I
$(matlab)/toolbox/matlab/strfun/ -I $(matlab)/toolbox/matlab/elmat/ -I
$(matlab)/toolbox/matlab/elfun/ -I $(matlab)/toolbox/matlab/datatypes/ shackProcessing.m

libdmshapegen.so: dm_shape_gen.m
    $(mcc) -t -W main -L C -h -T link:lib -o libdmshapegen dm_shape_gen.m

libconvolve_lenslet.so: convolve_lenslet.m
    $(mcc) -t -W main -L C -h -T link:lib -o libconvolve_lenslet convolve_lenslet.m

wfsCAD1.firmware: wfsFirmware/wfsFirmwareCAD1.o
    cp wfsFirmware/wfsCAD1.firmware .

wfsFirmware/wfsFirmwareCAD1.o:
    cd wfsFirmware; wine nmake

wfs1M150.firmware: wfsFirmware/wfsFirmware1M150.o
    cp wfsFirmware/wfs1M150.firmware .

wfsFirmware/wfsFirmware1M150.o:
    cd wfsFirmware; wine nmake

clean:
    rm -f rtpipeline *.so *~ *Var.c *Var.h *_main.c meshgrid.c meshgrid.h squeeze.c sque
eze.h WFS_CAD1.o WFS_1M150_EDT.o MirrorDriver.o centroidCapture1M150.o
```

Appendix E:
UVic AO Library Sample Code

```
% Create WFSA
cd /home/fischerm/repositories/linuxcode/matlabFramecaptureEDT-1M150/
WFSA =  shackHartmann(7,0.5,16)
WFSA = set(WFSA,'integrationTime',1.5,'driver',@(x)framecapture(x));
load valid36
WFSA = set(WFSA,'validLenslet',valid36);
WFSA = grab(WFSA);
WFSA = data(WFSA,piston(7*16));
WFSA = set(WFSA,'pixelBottomLeft',[20,1]);
WFSA = set(WFSA,'threshold',70);


% Get WFSA Camera Bias (pixelBiasMap)
biasA = get(set(WFSA,'nFrame',20),'frame');
WFSA     = set(WFSA,'pixelBiasMap',biasA);
ref_tmp = zeros(74,1)+8.;
WFSA = set(WFSA,'reference',ref_tmp);
load('biasA.mat')
WFSA = set(WFSA,'pixelBiasMap',biasA);
WFSA = imagesc(data(WFSA,'frame'),'CCD');


% Start Continuous WFSA Spot Diagram
startFun = 'warning(''off'',''shackHartmannData:centroids'');';
stopFun = 'warning(''on'',''shackHartmannData:centroids'');';
itACmd = 'WFSA = imagesc(data(WFSA,''frame''),''CCD'');';
itA = timer('TimerFcn',itACmd,'Period',1/4,'StartFcn',startFun,'StopFcn',stopFun,...
'ExecutionMode','fixedSpacing','Tag','volt timers');
start(itA)
stop(itA)


% Get WFSA Reference Centroids (refMeasurements)
WFSA = set( data( set(WFSA,'nFrame',100) ,'frame','reference') , 'nFrame',1);
refA=get(WFSA,'reference')
refA=refA/.485
WFSA = set(WFSA,'reference',refA);
% or load them from file:
load('refA.mat');
WFSA = set(WFSA,'reference',refA);


% Start Continuous WFSA Slope Diagram
SLCmdA = 'WFSA = imagesc(data(WFSA,''frame''),''data'');';
SLA = timer('TimerFcn',SLCmdA,'Period',1/4,'StartFcn',startFun,'StopFcn',stopFun,...
'ExecutionMode','fixedSpacing','Tag','volt timers');
start(SLA)
stop(SLA)
```



```
% Create Simple DM
nInfluenceFunction      = 8;
nWooferPupilPx      = get(WFSA,'nPx')*get(WFSA,'nLenslet');
[x,y,r,o]           = cartAndPol(nWooferPupilPx);
mechanicalCoupling  = 0.65;
nearestNeighbor     = 2./(nInfluenceFunction-1);
gif         = gaussInfluenceFunction(mechanicalCoupling,nearestNeighbor);
influenceFunctionMatrix = eye(64);
voltAlpao = zonalDeformableMirror(nInfluenceFunction,gif,x,y);
voltAlpao = set(voltAlpao,...
    'validActuator',wooferMask,...
    'mirrorLabel','ALPAO',...
    'commandMin',-0.5,...
    'commandMax',0.5,...
    'commandDefault',comDef6,...
    'realMirror',true);


% Flatten the DM
[nCoef,nWFSData] = size(command);
wfsData = zeros(nWFSData);
wfsData = get(WFSA,'data');
DMcoef = commandA*wfsData;
voltAlpao = set(voltAlpao,'coefficients',DMcoef);


% Poke a single actuator
voltAlpao = set(voltAlpao,'allCoefficients',0);
voltAlpao = set(voltAlpao,'coefficients',[4,4,-0.1])
voltAlpao = set(voltAlpao,'allCoefficients',0);


% Set up Closed Loop Operation (only applicable if WFSA is in appropriate position)
closedLoop = integrator(0.25,get(WFSA,'integrationTime'),1);
[closedLoop, voltAlpao] = run(closedLoop, 100, WFSA, voltAlpao, commandAV);
```