FOAM

A Modular Adaptive Optics Framework

Tim van Werkhoven
supervised by
Prof. Dr. C.U. Keller

Universiteit Utrecht
Department of Physics and Astronomy
Sterrekundig Instituut Utrecht

Master thesis by Tim van Werkhoven (t.i.m.vanwerkhoven@xs4all.nl), 2007–2008 supervised by Prof. Dr. C.U. Keller.

Abstract

Astronomical observations done from the ground are hampered in many ways by the Earth's atmosphere. Several techniques exist to remedy these problems, one of which is adaptive optics. The control mechanism behind adaptive optics has long been specialised hardware with little flexibility. This makes these systems highly non-portable and difficult to modify. Generic computers are now getting to a point where their performance is good enough to take over from these specialised pieces of hardware.

FOAM, backwards for Modular Adaptive Optics Framework, is a piece of software that aims to provide a platform which can be used on a wide variety of adaptive optics systems. To achieve this, the setup is modular such that pieces of code can be re-used easily. Furthermore, FOAM is open source and can be used by anyone, it is well-documented such that collaboration is possible.

In its current state, FOAM provides a modest library of modules which allow it to analyse and process Shack-Hartmann wavefront sensor output, and drive wavefront correctors with this. Additionally, it provides modules to simulate an adaptive optics system. The program has been implemented and tested on the McMath-Pierce telescope and is able to provide real-time tip-tilt correction.

Contents

1st Chapter

INTRODUCTION

When observing the skies in visible light there are two principal methods which can be used. The first is observing from the surface of the Earth; the second entails sending a telescope-equipped satellite into space. Both have their benefits and drawbacks.

Space-based observations are relatively expensive, since a telescope needs to be sent into an orbit around our planet. On the other hand, once a telescope is in orbit, it can observe celestial objects without any interference from the Earth's atmosphere, which greatly increases the quality of the images.

Ground-based observations on the other hand are cheaper to build, since a trip to space is not necessary in this case. Another advantage is that bigger telescopes can be built on the ground, since the size of satellites is limited by the rockets and space shuttles used. There is however a major drawback to this method, which is the interference of the Earth's atmosphere.

1.1 SEEING

When looking at stars with the naked eye during a clear night, they seem to twinkle: their intensity appears to vary slightly. This twinkling or scintillation is not intrinsic to the stars themselves, but is caused by the atmosphere which distorts the light in various ways. This effect is called seeing.

Seeing is image degradation caused by the fact that the atmosphere is not homogeneous, but consists of different pockets of air with varying temperature. Since the index of refraction (n) is a function of temperature, this means that light is refracted in the atmosphere, so that one line-of-sight does not have the same optical path length as another. This path difference results in the rippling of a flat wavefront after it passes through the atmosphere. Figure **??** schematically shows how this works.

This rippled wavefront results in different forms of image degradation. When observing point sources, it is possible that the source is imaged more than once, resulting in speckle patterns (**?**). This can happen because light following different paths through the atmosphere might form separate images of the same

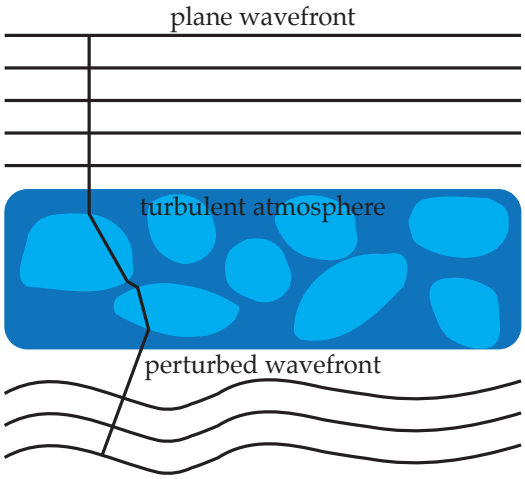


Figure 1.1: A simple diagram showing the aspects involved with seeing. The plane wavefront enters at the top, traverses through the inhomogeneous atmosphere, and comes out rippled. The different shades in the turbulent atmosphere denote differences in the index of refraction, n. A sample light ray has been drawn to show how the refraction of these relate to the wavefront perturbation.

object with varying intensity. On top of this, the various speckles may overlap and form interference patterns. Because the atmosphere's inhomogeneities change rapidly, this speckle pattern also varies with time. To observe speckles, one therefore needs to use a short exposure time not to blur them out.

If one uses a larger exposure time, the different patterns will overlap and cause overall image degradation: the image becomes more fuzzy and the resolution is degraded.

Another effect associated with seeing is that the speckles move around at random. Because the light is refracted differently by the atmosphere as time progresses, the speckles can appear at a different location each time.

All the above holds for point sources, but also applies to extended sources as the sun or planets. The effects are slightly more complicated there as the individual speckles are not distinguishable anymore, but overlap with neighbouring speckles. Instead, the surface of the object appears to shear, stretch and warp from frame to frame. Again, if the exposure is long enough, this deformation is averaged out and the result is an overall more blurry image.

Besides these distortions caused by the atmosphere there are also various other problems one has to face. These include scintillation, light pollution, spectral lines caused by the Earth's atmosphere etc. These phenomena are not unimportant, but are not part of image correction and therefore will not be discussed here.

1.2 CORRECTING SEEING

A solution to the previously mentioned problem could be to take short exposure images. Since this freezes the speckles, it prevents large distortions. Using clever techniques (**?**), the original image can be partially restored from a burst of these images. This method is computationally expensive and can only be done after the images are taken.

Another method to restore the images is adaptive optics. This technique constantly analyses the distortion at high frequency (in the order of a few kilohertz) and uses some correcting actuator, usually a deformable mirror, to correct the distorted wavefront in real time. The way this is done is that a sensor looks at the distortion of the wavefront, and feeds back this information to the mirror. This hopefully corrects the distortion, so that the original wavefront is restored. Of course in practice, this correction is never perfect. The principle of this method is shown in Fig. **??**.

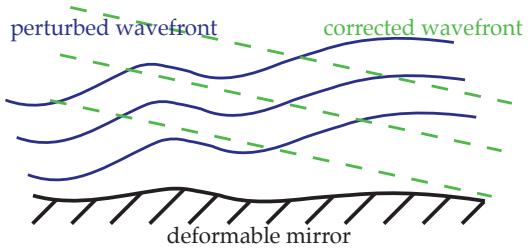


Figure 1.2: A schematic depiction of the correction mechanism behind adaptive optics. The incoming wavefront is disturbed by the atmosphere and the mirror is deformed in such a way that the distortion is cancelled.

1.3 NEED FOR FOAM

Adaptive optics (AO) has been in use on numerous telescopes since the 1990's when this technique emerged. Because a few years ago, computers were too slow to do real-time analysis and correction of the wavefront, such adaptive optics systems used specialist hardware specifically designed for this task. This made them highly non-portable, as everything was hardwired. Any modification to the setup required a large amount of work. Also, these systems were expensive to build, since designing and programming such hardware requires a lot of resources. Demand for a universal AO system is therefore high.

Since computers are much faster now and can easily perform the previously mentioned tasks required in real-time, the road is paved for open and universal AO software based on general purpose machines. Several attempts have already been made in the past, for example on the McMath-Pierce solar telescope (**??**), but although these systems use generic x86 architecture machines, they are usually built in a monolithic fashion and have their drivers entangled in the rest of the program. This therefore hampers reusability of the system.

The goal of this research is to provide an adaptive optics software implementation that is open, modular and scalable and is intended to be (re-)used over a wide variety of platforms and systems. This thesis presents the results of this research: FOAM, a piece of software aiming to provide such an implementation. The structure of this writing is as follows: Chapter **??** provides a theoretical background on seeing caused by a turbulent atmosphere. Chapter **??** discusses adaptive optics systems in general and explores some of the problems and caveats associated with this technique. Chapter **??** is devoted to the design of FOAM, while Chapter **??** treats a simulation implementation of FOAM and evaluating it by placing this research in a broader context. Chapter **??** provides some initial results obtained with the software on the McMath-Pierce telescope.

2nd Chapter

2 QUANTIFYING SEEING

The index of refraction (n) of air is not constant throughout the atmosphere. The reason for this is the temperature and pressure dependence of n. Since the atmosphere is not homogeneous, the temperature and pressure vary throughout the atmosphere, and thus also the index of refraction.

The cause for the inhomogeneity of the atmosphere are due to the weather, causing temperature- and pressure gradients. Specifically, things like wind shear, differential heating of the atmosphere, turbulence caused by wind passing along mountains, etc. give rise to these gradients. Because these phenomena change with time, the inhomogeneities do not only vary spatially, but also temporally.

When the light from a celestial object arrives at the earth just above the atmosphere, it can be considered a plane wave, meaning that the loci of equal phase lie on a plane. As this wavefront encounters the turbulent atmosphere, it is perturbed and becomes rippled. The rippling causes degradation in the quality of the image. This effect is called 'seeing'. To correct this distortion with adaptive optics, we first need to characterise these phenomena.

In this chapter I explain the nature of this distortion, starting with a brief analysis of the atmospheric turbulence, followed by the effect it has on the light passing through the atmosphere, and concluding with some requirements for adaptive optics systems. Details on the analysis can be found in both **?** and **?**.

2.1 KOLMOGOROV TURBULENCE

? proposed a hydrodynamical model for turbulence by assuming energy is inserted at low frequencies on large scales, characterised by an 'outer scale length' L_0. This energy is then transported to smaller scales in a cascading way, where the homogeneous clouds break up in smaller parts. Finally when this energy is transported to small enough scales, it is converted to heat by dissipation. This small scale is characterised by an 'inner scale length' l_0. Although originally not intended for the modelling of atmospheric turbulence, **?** found this model applicable to our atmosphere most of the time. In the rest of

this chapter, I assume this model to be valid. Typical values for l_0 range from a few millimetre to a few centimetre, while the value of L_0 is less certain and is believed to be several tens of meters.

When studying the wavefront perturbation caused by the variation of n throughout the atmosphere, the absolute value of n is of little importance, since it is the variation in n that causes the perturbation. The Kolmogorov model states that

$$D_n(\boldsymbol{\rho}) = \langle |n(\boldsymbol{r}) - n(\boldsymbol{r} + \boldsymbol{\rho})|^2 \rangle = C_n^2(z)\rho^{2/3}, \tag{2.1}$$

with $\boldsymbol{\rho}$ the distance between two points $\boldsymbol{r_1}$ and $\boldsymbol{r_2}$ and $\rho = |\boldsymbol{\rho}|$. This relation holds as long as ρ ranges from l_0 to L_0 and is called the index structure function. All other important quantities in atmospheric turbulence scale with the same two-thirds power law. In Eq. (**??**), $C_n^2(z)$ is the so-called refractive index structure constant, and mainly varies with height.

$C_n^2(z)$ characterises the turbulence in the atmosphere and is a measure for the variation of n. The total distortion of a wavefront at ground level is a function of the integral of this quantity along the line of sight. Typical values lie around 10^{15} m$^{-2/3}$ (**?**, pp. 10), and there are several different models describing this quantity. Two of these are plotted in Fig. **??**.

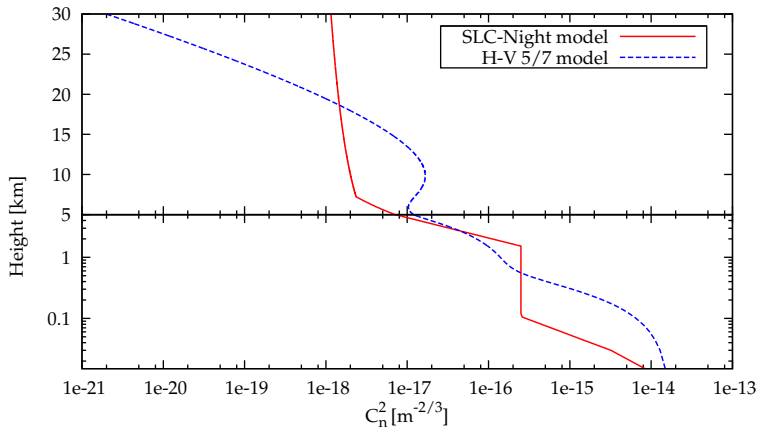


Figure 2.1: The $C_n^2(z)$ profile for two different models. The SLC-Night model is a stratified version, while the H-V 5/7 model is more continuous. This graph illustrates the general shape and variation in such profiles.

Besides the spatial variance of n, the temporal variance is just as important for adaptive optics systems. Given Eq. (**??**), this can be derived assuming that variations in n live longer than the time it takes for such in inhomogeneities to cross over a telescope driven by wind, so that

$$n(\boldsymbol{r}, t + \tau) = n(\boldsymbol{r} - \boldsymbol{v}\tau, t), \tag{2.2}$$

with $\boldsymbol{v}$ the wind speed. Continuing this reasoning, we find a temporal variance

for n of

$$\begin{aligned} D_n(\boldsymbol{\rho}) &= \langle |n(\boldsymbol{r},t) - n(\boldsymbol{r},t+\tau)|^2 \rangle \\ &= \langle |n(\boldsymbol{r},t) - n(\boldsymbol{r}-\boldsymbol{v}\tau,t)|^2 \rangle \\ &= C_n^2(z)|\boldsymbol{v}\tau|^{2/3}, \end{aligned} \tag{2.3}$$

and conclude that the temporal variance in n can be found by replacing $\boldsymbol{\rho}$ with $|\boldsymbol{v}\tau|$ in Eq. (**??**).

2.2 WAVEFRONT DISTORTION

Now that we have established the temporal and spatial variation of the index of refraction in a turbulent atmosphere following a Kolmogorov spectrum, we can investigate its effect on the wavefront perturbation at the telescope. Recall that the distortion is caused by varying optical path lengths throughout the atmosphere due to the inhomogeneity of n. Consider the optical path length given by

$$\delta = \int n(z)\,\mathrm{d}z, \tag{2.4}$$

integrating n along the line of sight. Although n is a function of temperature and pressure, it is more or less wavelength independent over the relevant range of the spectrum (visible and near infrared). The optical path length is therefore also wavelength independent. The phase φ of the wavefront however does depend on the wavelength:

$$\varphi = k\int n(z)\,\mathrm{d}z, \tag{2.5}$$

with k the wave number, $2\pi/\lambda$.

Note that although the optical path length is millimetrelength independent, the more important phase difference is not. If the path length of radio waves are offset by a few millimeter, the wave is still plane because of the enormous wavelength of radio waves. If the same distortion is applied to visible light however, a few millimetres equals thousands of wavelengths, hence the wave can no longer be considered plane, and the phase will vary randomly over the telescope aperture.

However, when correcting this distortion, it is the difference in path-length that is corrected. This means that the same correction can be applied to all wavelengths simultaneously, because of the wavelength independence of Eq. (**??**). If some part of the wavefront is offset by a distance Δx with respect to another part, a correction to this offset works for all wavelengths.

As we are not interested in absolute phases, we again look at the variation of the phase in a plane at the Earth's surface using the structure function. $\boldsymbol{x}$ and $\boldsymbol{\xi}$ are 2-d vector in this plane

$$D_\varphi(\boldsymbol{\xi}) = \langle |\varphi(\boldsymbol{x}) - \varphi(\boldsymbol{x}+\boldsymbol{\xi})|^2 \rangle. \tag{2.6}$$

Using Eq. (**??**), we can find an expression for D_φ in terms of $C_n^2(z)$ which leads to

$$D_\varphi(\boldsymbol{\xi}) = 2.91\, k^2 \int C_n^2(z)\, \mathrm{d}z\, \xi^{5/3}. \tag{2.7}$$

Since $C_n^2(z)$ is only dependent on height, the above integral along the line of sight can be rewritten to an integral going straight up:

$$D_\varphi(\boldsymbol{\xi}) = 2.91\, k^2 \sec\zeta \int C_n^2(h)\, \mathrm{d}h\, \xi^{5/3}, \tag{2.8}$$

with ζ the angle between the line-of-sight and zenith. Usually, the above expression is further rewritten as

$$D_\varphi(\boldsymbol{\xi}) = 6.88\, (\xi/r_0)^{5/3}, \tag{2.9}$$

with

$$r_0 = \left(0.423\, k^2 \sec\zeta \int C_n^2(h)\, \mathrm{d}h\right)^{3/5}, \tag{2.10}$$

the Fried parameter (**?**). The Fried parameter is a useful quantity for characterising the seeing quality, as will become clear in the next section.

Another quantity frequently used in describing atmospheric turbulence is the isoplanatic angle. This quantity characterises an angle across the sky for which the seeing conditions can be considered similar, and is frequently used to indicate the maximum field of view for which disturbance is corrected. Considering a turbulent atmospheric layer at a distance of $h/\cos(\zeta)$, we characterise the mean square error in the wavefront by replacing $\boldsymbol{\xi}$ by $\theta\, h \sec(\zeta)$ in Eq. (**??**). While this would be correct for a single turbulent layer, in reality there are several of such layers, so that h should be replaced by a weighed average $\bar{h}$, resulting in

$$\sigma^2{}_{\text{aniso}} = 6.88\, (\theta\, \bar{h}\, \sec(\zeta)/r_0)^{5/3}. \tag{2.11}$$

Equating this anisoplanicity rms error to 1 radian, we find that the isoplanatic angle is given by

$$\theta_0 = 0.314\, \frac{r_0}{\sec(\zeta)\, \bar{h}}, \tag{2.12}$$

so that the isoplanatic angle has the same wavelength dependence as r_0, and that it is much more dependent on the zenith angle ζ.

As for the temporal behaviour of seeing, recall that Eq. (**??**) gave the temporal variance of the index structure function. Using this expression, and performing the identical analysis as done previously, the variance of the phase difference between time t and $t + \tau$ is found to be

$$D_\varphi(\boldsymbol{\xi}) = 6.88\, (\bar{\boldsymbol{v}}\, \tau/r_0)^{5/3}, \tag{2.13}$$

with $\bar{\boldsymbol{v}}$ the average wind speed along the line of sight. This quantity can be used to calculate the mean square phase error if a correction to the wavefront is applied after a certain delay τ, i.e.

$$\sigma^2{}_{\text{time}} = 6.88\, (\bar{\boldsymbol{v}}\, \tau/r_0)^{5/3}. \tag{2.14}$$

j	Zernike mode j, Z_j	A_j (rad^2)	Optical aberration
1	1	1.0299	Piston
2	$2\,\rho\,\sin(\theta)$	0.582	Tip/tilt
3	$2\,\rho\,\cos(\theta)$	0.134	Tip/tilt
4	$\sqrt{3}\,(2\rho^2-1)$	0.111	Defocus
5	$\sqrt{6}\,(\rho^2\sin(2\theta))$	0.0880	Astigmatism
6	$\sqrt{6}\,(\rho^2\cos(2\theta))$	0.0648	Astigmatism
7	$\sqrt{8}\,(3\rho^3-2\rho)\sin(\theta)$	0.0587	Coma and Tilt
8	$\sqrt{8}\,(3\rho^3-2\rho)\cos(\theta)$	0.0525	Coma and Tilt

Table 2.1: The first few Zernike polynomials. A_j is used as a constant to determine the residual rms wavefront error after correcting j Zernike modes, see Eq. (**??**). The optical aberration associated with mode j is listed alongside the wavefront error. Taken from **?**.

Allowing a maximum rms error of 1 radian, Eq. (**??**) can be rewritten to

$$\tau_0 = 0.314\,\frac{r_0}{\overline{\boldsymbol{v}}}, \tag{2.15}$$

so that τ_0 gives a measure of the maximum time-delay allowed before applying a correction to the distorted wavefront. This delay is called the Greenwood time delay, and the reciprocal value is called the Greenwood frequency, f_{G}. Typical values for this frequency are around several hundred Hertz in the visible.

2.3 WAVEFRONT DECOMPOSITION

Decomposing the wavefront can help to identify what kind of distortions we are dealing with. One widely used example is the set of Zernike polynomials, $Z_j(\rho,\theta)$, which is an infinite series of orthogonal polynomials over a unit circle. These are set up in such a way that the first few polynomials directly relate to optical phenomena like tip-tilt, defocus, etc. The first eight of these are given in Table **??** with the associated optical error. Such a wavefront decomposition for a circular aperture with radius R is given by

$$\varphi(\rho R,\theta) = \sum_j a_j Z_j(\rho,\theta), \tag{2.16}$$

with a_j some coefficient. For details on Zernike modes I refer the reader to the literature (again, see **?** or **?**).

Now consider that we correct exactly N Zernike modes of the wavefront distortion. Taking the first N Zernike modes gives us

$$\varphi_{\mathrm{c}}(\rho R,\theta) = \sum_{j=1}^{N} a_j Z_j(\rho,\theta). \tag{2.17}$$

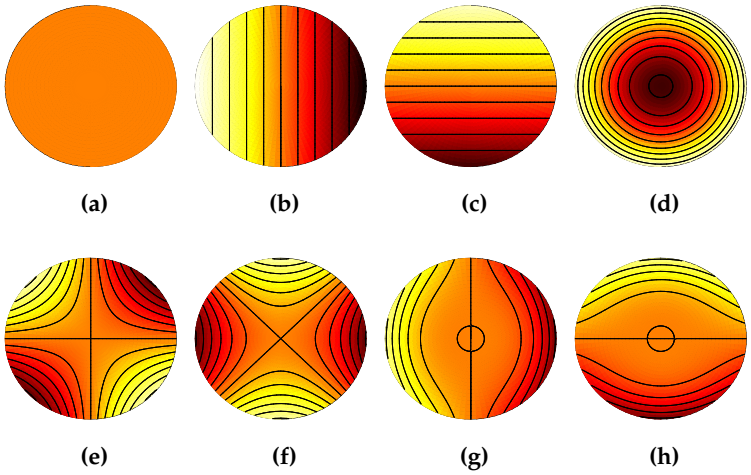


Figure 2.2: Visualisations of the first 8 Zernike modes. (a) piston, (b) and (c) tip-tilt, (d) defocus, (e) and (f) astigmatism, (g) and (h) coma. See Table **??** for details.

If we apply such a correction to a distorted wavefront, and assuming Eq. (**??**) holds, **?** and **?** found that the residual mean square phase distortion of the wavefront is given by

$$\begin{aligned}\sigma^2_{\varphi,N} &= \int_{\text{Aperture}} \langle|\varphi(\rho R,\theta) - \varphi_{\text{c}}(\rho R,\theta)|^2\rangle \\ &= A_j \left(\frac{D}{r_0}\right)^{5/3},\end{aligned} \tag{2.18}$$

with D the aperture diameter and r_0 the Fried parameter from Eq. (**??**). The first few values of A_j are listed in Table **??**. Thus, decomposing the wavefront helps us to predict the improvement gained by a certain optical correction. **?** found that for $N \gtrsim 10$ the residual phase error is given by

$$\sigma^2_{\varphi,N} = 0.2944\, N^{\sqrt{3}/2} \left(\frac{D}{r_0}\right)^{5/3}. \tag{2.19}$$

Looking at the residual errors listed in Table **??**, note that without correction, the residual error is given by

$$\begin{aligned}\sigma^2_{\varphi,0} &= A_j \left(\frac{D}{r_0}\right)^{5/3} \\ &= 1.0299 \left(\frac{D}{r_0}\right)^{5/3},\end{aligned} \tag{2.20}$$

so that if we use a telescope aperture of size r_0, the rms phase distortion is about 1 radian.

Besides the Zernike expansion there are other methods to decompose the wavefront error. One decomposition suitable for this is the Karhunen-Loève

expansion, which is similar to the Zernike decomposition in some respects, but is optimised for a given turbulence and is numerical instead of analytical. In practice, the Zernike expansion is used more often though, due to its analytical nature.

2.4 EFFECT OF SEEING ON IMAGE QUALITY

Now that we have established some statistics on the wavefront error and the phase variance, it is time to look at the influence this has on image quality. Since science is done with images and not with wavefronts, it is interesting to so see what effect the distortion has on the image quality.

Before we can analyse this at all, we need to define it first. A widely used quantity for determining image quality is the so-called Strehl ratio, the ratio between maximum intensity of a degraded image and the maximum intensity of a diffraction limited image. For large errors, it is defined as

$$S \approx \exp\left(-\sigma_\varphi^2\right), \tag{2.21}$$

with σ_φ^2 the rms wavefront error.

Recall that for a telescope with diameter $D = r_0$ the rms wavefront error is about 1 radian, it follows from Eq. (**??**) that the Strehl ratio is $\exp(-1) \approx 0.37$ in such a case. A rms wavefront phase variance of about 1 radian is considered a 'good' image, although this limit is somewhat arbitrary. The resolution that can be achieved in such a case is about r_0/λ. It can be shown that the image quality quickly degrades further if the telescope diameter is increased beyond r_0 so that this will not result in a higher resolution. Without correcting the wavefront error, r_0 determines the maximum useful telescope diameter for long exposure imaging.

From Table **??**, note that the first Zernike mode is a constant wavefront offset and thus does not influence the image quality. The second and third Zernike mode are the tip- and tilt modes which only result in an image displacement and not in overall image degradation *if the exposure is shorter than* τ_0. If the exposure is too long, the various displacements are stacked resulting in a blurry image. Therefor, when using short exposures we can increase the telescope diameter to $(1/0.134)^{3/5} = 3.3\, r_0$ before the rms wavefront error increases beyond 1 radian. This shows that using short exposure imaging theoretically can be a simple solution to increase the image resolution by a factor 3.3.

Considering that r_0 typically ranges from 5 – 15 cm in the visible, this still strongly limits the maximum resolution obtained by telescopes. To improve this, we either need space-based telescopes or some method to correct the atmospheric distortion. One method to correct such distortions is adaptive optics, which is discussed in the next chapter.

3rd Chapter

ADAPTIVE OPTICS

Adaptive optics is a mechanism to correct distorted wavefronts in real-time. The incoming distorted wavefront is fed to a wavefront sensor (WFS) which can detect the aberrations. This information is sent to a wavefront corrector (WFC) which is an actuator that corrects the distortion. The separate components are linked together using some control hardware, which can be anything from a piece of special purpose hardware to a general purpose PC. Such a system is depicted in Fig. **??**. This chapter will deal with the several aspects of an adaptive optics system one by one, beginning with the wavefront sensor in the next section.

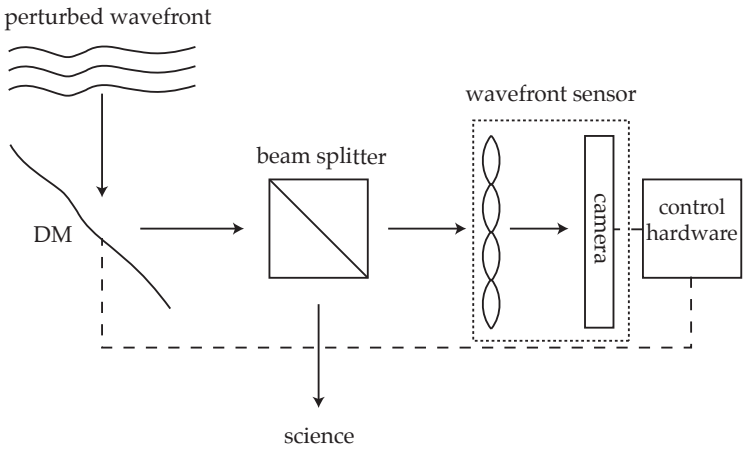


Figure 3.1: A typical adaptive optics system using one deformable mirror (DM) as wavefront corrector and one Shack-Hartmann wavefront sensor to detect the distortion. The control hardware takes input from the sensor and feeds this to the corrector, forming a closed-feedback loop.

3.1 WAVEFRONT SENSORS

Before correcting anything at all, the aberration to be corrected must be measured, which is done by so-called wavefront sensors. When imaging a wavefront, we measure the absolute amplitude squared of the electromagnetic waves. To detect the deformation, we need to know the phase, which at this time cannot be measured directly; there are no wavefront-phase sensors for visible light. Instead, the deformation is measured indirectly, which can be done in various ways.

The most commonly used wavefront sensor is the Shack-Hartmann (SH) wavefront sensor (**?**), which consists of an array of small lenses (or subapertures) which image different parts of the wavefront. These lenslets are typically smaller than a millimetre and typical arrays used are 8×8 and 16×16 arrays. The concept behind this sensor is that the lenslets image only a part of the wavefront, such that these only sense the local deformation. The local slope of the wavefront translates in a displacement of the image the lenslets form, such that this displacement is a direct measure for the local deformation. Using an array of these lenslets, the slope can be determined at an arbitrary number of points, such that the shape of the wavefront can be reconstructed. The principle of the Shack-Hartmann sensor is shown in Fig. **??**.

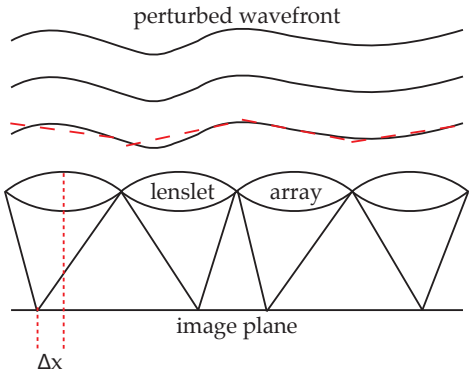


Figure 3.2: The principle of a Shack-Hartmann wavefront sensor. The wavefront arrives at the top, after it has been perturbed by for example the atmosphere. A 1 dimensional cut through the lenslet array is drawn, showing 4 individual lenslets. Each lenslet images a different part of the wavefront, thus sensing only the local slope. The slopes are accentuated by dashed lines, and the image displacement is annotated at the leftmost lenslet. If the wavefront would be plane, the image would be at the centre, instead it is offset by Δx. This direct relation between image displacement and wavefront slope allows reconstruction of the perturbation.

There are various ways to measure the position of the various images formed by the lenslet array. One method is calculating the centre of gravity (CoG) for each subaperture. This method works best on (more or less) circular symmetric sources (stars, planets) with a clear maximum, asymmetric objects that change over time like the solar surface are less suitable for this method. Another advantage of this method is that it is relatively insensitive to intensity variation

caused by scintillation or clouds.

Another method is using correlation between two images. First a reference image is taken, again by sending a flat wavefront through the AO system and recording this. After this calibration, subsequent subaperture images are cross correlated with the reference images at various displacements, and the displacement yielding the highest cross correlation is used as the subaperture offset. This method is more prone to intensity fluctuations as the reference image is static and is not automatically corrected for intensity variations.

3.2 WAVEFRONT CORRECTORS

After the wavefront distortion is measured by some sensor like the one discussed previously, it must be corrected. One type of wavefront corrector is a deformable mirror (DM), which typically has tens to hundreds of individual actuators deforming the mirror surface, so that it can correct many Zernike modes. Another corrector is a tip-tilt mirror, which is a flat mirror only capable of correcting the tip- and tilt modes.

3.2.1 TIP-TILT MIRROR

A tip-tilt mirror is usually the first corrector the distorted wavefront encounters. Tip-tilt mirrors are static mirrors which can rotate on two perpendicular axes. Using piezoelectric actuators these can function at frequencies up to several kilohertz. The advantage of correcting only tip- and tilt-modes is that such a mirror can correct these modes effectively. Since most of the distortion is located in these modes (see Table **??**), the mirror needs to move quite a distance, i.e. it has to have a large stroke. By correcting the first two modes with this mirror, a high-order corrector can focus on the remaining aberrations.

3.2.2 DEFORMABLE MIRROR

There are various types of deformable mirrors, which have gained complexity over the years. The early deformable mirrors consisted of mirror segments with piston-like actuators behind them, which would move the segments up and down. By adding two extra pistons to each mirror segment, these were able to correct tip- and tilt modes as well. Later continuous deformable mirrors were developed replacing their segmented predecessors, and these are still used today.

Deformable mirrors typically have less stroke than tip-tilt mirrors, and focus on correcting the higher order modes. If the low order modes also have to be corrected by this mirror, there is only little stroke left for the higher order modes, reducing the effectivity of the AO system.

3.3 SYSTEM CONTROL

Once the wavefront sensor delivers some input, this must be translated into actuator commands by some controlling unit. An example implementation of such a control mechanism using a Shack-Hartmann WFS with a DM as corrector is explained here.

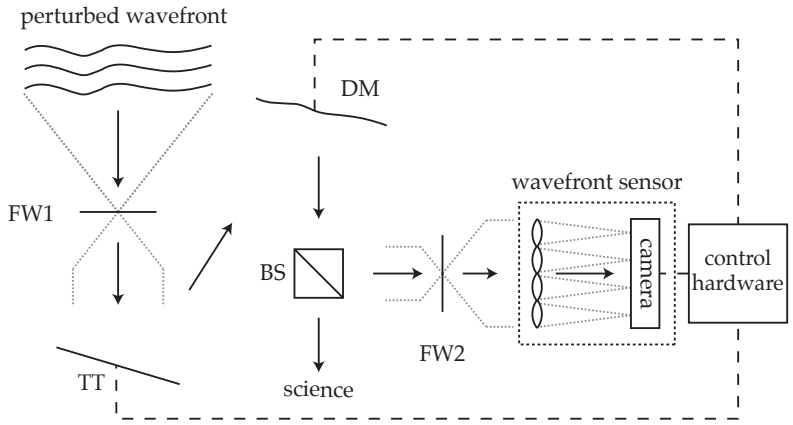


Figure 3.3: A more detailed description of an AO setup with one wavefront sensor, one tip-tilt mirror (TT) and one deformable mirror (DM). This scheme also shows the two filter wheels (FW) that can be used to place a pinhole in the beam. The beam splitter (BM) distributes the incoming light between the WFS and the science instruments. The dashed lines indicate the connection between the various sensors and actuators in the system. The dotted lines denote the beam of light, showing that the two filter wheels are located in the image plane and that the mirrors are located in the pupil plane, receiving a collimated beam. The lenslet array is also placed in the pupil plane, chopping up the aperture in several subapertures. The lenses (re-)imaging the beam are not drawn here.

3.3.1 WAVEFRONT SENSOR CALIBRATION

Before using the WFS output to drive the actuators, this sensor must first be calibrated. To do this, a plane wave is sent into the WFS which acts as a reference wavefront. This can be done by placing a pinhole just before the WFS, for example by using a filter wheel (FW2 in Fig. **??**). Such a pinhole generates a spherical wavefront which is converted into a flat wavefront after it passes a lens just before the wavefront sensor. The offsets measured using this flat reference wavefront with the WFS do not have to be zero, because of small alignment errors in the setup itself. These (non-zero) reference offsets thus define a flat wavefront.

The reference coordinates measured this way guarantee that a flat wavefront is also sent down into the science instruments. During normal operations, FW2 in Fig. **??** is open, sending anything that comes from the tip-tilt- and deformable mirror directly to the wavefront sensor. If the WFS therefore receives a flat wavefront, defined by the calibration discussed above, the same wavefront is sent to the science instruments. A positive side-effect of this approach is that

any optical aberration in front of the wavefront sensor is also corrected, because the sensor does not distinguish between different sources of perturbation.

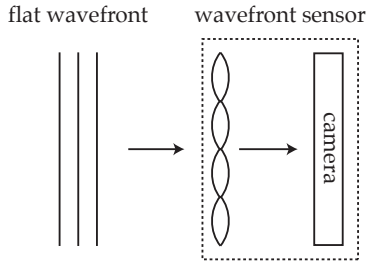


Figure 3.4: Calibration of the wavefront sensor is achieved by sending in a plane wave directly into the WFS, by placing a pinhole just before the sensor (FW2 in Fig. **??**). The sensor output measured this way defines a flat wavefront, and is used as a reference during AO operations.

3.3.2 WAVEFRONT CORRECTOR CALIBRATION

After the offsets for a flat wavefront are defined, the influence of the wavefront correctors must be measured. This is necessary because, if we wish to negate the wavefront distortion measured, we must know the response of the deformable mirror. Because these are not known a priori and can vary due to slight changes (alignment, temperature), calibration is used to measure the so-called influence matrix. This is done by placing a pinhole at the telescope focus (with FW1 in Fig. **??**) before the wavefront correctors so that a plane wave is sent through the complete AO system. This guarantees that we are indeed measuring the influence of the DM, and not an intrinsic change in the wavefront entering the AO system.

Once plane waves are running through the system, the actuators on the DM are driven back and forth one by one. While driving these actuators, the difference between the offsets measured by the WFS for the two actuator positions are stored. This gives the influence for each actuator on the offsets measured in each subaperture. Once this is done for each actuator, the result is a matrix called the influence matrix, which is $n \times m$ big, with n the number of measurements and m the number of actuators. The number of measurements n is twice the number of subapertures in the SH WFS, since each subaperture gives an x- and y-offset.

Now that the influence matrix is known, the offsets measured with the SH WFS can be calculated given a certain vector of actuator voltages:

$$\boldsymbol{p} = \mathbf{D}\,\boldsymbol{v}, \tag{3.1}$$

where $\boldsymbol{v}$ is an m-element actuator voltage vector, $\mathbf{D}$ is the influence matrix and $\boldsymbol{p}$ is the resulting measurement vector holding n wavefront sensor measurements. Since for an AO system $\boldsymbol{p}$ is the input from the WFS and $\boldsymbol{v}$ is the output that should be sent to the DM, we need to invert $\mathbf{D}$. Since this matrix is typically rectangular, with $n > m$, i.e. we have more measurements than variables,

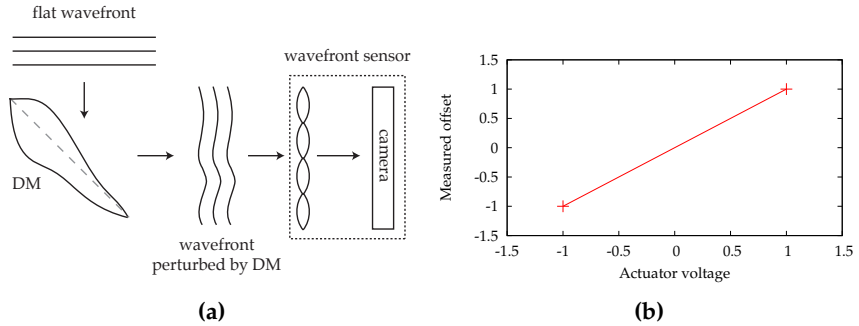


Figure 3.5: *(a)* Calibration of the wavefront corrector, such as a deformable mirror in this example, is achieved by sending a known (flat) wavefront through the adaptive optics system and measuring the wavefront sensor output as function of shape of the mirror. Two different shapes are drawn for the deformable mirror, with the relaxed position given by a dashed line. The shape of the mirror is defined by the voltage applied to the actuators. *(b)* Once the offsets for each subaperture are measured with one actuator moving between the two extreme positions, the relation between the actuator input and subaperture shift is found, which results in a graph such as the one here for each actuator-subaperture pair.

this is a typical singular value decomposition problem. For an insightful discussion on singular value decompositions, I refer to Numerical Recipes (**?**, Ch. 2.6). After singular value decomposing $\mathbf{D}$, the result is given by

$$\boldsymbol{v} = \mathbf{D}^* \boldsymbol{p}, \tag{3.2}$$

such that given the measurement vector $\boldsymbol{p}$, we can calculate the control signals (voltages) that need to be sent to the DM in order to correct the wavefront.

3.3.3 Closing the feedback loop

Once the system is calibrated, the feedback loop can be closed. In such a case, the image from the wavefront sensor camera is read out, and the offsets are determined and compared with the reference offsets for a plane wavefront. The offsets that remain are then multiplied with $\mathbf{D}^*$, resulting in control voltages for the wavefront corrector. If the system is successfully running in closed feedback loop, it is said to be locked.

Note that the wavefront sensor only senses the residual wavefront error, i.e. a correction has already been applied by the correcting actuators. To account for this, the voltages calculated using Eq. (**??**) must be added to the current voltages applied to the actuators.

3.4 System requirements

To successfully correct distorted wavefronts, there are several things to keep in mind. As already discussed in Chapter **??**, the strength of the seeing is a

function of wavelength. The spatial and temporal requirements of an adaptive optics systems are thus dependent on what one aims to observe. Recall two important quantities, the Fried parameter r_0 and the Greenwood time delay τ_0. These two quantities are again plotted in Fig. **??** for easy reference.

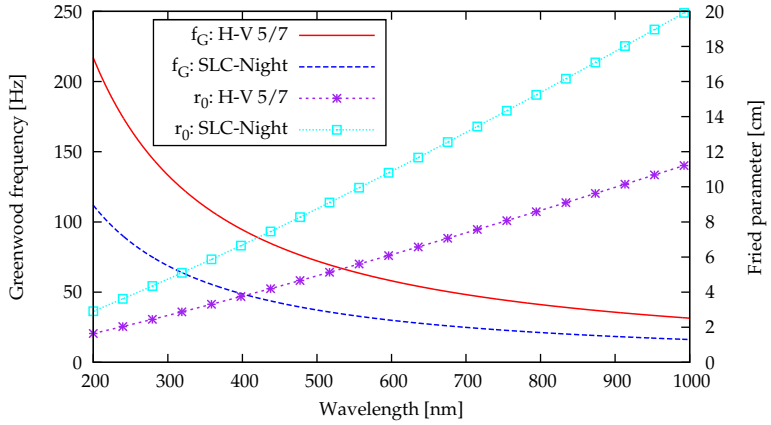


Figure 3.6: This graph shows the relation between r_0 and f_G versus wavelength.

3.4.1 SPATIAL REQUIREMENTS

To get an idea of how many actuators we need to correct, consider a small mirror with radius d, which is able to correct piston, tip- and tilt modes. To do so, this mirror needs 3 degrees of freedom, i.e. 3 actuators. If we fully correct these modes, Table **??** shows that the wavefront error scales as

$$\sigma^2_{\varphi,3} = 0.134 \left(\frac{d}{r_0}\right)^{5/3}. \tag{3.3}$$

Placing several of these mirrors together to make a bigger mirror of radius D requires about D^2/d^2 of these small mirrors, thus requiring $N = 3 \times (D/d)^2$ actuators in total. Putting this all together and combining it with Eq. (**??**) gives

$$\sigma^2_{\varphi,3} = 0.335 \left(\frac{D}{r_0}\right)^{5/3} N^{-5/6}, \tag{3.4}$$

which is also given in **?**, pp. 13.

This correction method is called a zonal correction, as the correction is applied for each part of the wavefront separately (i.e. the big mirror is not much more than several smaller mirrors working individually). Another possibility is the so-called modal correction, where the mirror corrects several modes in the wavefront at once, such that the different parts of the mirror do not work independently. The modes used in this case are usually specific system modes that a mirror has.

According to **?**, pp. 31, modal correction is slightly more effective than zonal correction. This makes sense as modal correction uses the deformable mirror

as a whole, instead of acting like several independent mirrors. The wavefront error when correcting Zernike modes decreases as $N^{-0.87}$ for $N \gtrsim 10$, as opposed to $N^{-5/6} = N^{-0.83}$ for zonal correction. When using a Karhuhen-Loève modes expansion instead of Zernike modes, this correction is again slightly more effective.

As for the spatial requirement for an adaptive optics system, keeping the rms wavefront error around 1 radian gives the following requirement for the number of independent actuators to be used:

$$\begin{aligned} N &= 0.269 \left(\frac{D}{r_0}\right)^2 \\ &\propto D^2 \, \lambda^{-12/5}, \end{aligned} \tag{3.5}$$

which means that bigger telescopes and shorter wavelengths are much more difficult to correct. For example when comparing adaptive optics in the infrared (1.5 micron) with visible (0.5 micron), correcting at both wavelengths with the same accuracy would require about 10 times as many actuators in the visible as in the infrared. When designing an AO system, one has to keep these requirements in mind.

3.4.2 TEMPORAL REQUIREMENTS

Besides spatial requirements, there are also temporal restrictions to an AO system. The Greenwood frequency f_G given by the reciprocal of Eq. (**??**) is a measure for the rate of change of the turbulence, which means that in practice an adaptive optics system should run at least a few times faster than this frequency to effectively correct the wavefront. From Fig. **??** we can see that this means that AO systems need to run at rates of up to 500–1000 Hz. Again, there is a strong wavelength dependence. While the Greenwood frequency is in the order of 25 Hz for infrared, it goes up to 100 Hz for the visible.

This requires that the wavefront correctors must be able to run at these speeds, while this also means that the wavefront sensor, i.e. the camera, must be able to deliver 500–1000 frames per second. Consequently, this data stream must be processed to analyse the sensor output and generate actuator voltages. Since this data stream is in the order of tens to hundreds of megabytes per second, this puts a strain on the controller. Considering the image must be dark- and flatfielded as well in certain cases, adaptive optics can be computationally expensive.

The solution to this problem has for a long time been to use dedicated hardware which was tailor made for controlling adaptive optics. The hardware used for this were FPGA's, or Field Programmable Gate Arrays, which contain programmable logic such that the algorithms used during adaptive optics can be directly implemented in the hardware itself. This type of systems is relatively fast, but unfortunately also difficult to build or customise.

Nowadays, general purpose computers are beginning to become fast enough to run these algorithms instead. FOAM attempts to fill the gap of adaptive optics controlling software running on general purpose machines, which is the topic of the next chapter.

3.5 EXAMPLE: MERCURY

As a case-study to give insight on how adaptive optics help observations in practice, this section provides example observations of Mercury with and without adaptive optics.

Since Mercury is the innermost planet in our solar system, it is always close to the sun. This gives rise to some problems when observing this planet. Either it is observed during the day, when seeing is generally bad due to surface heating of the sun, or it is observed just after sunset through a large air mass. In both cases, atmospheric seeing is especially problematic when observing Mercury. On top of that, Mercury only subtends a few arcseconds across the sky, which is not much more than the highest resolution obtained when observing without adaptive optics.

The sodium distribution over the Mercury surface appears non-uniform and changes with time. To better understand the Mercury exosphere and exospheres in general, sodium is a suitable candidate for study because of its strong resonance. Furthermore, since the sodium interacts with the solar wind, studying the distribution of this compound provides insight in the interaction of the wind with the Mercury surface and exosphere. The following is a summarised version of **?** and **??**, with focus on adaptive optics.

3.5.1 BOWEN IMAGE SLICER

To study the spatial distribution of sodium over the Mercury surface, a Bowen image slicer was used. This device takes a two-dimensional image and slices it up in a complete set of several rectangles and uses prisms to re-arrange the slices such that they line up top to bottom. This elongated image is then passed to a spectrograph, which disentangles the light into spectra. The result is thus a two-dimensional image with a spectrum for each pixel. The procedure is depicted in Fig. **??**, and is also discussed in **?**, footnote 13.

3.5.2 ADAPTIVE OPTICS

Since Mercury observations are observationally challenging, adaptive optics can help out in correcting some of the distortions caused by the atmosphere. This was done in May of 2008 using the McMath-Pierce solar telescope at Kitt Peak, Arizona. Although this run did not yet use FOAM for driving the AO system, new hardware was used in this setup which required adaptations to the old software in order to get it working. The experience gained during the development of FOAM proved to be quite useful in this situation as it was used in some preliminary tests on this new hardware. An adaptive optics setup similar to the one used during this run is described in more detail in **?**.

To illustrate the benefit from the use of adaptive optics, some results of two different Mercury observation runs are presented in Fig. **??**. One was done without the use of any optical correction, while the other used full adaptive optics with a tip-tilt- and deformable mirror. Although the two observations were taking at different times, the seeing was approximately the same during

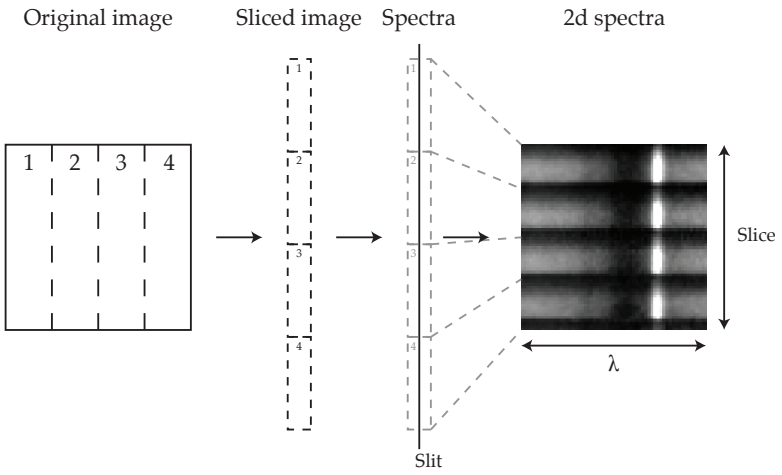
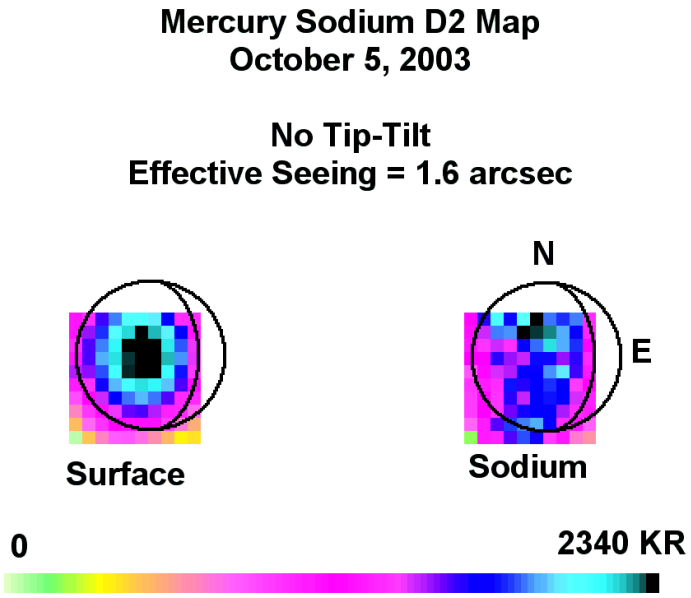
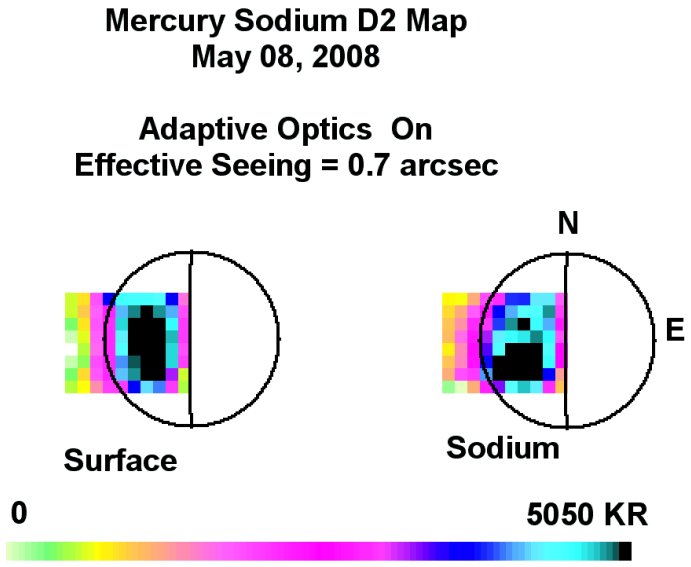


Figure 3.7: The mechanism of a Bowen image slicer. A 2-dimensional spatial image is fed to the slicer, which chops it up in a set of rectangular slices, as shown on the left. These are then rearranged top-to-bottom and fed to a slit which disentangles the light into spectra, shown in the two centre images. The result (on the right) is again a 2-dimensional image with wavelength on the horizontal axis and position on the vertical. By taking a certain spectral band in this image and again rearranging it back to the original configuration, one can construct a spatial 2d image showing the intensity of that spectral band. The resolution of the final image is limited in one direction by the amount of slices produced, while in the other direction this is governed by the resolution of the slit-CCD combination.

the two observations. Figure **??** shows that the effective seeing is much better with the correction turned on, and as the image is concerned the non-uniform distribution of sodium is much clearer in the right image. Thanks to adaptive optics, this kind of research is possible in the first place.



(a)



(b)

Figure 3.8: Comparison of image quality of Mercury observations with and without using adaptive optics. In both (a) and (b), the left image shows Mercury in integrated light, while the right image shows the planet in a sodium line. Both images were obtained through private correspondence with A. Potter, but the results of (a) have been published in **?**.

4th Chapter

FOAM DESIGN

FOAM is a reverse acronym for Modular Adaptive Optics Framework, and tries to provide exactly that. In light of the relative absence of this type of software, FOAM aims to fill this niche.

To this end, I designed FOAM with an open and portable adaptive optics (AO) system in mind, based on general purpose computers using Unix-like operating systems. The architecture describes a general framework which should apply to as many AO systems as possible, such that implementation of this architecture is possible across a wide range of platforms. To achieve this goal, I designed the interfaces between different components to be as general as possible, internally using generalised commands which are translated to hardware-specific commands by separate software modules.

This chapter describes what the requirements for FOAM are and how this is translated into an architecture.

4.1 REQUIREMENTS

Before designing the architecture of a program, the requirements for the system must first be considered. This section contains a brief overview of what FOAM must be capable of and thus what must be included in the architecture.

Portability The system should be portable across different Unix platforms.

Scalability The AO system should be scalable in the sense that it should run efficiently on large machines (multi-core or -CPU machines).

Extensibility The system must easily be extensible, i.e. it must be easy to add new wavefront sensors or -correctors, such as the case in a multi-conjugate AO setup. Another example of this would be that the system can be simulated and that different modes of operation can be tested without hardware.

Usability It must be possible that the system is controlled by a user on the same or a different computer than what the AO system runs on. Ad-

ditionally, it must be possible to automate this control such that other software or hardware can also (partially) drive the AO system.

Public Because FOAM aims to be used in a collaborative and open environment, FOAM must be free of any licensing issues. The license of FOAM should thus be an open source version, such that anyone can modify and adapt the software to their liking.

In the conclusions at the end of next chapter I review this list and evaluate the degree to which these requirements have been met.

4.2 SYSTEM COMPONENTS OVERVIEW

An AO system consists of different components that work together. Figure **??** shows how the components are connected, and a description of each component is given below that. Each component can be something physical, but can also be simulated by software.

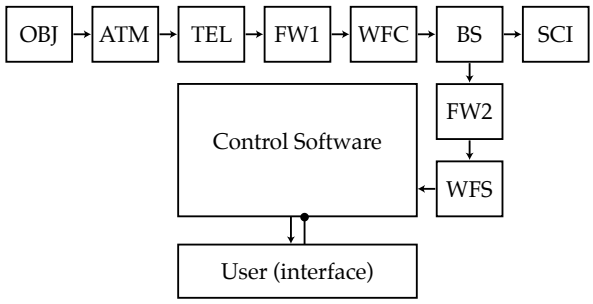


Figure 4.1: The setup of a typical AO system. The arrows indicate the data-streams (light, images, wavefronts), the circle-ended arrow denotes the control-stream from the user to the software. Note that the logical connections from the control software to the components are not drawn here yet.

Object (OBJ) The object is strictly speaking part of the system, although usually one has little to do with it. Only in the case of simulation does this become important.

Atmosphere (ATM) Here the (synthesised) image is be distorted. This component is usually of great influence since it is responsible for the distortion, but under little control of the observer. During simulation this component is again simulated by software.

Telescope (TEL) This is the telescope which takes care of gathering the light and focussing it. During AO operations, the telescope is able to perform the same function as the tip-tilt mirror, and must be included during long runs to prevent large image drifting.

Filter wheel 1 (FW1) The first filter wheel is placed in the telescope focus in front of the AO system and is used for calibration purposes. During normal operation, this filter wheel is open.

Wavefront corrector (WFC) This component will correct the wavefront distorted by the atmosphere and optical components in front of it. There can be multiple WFCs in one AO system, and usually there are at least two: a tip-tilt- and a deformable mirror.

Beam splitter (BS) This passive component redirects some of the light to the AO sensors and sends most of the light to the actual science instrumentation. This is essential because we need part of the light for AO, but most of the light will be used for science.

Filter wheel 2 (FW2) This filter wheel placed behind the beam splitter is only experienced by the AO sensors and is used for calibration purposes. This filter wheel is also open during normal operation.

Wavefront sensor (WFS) The wavefront sensor measures the wavefront distortion. This can for example be a Shack-Hartmann wavefront sensor, but again, this can also be something else. An imaging camera is strictly speaking also a WFS, but one with only one subaperture.

Control Software This is the centre of the AO system as it uses the input from a sensor to drive the other components. Additionally, the software must take care of user interaction.

Science instrumentation (SCI) This single component encompasses all science instrumentation behind the AO system. This component is of interest because in simulation we might want to be able to simulate science instruments as well.

The first two components are a bit unusual in the sense that during operation they are not anywhere near the telescope, and are not under any control of the observer. These are included for completeness however, since they play a crucial role during simulation. The other components can either be real or simulated. It is important to understand that only the whole AO system can be simulated though, and that simulation of individual components is senseless.

4.3 ARCHITECTURE

Now that the components that are used in the AO system are defined, this section presents the architecture of FOAM providing the requirements mentioned before.

4.3.1 MODULAR SETUP

To achieve portable and extensible software, I kept the basic program in FOAM to a minimum. This framework can then be extended by several modular

pieces of code (modules) which attach to hooks provided by the framework. A bundle of the framework and some modules is linked together by a prime module. This prime module also contains configuration on how to run the modules, i.e. if a camera module is used, what is the resolution, if a deformable mirror module is used, how many actuators do we use etc. Together, the framework, the prime module and the software modules form one package which can do all the work. This is depicted in Fig. **??**.

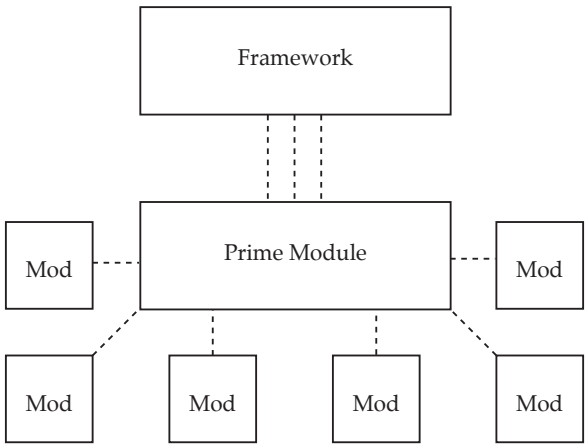


Figure 4.2: The FOAM architecture, showing the framework at the top which only links to the prime module, which in turn employs a number of modules to provide a functional program. This whole setup is dubbed a 'package'.

This FOAM framework can be run in three different modes. The first mode is open loop, which only performs measurements and nothing else. In this mode, no adaptive optics is performed and FOAM functions as a measurement platform. The second mode is closed loop and in this mode the measurements are used to drive the actuators and correct the incoming wavefront. This feedback loop is the adaptive optics loop. The third mode is calibration mode which is only run once and can be used to perform various calibrations on the system.

4.3.2 HOOKS

The framework provides several hooks that can be used to attach the prime module to. The prime module can then decide what to do at each of these hook functions, and use various modules to process data or drive actuators.

There are program-wide hooks which are called at the beginning of the program to initialise the (prime) modules, at the end to clean things up and to process messages when commands are received from a user. Additionally, there are hooks for the open and closed loop modes. A hook is called at the beginning and end of these modes, and additionally, one is called during the main loop. For the calibration mode, there is only one hook which is called as this mode does not provide a loop.

The following piece of pseudocode illustrates the use of hooks. For a complete

skeleton, see Appendix **??**.

Code snippet 4.1: FOAM hook illustration

```
main() {
    modInitModule()

    fork (listenLoop())

    while (true) processUserInput()
    modStopModule()
    exit
}

listenLoop() {
    while (mode != shutdown) {
        switch (mode) {
            case 'open': modeOpen()
            case 'closed': modeClosed()
            case 'calibration': modeCal()
        }
    }
    modStopModule()
}

modeOpen() {
    modOpenInit()
    while (mode == open) {
        modOpenLoop()
    }
    modOpenFinish()
}
```

In this example, all `mod*` functions are hooks that are not defined by FOAM itself, but must be defined in the prime module. `modInitModule()` and `modStopModule()` are the hooks called to initialise and clean up the whole program, while `modOpenInit()`, `modOpenLoop()` and `modOpenFinish()` are the three hooks used during open loop mode. The same method is used for closed loop mode, and during calibration mode only a hook similar to `modOpenInit()` is available. A more detailed description of these hooks is provided in Appendix **??**.

4.3.3 PROGRAM FLOW

In this section, I briefly discusses the general program flow to get an overview of FOAM.

After initialisation, FOAM splits into two threads. The threads share the same memory, so the variables initialised before the threading are available to both threads. One thread runs in a high priority mode, ensuring access to computing resources, while the other runs with a lower priority. The high priority thread does the hard work and control the AO, while the secondary thread opens a socket and listens for incoming connections. Additionally, the first

thread can again use multiple threads to provide scalability on multi-core or -CPU platforms.

To send commands to FOAM, a user or program can connect to it using TCP/IP sockets on a user-configurable port. Once a connection is established and valid commands are received by FOAM, the shared variables are used to change the mode of operation of the high priority thread, in combination with condition variables in certain cases. When a valid command is received, this is broadcasted to all connected clients. There is no direct feedback to the client sending the command, instead any change in the software is broadcasted independently of the clients connected (e.g. model-view-controller). This allows simultaneous control of FOAM by multiple clients. Illegal commands are reported to the client that sent them only.

The advantage of using TCP/IP sockets to connect to FOAM is that this provides a scalable way to handle multiple users. FOAM internally uses `libevent` to multiplex network I/O to handle multiple clients at the same time. `libevent` is capable of multiplexing I/O for webservers which typically deal with much more than FOAM will, hence this is unlikely to pose a limitation in the future. Besides this scalability, one can easily make a (G)UI compatible with FOAM since the commands are standardised. At the moment, telnet is used to connect to FOAM. Automating control is also easy, as long as the software trying to interact with FOAM can talk TCP/IP.

While the flow of the low priority user I/O thread is more or less fixed, the high priority thread is highly customisable using prime module and modules. FOAM provides a bare minimum for the open loop-, closed loop- and calibration modes, but the specifics must be filled in by the prime module using the hooks discussed above.

4.3.4 HARDWARE ABSTRACTION

To provide a flexible setup, I abstracted the hardware from FOAM to such a level that all wavefront sensors and -correctors can be described with the same parameters. The abstraction for WFSs and WFCs is discussed in this section. The technical details are given in Appendix **??**.

WFS

A wavefront sensor is generalised by FOAM as a sensor with the following characteristics:

Resolution The pixel-resolution attributed to the sensor, which states a horizontal and vertical resolution, as well as the bit depth for each pixel,

Calibration The dark- and flat-field calibration images used to correct the incoming sensor images.

What becomes clear from the above list is that nothing relates the above characteristics to anything more than a camera. In order to keep the system as general

and flexible as possible, I have chosen to store the specific WFS details in modules. For example, if a system uses a Shack-Hartmann WFS, a specific module will be used which adds relevant attributes to the sensor. These include:

Cell grid The resolution of the subapertures used, i.e. 8×8,

Track size The pixel size of the windows used to track the offsets in each subaperture,

Calibration The influence function stored in singular value decomposition format,

Measurements The matrix of offsets the SH WFS measures for each subaperture.

The combination of these two sets of properties allow the software to use the WFS to correct the wavefront. The specific details are not relevant here, but can be found in the software documentation.

WFC

The wavefront corrector, which will often be a deformable mirror, is the actuator correcting the wavefront. In FOAM, this actuator is generalised with the following properties:

 # Actuators The number of actuators a WFC has,

Control A vector of control signals to be sent to each actuator, which range from -1 to 1,

Gain The different gains used for this actuator.

Again, to keep the system as general as possible, the control signals cover the range from -1 to 1. The module responsible for interacting with the hardware like a deformable- or tip-tilt mirror can then translate these control signals to voltages. One thing to keep in mind here is that some actuators might not respond linearly in the voltage range they accept. The control signals used here are linear however, such that a conversion might be necessary in the module. An example is a membrane mirror, which responds linearly in voltage squared.

Besides these interfaces, each module uses its own data type to characterise a device it will work with. Examples of this include cameras, filter wheels, display routines etc.

4.3.5 PARALLELISATION

To provide scalability to the code, the computationally intensive tasks should be threaded such that the load can be distributed over multiple CPUs CPU-cores. Since FOAM itself does not perform any computational intensive tasks however, threading has been kept to a minimum here.

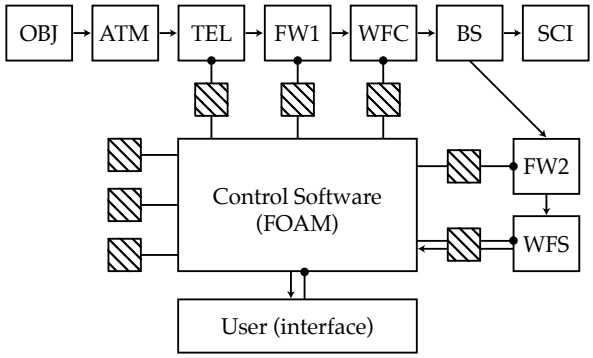


Figure 4.3: An implementation of FOAM, showing the framework at the centre, which communicates with the user and modules (dashed boxes) only. The hardware itself is shielded from FOAM, the specific hardware modules form a shell around the framework itself. This implementation allows for easy replacement of any hardware component, the framework only needs minor modifications as all hardware related commands are implemented in the modules.

The computationally intensive tasks such as dark- and flat-fielding, correlation- or centre of gravity-tracking etc., are performed in modules. The modules currently supplied with FOAM are not threaded, but the architecture of FOAM is designed in such a way that threading these modules only requires rewriting the modules themselves.

Although threading is not implemented, FOAM internally uses data types that are suitable to use in combination with a BLAS[1], like ATLAS[2]. These highly optimised libraries are available on many platforms and are used for several linear algebra operations involving vectors or matrices. These libraries are for example used during the vector-matrix multiplication necessary to calculate the control signals. Even without threading, this provides a significant performance boost.

[1] Basic Linear Algebra Subprograms, `http://www.netlib.org/blas/`.
[2] `http://math-atlas.sourceforge.net/`

5th Chapter
FOAM IMPLEMENTATION

This chapter describes some results obtained with a simulation implementation of FOAM, looks into possible future extensions, and finally concludes by reviewing the requirements stated at the beginning of the previous chapter.

5.1 SIMULATIONS

Even though FOAM has not been used in a real adaptive optics setup yet, it does provide simulation capabilities that allow it to simulate seeing and correct this artificial error afterwards.

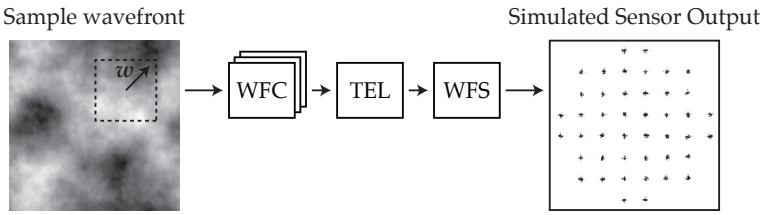


Figure 5.1: Schematic depiction of the simulation process in FOAM. A simulated wavefront is cropped to the size of the sensor resolution, with the cropping windows moving over the simulated wavefront with speed $\boldsymbol{w}$ to simulate wind. The cropped wavefront is passed through the wavefront correctors, after which the wavefront is cropped with an image of the telescope aperture. After this, the wavefront is passed through a simulation of a wavefront sensor which in this case is a Shack-Hartmann WFS.

The simulation starts with a sample artificial wavefront, which is then passed through the various elements in the simulated AO system. This process is shown in Fig. **??**. The simulation can be tweaked using several variables available through FOAM. For example, the wind speed can be altered, as well as a seeing factor, which is multiplied with the raw input wavefront to worsen or lessen the seeing.

Besides this type of simulation, FOAM can also use WFCs to simulate an error. This is done constructing a fake actuator signal, either periodic or random,

which is fed to the simulation routine for a WFC. The output this gives is then passed through the rest of the simulation. The advantage of this method is that an error produced by such a simulated WFC can be perfectly corrected by that same WFC. This is an acid test for the whole calibration-measurement-reconstruction loop performed by FOAM: this error should be easy to correct.

5.1.1 TIP-TILT SIMULATIONS

Using revision 477 of FOAM with prime-module `simdyn`, I simulated a sawtooth-shaped tip-tilt error signal with amplitude 1 and a 40-frame periodicity which was corrected using the same routine that produced the error. The results of this simulation are presented in Fig. **??**, which shows the difference in error- and control signal. The mean value of this difference is about 1×10^{-5} with a variance of 4×10^{-6}.

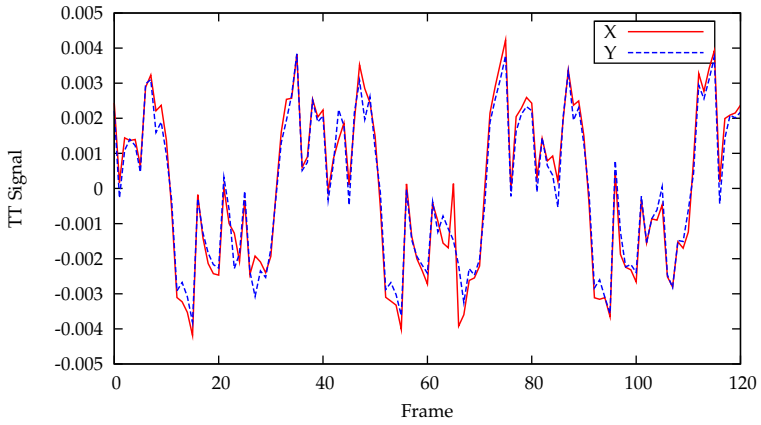


Figure 5.2: This graph shows the difference between the signal going to the tip-tilt routine to simulate an error and the signal that was reconstructed after measuring the subaperture offsets. The signal shows the 40-frame periodicity which was used to generate the sawtooth error signal for the tip-tilt mirror.

Using the same error, instead of using a regular sawtooth error signal, I used a randomly drifting error, to analyse the correction capabilities for FOAM with a less regular error. The result of this simulation is given in Fig. **??**. Instead of plotting the difference, the error- and correction signal itself were plotted this time.

A more realistic test is to look at the simulation of a real error, i.e. the situation shown in Fig. **??**. I have also done this and the results are in accordance with the expectations based on the previous analysis. Because this error was simulated using an artificial wavefront, comparison of the error signals with the correction signals for the WFC is not possible. Instead, I summed the offset-vectors measured for each subaperture, and divided this sum by the number of subapertures. The result is thus an average tip-tilt error over all subapertures, which I've compared between open-loop without correction and closed-loop with correction in Fig. **??**.

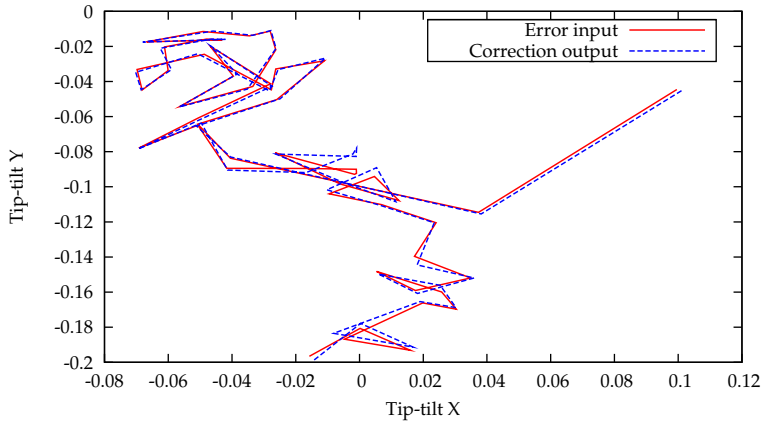


Figure 5.3: A random tip-tilt error corrected by FOAM. The correction signal closely follows the error, as expected.

Note that there is a one-frame lag between the error and the correction of that error, such that a spread around the origin is expected. Because of this one-frame delay, the correction applied to frame N is based on the error in frame $N-1$. Therefore, the correction on frame N should be perfect for frame $N-1$. Keeping this in mind, the spread around the origin of the corrected displacement must be caused by the change in the error between frame N and $N+1$. Thus, the difference in the (x,y) error vector between frames $N-1$ and N should be equal to the offset from the origin of the correction for frame N. Calculating the mean and variance for both the length of the correction offset vectors and the length of the error-difference vectors, I found that these are quite similar. For the corrected offsets, the mean was about 0.11 pixels, with a variance of 4.0×10^{-3}, while for the error-difference vectors, this was 0.11 and 9.4×10^{-3}. Thus the statistics of the spread are in accordance with expectations.

Thus, in simulation mode, FOAM works as expected, it can simulate and correct tip-tilt errors with a high precision. This shows that the underlying mechanisms of FOAM are working and that this means that FOAM will also work on real adaptive optics systems.

5.2 AVAILABILITY

To encourage the development of FOAM, the source code is available under the GNU GPL[1]. In a nutshell, this means that anyone can take the code, adapt it as they like, and redistribute it, as long as the redistributed code is also released under the GPL. This provides freedom to develop the source code on the one hand, while ensuring that the fruits of development will be returned to the community on the other hand. Since this software will most likely be used in an academic environment and the development was funded with public

[1]Details of the GNU General Public License can be found at `http://www.gnu.org/licenses/gpl.html`.

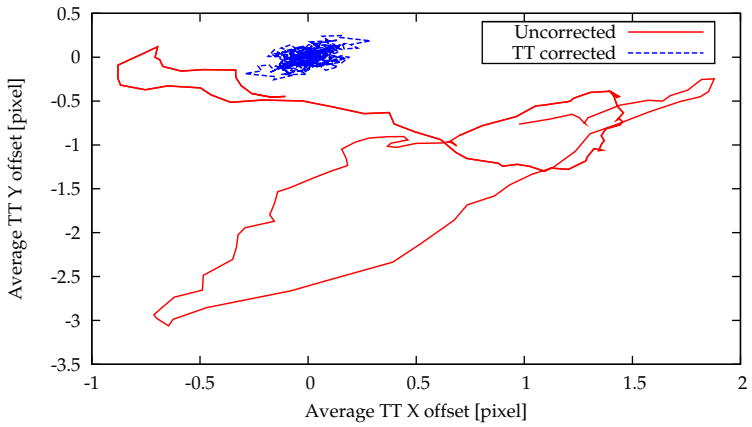


Figure 5.4: Correction of simulated seeing using an artificial wavefront. Without correction, the image drifts around, but when tip-tilt correction is turned on, the image displacement is cancelled. Note the expected spread around the origin.

money, this does not seem more than fair.

If one does not wish to return the modifications to the community, this is also possible, but in that case the software must be kept in-house. Additionally, the source code may be released under a different license by request, if a specific situation requires a tailor-made solution.

5.3 FUTURE EXTENSIONS

Although FOAM is still in work in progress, it is already clear that several extensions can be made to the current modules supplied. Some of these will be discussed here, considering the possibilities and possible issues involved.

5.3.1 MCAO

While adaptive optics itself is now used more and more, an improvement to this system can be made by using several different wavefront sensors at the same time. Placing these in different planes allows to analyse the atmospheric turbulence at different heights in the atmosphere.

This so called multi-conjugate adaptive optics is more challenging than regular AO because of the added wavefront sensor. It is not a priori clear how the data from these different sensors should be combined. An extension for FOAM would thus be to add routines which can combine the output from several sensors and improve the correction of the wavefront.

5.3.2 PARALLELISATION

Currently, the only threading is done in the framework, where one thread handles user input while the other does the actual AO. As noted earlier, there is still room for improvement performance-wise, as the number crunching modules are still single threaded. Although for some procedures threading is rather straightforward (i.e. correlation- or centre of gravity tracking can be done in parallel for all subapertures), for others this is far from trivial.

5.3.3 C++

Another improvement would be to (partially) port FOAM to C++. Since C++ provides some language constructs that have quite a few benefits for the modular approach (i.e. classes), this would be a logical step. Unfortunately, at the beginning of the development, I was not aware of this and thus FOAM is written in C only. Fortunately, C is a subset of C++, such that porting will not be as difficult as porting from C to, say, Fortran.

5.3.4 REAL-TIME LINUX

Since a computer can do only one thing at a time (which is not entirely true anymore for newer multi-core machines), time is sliced up and given to processes requiring the CPU in turn. Higher priorities get longer time slices, but all processes are given at least some processing time. When running adaptive optics control software, it is possible that some network I/O needs to be taken care of, or some file on the hard disk needs to be accessed. Because network and hard disk activity are relatively slow, this can introduce a sudden delay in the AO software, causing it to miss up to a few frames, potentially causing the software to lose the lock.

To provide tighter control over which processes get CPU time, a real-time operating system can be used. For simple controlling algorithms this is not crucial, but smarter algorithms can benefit from a tighter timing control.

5.4 CONCLUSIONS

To review the requirements made earlier, here is a list which discusses to what extent they have been addressed:

Portability FOAM is written in C and specific hardware code will not be included. The libraries used in FOAM are at least as portable as FOAM itself. Furthermore, FOAM internally uses generalised controls which are translated into hardware controls by the drivers, providing abstraction from the hardware.

Scalability FOAM is be written in C and can provide threaded modules for the computationally intensive tasks. The OS can then easily distribute these tasks over different cores (single- or multi-CPU).

Extensibility Because of the modular setup, big pieces of software can be reused in other AO systems. Possible extensions can include the use of multi-conjugate AO.

Usability The use of a TCP/IP sockets makes interfacing to the software easy, both for humans and other software.

Public The source code of FOAM is released under the GPL, such that anyone is free to take and modify the code to their liking. Furthermore, extensive documentation is supplied to make this possible.

This architecture thus meets the requirements stated at the beginning of this chapter. The architecture is flexible and general enough to cope with both a simple tip-tilt tracking system or a sophisticated multi-conjugate AO system. In the first case, we have one WFC with two degrees of freedom (tip and tilt) and a WFS with one subaperture, i.e. a camera, tracking the image displacement. The second system is described by having two (or more) WFCs which are driven using the output of several wavefront sensors. Even more generally speaking, we can also describe a simple telescope tracking mechanism with this architecture. In this case there are zero WFCs and a WFS with one subaperture. The image motion is then corrected by the telescope itself at low frequency. As the reader might notice, these three different systems all fall nicely within the design of this architecture.

Most, if not all new telescopes under construction today incorporate some kind of adaptive optics systems. Examples of this include the ATST (**?**) and the E-ELT (**?**), but there are many more. With the ever increasing diameter of telescopes, this is not remarkable. To increase the resolution, one cannot go without some mechanism to correct the atmospheric distortions, and adaptive optics plays a crucial role in fulfilling this task.

With this in mind, software to run such systems will be in high demand as these complex systems need to be controlled. Putting FOAM in this broader context, it could potentially be quite useful indeed.

6th Chapter

APPLICATIONS

6.1 FOAM AT THE MCMATH-PIERCE TELESCOPE

FOAM has been tested on the the McMath-Pierce solar telescope at Kitt Peak (**?**). This AO system consists of one Shack-Hartmann wavefront sensor, one deformable mirror and one tip-tilt mirror, and some miscellaneous hardware such as filter wheels. The system is similar to the one depicted in Fig. **??**, and served as a basis for the development of FOAM. The details of the system are listed in Table **??**.

6.1.1 AO SETUP

Hardware	Model and Vendor	Driver
Camera	CA-D6 260 × 260 8-bit CCD, 955 fps (Dalsa)	-
Frame grabber	PC-Dig PCI board (Coreco Imaging)	ITIFG 8.4.0-0
Deformable mirror	37-actuator membrane mirror (Okotech)	-
DM Controller	PCI board (Okotech)	Direct I/O to `/dev/pci`
Tip-tilt mirror	PSH 8 Tilting platform (piezosystem jena)	-
DAC board	DaqBoard/2000 Series (IOtech)	Linux 2.6 driver
Computer	Intel Core 2 processor / Debian 4.0	-

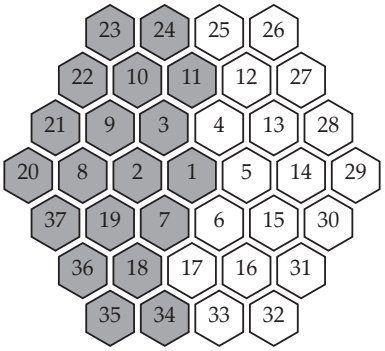
Table 6.1: The hardware used in the adaptive optics system used at the McMath-Pierce telescope, which served as a testbed for FOAM. This setup closely resembles a system used previously at the same telescope for infrared adaptive optics, see `http://www.noao.edu/noao/staff/keller/irao/` for details.

The Dalsa camera is part of a Shack-Hartmann wavefront sensor, with a lenslet array in front of the camera. It provides the input for the adaptive optics system. The camera output is first fed to a frame grabber which directly interfaces with the camera and temporarily buffers this data. After that, the frames are available to the rest of the system using the ITI frame grabber driver by GOM GmbH[1]. There are two actuators correcting the wavefront, one tip-tilt mirror

[1] See `http://sourceforge.net/projects/itifg/` for details.

for correcting the lower order modes, and one 37-actuator deformable mirror for correcting the higher order modes. The tip-tilt mirror is controlled by two digital to analogue ports on a DaqBoard/2000 by IOtech. The deformable mirror is controlled by a PCI card which can be accessed directly through `/dev/pci` on Linux systems.

(a)



(b)

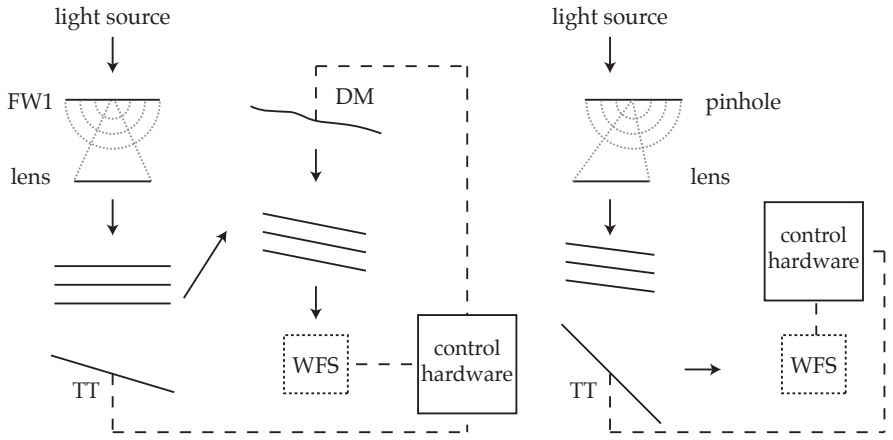
Figure 6.1: (a) An example of the 37-channel membrane mirror by Okotech. (b) The actuator layout of the mirror. For the tip-tilt simulation, the greyed actuators were set to voltage x while the other actuators were set to $-x$.

The implementation of FOAM for this setup began by writing modules for the hardware. Specifically, modules were written for interfacing with the various drivers used in the setup, which are listed in Appendix **??**. After writing these hardware interfacing modules, I wrote a prime module linking all this together which configures the hard- and software so it can be properly used. Unfortunately the deformable mirror was not used for wavefront correction in this implementation yet. After implementing the system, it was tested on the McMath-Pierce telescope using artificial sources of wavefront errors as discussed below.

6.1.2 TIP-TILT PERFORMANCE

Without the deformable mirror, the tip-tilt performance could still be tested. For this test, the camera ran at a 100 frames per second since the ITIFG 8.4.0 driver seems to have problems at high frame rates. However, a high frame rate was not crucial and thus this limitation did not pose a problem.

In the first series of tests an error in the wavefront was introduced by the deformable mirror. By setting the left actuators to a voltage x and the right actuators to voltage $-x$, this produces a tip-tilt error in the wavefront (see Fig. **??**). This error was consequently corrected by the tip-tilt mirror. The second test consisted of manually moving a pinhole around in the telescope focus resulting in an image displacement. This was done in two ways, one error was of low frequency and high amplitude ('big shake'), while the other had a higher



(a) DM error test setup **(b)** 'Shake' test setup

Figure 6.2: (a) The test setup for FOAM in which the deformable mirror generates a tip-tilt error in the wavefront. A pinhole is used in the first filter wheel (FW1), resulting in a flat wavefront arriving at the tip-tilt mirror, which is initially in its centre position. The same flat wavefront encounters the deformable mirror (DM), which tilts it slightly, such that an error is measured by the wavefront sensor (WFS). The control system then positions the tip-tilt mirror such, that it corrects the error introduced by the DM. For this test, FW2 is not used. (b) The 'shake' test, where a pinhole was moved around to generate an error. The offset pinhole produces a spherical wavefront, which is focused by a lens. Because the pinhole is not centred above the lens, the focused wavefront is not flat, but tilted. Moving the pinhole around therefore results in a dynamical error, which is consequently corrected by the tip-tilt mirror. The DM was not used in this test and is therefore not drawn.

frequency but lower amplitude ('little shake'). This generates an image offset as well, but the amplitude of the error is not restricted to the rather limited stroke of the deformable mirror. See Fig. **??** for a schematic depiction of the test setup used.

To analyse the performance of FOAM in these tests, the image offsets for each Shack-Hartmann subaperture was recorded. As only the tip-tilt correction was used during these tests, it was expected that the overall image displacement would be minimised. To test this, the offsets for half of the subapertures was summed to get measure for this displacement. Summing over a few subapertures is useful because this averages out some errors between the subapertures.

The DM-induced error test

During the first test, the deformable mirror introduced a tip-tilt error in the wavefront. The results of this test are presented in Figs. **??**–**??**. For this test, the periodicity of this error was set to 300 frames, equivalent to 3 seconds.

The correction was applied using a differential gain of 0.3, meaning that only 30% of the correction was applied each frame. This effectively means the correction is averaged out over a few frames, which results in a more stable cor-

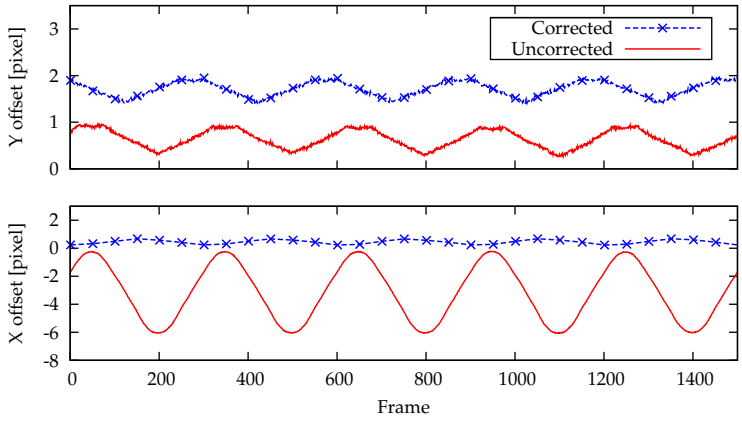


Figure 6.3: This graph displays the summed offset of several Shack-Hartmann subapertures. The top graph shows the Y-offset, the bottom graph shows the X-offset. The uncorrected error in the Y offset has been shifted up by 6 pixels to allow for better comparison. As the error introduced with the DM was one-dimensional, a majority of the shift is only visible in the X offset signal, the Y offset remained rather static and hardly needed correction. These results were obtained with revision 544 of FOAM.

rection. The downside of this approach is that not the complete error can be corrected. This is visible from the graphs in Fig. **??**, as there is still a slight wobble in the corrected signal. As the data sets were taken sequentially, a frame-by-frame comparison is not useful. The phase difference between the plots is therefore insignificant.

The shaking pinhole tests

The results of the second pinhole shaking tests are presented in Figs. **??**–**??**, which show two continuous data sets for the little-shake and big-shake errors with increasing differential gains applied for the correction of the tip-tilt mirror, and a power spectrum for those data sets.

Although there are some residual errors in the corrected images, the amplitude of the error is greatly reduced in both tests. Of the two tests, the system seems to perform better in the little-shake test. The increasing gain does not seem to have any effect on the correction in the big-shake test, as is visible from Figs. **??**–**??**. Even tough the power spectrum shows a strong decrease in power when enabling the system, the correction appears to be equally effective for gains 0.3, 0.7 and 1.0. The influence of the gain appears to be stronger for the little-shake test, where the fluctuation in the X- and Y-offsets seems to decrease with increasing gain. This is further supported by the power spectrum of this signal, which shows that the peak at the dominant frequency of 0.012 is reduced much more at higher gains.

For the big-shake test, it appears that though the image is stabilised most of the time, it regularly jumps to an -10 pixel offset during the whole test for all gains. This is probably due to the rapid motion of the pinhole at these times. As the data set is continuous, we can extrapolate the error shown during roughly the

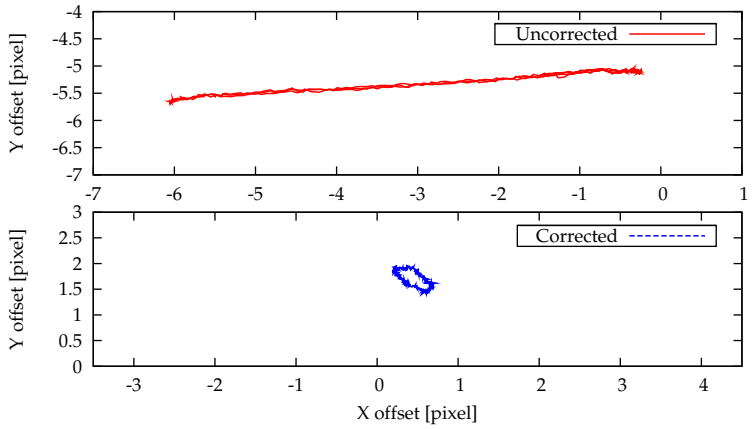
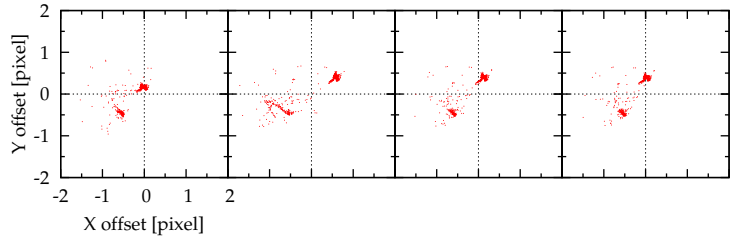


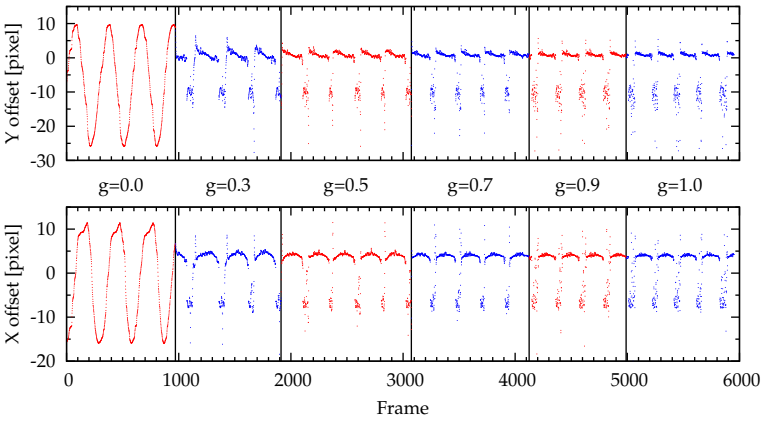
Figure 6.4: This plot shows the summed offset in both X- and Y directions with and without correction for the first 500 frames. Again there is little variation in the Y-offset while the X-offset is clearly corrected.

first thousand frames to the rest of the data set. In doing so it appears that these 10-pixel-offsets coincide with the rapid motion of the pinhole. As this motion takes about 100 frames, and moves from a +10 to a -25 pixel offset in that time, this corresponds to approximately one pixel per three frames. This rapid motion is apparently too much for the system too handle. Fortunately, the errors in the big-shake test are far worse than anything encountered even during bad seeing. The same effect is also visible with the little-shake test, although it appears that the increasing gain dampens these offsets, there are spikes clearly visible with gain at 0.3, but these dampen out as the gain increases.

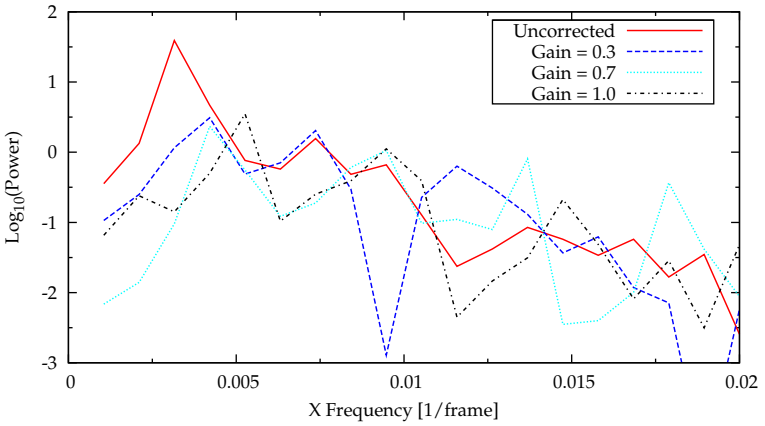
Overall, these tests show that the system indeed works in practice. Even though these test only used tip-tilt correction, the routines used are generalised such that they should work with any wavefront corrector. Future tests will hopefully show FOAM indeed works with for example deformable mirrors. After proving that the concept works, it can hopefully be incorporated in live adaptive optics systems.



(a)

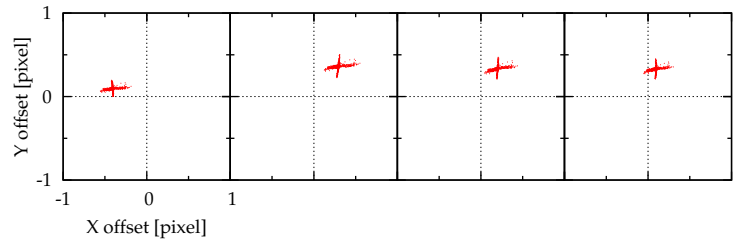


(b)

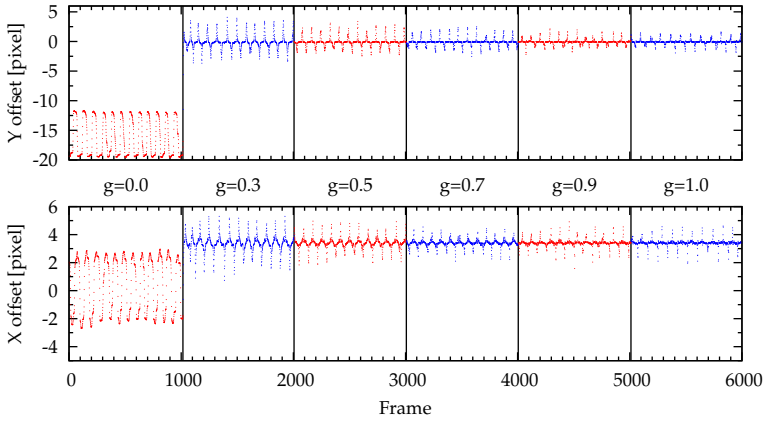


(c)

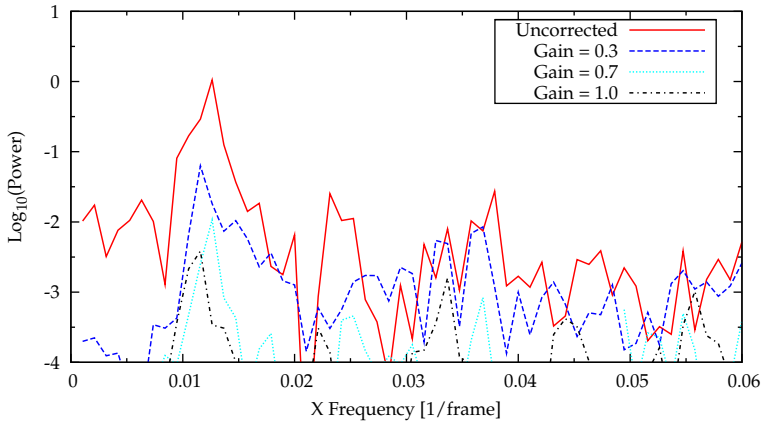
Figure 6.5: Results for the 'big-shake' test. (a) The offsets for some of the individual subapertures corrected with a gain of 0.5. (b) Summed X- and Y-offsets, with various differential gains set for the correction denoted by the g in between the two plots. Note that a gain of 0 effectively disables correction and shows the uncorrected error. (c) The power spectrum of the uncorrected and corrected offsets for various gains. These results were obtained using revision 553 of FOAM.



(a)



(b)



(c)

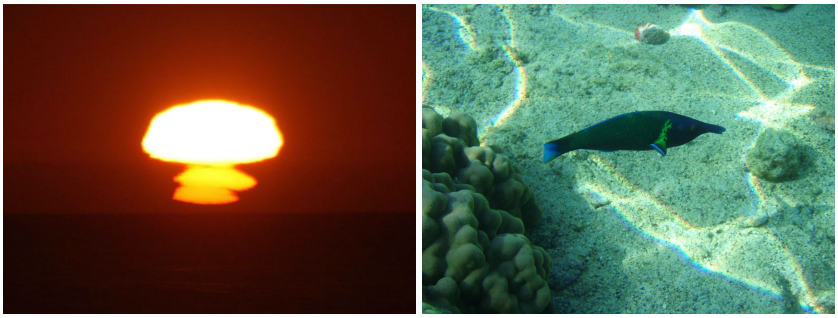
Figure 6.6: Results for the 'little-shake' test, same plots as in Fig. **??**.

A NEDERLANDSE SAMENVATTING

A.1 ATMOSFERISCHE STORINGEN

Als we naar de sterren kijken is er een onderscheid te maken tussen waarnemingen vanaf het aardoppervlak en vanuit de ruimte. Vanuit de ruimte is het zicht op de sterren veel beter door de afwezigheid van de atmosfeer die het licht verstoort, alleen zijn de kosten hiervan vele malen groter. Telescopen op aarde kunnen daarnaast een stuk groter zijn, omdat ze niet met een space shuttle of raket mee de ruimte in hoeven. Het probleem is echter dat de atmosfeer het zicht belemmert, waardoor plaatjes minder scherp worden dan ze zouden kunnen zijn.

De storing veroorzaakt door de atmosfeer is 's nachts duidelijk te zien. Als er al geen wolken zijn die de sterren voor ons verbergen, lijken ze een beetje heen en weer te bewegen, en variëren ze in helderheid: de sterren twinkelen. Met het blote oog lijkt dit effect misschien niet zo sterk, maar als je gedetailleerde plaatjes wilt maken van hemelse objecten beperkt dit je mogelijkheden sterk.



(a) **(b)**

Figuur A.1: Een voorbeeld van optische verstoringen. Links een vervormde ondergaande zon en rechts een licht-donker patroon op de bodem van een aquarium. Foto's van Wikipedia, door Mila Zinkova, vrijgegeven onder de GNU FDL, `http://www.gnu.org/copyleft/fdl.html`.

De reden van deze verstoringen zit in het feit dat de atmosfeer niet een goed gemengde massa is. De temperatuur verschilt van plek tot plek, en daardoor varieert de brekingsindex van de lucht ook. Deze eigenschap is namelijk afhankelijk van de temperatuur, en geeft de mate aan waarmee de lucht in de atmosfeer het licht afbuigt. Omdat de brekingsindex door de atmosfeer heen varieert, worden verschillende lichtstralen die op je telescoop vallen anders vervormd.

Een voorbeeld is de ondergaande zon, die soms sterk vervormd kan zijn als deze laag aan de horizon staat (zie Fig. **??**). Dit effect is extra sterk in deze situatie, waardoor het goed te zien is. Een soortgelijke vervorming kun je zien als je vanaf de kant van een zwembad onder water probeert te kijken, alles lijkt dan vervormd omdat het wateroppervlak niet mooi glad is. Daarnaast zie je op de bodem van zwembaden ook licht-donker patronen, die ook door het golvende wateroppervlak worden gevormd (zie Fig. **??**).

Het is hierbij belangrijk om op te merken dat het om het verschil in temperatuur is die het probleem veroorzaakt. Als de hele atmosfeer heel warm of koud is (zomer/winter), heb je deze namelijk niet. Ook als de oppervlakte van het water in een zwembad glas is, wordt het beeld niet vervormd, maar hooguit wat verplaatst.

A.2 DE STORING CORRIGEREN

Een methode om de storing te corrigeren heet adaptieve optiek. Adaptief omdat het systeem zich aanpast aan de storing, en optiek omdat het gebruikt maakt van spiegels en lenzen.

Het systeem kijkt constant naar de verstoring veroorzaakt door de atmosfeer met speciale sensoren. Dit gebeurt in hoog tempo omdat de verstoring erg snel verandert, typisch wordt de storing honderden tot duizenden keren per seconde gemeten. Als deze storing is gemeten zorgt een stukje hardware zoals een computer ervoor dat speciale spiegels worden vervormd om de storing te corrigeren. Deze spiegels zijn een beetje te vergelijken met lachspiegels, het oppervlak is vervormd waardoor als je in een lachspiegel kijkt het beeld ook vervormd is. De spiegels die bij adaptieve optiek worden gebruikt zijn echter speciaal ontworpen om de atmosferische storing te corrigeren, en kunnen heel snel vervormen om de storing bij te houden.

Zoals eerder gezegd verandert de storing in hoog tempo, waardoor ook de correctie snel moet worden toegepast. Naast deze snelheid is er nog een ander criterium dat belangrijk is voor deze correctie. Dit is de precisie waarmee de spiegel kan vervormen. Aan de achterkant van de spiegels zitten kleine pootjes die de spiegel vervormen. Hoe meer van deze pootjes er zijn, hoe preciezer de vorm van de spiegel bepaald kan worden en hoe beter de correctie is.

A.3 HET CONTROLESYSTEEM

Voordat de spiegel de storing kan corrigeren, moet eerst bepaald worden hoe deze vervormd moet worden. Dit gebeurt aan de hand van de metingen die de sensoren doen. De metingen hiervan worden ingelezen in een computer systeem, en vervolgens wordt de vervorming die de spiegel moet krijgen berekend.

Deze aansturing werd lange tijd echter door zeer specialistische systemen gedaan, welke moeilijk waren om te ontwerpen, en haast nog moeilijker om aan te passen mocht het systeem iets veranderd worden. Dit werd gedaan omdat normale PC's niet snel genoeg zijn om de berekeningen uit te voeren. Er moet immers ongeveer duizend keer per seconde ingewikkelde berekeningen uitgevoerd worden die voor PC's erg zwaar kunnen zijn.

De computers zijn nu op een punt dat ze snel genoeg zijn om deze berekeningen uit te voeren. Dit heeft dus als voordeel dat er geen specialistische hardware meer gebruikt hoeft te worden. Voordat dat kan is er echter software nodig die die berekeningen uit kan voeren.

A.4 FOAM

FOAM, wat een omgekeerde afkorting is voor *Modular Adaptive Optics Framework*, is een stuk software dat op dit soort systemen gebruikt kan worden. Het is een poging om de gebreken van eerdere software op te lossen, zodat het een universeel stuk software biedt dat op een uiteenlopend scala aan systemen gebruikt kan worden.

A.4.1 EISEN AAN DE SOFTWARE

Om dit te bereiken heb ik FOAM ontworpen met de volgende eisen in mijn achterhoofd:

Portabiliteit De software moet makkelijk op andere systemen te gebruiken zijn.

Schaalbaar FOAM moet de snelheid van grote computers efficiënt kunnen benutten.

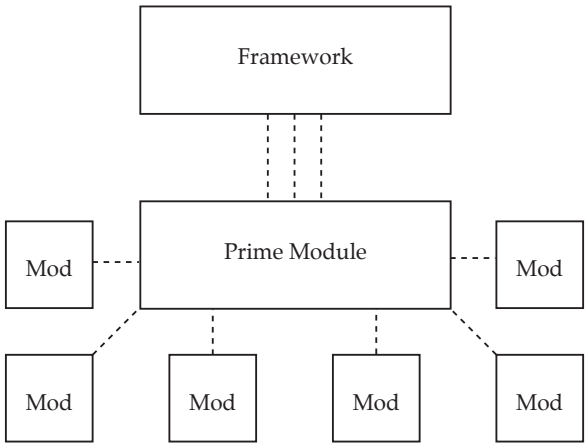
Uitbreidbaar Het moet makkelijk zijn om een systeem uit te breiden of aan te passen.

Bruikbaar Het moet eenvoudig te bedienen zijn door zowel mensen als andere software.

Publiek De software moet vrij beschikbaar zijn voor ieder die er wat mee wil doen.

A.4.2 DE SOFTWARE ZELF

FOAM is modulair opgebouwd, zodat veel stukken software makkelijk kunnen worden hergebruikt of worden vervangen, mocht dit nodig zijn. Dit is handig omdat op deze manier eenvoudig stukken hardware kunnen vervangen (bijvoorbeeld een andere spiegel) zonder dat het hele programma herschreven hoeft te worden. Ook kun je op deze manier makkelijk functionaliteit toevoegen. De globale architectuur van FOAM is weergegeven in Fig. **??**.



Figuur A.2: De FOAM architectuur. Het kernprogramma van FOAM, het *Framework* zorgt voor de basisfunctionaliteit. De verschillende modules kunnen dan via een *Prime Module* aan elkaar gekoppeld worden. Deze is per systeem anders, en maakt het mogelijk om FOAM op verschillende systemen te gebruiken.

De schaalbaarheid kan worden worden vergroot door meerdere dingen tegelijk uit te rekenen. Dit is op een module-niveau mogelijk door reken-intensieve taken parallel uit te voeren. Op dit moment werken de modules nog serieël, er gebeurt nog weinig tegelijkertijd. Door de modulaire opbouw is dit echter eenvoudig te implementeren, en dit staat ook op de planning om toegevoegd te worden.

Verder kan FOAM via een netwerkverbinding bestuurd worden, zodat gebruikers verbinding met het programma kunnen leggen, maar tegelijkertijd kan ook een ander stuk software dat moet samen werken met het adaptieve optiek-systeem met FOAM communiceren. Op deze manier kan het programma naadloos in een groter systeem functioneren, mocht dat nodig zijn.

Om het gebruik van FOAM te bevorderen is gekozen voor een open source licentie, en wel de GNU General Public Licentie[1]. In een notendop houdt dit in dat iedereen de software mag gebruiken en aanpassen, zolang de aangebrachte wijzigingen ook weer vrij beschikbaar zijn voor anderen. Op die manier profiteert iedereen van de verbeteringen die worden gemaakt. Ik heb voor deze licentie gekozen omdat dit waarschijnlijk het meest geschikt is in een academische omgeving, waar samenwerking erg belangrijk is.

[1] `http://www.gnu.org/licenses/gpl2.html`

A.4.3 FOAM IN EEN BREDERE CONTEXT

Zoals in de vorige paragraaf beschreven, is de architectuur van FOAM zo opgezet dat deze aan alle eisen die eerder gesteld werden kan voldoen, als dat al niet gebeurt. Ondanks dat FOAM an sich nog grotendeels serieël werkt, en dus nog niet goed schaalt op computers met veel verschillende rekeneenheden, kan dit dankzij de modulaire structuur later relatief eenvoudig ingebouwd worden.

Hopelijk kan FOAM met zijn modulaire en gestructureerde opbouw de weg vrij maken voor open en universele adaptieve optiek software. Omdat de telescopen steeds groter worden, zal adaptieve optiek in de toekomst ook steeds meer gebruikt worden. Hopelijk kan FOAM hier een steentje aan bijdragen.

B BRIEF FOAM MANUAL

This appendix provides a brief manual for end-users. Developers should also look at the other appendices. This manual assumes the implementation for FOAM on a specific system is already available, or that the software is only used in simulation mode.

B.1 PREREQUISITES

FOAM depends on certain libraries, which need to be installed prior to installation of FOAM itself. The libraries used are all freely available and comply with the 'open and free' mentality of FOAM.

The libraries necessary for basic functionality are:

- `libm` - the general purpose math library available on many systems,
- `pthread` - the POSIX threading library, used to provide threading, available on many systems,
- `libevent` 1.4 - an event multiplexing library available at `http://monkey.org/~provos/libevent/`,
- `SDL` 1.2.11 - a graphical library providing a means to display sensor output, available at `http://www.libsdl.org/`,
- `GSL` 1.8.2 - the GNU Scientific Library, which provides useful mathematical routines as well as an API to BLAS routines, available at `http://www.gnu.org/software/gsl/`,
- `GSLBlas` 1.2 - a BLAS to be linked to `GSL`. Can be a statically compiled version, such as `refblas3`, but `ATLAS` is recommended for its superior performance.

In certain cases, other libraries are necessary as well, including:

- `SDL_Image` 1.2 - used for image I/O, necessary during simulation. Available at `http://www.libsdl.org/projects/SDL_image/`,

- `fftw3` 3.1.2 - used for Fourier transforms to simulate imaging by lenses, necessary during simulation. Available at `http://www.fftw.org/`,
- `OpenGL` - used for fast display routines, useful because of reduced CPU load. Different OpenGL implementations exist, and any implementation adhering to the OpenGL specifications will suffice,
- `libGLU`, `libGLUT` - includes some extra OpenGL routines that FOAM depends on when using OpenGL.

FOAM does not require the latest versions of all libraries listed. The minimum version requirements of the libraries have not been established, as default versions supplied with distributions are typically new enough. The versions that FOAM has been verified to work with are listed above for reference. One exception to this is `libevent` for which at least version 1.4 is necessary which is not supplied with Debian Stable, and thus might be missing on other systems as well.

If the libraries are not available on a specific system, the `./configure` process will diagnose this and show a list of missing libraries necessary for certain build targets.

B.2 OBTAINING FOAM

To obtain FOAM, either download one of versions available at `http://dotdb.phys.uu.nl/~tim/foam/`, or obtain any version from the Subversion repository at `http://dotdb.phys.uu.nl/svn/foam/`. To check out the most recent version version of FOAM, use

```
svn co http://dotdb.phys.uu.nl/svn/foam/trunk/code/ foam-latest
```

which will download the code to `./foam-latest`. Alternatively, you can check out the revision associated with this thesis with

```
svn co http://dotdb.phys.uu.nl/svn/foam/tags/ foam-thesis
```

After checking out a version from the Subversion repository, you need to run

```
autoreconf -sfi
```

to install the `configure` script for your system. This is not necessary when downloading a pre-processed package of FOAM. After this step, continue with the installation.

B.3 INSTALLATION

FOAM uses the GNU build system and therefore configuration and installation of this software is the same as installing any other software using this system.

To compile FOAM you will need an ANSI C-compiler. After unpacking the distribution, the `Makefiles` can be prepared using the configure command

```
./configure
```

You can then build the program by typing

```
make
```

The default set of packages (i.e. sets of software and hardware modules) to be compiled, might not be possible or desired in your situation. The configure script takes extra command line options such that you can specify which packages you do or do not want to build. You can also specify the location where the FOAM binaries should be installed after compilation. For an overview of these options, run

```
./configure --help
```

If you change any compilation options you will need to remove any existing compiled files with

```
make clean
```

and re-run `make`.

If everything works as expected, you can install the software system-wide with

```
make install
```

B.4 RUNNING FOAM

The programs are built in the `bin/` sub-directory of where FOAM was, and can only be run from there by default, because the configuration files are expected to be in `../config`. Therefor, to run the software, `cd` into `bin/`, and call `./foamcs-<name>`, with `<name>` the name of the package you want to run.

After calling this program, FOAM will notify you of the initialisation process. During this phase, be sure to check for warnings and errors.

C FOAM SKELETON

This appendix shows the flow of FOAM at a functional level. This illustrates where the hooks are called and how these can be used to attach to.

Code snippet C.1: FOAM code skeleton

```
main() {
    modInitModule()              // Initialize modules,
                                 // memory, cameras etc.

    fork (startThread())         // Branch into two threads,
                                 // one for AO, one for user
                                 // I/O

    sockListen()                 // Read & process user I/O.
    exit
}

sockListen() {
   while (true) {                // In a continuous loop,
      parseCmd()                 // read the user input and
      modMessage()               // process it.
   }
}

startThread() {
   modPostInitModule()           // After threading, provide
                                 // an additional init hook.
}

listenLoop() {
   while (ptc.mode != shutdown) { // Run continuously until
                                 // shutdown.
      switch (ptc.mode) {        // Read the mode requested
                                 // and switch to that mode.
         case 'open': modeOpen()
         case 'closed': modeClosed()
         case 'calibration': modeCal()
      }
```

```
34      }
35      modStopModule()                 // Shutdown the modules.
36  }
37
38  modeOpen() {
39      modOpenInit()                   // Open loop init hook.
40
41      while (ptc.mode == AO_MODE_OPEN) {
42                                      // Loop until mode changes.
43          modOpenLoop()               // Actual work is done here.
44      }
45
46      modOpenFinish()                 // Clean up after open loop.
47  }
48
49  modeClosed() {
50      modClosedInit()                 // Closed loop works the
51                                      // same as open loop.
52
53      while (ptc.mode == AO_MODE_CLOSED) {
54          modClosedLoop()
55      }
56
57      modClosedFinish()
58  }
59
60  modeCal() {
61      modCalibrate()                  // Calibration mode provides
62                                      // only one hook.
63  }
```

Note that all overhead and housekeeping is not included here, and that the subroutine prototypes are incomplete. The hook functions that must be specified in the prime module are prefixed by `mod`.

At the start of the program, FOAM calls `modInitModule()`, which should be used to initialise any modules used, i.e. allocate memory, configure cameras and things like that. Directly afterwards, the program splits up into two (POSIX) threads. One thread is used for the adaptive optics routines that intensively use the modules and should be programmed by the developer, while the other provides standardised user input and output.

After threading, the AO thread runs through a second initialisation which is called after threading. This can be useful for initialisation of libraries that can only be initialised from the thread they will be called from later on (i.e. SDL). After that, the program decides what mode to switch to, depending on the value of the shared global variable `ptc.mode`.

For a detailed description of the functions used in the above example, please consult the FOAM reference documentation.

D FOAM SUMMARY

This appendix provides a short summary of FOAM. The data types and routines used in the FOAM framework are discussed in some detail, while the routines used modules are only listed here. For a details, consult the FOAM documentation.

D.1 FOAM FRAMEWORK

D.1.1 DATA TYPES

FOAM includes some data types used to run the control software. These are discussed here.

control_t

The following data type is used to track AO operations of FOAM. It contains relevant information on the hardware (wavefront sensors and -correctors, filter wheels) and contains some overhead information on the AO operations itself. A pointer to this struct is always passed on to the prime module functions, where it can be used to the liking of the programmer. A short description of each struct member is given as a comment. '`(user)`' prefixes denote fields that need to be filled out by the user (i.e. programmer), '`(foam)`' prefixes indicate fields that only FOAM worries about.

Code snippet D.1: `control_t` data type

```
1 typedef struct {
2     aomode_t mode;       // (user) the AO system mode
3     calmode_t calmode;   // (user) the calibration mode
4     time_t starttime;    // (foam) starting time of the system
5     time_t lasttime;     // (foam) tracks the framerate
6
7     long frames;         // (foam) number of frames parsed
8     long capped;         // (foam) number of frames captured
9     unsigned long saveimg; // (user) capture this many frames
```

```
10      float fps;          // (foam) the current FPS
11
12      int logfrac;        // (user) log certain info and debug
13                          // messages 1/logfrac frames.
14                          // 1/50 times), default 1000
15
16      FILE *misclog;      // (user) general purpose logfile
17      char *misclogfile;  // (user) filename for the above
18      bool domisclog;     // (user) toggle for the above logging
19
20      // WFS variables
21      int wfs_count;      // (user) number of WFSs
22      wfs_t *wfs;         // (user) wfs_t structs array
23
24      // WFC variables
25      int wfc_count;      // (user) number of WFCs
26      wfc_t *wfc;         // (user) wfc_t structs array
27
28      // Filterwheel variables
29      int fw_count;       // (user) number of fwheels
30      filtwheel_t *filter; // (user) filtwheel_t structs array
31
32  } control_t;
```

config_t

This struct stores things like the IP and port it should be listening on, the files to log error, info and debug messages to and whether or not to use the syslog facility. Again, '(user)' prefixes denote fields that need to be filled out by the user (i.e. programmer), '(foam)' prefixes indicate fields that only FOAM worries about.

Code snippet D.2: `config_t` data type

```
1   typedef struct { // config_t
2       char *listenip;             // (user) IP to listen on
3       int listenport;             // (user) port to listen on
4
5       char *infofile;             // (user) info logfile
6       FILE *infofd;               // (foam) associated FP
7       char *errfile;              // (user) error logfile
8       FILE *errfd;                // (foam) associated FP
9       char *debugfile;            // (user) debug logfile
10      FILE *debugfd;              // (foam) associated FP
11
12      bool use_syslog;            // (user) use syslog?
13      char *syslog_prepend;       // (user) syslogs prefix
14      bool use_stdout;            // (user) use stdout?
15      level_t loglevel;           // (user) loglevel
16
17      pthread_t threads[MAX_THREADS]; // (foam) thread ids array
18      int nthreads;               // (foam) number of threads
19  } config_t;
```

D.1.2 SUBROUTINES

The following hooks are available using the FOAM framework, and must be defined in the prime module. The prototypes are fixed, however.

- **int modInitModule(** `control_t * ptc, config_t * cs_config` **)**

  `modInitModule()` is run at the beginning of FOAM and is one of the cornerstones of the modular design. This hook provides a standardised means to initialise the prime module before anything has been done, like allocate memory, read in some configuration files, start cameras or anything else.

  Returns:
  `EXIT_SUCCESS` or `EXIT_FAILURE` depending on success or not.

- **int modPostInitModule(** `control_t * ptc, config_t * cs_config` **)**

  `modPostInitModule()` is run just after the framework split into two separate threads. This routine can be used to initialise things that are not entirely thread-safe, such as OpenGL.

  Returns:
  `EXIT_SUCCESS` or `EXIT_FAILURE` depending on success or not.

- **void modStopModule(** `control_t * ptc` **)**

  `modStopModule()` is run at the end of FOAM and can be used to wrap up things related to the module, like stopping cameras, setting filter wheels back or anything else. If this module fails, FOAM *will* exit anyway.

- **int modMessage(** `control_t * ptc, const client_t * client, char * list[], const int count` **)**

  This routine is run if a client sends a message over the socket.

  If the networking thread of the control software receives data, it will split the string received in space-separated words. After that, the routine will handle some default commands itself, like 'help', 'quit', 'shutdown' and others (see `parseCmd()`). If a command is not recognised, it is passed to this routine, which must then do something, or return 0 to indicate an unknown command. `parseCmd()` itself will then warn the user.

  Besides parsing commands, this routine must also provide help information about which commands are available in the first place. FOAM itself already provides some help on the basic functions, and after sending this to the user, `modMessage()` is called with `list[0] == 'help'` such that this routine can add its own help info.

  If a client sends 'help ⟨topic⟩', this is also passed to `modMessage()`, with `list[0] == 'help'` and `list[1] == '<topic>'`. It should then give information on that topic and return 1, or return 0 if it does not 'know' that topic.

For an example of `modMessage()`, see `foam_primemod-dummy.c`.

Returns:
`EXIT_SUCCESS` or `EXIT_FAILURE` depending on success or not.

• **int modCalibrate(** `control_t * ptc` **)**

This routine is run during calibration mode.

Slightly different from open and closed mode is the calibration mode. This mode does not have a loop which runs forever, but only calls `modCalibrate()` once. It is left to the programmer to decide what to do in this mode. `control_t` provides a flag (`.calmode`) to distinguish between different calibration modes.

Returns:
`EXIT_SUCCESS` or `EXIT_FAILURE` depending on success or not.

• **int modOpenInit(** `control_t * ptc` **)**

This routine is run once before entering open loop.

`modOpenInit()` should be provided by a module which does the necessary things just before open loop.

Returns:
`EXIT_SUCCESS` or `EXIT_FAILURE` depending on success or not.

• **int modOpenLoop(** `control_t * ptc` **)**

This routine is run during open loop.

`modOpenLoop()` should be provided by a module which does the necessary things in open loop.

Returns:
`EXIT_SUCCESS` or `EXIT_FAILURE` depending on success or not.

• **int modOpenFinish(** `control_t * ptc` **)**

This routine is run after open loop.

`modOpenFinish()` can be used to shut down cameras temporarily, i.e. to stop grabbing frames or something similar.

Returns:
`EXIT_SUCCESS` or `EXIT_FAILURE` depending on success or not.

• **int modClosedInit(** `control_t * ptc` **)**

This routine is run once before entering closed loop.

`modClosedInit()` should be provided by a module which does the necessary things just before closed loop.

Returns:
`EXIT_SUCCESS` or `EXIT_FAILURE` depending on success or not.

- **int modClosedLoop(** `control_t * ptc` **)**

  This routine is run during closed loop.

  `modClosedLoop()` should be provided by a module which does the necessary things in closed loop.

Returns:
`EXIT_SUCCESS` or `EXIT_FAILURE` depending on success or not.

- **int modClosedFinish(** `control_t * ptc` **)**

  This routine is run after closed loop.

  `modClosedFinish()` can be used to shut down cameras temporarily, i.e. to stop grabbing frames or something similar.

Returns:
`EXIT_SUCCESS` or `EXIT_FAILURE` depending on success or not.

D.2 OPENGL DISPLAY MODULE

File:	`foam_modules-dispgl.c`
Header:	`foam_modules-dispcommon.h`
Data type:	`mod_display_t`
Dependencies:	Shack-Hartmann module
Purpose:	Provide display routines such that wavefront sensor images can be displayed, along with some information such as a subaperture grid, the tracking subapertures in use, the displacements measured etc.

- **int displayInit(** `mod_display_t *disp` **)**
- **void displaySDLEvents(** `mod_display_t *disp` **)**
- **int displayDraw(** `wfs_t *wfsinfo, mod_display_t *disp, mod_sh_track_t *shtrack` **)**
- **void displayBeginDraw(** `mod_display_t *disp` **)**
- **int displayGSLImg(** `gsl_matrix_float *gslimg, mod_display_t *disp, int doscale` **)**
- **int displayImgByte(** `uint8_t *img, mod_display_t *disp, mod_sh_track_t *shtrack` **)**
- **int displayGrid(** `coord_t gridres, mod_display_t *disp` **)**

- **int displaySubaptLabels(** `mod_sh_track_t *shtrack, mod_display_t *disp` **)**
- **int displaySubapts(** `mod_sh_track_t *shtrack, mod_display_t *disp` **)**
- **int displayVecs(** `mod_sh_track_t *shtrack, mod_display_t *disp` **)**
- **void displayFinishDraw(** `mod_display_t *disp` **)**
- **int displayFinish(** `mod_display_t *disp` **)**

D.3 OKOTECH DEFORMABLE MIRROR MODULE

File:	`foam_modules-okodm.c`
Header:	`foam_modules-okodm.h`
Data type:	`mod_okodm_t`
Dependencies:	none
Purpose:	Provide an interface from the internal FOAM control signals to the 37 channel Okotech mirror controlled through a PCI-board.

- **int okoInitDM(** `mod_okodm_t *dm` **)**
- **int okoSetDM(** `gsl_vector_float *ctrl, mod_okodm_t *dm` **)**
- **int okoSetAllDM(** `mod_okodm_t *dm, int volt` **)**
- **int okoRstDM(** `mod_okodm_t *dm` **)**
- **int okoCloseDM(** `mod_okodm_t *dm` **)**

D.4 SHACK-HARTMANN MODULE

File:	`foam_modules-sh.c`
Header:	`foam_modules-sh.h`
Data type:	`mod_sh_track_t`
Dependencies:	none
Purpose:	Provide routines to perform analysis on Shack-Hartmann wavefront sensor output, such as tracking the centre of gravity displacement for subapertures, and calculating control signal output given these displacements.

- **int shInit(** `wfs_t *wfsinfo, mod_sh_track_t *shtrack` **)**
- **int shSelSubapts(** `void *image, foam_datat_t data, mod_sh_align_t align, mod_sh_track_t *shtrack, wfs_t *shwfs` **)**
- **int shCogTrack(** `void *image, foam_datat_t data, mod_sh_align_t align, mod_sh_track_t *shtrack, float *aver, float *max` **)**

- **int shCalcCtrl(** `control_t *ptc, mod_sh_track_t *shtrack, const int wfs, int nmodes` **)**

D.5 IMAGE I/O MODULE

File:	`foam_modules-img.c`
Header:	`foam_modules-img.h`
Data type:	`mod_imgbuf_t`
Dependencies:	none
Purpose:	Provide routines to read and write image files stored on disk. Can for example be used to read simulated wavefront images and actuator patterns, or can be used to write the wavefront sensor output to disk.

- **int imgInitBuf(** `mod_imgbuf_t *buf` **)**
- **int imgSaveToBuf(** `mod_imgbuf_t *buf, void *img, foam_datat_t datatype, coord_t res` **)**
- **int imgDumpBuf(** `mod_imgbuf_t *buf, control_t *ptc` **)**
- **void imgFreeBuf(** `mod_imgbuf_t *buf` **)**
- **void imgGetStats(** `void *img, foam_datat_t data, coord_t *size, int pixels, float *stats` **)**
- **int imgReadIMGArrByte(** `char *fname, uint8_t **img, coord_t *outres` **)**
- **int imgReadIMGSurf(** `char *fname, SDL_Surface **surf` **)**
- **int imgWritePGMArr(** `char *fname, void *img, foam_datat_t datatype, coord_t res, int maxval, int pgmtype` **)**
- **int imgWritePGMSurf(** `char *fname, SDL_Surface *img, int maxval, int pgmtype` **)**

D.6 DATA LOGGING MODULE

File:	`foam_modules-log.c`
Header:	`foam_modules-log.h`
Data type:	`mod_log_t`
Dependencies:	none
Purpose:	Used to log data to disk, which can afterwards be analyze.

- **int logInit(** `mod_log_t *log, control_t *ptc` **)**
- **void logGSLVecFloat(** `mod_log_t *log, gsl_vector_float *vec, int nelem, char *prep, char *app` **)**

- **void logVecFloat(** `mod_log_t *log, float *vec, int nelem, char *prep, char *app` **)**
- **void logMsg(** `mod_log_t *log, char *prep, char *msg, char *app` **)**
- **void logPTC(** `mod_log_t *log, control_t *ptc, char *prep` **)**
- **int logReset(** `mod_log_t *log, control_t *ptc` **)**
- **int logFinish(** `mod_log_t *log` **)**

D.7 SIMULATION MODULE

File:	`foam_modules-sim.c`
Header:	`foam_modules-sim.h`
Data type:	`mod_sim_t`
Dependencies:	Calibration module, Shack-Hartmann module, image I/O module
Purpose:	Can be used to perform dynamical adaptive optics systems simulations, starting with a simulated perturbed wavefront and consecutively simulating the telescope, the wavefront correctors, and the Shack-Hartmann wavefront sensor.

- **int simInit(** `mod_sim_t *simparams` **)**
- **int simFlat(** `mod_sim_t *simparams, int intensity` **)**
- **int simNoise(** `mod_sim_t *simparams, int var` **)**
- **int simWind(** `mod_sim_t *simparams` **)**
- **int simAtm(** `mod_sim_t *simparams` **)**
- **int simWFC(** `wfc_t *wfc, mod_sim_t *simparams` **)**
- **int simTT(** `mod_sim_t *simparams, gsl_vector_float *ctrl, int mode` **)**
- **int simDM(** `mod_sim_t *simparams, gsl_vector_float *ctrl, int nact, int mode, int niter` **)**
- **int simWFCError(** `mod_sim_t *simparams, wfc_t *wfc, int method, int period` **)**
- **int simTel(** `mod_sim_t *simparams` **)**
- **int simSHWFS(** `mod_sim_t *simparams, mod_sh_track_t *shwfs` **)**

D.8 DAQBOARD/2000 MODULE

File:	`foam_modules-daq2k.c`
Header:	`foam_modules-daq2k.h`
Data type:	`mod_daq2k_board_t`
Dependencies:	none
Purpose:	A module that provides routines to interface with the IOtech DaqBoard/2000 PCI-board, which provides some high-speed digital and analogue I/O ports.

- **int daq2kInit(** `mod_daq2k_board_t *board` **)**
- **void daq2kSetDACs(** `mod_daq2k_board_t *board, int val` **)**
- **void daq2kSetDAC(** `mod_daq2k_board_t *board, int chan, int val` **)**
- **int daq2kSetP2(** `mod_daq2k_board_t *board, int port, int bitpat` **)**
- **void daq2kClose(** `mod_daq2k_board_t *board` **)**

D.9 ITIFG MODULE

File:	`foam_modules-itifg.c`
Header:	`foam_modules-itifg.h`
Data type:	`mod_itifg_cam_t`, `mod_itifg_buf_t`
Dependencies:	none
Purpose:	A module interfacing with the ITI frame grabber driver which in turn is used to talk to frame grabber boards.

- **int itifgInitBoard(** `mod_itifg_cam_t *cam` **)**
- **int itifgInitBufs(** `mod_itifg_buf_t *buf, mod_itifg_cam_t *cam` **)**
- **int itifgInitGrab(** `mod_itifg_cam_t *cam` **)**
- **int itifgGetImg(** `mod_itifg_cam_t *cam, mod_itifg_buf_t *buf, struct timeval *timeout, void **newdata` **)**
- **int itifgStopGrab(** `mod_itifg_cam_t *cam` **)**
- **int itifgStopBufs(** `mod_itifg_buf_t *buf, mod_itifg_cam_t *cam` **)**
- **int itifgStopBoard(** `mod_itifg_cam_t *cam` **)**

D.10 CALIBRATION MODULE

File:	`foam_modules-calib.c`
Header:	`foam_modules-calib.h`
Data type:	none
Dependencies:	Shack-Hartmann module
Purpose:	Provides some calibration routines for Shack-Hartmann wavefront sensors.

- **int calibPinhole(** `control_t *ptc, int wfs, mod_sh_track_t *shtrack` **)**
- **int calibSVDGSL(** `control_t *ptc, int wfs, mod_sh_track_t *shtrack` **)**
- **int calibWFC(** `control_t *ptc, int wfs, mod_sh_track_t *shtrack` **)**