

Georgia State University

ScholarWorks @ Georgia State University

Physics and Astronomy Dissertations

Department of Physics and Astronomy

1-12-2006

A Prototype Visible to Near-Infrared Spectrograph for the CHARA Array, a Long-Baseline Stellar Interferometer

Chad Elliott Ogden

Follow this and additional works at: https://scholarworks.gsu.edu/phy_astr_diss



Part of the [Astrophysics and Astronomy Commons](#), and the [Physics Commons](#)

Recommended Citation

Ogden, Chad Elliott, "A Prototype Visible to Near-Infrared Spectrograph for the CHARA Array, a Long-Baseline Stellar Interferometer." Dissertation, Georgia State University, 2006.

doi: <https://doi.org/10.57709/1059806>

This Dissertation is brought to you for free and open access by the Department of Physics and Astronomy at ScholarWorks @ Georgia State University. It has been accepted for inclusion in Physics and Astronomy Dissertations by an authorized administrator of ScholarWorks @ Georgia State University. For more information, please contact scholarworks@gsu.edu.

**A PROTOTYPE
VISIBLE TO NEAR-INFRARED SPECTROGRAPH
FOR THE CHARA ARRAY,
A LONG-BASELINE STELLAR INTERFEROMETER**

by

CHAD E. OGDEN

Under the Direction of Theo A. ten Brummelaar

ABSTRACT

This work is a description of the Visible to near Infrared Spectrograph system for the CHARA array. The CHARA Array is a 6-telescope interferometer at the Mount Wilson Observatory in the mountains north of Pasadena, California. It combines the light from the 1-meter telescopes, and measures the visibility of the resulting interference fringes, which gives information about the source intensity distribution on the sky. The resolution of the instrument is proportional to the telescope separation, or baseline, divided by the wavelength. The VIS system operates in the 600-1000 nm wavelength range, a factor of 3 to 4 shorter than the standard operating wavelength at CHARA, $2.13\ \mu\text{m}$. An introduction to interferometry is given, with a description of the CHARA Array. The effects of diffraction through the system combined with atmospheric turbulence are described, and the results of a computer model given. The VIS system design is described, and results of the first fringe data are presented, including system visibility and throughput estimates.

INDEX WORDS: Interferometer, CHARA, Spectrograph, Visible, Infrared, Diffraction

**A PROTOTYPE
VISIBLE TO NEAR-INFRARED SPECTROGRAPH
FOR THE CHARA ARRAY,
A LONG-BASELINE STELLAR INTERFEROMETER**

by

CHAD E. OGDEN

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

Doctor of Philosophy

Georgia State University

2005

Copyright by
Chad Elliott Ogden
2005

**A PROTOTYPE
VISIBLE TO NEAR-INFRARED SPECTROGRAPH
FOR THE CHARA ARRAY,
A LONG-BASELINE STELLAR INTERFEROMETER**

by

CHAD E. OGDEN

Major Professor:	Theo A. ten Brummelaar
Committee:	Harold A. McAlister
	Douglas R. Gies
	William G. Bagnuolo
	Todd J. Henry
	Brian D. Thoms

Electronic Version Approved:

Office of Graduate Studies
College of Arts and Sciences
Georgia State University
December 2005

To my wife Rebecca,
and our children, Cyrus and Piper.

Acknowledgements

I must thank my wife Rebecca, who propped me up and encouraged me throughout the arduous and sometimes panic-stricken writing process. Without her support I would probably have given up and spent all of my time trying to escape reality.

Thanks also to my two children. Cyrus is my little pal who runs to hug me every evening when I walk in the door. Those hugs remind me what it's all about. I look forward to knowing the man he will become some day. My little Piper is just a baby right now, but she smiles when she looks at me, and that is more than enough.

Special thanks to my advisor,

Theo ten Brummelaar, whose scientific and technical prowess I will spend the rest of my life trying to attain. He also introduced me to the writing of Terry Pratchett, a reminder not to take things too seriously.

Thanks to the rest of my dissertation defense committee:

Hal McAlister, Bill Baguolo, Doug Gies, Todd Henry, and Brian Thoms.

Also thanks to:

My friend and office mate David Berger, who provided me with countless hours of shop talk, lent me a kind ear when I was frustrated, provided me with computer and T_EX support, and helped me keep sane via numerous dawn patrol surf sessions. The next wave is all yours, Dave.

The CHARA Array staff and denizens both current and former: Laszlo & Judit Sturmann, Nils Turner, Bob Cadman, PJ Goldfinger, Sandy Land, Steve Golden, Mike Fisher, Neda Safizadeh and more. You have been a pleasure to work with.

The people at JPL's Michelson Science Center. They have contributed invaluable support to the interferometry community via the Michelson Fellowships, Michelson Summer Schools, and by publishing *Principles of Long Baseline Stellar Interferometry* (Lawson, 2000b). I owe many thanks as well to those who contributed to the book. As a graduate student I needed an interferometry text which spoke to my level of understanding, and it did the job well. I use it often as a reference.

John McFarland and again to David Berger, who let me use their respective dissertation T_EX files as a template upon which to start my own, thus saving me many days of needless L^AT_EX hacking.

I was financially supported by the Michelson Fellowship Program funded by the

NASA Jet Propulsion Laboratory, the Department of Physics & Astronomy at Georgia State University, and the Center for High Angular Resolution Astronomy.

Contents

Acknowledgements	v
List of Tables	ix
List of Figures	x
List of Abbreviations	xiii
1 Introduction	2
1.1 A Brief Explanation of Stellar Interferometry	6
1.1.1 The Van Cittert-Zernike Theorem	6
1.1.2 A Simple Interferometer	11
1.1.3 Atmospheric Turbulence	22
1.2 The History of Stellar Interferometry	36
1.2.1 The Time Line	37
1.3 The CHARA Array	64
1.3.1 The Telescopes	69
1.3.2 The Light Pipes	71
1.3.3 The POPs	72
1.3.4 The OPLEs	75
1.3.5 The BRTs	77
1.3.6 The LDCs	79
1.3.7 The Beam Sampling System	81
1.3.8 The Beam Combining Laboratory	83
1.3.9 Tip-tilt Compensation Systems	86
1.3.10 Beam Combiners	87
2 CHARA Array Diffraction Effects in the Visible to Near-Infrared	93
2.1 Fresnel Zones and Diffraction	95
2.1.1 The Derivation of Fresnel Zones	98
2.1.2 Taking Beam Reduction Into Account	106
2.2 An Angular Spectrum Diffraction Model	117
2.2.1 The Angular Spectrum Approach to Diffraction	118
2.2.2 The Propagation Transfer Function	122

2.2.3	Execution of the Model	128
2.3	Model Predictions	136
2.3.1	Unequal Paths	137
2.3.2	Comparison of Unequal With Equal Paths	148
2.3.3	Conclusion	160
3	The CHARA VIS System Design	162
3.1	The VIS Table Layout	163
3.2	Dither Mirror	168
3.3	Beam Combining Splitter	174
3.4	Fiber Injectors	175
3.4.1	Fiber Injector Alignment	178
3.4.2	Fiber Switchboard	182
3.5	Spectrograph	183
3.5.1	Fiber Slit	184
3.5.2	Collimator and Camera Lenses	188
3.5.3	Prism	192
3.5.4	ARC CCD	195
3.6	Spectrograph Calibration	201
3.7	Software	203
4	Experimental Results	207
4.1	The Stars	207
4.2	The Fringe Data	211
4.2.1	Throughput	213
4.2.2	The Standard Data Reduction Method	215
4.2.3	Power Spectrum Peak Broadening Due To τ_o	219
4.2.4	Uncalibrated Visibilities From Fringe Packet Averages	221
4.2.5	System Visibility Estimates	228
4.3	Further Improvements	230
4.4	Conclusion	236
A	Diffraction Model IDL Code	
	(on CD-ROM)	237
A.1	The Master Files	237
A.1.1	Visibility and SNR Measurement With Variable Aperture Size and Propagation Distance, Averaged Over Many Model Iterations: chara_diff_vis.pro	237

A.1.2	A Single Iteration of Two Beam Diffraction and Interference Through The Sytem Which Generates Jpeg Images of All Array Values at Several Propagation Distances: chara_diff2.pro	243
A.2	Building the Turbulent Wavefront	261
A.2.1	Generate the Turbulent Phase Array: turbulent_phase.pro	261
A.2.2	Remove Tip/Tilt: remove_tip_tilt.pro	262
A.2.3	Apply the Phase Offset Array to the Wavefront Array: apply_phase.pro	264
A.3	Propagating the Wavefront Through the System	264
A.3.1	Apply the Telescope Aperture to the Wavefront: round_aperture.pro	264
A.3.2	Rescale the Arrays at a Beam Reducer: rescale_pupil.pro	265
A.3.3	Propagate the f_x, f_y Array a Given Distance: propagator.pro	265
A.4	Visibility and SNR Measurement	266
A.4.1	Calculate X and Flux in a Given Aperture Size and Position For Two Wavefront Arrays: vis_aperture.pro	266
A.5	General Utilities	267
A.5.1	Array Setup: setup_xy.pro	267
A.5.2	Update the f_x, f_y Plane After Changing the x, y Plane: update_fx fy.pro	267
A.5.3	Update the x, y Plane After Changing the f_x, f_y Plane: update_xy.pro	268
A.5.4	Normalize the Power in an Array: normalize_power.pro	269
A.5.5	Scale the Power in an Array to That of Another Array: scale_power.pro	270
A.5.6	Generate a Plot of V, SNR, or Flux Vs. Sub-Aperture Radius: plot_vis_stats.pro	270
A.5.7	Generate an Intensity Jpeg Plot of an Array: plot_intensity.pro	271
A.5.8	Read a Saved File of Visibility and SNR Data: read_vis_stats.pro	271

B	ARC CCD Control Code	
	(on CD-ROM)	273
B.1	The Header Files	273
B.1.1	arc_ccd.h	273
B.1.2	std_arc_ccd_functs.h	280
B.1.3	arc_ccd_channel_cal.h	282
B.2	The Real-Time Linux Module, arc_ccd_rt.c	284
B.3	The Non-Real-Time Control Code	332
B.3.1	arc_ccd.c	332
B.3.2	arc_ccd_background.c	333
B.3.3	arc_ccd_bias.c	337
B.3.4	arc_ccd_channels.c	341
B.3.5	arc_ccd_comms.c	388
B.3.6	arc_ccd_control.c	403
B.3.7	arc_ccd_spect.c	460
B.4	The Float ARCTAN Lookup Function For Real-Time Use	481
B.4.1	f_arctanlookup.h	481
B.4.2	f_arctanlookup.c	481
B.4.3	f_mktantable.c	483
B.4.4	f_tantable.h	485
B.5	The Data File Reader, arc_ccd_read.c	485
C	VIS Dither Control Code	
	(on CD-ROM)	494
C.1	The Header File, vis_dither.h	494
C.2	The Real-Time Linux Module, vis_dither_rt.c	497
C.3	The Non-Real Time Control Code, vis_dither_control.c	510
	References	523

List of Tables

1.1	τ_o and the characteristic atmospheric turbulence frequency as functions of r_o and wind velocity.	35
2.1	CHARA array values for z_{tel} and z_{brt}	114
2.2	CHARA Array values for α , β , and γ	115
2.3	CHARA Array typical values for r_o , $R_{1\text{ tel}}$, $R_{2\text{ tel}}$, $2\Delta R_{2\text{ tel}}$	115
3.1	Effective wave number κ and wavelength λ for VIS spectral channels.	202
4.1	Throughput calculated from Vega flux	213

List of Figures

1.1	The geometry of the van Cittert-Zernike theorem.	7
1.2	A plot of $2 \frac{J_1(x)}{x}$, the radially symmetric 2-dimensional Fourier transform of a uniform disc.	12
1.3	A simple stellar interferometer.	13
1.4	A plot of $\frac{\sin x}{x}$, the 1-dimensional Fourier transform of a “boxcar” function.	16
1.5	A CAD rendering of a 3D model of the CHARA Array and the surrounding grounds of the Mount Wilson Observatory.	65
1.6	A CAD rendering of a CHARA telescope 3D model.	70
1.7	The M7-M10 configuration.	72
1.8	The POPs, periscope, and OPLE.	73
1.9	The OPLE cart.	74
1.10	The Beam Reducing Telescopes and Beam Switchers.	78
1.11	The Longitudinal Dispersion Compensators.	79
1.12	The Beam Sampling System	82
1.13	The Beam Combining Laboratory layout	83
2.1	Point P in the beam combiner pupil plane corresponds to point O in the telescope pupil plane if diffraction effects can be neglected.	96
2.2	The geometry for determining the field at a point behind an aperture.	97
2.3	A seeing cell (dashed circles) will not contribute to the field at point P if its area is evenly divided between positive and negative Fresnel zones.	102
2.4	The geometry for determining the field at a point at the CHARA beam combiner.	106
2.5	A simple beam expander	111
2.6	Unit vector $\hat{\mathbf{k}}$ with angle ϕ from the z axis and angle θ from the x axis in the xy plane.	120
2.7	A plane wave traveling at an angle ϕ from the z axis only travels a distance of $z \cos \phi$ before the wave front reaches distance z on the z axis.	123
2.8	E1-W2 beams a and b vs. propagation distance. λ : 750 nm. r_o (550 nm): 10 cm.	139
2.9	E1-W2 combined quantities vs. propagation distance. λ : 750 nm. r_o (550 nm): 10 cm.	141

2.10	r_o dependence of E1-W2 combined quantities at the VIS table beam combiner. λ : 750 nm.	142
2.11	λ dependence of E1-W2 combined quantities at the VIS table beam combiner. $r_o(550\text{ nm})$: 10 cm.	143
2.12	Sub-apertures taken at the beam combiner.	144
2.13	E1-W2 V_{rms} (top) and SNR (bottom) for poor seeing. $r_o(550\text{ nm})$: 5 cm. λ : 750 nm.	145
2.14	E1-W2 V_{rms} (top) and SNR (bottom) for good seeing. $r_o(550\text{ nm})$: 10 cm. λ : 750 nm.	147
2.15	E1-W2 V_{rms} (top) and SNR (bottom) for excellent seeing. $r_o(550\text{ nm})$: 15 cm. λ : 750 nm.	149
2.16	S1-S2 SNR (top) compared to E1-W2 SNR (bottom) for poor seeing. $r_o(550\text{ nm})$: 5 cm. λ : 750 nm.	150
2.17	E1-W2 and S1-S2 SNR at the VIS table combiner vs. sub-aperture radius over a range of r_o . λ : 750 nm.	152
2.18	E1-W2 V_{rms} (top) and SNR (bottom) vs. $r_o(550\text{ nm})$ for various sub-aperture sizes. λ : 750 nm.	154
2.19	S1-S2 V_{rms} (top) and SNR (bottom) vs. $r_o(550\text{ nm})$ for various sub-aperture sizes. λ : 750 nm.	155
2.20	E1-W2 and S1-S2 SNR at the VIS table combiner vs. sub-aperture radius over a range of λ . $r_o(550\text{ nm})$: 10 cm.	157
2.21	E1-W2 V_{rms} (top) and SNR (bottom) vs. λ for various sub-aperture sizes. $r_o(550\text{ nm})$: 10 cm.	158
2.22	S1-S2 V_{rms} (top) and SNR (bottom) vs. λ for various sub-aperture sizes. $r_o(550\text{ nm})$: 10 cm.	159
3.1	The VIS table layout.	164
3.2	The VIS dither mirror.	169
3.3	The VIS dither mirror hysteresis curve.	170
3.4	The geometry of the OPD introduced by moving the dither mirror.	171
3.5	The VIS beam combining splitter.	174
3.6	The fiber injector.	176
3.7	Fiber injector alignment.	178
3.8	Reflected and projected beams from the fiber injector.	182
3.9	The spectrograph, top and side view.	184
3.10	The fiber slit viewed face-on.	185
3.11	Vertical spreading of collimated beams due to slit length, side view.	190
3.12	The ARC CCD photon transfer curve.	199
3.13	The ARC CCD electronics cabinet.	200
4.1	Normalized, filtered fringes from HR 7924, Deneb.	217
4.2	Power spectrum of fringe data from HR 7001, Vega	220
4.3	Uncalibrated Visibility vs. B/λ for HR 5191, Alcaid.	224

4.4	Uncalibrated Visibility vs. B/λ for HR 7001, Vega.	226
4.5	Uncalibrated Visibility vs. B/λ for HR 7924, Deneb.	227
4.6	An unusually high visibility fringe scan from HR 7924, Deneb	229
4.7	The VIS system visibility vs. λ from observations of Alcaid, Deneb, and Vega.	230

List of Abbreviations

ARC	Astronomical Research Cameras, Inc.
BC	Beam Combiner
BCL	Beam Combining Laboratory
BRT	Beam Reducing Telescope
BSF	Beam Synthesis Facility
CCD	Charge Coupled Device
CHARA	Center for High Angular Resolution Astronomy
DMA	Direct Memory Access
FFT	Fast Fourier Transform
FLUOR	Fiber Linked Unit for Optical Recombination
FIFO	First In, First Out
FWHM	Full Width, Half Maximum
GUI	Graphical User Interface
IDL	Interactive Data Language
IR	Infrared
ISA	Industry Standard Architecture
LBTI	Large Binocular Telescope Interferometer
LDC	Longitudinal Dispersion Compensator
MIRC	Michigan Infrared Combiner
NII	Near Infrared Imaging
NIRO	Near InfraRed Observer (CHARA's IR camera)
NPOI	Navy Prototype Optical Interferometer
OPD	Optical Path Difference
OPLE	Optical Path Length Equalization
PCI	Peripheral Component Interconnect
PTI	Palomar Testbest Interferometer
PoP	Pipes of Pan (CHARA's fixed delay lines)
PZT	Piezoelectric Transducer
RT	Real-Time
SNR	Signal-to-Noise Ratio
SUSI	Sydney University Stellar Interferometer
TECA	ThermoElectric Cooling America Corporation
UT	Universal Time
VIS	Visible to near-Infrared Spectrograph
ZPD	Zero Path Difference

Nobody climbs mountains for scientific reasons.

Science is used to raise money for the expeditions,
but you really climb for the hell of it.

— Edmund Hillary

Chapter 1

Introduction

Much of modern astronomy is driven by a desire to look ever deeper into space. As we see farther away, we see back in time to the period just after the Big Bang, when the universe first cooled enough to become transparent. We also have recently become aware of the existence of planets around many other stars. The possibility that there may be another planet like Earth in our immediate vicinity has captured the imaginations of many.

If we want to study these far-off objects, we need to be able to image them with high detail. This requires large telescopes. The angular resolution of a telescope, when there is no perturbation to the light waves entering it and the optics are geometrically perfect, is

$$\theta = 1.22 \frac{\lambda}{D} ,$$

where θ is the angle of the smallest resolvable detail in radians, λ is the wavelength of the light, and D is the diameter of the telescope aperture, which is assumed to be circular.

Stars subtend very small angles in the sky. The one which appears largest to us is Betelgeuse, the brightest star in Orion. It is only 0.047 seconds of arc in diameter,

or 1.31×10^{-5} degrees, the angular size of a dime at a distance of 46 miles.

A telescope with a diameter of 3 meters is required to resolve such a small angle. As we desire to see things in more detail, the size of our telescopes must increase. At some point, building a larger telescope becomes prohibitively expensive.

A technique has been developed, first proposed by Fizeau in 1868 and successfully implemented by Michelson in 1891, by which the light from two apertures may be mixed together, and the resolution information obtained is equivalent to that of a telescope whose diameter is equal to the separation between the apertures. The light from the two apertures forms interference fringes, light and dark bands, across the image of a star. The contrast of those fringes decreases as the separation of the apertures increases. The rate at which the fringe contrast, or *visibility*, decreases depends directly on the angular size of the star in the direction of the line separating the two apertures. By measuring the fringe visibility as a function of aperture separation, the size of a star may be measured. An explanation of the math is given in section 1.1.1 of this chapter.

This method may be extended beyond stars, and any source may be imaged, provided the fringe visibility (and fringe position with respect to the center of the image) is known over an extensive enough range of aperture separations and orientation angles. This technique, known as *aperture synthesis imaging*, has been used by radio interferometry for some time to build a large radio telescope out of an array of many smaller radio antennas.

The measurement of interference fringes, called *interferometry*, requires that the instrument be stable on the order of a fraction of the wavelength (denoted λ) of the light being detected. Radio wavelengths are on the order of meters and longer, so

radio interferometers do not require high mechanical tolerances to build. In addition, the radio waves oscillate slowly enough that their wave patterns may be recorded directly, then mixed together electronically to find the interference fringes.

Visible light has a much shorter wavelength, 550 nm at the peak response of the human eye. An interferometer which combines visible light must be stable on the order of 100 nm or less, and the technology to control distances to this precision over a scale of tens or hundreds of meters has only become possible in the last 40 years with the advent of lasers and computers.

In the last few decades, interferometric arrays have been built on increasingly larger scales, measuring fringes from stars to determine their angular diameters, as well as measuring the relative positions of binary stars. These methods provide a geometric determination of star diameters and binary star orbits, when combined with parallaxes and radial velocity data. Direct determinations of star sizes (and masses in the case of spectroscopic binaries) provide a reality check for astrophysical models.

In the long run, interferometric methods will provide a powerful approach to the search for earth-like planets around other stars. Interferometric arrays provide the resolution of a telescope well beyond the practical limits of single telescope size, at the expense of increased complexity in the precision control of the optical paths involved.

Georgia State University's Center for High Angular Resolution Astronomy (CHARA) Array is one such interferometer, located at the Mount Wilson Observatory just north of Pasadena, California. The CHARA Array utilizes six 1-meter telescopes at separations up to 330 m to measure stellar diameters, binary star orbits, and other objects with simple morphologies. The CHARA array has previously operated only at wave-

lengths of $1.6\ \mu\text{m}$ (H-band) and $2.13\ \mu\text{m}$ (K'-band). This dissertation describes the design and operating characteristics of the CHARA Visible to near Infrared Spectrograph system, which operates in the 600-1000 nm wavelength range, and will increase the array's angular resolving power by a factor of 3 to 4 when used on the longest baseline.

Chapter 1 is an introduction to interferometry. Section 1.1 covers the math behind aperture synthesis, a description of the basic design of an interferometer and how it works, and a discussion of the effects of atmospheric turbulence on the interference fringes. Section 1.2 lays out the history of stellar interferometry at visible and infrared wavelengths, from Fizeau to the present day. Section 1.3 is a description of the CHARA array and its subsystems.

Chapter 2 is a discussion of the effects of diffraction in the wavefront as the light propagates from the telescopes to the beam combiner. Section 2.1 covers the basic math and geometry, Section 2.2 describes the method used to mathematically model diffraction effects in the CHARA array, and Section 2.3 gives the results of the diffraction model.

Chapter 3 focuses on the design of the VIS system. The various sections cover components of the system.

Chapter 4 gives details of the first fringes data obtained with the VIS system. Section 4.1 is a summary of the three stars, Alcaid, Vega, and Deneb, whose fringe data was used. Section 4.2 presents an analysis of the fringe data. System throughput and system visibility estimates are presented. Section 4.3 gives recommendations on how the VIS system may be improved, and Section 4.4 is a brief conclusion.

1.1 A Brief Explanation of Stellar Interferometry

Stellar interferometry is the measurement of star images via the use of interference phenomenon. Starlight is commonly collected via two or more telescopes or siderostat mirrors, then combined in such a way that the light beams interfere to form a fringe pattern. A fringe pattern is a series of bright and dark alternating bands; bright where the starlight interferes constructively, dark where the starlight interferes destructively. The contrast and position of the fringes contain information about the brightness distribution across the star being observed.

1.1.1 The Van Cittert-Zernike Theorem

When light from some source is gathered at two different points and then caused to interfere, the relationship between the interference fringes and the brightness distribution across the source was derived by van Cittert (1934) and Zernike (1938), and is referred to as the *van Cittert-Zernike theorem*.

Let us first make some geometrical assumptions before stating the theorem. Figure 1.1 represents an incoherent light source σ centered around point O , whose light is sampled at two points (e.g. telescopes) P_1 and P_2 . Let P_1 and P_2 and O' lie in a plane, the plane being perpendicular to the line connecting O and O' . Let O' be the origin of the plane containing P_1 and P_2 , and the coordinates X, Y be used to indicate the position of points in that plane. Let us also assume that σ lies in a plane parallel to the X, Y plane, with O as the plane's origin, and the coordinates ξ, η used to indicate the position of points in the plane.

The source σ may be thought of as a collection of smaller sources $d\sigma$, and the

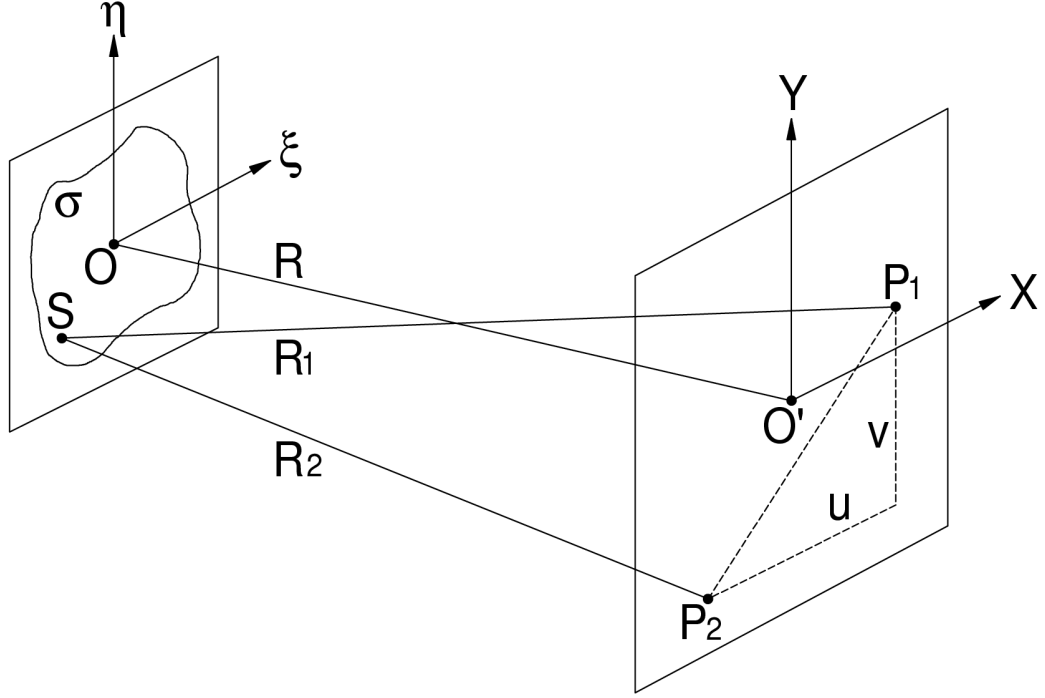


Figure 1.1: The geometry of the van Cittert-Zernike theorem.

A source σ in the ξ, η plane may be described as the sum of smaller sources $d\sigma$ at point S , with brightness $I(S)$. Light from $d\sigma$ is gathered at points P_1 and P_2 , which are distances R_1 and R_2 from point S . P_1 and P_2 lie in the X, Y plane, which is distance R from the ξ, η plane. The relative separation between P_1 and P_2 may be expressed in the coordinates u, v .

The van Cittert-Zernike theorem states that the interference fringes caused by combining the light from P_1 and P_2 have a complex degree of coherence $\gamma(u, v)$ which is the Fourier transform of $I(S)$.

effect of the entire source found as a sum over all of its parts.

Let the average intensity at point P_i be

$$I_i = \int_{\sigma} \frac{I(S)}{R_i^2} dS, \quad (1.1)$$

where $I(S)$ is the function which describes the intensity per unit area coming from

the portion of the source $d\sigma$ at point S in the ξ, η plane. The average intensity is simply the sum of all the intensities of all the sources $d\sigma$ over the entire source σ . The *mutual* intensity $J(P_1, P_2)$ is the average intensity measured when combining the light from points P_1 and P_2 .

The fringe contrast, also called the *visibility* V , is defined as

$$V = \frac{I_{\max} - I_{\min}}{I_{\max} + I_{\min}} = \frac{2\sqrt{I_1 I_2}}{I_1 + I_2} |\gamma_{12}|. \quad (1.2)$$

The complex number γ_{12} is called the *complex degree of coherence*.

The van Cittert-Zernike theorem states that

$$J(P_1, P_2) = \int_{\sigma} I(S) \frac{e^{i2\pi \bar{\kappa} (R_1 - R_2)}}{R_1 R_2} dS \quad (1.3)$$

and

$$\gamma_{12} = \frac{1}{\sqrt{I_1 I_2}} \int_{\sigma} I(S) \frac{e^{i2\pi \bar{\kappa} (R_1 - R_2)}}{R_1 R_2} dS, \quad (1.4)$$

where $\bar{\kappa}$ is $1/\bar{\lambda}$, $\bar{\lambda}$ in turn being the mean wavelength.

Equation 1.4 is in the same form as the equation which describes the complex amplitude in a diffraction pattern on a screen in the X, Y plane. Specifically, *if the light source σ is replaced by a σ -shaped aperture at which there is spherical wave with intensity distribution $I(S)$ converging on point P_1 , then the complex amplitude of the electric field as a function of the separation $\overline{P_2 P_1}$ is identical to γ_{12}* . The diffraction formulation to which this refers is that predicted by the Huygens-Fresnel principle, and is covered in Section 2.1.

The most important implication of the van Cittert-Zernike theorem is found when

the R_1 and R_2 dependence is removed from equation 1.4.

$$R_i = (X_i - \xi)^2 + (Y_i - \eta)^2 + R^2$$

which can be expressed via a Taylor expansion as

$$R_i \approx R + \frac{(X_i - \xi)^2 + (Y_i - \eta)^2}{2R} \quad (1.5)$$

by using only the 0th and 1st order terms. The distance between the points P_1 and P_2 is therefore

$$\begin{aligned} R_1 - R_2 &\approx \frac{(X_1 - \xi)^2 - (X_2 - \xi)^2 + (Y_1 - \eta)^2 - (Y_2 - \eta)^2}{2R} \\ &= \frac{(X_1^2 + Y_1^2) - (X_2^2 + Y_2^2)}{2R} - \frac{(X_1 - X_2)\xi + (Y_1 - Y_2)\eta}{R} . \end{aligned} \quad (1.6)$$

Define the following variables:

$$\begin{aligned} u &= \bar{\kappa}(X_1 - X_2) , \\ v &= \bar{\kappa}(Y_1 - Y_2) , \\ \alpha &= \frac{\xi}{R} , \\ \beta &= \frac{\eta}{R} . \end{aligned} \quad (1.7)$$

u and v are dimensionless quantities representing the separation between points P_1 and P_2 divided by the mean wavelength. α and β are the angular sizes of ξ and η measured from point O' . The definitions of α and β assume that they are small angles, such that $\tan \alpha \approx \alpha$ and the same for β .

By assuming that ξ, η , X_i , and Y_i are much smaller than R , R_1 and R_2 may be

replaced with R in the denominator of the integral in (1.4). After substituting in (1.6) and (1.7), the complex degree of coherence becomes

$$\gamma_{12} = \frac{e^{i\psi} \iint_{\sigma} I(\alpha, \beta) e^{-i 2\pi (u\alpha + v\beta)} d\alpha d\beta}{\iint_{\sigma} I(\alpha, \beta) d\alpha d\beta} \quad (1.8)$$

where

$$\psi = \frac{2\pi\bar{\kappa}[(X_1^2 + Y_1^2) - (X_2^2 + Y_2^2)]}{2R}. \quad (1.9)$$

Note that the intensity distribution $I(\alpha, \beta)$ is now expressed in terms of the angular size of ξ and η , which is more useful when working with distant sources where R is not known. ψ is simply a phase shift induced by the difference in the distances from point O to points P_1 and P_2 . ψ is equivalent to $2\pi\bar{\kappa}(\overline{OP_1} - \overline{OP_2})$.

Equation 1.8 is a powerful result. It states that *the complex degree of coherence γ_{12} between P_1 and P_2 is the normalized Fourier transform of the source brightness distribution $I(\alpha, \beta)$ and a small phase term.* This is the keystone of imaging interferometry. By measuring the complex degree of coherence throughout the u, v plane, we may build up the function $\gamma(u, v)$. Then we inverse transform $\gamma(u, v)$ to obtain $I(\alpha, \beta)$, an image of the object whose fringes we have been measuring.

The most common intensity distribution of interest to the astronomer is that of a star, which may be approximated by a disc of uniform intensity. To take advantage of the circular symmetry of the disc, let

$$\rho = \sqrt{\alpha^2 + \beta^2}$$

and

$$r = \sqrt{u^2 + v^2} .$$

If $I(\rho)$ is 1 for $\rho \leq \rho_o$ and 0 everywhere else, then its (un-normalized) Fourier transform is

$$\gamma(r) = \frac{J_1(\pi \rho_o r)}{\pi \rho_o r} . \quad (1.10)$$

$J_1(x)$ is the first order Bessel function. Figure 1.2 is a plot of $\frac{2J_1(x)}{x}$. It is a decaying, oscillating function which starts equal to 1 at $x = 0$, then dives to 0 at $x = 1.22$, oscillating about 0 as its amplitude decays with increasing x . By measuring γ across a range of r , one may find a best fit form of (1.10) to the data, and solve for $2\rho_o$, the angular diameter of the star. Real stars exhibit features which deviate from the uniform disk model, such as limb darkening, hot and cool spots, and oblateness. These features change $I(\alpha, \beta)$ and hence $\gamma(u, v)$.

In general, P_1 and P_2 do not actually have to lie in a plane perpendicular to $\overline{O'O}$. The problem may still be broken down into the component of $R_1 - R_2$ parallel to the X, Y plane, and the component parallel to $\overline{O'O}$. The latter component simply introduces a delay between the P_1 and P_2 signals, which does not affect the complex visibility of the fringes.

For a detailed derivation of the van Cittert-Zernike theorem, consult Born and Wolf (1998, Section 10.4), upon which this summary is based.

1.1.2 A Simple Interferometer

In order to measure $\gamma(u, v)$, we need a device which can gather light from at least 2 points in the u, v plane and combine the light to form interference fringes.

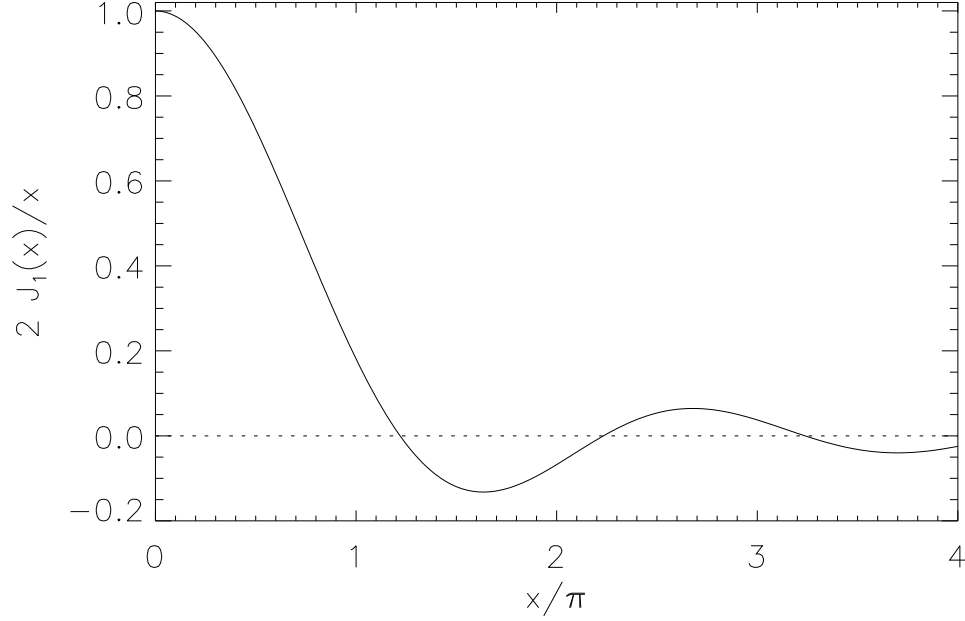


Figure 1.2: A plot of $2 \frac{J_1(x)}{x}$, the radially symmetric 2-dimensional Fourier transform of a uniform disc. The horizontal axis is in units of π radians. The first three zeros are at $x = 1.2196\pi$, 2.2332π , and 3.2384π .

Figure 1.3 is a diagram of a basic stellar interferometer. Light coming from a star at angle θ from the vertical is collected at the points P_1 and P_2 via telescopes or siderostats.

The separation between points P_1 and P_2 is called the baseline, B . The van Cittert-Zernike theorem gives the relationship of $\gamma(u, v)$ to the brightness distribution $I(\alpha, \beta)$ of the source. The u, v plane is assumed to be parallel to the plane which contains the source (referred to as the plane of the sky), and perpendicular to the incoming starlight. The component of B perpendicular to the incoming starlight is $B \cos \theta$, and that is the value used to compute the u, v coordinates of the $P_1 - P_2$ separation.

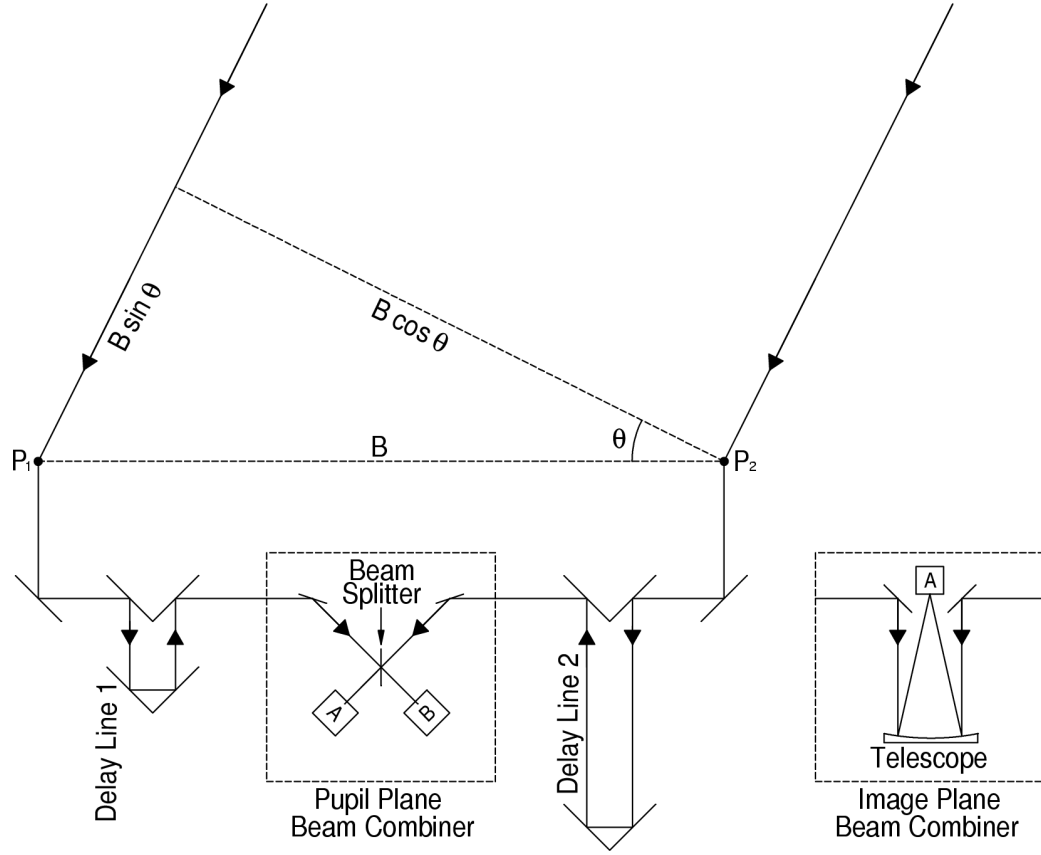


Figure 1.3: A simple stellar interferometer.

Light from a star at angle θ from the vertical is collected at points P_1 and P_2 by telescopes or siderostats. B is the baseline, the separation between P_1 and P_2 . The projection of B onto a plane perpendicular to the incoming starlight is $B \cos \theta$.

The starlight reaches P_2 before P_1 , with the extra path length $B \sin \theta$ in the #1 arm of the interferometer. Delay lines of variable length are used to remove the path difference. The path in delay line 2 must be $B \sin \theta$ longer than delay line 1 to equalize the total path from the star to the combiner for both arms of the interferometer.

The two methods of beam combination are described in the text.

The starlight reaches P_2 before P_1 because the two points do not lie in a plane perpendicular to the incoming light. The light traveling to P_1 must go an extra distance of $B \sin \theta$ before it reaches the collector. The extra optical path in the #1

arm of the interferometer must be compensated for in order to detect fringes.

To understand the need for equal paths for fringe detection, consider the effect of multiple wavelengths of light on the fringes. Let a simplified equation describing the fringes be

$$I_{12}(z) = 1 + \gamma_{12} \cos \left(2\pi \frac{z}{\lambda} + \phi \right) \quad (1.11)$$

where λ is the wavelength, z is the total optical path difference (OPD) between the two arms of the interferometer, and ϕ is a phase term due to the star, instrument, and atmosphere. It is assumed that the intensity of the starlight is 1 in both beams.

When the OPD is zero, the phase of the fringe is ϕ regardless of the wavelength of the starlight. When the OPD is not zero, the phase of the fringes depends upon the wavelength. Say the OPD is 1200 nm. If λ is 600 nm, the phase of the fringe is $4\pi + \phi$, or just ϕ . If λ is 900 nm, however, the fringe phase is $3\pi + \phi$, or just $\pi + \phi$. The 900 nm fringes have exactly the opposite phase of the 600 nm fringes. The effect of using a range of wavelengths is that the fringes of different wavelengths cancel each other out after a small OPD.

Born and Wolf (1998, Section 10.4.1) show that the complex degree of coherence measured at a single point, with delay τ is

$$\gamma_{11}(\tau) = \frac{\int_0^\infty S(\nu) e^{-i2\pi\nu\tau} d\nu}{\int_0^\infty S(\nu) d\nu}, \quad (1.12)$$

where ν is the frequency of the light, and $S(\nu)$ is the spectral density, or the intensity distribution as a function of frequency. Equation 1.12 shows that $\gamma_{11}(\tau)$ is the normalized one-dimensional Fourier transform of $S(\nu)$.

Assume $S(\nu)$ is a “boxcar” function,

$$S(\nu) = \begin{cases} 1, & |\nu - \nu_o| < \frac{1}{2}\Delta\nu, \\ \frac{1}{2}, & |\nu - \nu_o| = \frac{1}{2}\Delta\nu, \\ 0, & |\nu - \nu_o| > \frac{1}{2}\Delta\nu, \end{cases} \quad (1.13)$$

centered at ν_o and having a spectral bandwidth $\Delta\nu$. The Fourier transform of $S(\nu)$ is

$$\gamma_{11}(\tau) = \frac{\sin(\pi\Delta\nu\tau)}{\pi\Delta\nu\tau}. \quad (1.14)$$

A more concise form of notation for $\frac{\sin x}{x}$ is $\text{sinc}(x)$. Let the wavenumber κ be $\frac{1}{\lambda}$. Plugging $c\Delta\kappa = \Delta\nu$ and $\frac{z}{c} = \tau$ into (1.14) makes it

$$\gamma_{11}(z) = \text{sinc}(\pi\Delta\kappa z). \quad (1.15)$$

Thus we see that for an interferometer with bandwidth $\Delta\kappa$, centered at κ_o , the coherence of the fringes is modulated by a sinc function envelope. The distance $\frac{1}{\Delta\kappa}$ is known as the *coherence length*, commonly noted as Λ_{coh} . The sinc function is plotted in Figure 1.4. It starts out with a value of 1 when its argument equals 0, then dives to 0 when $z = \Lambda_{\text{coh}}$. Because it is a sine function with a $\frac{1}{x}$ envelope, sinc has decaying oscillations outside $z = \Lambda_{\text{coh}}$.

Equation 1.11 may now be written more generally as

$$I_{12}(z) = 1 + \frac{2\sqrt{I_1 I_2}}{I_1 + I_2} \gamma_{12} \text{sinc}(\pi\Delta\kappa z) \cos(2\pi\kappa_o z + \phi) \quad (1.16)$$

where beams 1 and 2 are assumed to have the normalized intensity $\frac{I_i}{I_1 + I_2}$, $i = 1, 2$. We

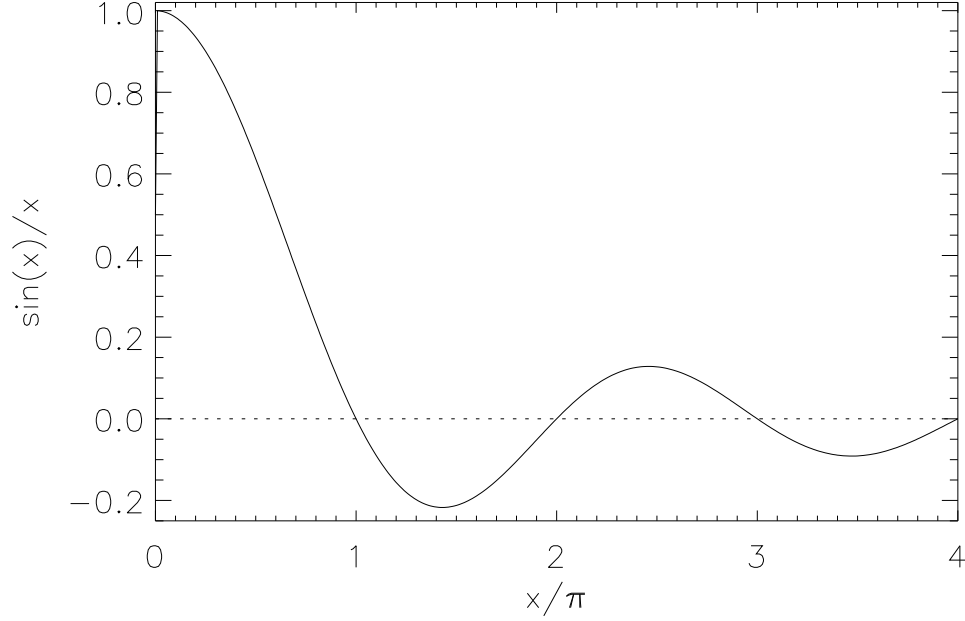


Figure 1.4: A plot of $\frac{\sin x}{x}$, the 1-dimensional Fourier transform of a boxcar function. The horizontal axis is in units of π radians. The zeros are at $x = n\pi$, where n is any integer.

will therefore only see fringes in an envelope $2\Lambda_{\text{coh}}$ wide, centered on zero OPD, plus a few small side lobes from the ringing of the sinc function.

The mechanism for changing the OPD is a pair of movable delay lines, one in each arm of the interferometer (see Figure 1.3). The typical design for a delay line is a cart which rides on rails, upon which is mounted a retroreflector, whose return beam is always parallel to its incoming beam. The cart may move along the rails, changing the optical delay, without changing the angle of the incoming or outgoing beams. When delay line 2 has $B \sin \theta$ more delay than delay line 1, it cancels out the extra $B \sin \theta$ path in arm #1 due to the relative positions of P_1 and P_2 with respect

to the star. Once the OPD is equalized, the beams are ready for combination.

Two methods of beam combination are shown in Figure 1.3. Each has its merits and drawbacks. A discussion of each follows.

Pupil Plane Beam Combination

Pupil plane beam combination interferes the two input beams by overlapping them. The cross-section of the beam is also known as the pupil plane, hence the name of this combination scheme.

The two input beams are directed onto opposite sides of a beam splitter. The splitter is of the 50-50 type, meaning that 50% of the incident light is reflected, and 50% is transmitted. In Figure 1.3, beam 1 hits the left side of the splitter, and the reflected component of beam 1 passes into detector *A*. The transmitted component of beam 1 continues straight into detector *B*. Likewise, beam 2's transmitted light hits detector *A*, and its reflected light hits detector *B*. Detectors *A* and *B* each have input beams whose intensities are half beam 1, half beam 2. In practice, the split may not be perfectly 50-50, and I_1 may be different than I_2 , but these inequities may still be calibrated.

Because light acquires a π phase shift when it reflects, the fringe pattern measured in detector *B* is π out of phase with the fringes in detector *A*. In other words, if *A* sees a bright fringe, *B* sees a dark fringe. The total power is conserved.

Pupil plane combination only measures the fringe pattern at one particular OPD at a time, so in order to measure the entire fringe pattern $I_{12}(z)$, the delay lines must be scanned through a small range of OPD, typically larger than a few Λ_{coh} , on either side of zero OPD.

The beneficial side of this configuration is that as few as 1 pixel per detector is required. Each detector's input beam may be focused down to small spot on the detector, concentrating all of the input beam's light onto one or a few pixels. This helps the signal climb above detector noise quickly.

The down side of pupil plane combination is that exposures at several OPD positions are required to adequately sample the fringe pattern. If the interferometer is trying to lock onto the central fringe of the fringe packet, at least 4 samples per fringe are needed. This may be done with 2 OPD positions $\frac{1}{4}$ of a fringe apart and using both sides of the beam splitter, which are $\frac{1}{2}$ of a fringe out of phase.

In order to work reliably, locking onto the central fringe requires more light per exposure than open-loop scanning through the fringe packet. However, scanning through the entire fringe packet requires many more exposures. The number of fringes in the main lobe of the fringe envelope is

$$N_{\text{fringes}} = \frac{2\Lambda_{\text{coh}}}{\lambda_o} = \frac{2\kappa_o}{\Delta\kappa} . \quad (1.17)$$

If $\Delta\kappa$ is 20% of κ_o , N_{fringes} is 10. With 4 exposures per fringe, at *least* 40 exposures across the fringe envelope are required. That many exposures takes a significant amount of time, and we will see in section 1.1.3 that the amount of time in which a fringe pattern may be assumed to be stationary is limited.

Image Plane Beam Combination

Image plane beam combination works by sending the input beams into a telescope, as shown in Figure 1.3. If the input beams are parallel, they will form images super-

imposed on exactly the same spot on the detector.

This combination scheme samples the interference pattern across a range of OPD all at the same time. In brief, the two beams work like Young's double slit experiment. The image of the star is crossed with fringes which are perpendicular to the line containing both beam centers at the telescope input pupil. The fringe spacing s depends upon the geometry of the telescope.

$$s = \frac{f\lambda}{D}$$

where f is the telescope focal length and D is the separation between the centers of beams 1 and 2.

Section 2.2.1 will show that the image at the detector is the Fourier transform of the beam aperture. Because each beam aperture is assumed to be a circle of uniform intensity with diameter d , the image of each aperture is proportional to

$$I(\rho) = \left[\frac{J_1(\pi\rho\frac{d}{\lambda})}{\pi\rho\frac{d}{\lambda}} \right]^2, \quad (1.18)$$

where ρ is the distance from the center of the image, measured in the image plane. $J_1(x)$ is the 1st order Bessel function of the first type.

Equation 1.18 is commonly known as the Airy disk. It has a peak at the origin, and drops to 0 when $\rho_{\text{disk}} = 1.22\frac{\lambda}{d}$. The central peak is surrounded by weak secondary intensity rings outside of the first zero, due to the oscillatory nature of $J_1(x)$.

The Airy disk acts as an envelope for the fringes when ρ_{disk} is smaller than Λ_{coh} . The image appears as an Airy disk with fringes across it. The number of fringes

which fit across the Airy disk is

$$N_{\text{fringes}} = \frac{2\rho_{\text{disk}}}{s} = 2.44 \frac{D}{d} , \quad (1.19)$$

and the OPD across the diameter of the Airy disk is

$$z_{\text{disk}} = \frac{2\rho_{\text{disk}} D}{f} = 2.44 \lambda \frac{D}{d} = \lambda N_{\text{fringes}} . \quad (1.20)$$

The fractional bandwidth $\frac{\Delta\kappa}{\kappa_o}$ may be as high as

$$\frac{\Delta\kappa}{\kappa_o} \leq \frac{2}{N_{\text{fringes}}} = \frac{d}{1.22D}$$

before Λ_{coh} becomes smaller than the OPD across the radius of the Airy disk.

If the beam diameters d are $\frac{1}{3}$ of the distance between their centers D , then there are 7.3 fringes across the Airy disk. The bandwidth $\Delta\kappa$ may be up to 27% of κ_o before the coherence length Λ_{coh} is shorter than the OPD from the center of the Airy disk to its edge.

Image plane beam combination has the advantage that the entire fringe packet is sampled at once, so only one exposure is necessary. The disadvantage is that many pixels are required across the image, at least 4 per fringe, totaling at least $4N_{\text{fringes}}$. Because the light from both beams is spread out across several pixels, a much longer exposure time is required to reach the same signal-to-noise ratio as the pupil plane combination data. Some improvement in the intensity per pixel may be gained by using a cylindrical lens to collapse the image dimension parallel to the fringes down to 1 pixel.

The trade-off between pupil and image plane combination really depends upon how much time one has until the fringe pattern moves, and how noisy the detector pixels are. We will see in section 1.1.3 that atmospheric turbulence causes the fringes to move after a characteristic time τ_o . Pupil plane combination gets more photons per pixel per τ_o , but the entire fringe pattern may take several τ_o to sample, and it will move around during that time, distorting the fringe data. Image plane combination samples the whole fringe pattern at once, but each exposure is limited to a few τ_o , or the fringes will move around enough to blur out completely.

Some interferometers are built to combine more than 2 beams at once. In a pupil plane combiner, this may be done by combining different pairs of beams in a chain, or by combining all beams at once in a hierarchical arrangement of beam splitters. The all-in-one method requires that each beam pair has its OPD scanned at a unique speed. The all-in-one fringe signal may then be Fourier transformed, and each beam pair will have a unique signal peak in frequency space.

Image plane combination of multiple beams may be done by having the beams enter the telescope at unique separations. If all of the beams are in a row, setting them at non-redundant spacings ensures that each beam pair's fringe spacing is unique, and a Fourier transform of the image will yield a unique signal peak for each beam pair in pixel frequency space.

The beams can also be placed to take advantage of the 2-dimensional nature of the image, although this precludes collapsing the image to 1 dimension to increase the intensity per pixel. If 3 beams enter the telescope in an equilateral triangle, their fringe spacings will be identical, but the fringes for each beam pair are at a unique angle. A 2-dimensional Fourier transform of the image yields unique signal peaks

in different directions but the same distance from the center of the pixel frequency plane.

1.1.3 Atmospheric Turbulence

The atmosphere is in constant motion. The temperature gradient from the equator to the poles, coupled with the rotation of the Earth, are the driving force behind our planet's weather patterns. No matter where on the planet we might place a stellar interferometer, the incoming starlight will have to first pass through moving atmospheric layers before reaching the light collectors.

Atmospheric motion in and of itself is not a problem, but that motion is in response to temperature gradients in the airmass. The index of refraction of air is dependent upon temperature, and so parcels of air with different temperatures have different refractive indices. The colder parcels of air have a higher density, which corresponds to a higher index of refraction. The speed of light in a substance is inversely proportional to its refractive index, so parts of the incoming starlight wavefront are slowed down and bent by the colder parcels of air.

The wavefront which reaches the interferometer is therefore a history of the 3-dimensional refractive index distribution through which the light has passed. If the atmosphere above the interferometer is very turbulent, the wavefront is extremely distorted by the time it reaches the ground.

This section is a brief review of the basics of atmospheric turbulence, and the limitations it imposes on stellar interferometry. For a more detailed treatment, see the works by Coulman (1985) and Roddier (1981). In his doctoral dissertation, ten Brummelaar (1992) wrote a review of atmospheric turbulence which is more easily under-

standable to the student, as is the summary in *Principles of Long Baseline Interferometry* by Quirrenbach (2000). Both were referred to extensively in the writing of this section.

Any fluid flow may be classified by its dimensionless Reynolds number:

$$R = \frac{vL}{\nu_{\text{mol}}} , \quad (1.21)$$

where v is the fluid velocity, L is the typical length of the flow, and ν_{mol} is the kinematic molecular viscosity of the fluid. Low Reynolds numbers indicate smooth, laminar flow, and high Reynolds numbers indicate turbulent flow. Air typically has a ν_{mol} of $1.5 \times 10^{-5} \text{ m}^2 \text{ s}^{-1}$. If the wind velocity is one meter per second, the Reynolds number exceeds 1×10^6 after traveling only 15 meters. The flow of air in the atmosphere is therefore almost always turbulent.

As a mass of air moves along, it is broken up into eddies of typical size L_{\circ} , called the *outer scale length*. L_{\circ} is approximately the distance to the nearest boundary to the flow, such as buildings, trees, and the ground. With no boundaries nearby, the largest eddies are limited to around 100 m, which becomes the upper limit of L_{\circ} . Several attempts have been made to measure L_{\circ} at various observatories, with a wide range of results; a few meters to a few kilometers.

The equations that describe fluid flow are non-linear, and the process they describe results in the formation of eddies starting at the lowest spatial frequency available, $\frac{1}{L_{\circ}}$, which is why the large eddies form first. Once the L_{\circ} sized eddies have been created, they in turn are unstable and break up into smaller eddies. These secondary eddies have a smaller characteristic length L , so their Reynolds numbers are lower,

and they have weaker turbulence. Eddies continue to form by this process at smaller and smaller scales until some size l_o is reached where the Reynolds number is too low for the flow to drive turbulence. l_o is referred to as the *inner scale length*, and is typically a few millimeters.

Thus we see that the kinetic energy of the bulk movement of the flow is transformed into kinetic energy of turbulent cell movement at the lowest spatial frequency $\frac{1}{L_o}$. This energy cascades through increasing spatial frequencies up to $\frac{1}{l_o}$, where no smaller eddies are formed. The kinetic energy of the l_o -sized eddies is dissipated as heat due to the viscosity of the fluid. Richardson (1922) wrote a verse which summarizes the cascade of energy from the outer to the inner scale length:

*Big whirls have little whirls that feed on their velocity
and little whirls have lesser whirls and so on to viscosity.*

The turbulent energy spectrum, a function which describes the energy in the flow across the spatial frequency spectrum, reaches a dynamic equilibrium when the energy dissipated at the l_o scale equals the energy introduced into the flow at the L_o scale. Inside the inertial range, the energy spectrum may be described by a simple model developed by Komolgorov, and turbulent flows which follow this model are commonly called Komolgorov turbulence.

The Komolgorov Spectrum

In order to derive the Komolgorov turbulence spectrum, we must first familiarize ourselves with two types of functions which are useful in describing the average shape of a function. The first is called a *covariance*. For a function $g(\tau)$, the covariance B_g

of g is

$$B_g(t) \equiv \int_{-\infty}^{\infty} g(\tau + t) g(\tau) d\tau = \langle g(\tau + t)g(\tau) \rangle .$$

If g is complex, $\langle g(\tau + t)^*g(\tau) \rangle$ is used.

We are interested in the covariance of the function describing the flow velocity. Let the function $v(\mathbf{x})$ be the flow velocity at position \mathbf{x} . Then $\langle v(\mathbf{x} + \mathbf{r})v(\mathbf{x}) \rangle$ may be more concisely noted as $B_v(\mathbf{r})$.

The covariance is a bit of a problem at low spatial frequencies, because on scales longer than L_o the variance goes to infinity, as the contribution of those scales to the turbulence spectrum is unknown. For this reason, the *structure function* D_v is used.

For a function $g(\tau)$, the structure function D_g is

$$D_g(t) \equiv \langle |v(\tau + t) - v(\tau)|^2 \rangle . \quad (1.22)$$

The structure function is related to the covariance by

$$D_g(t) = 2(B_g(0) - B_g(t)) . \quad (1.23)$$

The structure function of the velocity field is

$$D_v(\mathbf{r}) = \langle |v(\mathbf{x} + \mathbf{r}) - v(\mathbf{x})|^2 \rangle$$

and has units of velocity squared, or $\text{m}^2 \text{s}^{-2}$. We will assume that small scale motions do not depend on the direction of r , and so we can simply use the magnitude r instead of the vector form.

Next we assume that the only parameters that affect the turbulent flow aside from L_o and l_o are the kinematic viscosity ν_{mol} and the rate at which energy is dissipated per unit mass through viscous heating at the inner scale, ϵ_o . ϵ_o has units of $\text{J s}^{-1} \text{kg}^{-1} = \text{m}^2 \text{s}^{-3}$. We assume that the turbulent cascade has reached a dynamic equilibrium where the turbulent energy introduced at the outer scale is equal to the energy dissipated at the inner scale.

Komolgorov (1941) assumed that the velocity at a certain scale length r is a function which depends on ϵ_o and r each to some power. By looking at the units of ϵ_o we see that $(\epsilon_o r)^{\frac{1}{3}}$ has units of velocity: $(\text{m}^2 \text{s}^{-3} \text{m})^{\frac{1}{3}} = \text{m s}^{-1}$. Because D_v has units of velocity squared, it must depend upon $r^{\frac{2}{3}}$.

We can now write the velocity structure function as

$$D_v(r) = C_v^2 r^{\frac{2}{3}}, \quad (1.24)$$

where C_v is called the *structure constant*, which incorporates $\epsilon_o^{\frac{1}{3}}$ and describes the turbulence strength.

We are interested in the velocity structure function because of the relation it can give us to the structure functions of other quantities, namely temperature and refractive index. Tatarski (1961) points out that temperature may be treated as a *conserved passive additive*. A passive additive is something that does not change the turbulent flow, and it is conserved if it does not change. Any conserved passive additive has the same form of structure function as the velocity distribution.

While temperature gradients are indeed the driving force behind the turbulent flow, small scale temperature fluctuations within the flow may be treated as a passive

additive. As a result, the temperature structure function has the form

$$D_T(r) = C_T^2 r^{\frac{2}{3}} , \quad (1.25)$$

where C_T is the structure constant for temperature, similar in function to C_v .

The refractive index n is proportional to the density of the gas, and according to the ideal gas law, density is inversely proportional to temperature. The refractive index is therefore also a conserved passive additive, and its structure function is

$$D_N(r) = C_N^2 r^{\frac{2}{3}} , \quad (1.26)$$

with C_N as the structure constant for $N \equiv (n - 1)$, and

$$C_N = 7.8 \times 10^{-5} \frac{P}{T} C_T$$

with the pressure P in millibars and temperature T in Kelvin, according to Coulman (1985).

Tatarski (1961) shows that the refractive index structure function is related to the power spectral density $\Phi(\rho)$, the energy density at spatial frequency $\rho = \frac{1}{r}$:

$$D_N(r) = 2 \int_{-\infty}^{\infty} (1 - e^{i2\pi\rho r}) \Phi(\rho) d\rho , \quad (1.27)$$

which yields

$$\Phi(\rho) = 0.0365 C_N^2 \rho^{-\frac{5}{3}} . \quad (1.28)$$

This is for the one-dimensional distance r . If the integration is carried out in three

dimensions, the vector \mathbf{r} corresponds to the spatial frequency vector $\boldsymbol{\rho}$, and the volume element becomes $4\pi|\boldsymbol{\rho}|^2 d|\boldsymbol{\rho}|$. Then

$$\Phi(\boldsymbol{\rho}) = 0.0365 C_N^2 \rho^{-\frac{11}{3}} . \quad (1.29)$$

Equations 1.28 and 1.29 are the 1- and 3-dimensional Komolgorov spectra for turbulent flows with refractive index structure constant C_N in the inertial sub-range from L_o down to l_o . The turbulent flows near the ground do not have the same C_N as those higher up. A model of the atmosphere made up of several thin turbulent layers at various heights and values of C_N is generally able to reproduce the statistical behavior of the real thing. C_N generally drops with height, from 10^{-15} at tens of meters to 10^{-16} at around 1 km to 10^{-17} near 10 km. C_N can vary drastically in the first few kilometers above the ground, where the air is more dense and closely coupled with heating and cooling effects from the ground.

The Phase Structure Function

Now that we know the shape of the Komolgorov spectrum, we are equipped to find the structure function of the phase of the incoming starlight. The starlight wavefront is expressed as a wave with unit amplitude

$$\psi(x) = e^{i\phi(x)} , \quad (1.30)$$

and $\phi(x)$ is the phase of the wavefront along the x -axis, which is parallel to the ground.

We assume that each turbulent layer will only introduce changes in the phase of the wavefront, and not the intensity. Intensity changes, known as *scintillation*, are caused by the bending of various rays across the wavefront, causing rays to cluster more closely in some areas than others. In practice, scintillation is present in the wavefront, and is a significant source of noise in interferometric data. For the purposes of this derivation, however, we will ignore scintillation and assume that the amplitude of $\psi(x)$ remains constant. We also assume that the incoming starlight travels straight down (which we will call the $-z$ direction), and not at an angle.

The phase shift induced by a turbulent layer at height h and thickness δh is

$$\phi(x) = k \int_h^{h+\delta h} n(x, z) dz , \quad (1.31)$$

where k is the wavenumber $\frac{2\pi}{\lambda}$ with units of radians per meter. If the turbulent layer is much thicker than the individual eddies, then many eddies affect the phase at any x and $\phi(x)$ has a Gaussian distribution.

The covariance of the wavefront after passing through the turbulent layer at height h is

$$B_{\psi,h}(r) \equiv \langle \psi(x) \psi^*(x+r) \rangle = \langle e^{i[\phi(x) - \phi(x+r)]} \rangle .$$

Because $\phi(x)$ has a Gaussian distribution, $[\phi(x) - \phi(x+r)]$ does also, and is centered on zero. Any variable χ with a Gaussian distribution about zero obeys the rule

$$\langle e^{\alpha\chi} \rangle = e^{\frac{1}{2}\alpha^2 \langle \chi^2 \rangle} ,$$

and so

$$\langle e^{i[\phi(x)-\phi(x+r)]} \rangle = e^{-\frac{1}{2}\langle |\phi(x)-\phi(x+r)|^2 \rangle} .$$

The averaged quantity in the second half of the above equation is the definition of the phase structure function $D_\phi(r)$, so

$$B_{\psi,h}(r) = e^{-\frac{1}{2}D_\phi(r)} . \quad (1.32)$$

This relationship between the wavefront covariance and the phase structure function will prove useful very soon.

Now consider the phase covariance

$$B_\phi(r) \equiv \langle \phi(x+r)\phi(x) \rangle .$$

Using (1.31) this becomes

$$B_\phi(r) = k^2 \int_h^{h+\delta h} \int_h^{h+\delta h} \langle n(x, z') n(x+r, z'') \rangle dz' dz''$$

Change variables in the second integral to $\zeta = z'' - z'$ and define

$$B_N(r, \zeta) = \langle n(x, z') n(x+r, z' + \zeta) \rangle$$

to get

$$B_\phi(r) = k^2 \int_h^{h+\delta h} dz' \int_{h-z'}^{h-z'+\delta h} B_N(r, \zeta) d\zeta .$$

We have assumed that the turbulence layer is much thicker than the eddies which

it contains, meaning that the covariance B_N reaches zero before it gets to the edges of the turbulent layer. The integral over $d\zeta$ may therefore extend its limits to infinity without any trouble. Now

$$\begin{aligned} B_\phi(r) &= k^2 \int_h^{h+\delta h} dz' \int_{-\infty}^{\infty} B_N(r, \zeta) d\zeta \\ &= k^2 \delta h \int_{-\infty}^{\infty} B_N(r, \zeta) d\zeta . \end{aligned} \tag{1.33}$$

Now we use (1.23) which relates any covariance to its structure function. In the case of $\phi(r)$,

$$D_\phi(r) = 2[B_\phi(0) - B_\phi(r)] ,$$

into which we substitute (1.33) to obtain

$$D_\phi(r) = 2k^2 \delta h \int_{-\infty}^{\infty} [B_N(0, \zeta) - B_N(r, \zeta)] d\zeta .$$

If we add and subtract $B_N(0, 0)$ to the terms in brackets the integral remains unchanged. This allows us to use (1.23) again, this time to switch from B_N to D_N :

$$\begin{aligned} D_\phi(r) &= 2k^2 \delta h \int_{-\infty}^{\infty} [(B_N(0, 0) - B_N(r, \zeta)) - (B_N(0, 0) - B_N(0, \zeta))] d\zeta \\ &= k^2 \delta h \int_{-\infty}^{\infty} [D_N(r, \zeta) - D_N(0, \zeta)] d\zeta . \end{aligned}$$

Remember that in (1.26), $D_N(x) = C_N^2 x^{\frac{2}{3}}$. We substitute $r^2 + \zeta^2 = x^2$ so that

$$D_N(r, \zeta) = C_N^2 (r^2 + \zeta^2)^{\frac{1}{3}} ,$$

and substitute that into our equation for $D_\phi(r)$ to obtain

$$D_\phi(r) = k^2 \delta h C_N^2 \int_{-\infty}^{\infty} \left[(r^2 + \zeta^2)^{\frac{1}{3}} - \zeta^{\frac{2}{3}} \right] d\zeta . \quad (1.34)$$

Equation 1.34 integrates to

$$D_\phi(r) = 2.914 k^2 \delta h C_N^2 r^{\frac{5}{3}} , \quad (1.35)$$

which is the phase structure function on the ground after the wave ψ propagates through a single turbulent layer of thickness δh .

If we define the zenith angle z as the angle between the incoming starlight and the vertical, we may integrate (1.35) through the entire atmosphere:

$$D_\phi(r) = 2.914 k^2 r^{\frac{5}{3}} \sec z \int C_N^2(h) dh .$$

We may simplify things greatly by defining the parameter

$$r_o \equiv \left[0.423 k^2 \sec z \int C_N^2(h) dh \right]^{-\frac{3}{5}} , \quad (1.36)$$

then

$$D_\phi(r) = 6.88 \left(\frac{r}{r_o} \right)^{\frac{5}{3}} \quad (1.37)$$

and by (1.32),

$$B_\psi(r) = e^{-3.44 \left(\frac{r}{r_o} \right)^{\frac{5}{3}}} . \quad (1.38)$$

The Komolgorov power spectrum of phase fluctuations $\Phi_\phi(r)$ may be obtained by

an integral of the same form as (1.27):

$$D_\phi(r) = 2 \int_{-\infty}^{\infty} (1 - e^{i2\pi\rho r}) \Phi_\phi(\rho) d\rho ,$$

which is set equal to (1.37) and integrated to obtain

$$\Phi_\phi(\rho) = 0.0229 r_\circ^{\frac{5}{3}} \rho^{-\frac{11}{3}} . \quad (1.39)$$

Fried's Coherence Length, r_\circ

First proposed by Fried (1965), the parameter r_\circ (defined in Equation 1.36) *is the diameter of a circular area of the wavefront in which the rms phase distortion is about 1 radian.*

This has some significant physical consequences:

1. The limiting image resolution for long exposures with a telescope whose diameter D is larger than r_\circ is $1.22 \frac{\lambda}{r_\circ}$ rather than $1.22 \frac{\lambda}{D}$, so telescopes typically do not achieve their diffraction-limited imaging capabilities.
2. Telescopes of diameter $3.8 r_\circ$ are the optimum size for the highest possible quality images taken with short exposure times. Telescope diameters larger than this do not benefit from higher resolution, as the perturbed wavefront degrades the image quality.
3. Buscher (1988a) found that for an interferometer that tracks the zero OPD position using the power spectrum of spectrally dispersed fringes, the signal-to-noise ratio reaches its highest value for apertures about $3r_\circ$ in size.

The wavefront may be roughly approximated as a collection of r_o -sized patches of different phase, with the phase nearly constant over each patch. In this approximation, r_o is referred to as the *isoplanatic patch size*.

Equation 1.36 shows that r_o is proportional to $k^{-\frac{6}{5}}$. k is $\frac{2\pi}{\lambda}$, so

$$r_o \propto \lambda^{\frac{6}{5}}.$$

This means that as one goes to shorter wavelengths, the wavefront distortions due to atmospheric turbulence become worse. This work represents some degree of ambition, as it is an attempt to reduce the CHARA Array's operating wavelength by a factor of 3 to 4. This corresponds to an r_o that is $\frac{1}{4}$ to $\frac{1}{5}$ the size of r_o in K' band ($2.13 \mu\text{m}$).

A typical value of r_o for good seeing is 10 cm at a wavelength of 550 nm. At 750 nm this corresponds to an r_o of 14.5 cm, and a 50 cm r_o at a wavelength of $2.13 \mu\text{m}$.

The Coherence Time, τ_o

For simplicity, we adopt Taylor's hypothesis of frozen turbulence, the assumption that the turbulent eddies evolve slowly enough that the turbulent layer may be treated as a static phase screen which is moved past the interferometer with wind velocity v . The phase of the incoming starlight therefore changes by about 1 radian when the turbulent layer has moved a distance of r_o .

The coherence time τ_o *is the time in which the phase of the wavefront at a given*

Table 1.1: τ_o and the characteristic atmospheric turbulence frequency as functions of r_o and wind velocity.

Wind Velocity	r_o 5 cm	r_o 10 cm	r_o 15 cm	r_o 20 cm
5 m/s	10 ms, 100 Hz	20 ms, 50 Hz	30 ms, 33.3 Hz	40 ms, 25 Hz
10 m/s	5 ms, 200 Hz	10 ms, 100 Hz	15 ms, 66.7 Hz	20 ms, 50 Hz
15 m/s	3.3 ms, 300 Hz	6.7 ms, 150 Hz	10 ms, 100 Hz	13.3 ms, 75 Hz
20 m/s	2.5 ms, 400 Hz	5 ms, 200 Hz	7.5 ms, 133.3 Hz	10 ms, 100 Hz

point changes by 1 radian. It is simply

$$\tau_o = \frac{r_o}{v} . \quad (1.40)$$

Because r_o is proportional to $\lambda^{\frac{6}{5}}$,

$$\tau_o \propto \lambda^{\frac{6}{5}}$$

as well. r_o is calculated as an integral over all turbulent layers, and τ_o is valid for all turbulent layers as well if v is a characteristic velocity that describes the effective velocity of the entire turbulence distribution, weighted by the optical turbulence strength of each layer. A very precise measurement of τ_o would require the incorporation of the velocity distribution with height $v(h)$ into the integral of the refractive index structure constant $C_N(h)$ in the definition of r_o , Equation 1.36.

In order for an interferometer to record fringes, they must be sampled within a few τ_o , or the fringes will move around enough that they will average out. For scanning of the entire fringe packet with a pupil plane beam combiner, the whole fringe packet should be sampled in a few τ_o in order to avoid excessive distortion of the fringe packet which makes the data unusable. This problem is discussed further in section 4.2.3.

If the interferometer must acquire its fringe data within a few τ_o , then the required data gathering rate is a few times the characteristic atmospheric turbulence frequency $\frac{1}{\tau_o}$. Values for τ_o and $\frac{1}{\tau_o}$ are given in Table 1.1 for a range of wind velocities and r_o sizes.

1.2 The History of Stellar Interferometry

In this section I seek not to lay out the history of man's understanding of the wave nature of light and interference phenomenae, but rather to give an account of the application of the van Cittert-Zernike theorem to the measurement of stars via fringe visibility measurements.

The history is presented as a simple list of events in chronological order, starting from Fizeau in 1868 and working up through the various efforts in visible and infrared interferometry to the present day, with a brief mention of future projects. The further in the future the project the less mention it receives, for they change in design from year to year. The focus of this history is more on the architecture of the instruments themselves than on the science they produce. References are given for each project wherever possible.

The time line published by Lawson in *Principles of Long Baseline Interferometry* (Lawson, 2000a) was a useful starting point for the compilation of this history, giving key references and noting smaller projects of which I was not previously aware.

The review paper by Monnier (2003) is also useful for familiarizing oneself with the landscape of the interferometry community, as well as providing an excellent summary of key scientific results to date from the various interferometers that are

producing publishable results.

1.2.1 The Time Line

- 1868: Fizeau proposes stellar interferometry

Hippolyte Fizeau describes how a two-aperture mask may be used in a telescope to produce fringes. The separation between the apertures is gradually increased, causing the fringe visibility to decline. By finding the separation where the fringe visibility first reaches zero (the 1st zero of $\frac{J_1(x)}{x}$), the diameter of the star may be calculated (Fizeau, 1868).

- 1872-1873: Stéphan attempts to measure stellar diameters

Edouard Stéphan tries out Fizeau's method for measuring stellar diameters with the 80-cm refractor at the Observatory of Marseille. He has an insufficient baseline to resolve any stars, but establishes that they must have angular diameters smaller than 0.158 arc seconds (Stéphan, 1874).

- 1890: Michelson publishes the math

Albert Michelson writes a paper laying out the mathematical basis for stellar interferometry (Michelson, 1890). He does this with apparently no knowledge of Fizeau's or Stéphan's work; no reference is made to their work in the English language until 1928. See Lawson (2000a) for a discussion of the likelihood that Michelson knew of the previous French interferometry efforts.

- 1891: Michelson measures the Galilean moons

Albert Michelson measures the diameters of the 4 Galilean moons of Jupiter using a cap constructed with two movable slits, spaced about 4 inches apart,

on the 12-inch equatorial telescope at Lick Observatory. The angular diameters he measures range from 1.73 to 1.19 arc seconds, right at the diffraction limit for his operating wavelength of 550 nm, and are verified by angular diameters measured using the Lick Observatory 36-inch telescope (Michelson, 1891).

- 1896: Schwartzschild resolves binary stars

Karl Schwarzschild measures the separations of 13 binary stars using multiple-slit masks on the 10-inch telescope at Munich Observatory. The mask is made of two pieces that fold at the center like a rooftop over the telescope aperture. By changing the opening angle of the “roof” a variable slit separation is projected onto the telescope aperture. At a wavelength of 570 nm, he uses three masks with different slit spacings to measure binary separations from 0.85 to 4.35 arc seconds (Schwarzschild, 1896).

- 1898: Hamy measures Vesta

Maurice Hamy measures the moons of Jupiter and Vesta using the Grand Coudé telescope at the Observatory of Paris.

- 1920: Anderson plots the orbit of Capella

J. A. Anderson measures the orbit of Capella using a rotating plate with two apertures of variable separation mounted near the image plane of the 100-inch Hooker telescope at Mount Wilson Observatory. By taking several observations at many aperture orientations and separations, he is able to determine both the position angle and separation (about 0.05 arc seconds) at 6 points distributed around the orbit. His paper contains a detailed discussion of atmospheric dispersion effects, and how if uncorrected, they can blur out the fringes completely

for aperture orientations with a large vertical component (Anderson, 1920).

- 1921-1931: The 20-Foot Interferometer

Albert Michelson and Francis Pease measure the diameter of Betelgeuse using a pair of mirrors mounted on a 20-foot boom placed at the aperture of the 100-inch Hooker telescope at Mount Wilson Observatory. The two mirrors serve as a pair of apertures, reflecting their light to 2 fixed inner mirrors which inject their beams through holes in an aperture cover and into the standard optical path of the telescope. Interference fringes are formed on the image of the star, and compared to reference fringes from a pair of beams whose separation is much smaller. A pair of rotating prisms is used to compensate for differential atmospheric dispersion between the beams. With the full 20-foot separation between the outer mirrors, the resolution limit of the interferometer at a wavelength of 550 nm is 23 milli-arc seconds (mas) for a single star diameter, and 6 mas for a binary star separation. The diameter of Betelgeuse is measured to be 47 mas in December of 1921 (Michelson and Pease, 1921). The 20-foot interferometer remains in use, measuring 6 more stellar diameters until 1931.

- 1931-1938: The 50-Foot Interferometer

Francis Pease builds a 50-foot interferometer at Mt. Wilson Observatory, separate from the 100-inch telescope. He never achieves satisfactory results due to flexure problems in the instrument (Pease, 1931). The project is abandoned after his death in 1938. Interferometry at visible wavelengths is not pursued for several years as the development of RADAR prompts a focus on interferometry at radio wavelengths.

- 1956 Hanbury Brown and Twiss propose and test intensity interferometry

Hanbury Brown and Twiss predict that visibility can be measured using intensity correlations rather than direct combination of wavefronts (Hanbury Brown and Twiss, 1956b). They build a prototype interferometer which detects photons separately at each aperture with photomultiplier tubes and correlates the signals using radio interferometry techniques. Intensity correlation has a much higher tolerance for errors in the relative delay between the two apertures, and so represents a massive gain in simplicity of the instrument, but the technique is very insensitive. First fringes are found on Sirius (Hanbury Brown and Twiss, 1956a), prompting the design and construction of a much larger intensity interferometer.

- 1964-1976: The Narrabri Stellar Intensity Interferometer

Hanbury-Brown, Twiss, and others build and operate the Narrabri Stellar Intensity Interferometer (NSII), a pair of 6.5 m reflectors which are able to move around a circular track 188 m in diameter (Brown et al., 1967a). The interferometer requires very narrow bandwidths, which severely limits the instrument's sensitivity. At 440 nm a 2nd magnitude star requires a 50-hour integration time to build a sufficient signal to noise ratio. Two sets of stellar diameters are published, the first containing the angular diameters of 15 stars (Brown et al., 1967b), and the second representing the full set of stars bright enough for the instrument, 32 stars of B magnitude less than 2.5 (Hanbury Brown et al., 1974).

- 1970: Labeyrie proposes speckle interferometry

Antoine Labeyrie points out that each seeing cell across a telescope aperture

functions approximately as a separate aperture with its own phase shift. A phase shift at the aperture translates into a shift in the position of the image. The result of many seeing cells across the aperture is an image made up of many star images, each at a different position. By adding the Fourier transform of many speckle images together, one may build up the Fourier transform of the auto-correlation of the original unperturbed image (Labeyrie, 1970).

- 1974: First infrared heterodyne stellar interferometry

Johnson, Betz, and Townes measure the first heterodyne fringes with separated telescopes at $10\ \mu\text{m}$ with the McMath auxiliary telescopes at Kitt Peak (Johnson et al., 1974). Fringes were formed by mixing incoming light from the limb of Mercury with a laser reference, and using radio receivers to measure the beat signal at the frequency equal to the difference between the incoming light and the reference. The method requires very narrow bandwidths, and so has a very bright magnitude limit like intensity interferometry. It has the advantage, like intensity interferometry, of using radio astronomy correlation techniques, which are much more tolerant of delay errors. With a telescope separation of 5.5 m, the resolution of the instrument is 0.45 arc seconds at $10\ \mu\text{m}$. The system is used to measure dust shells around several stars.

- 1974-1987: The Interféromètre à 2 Télescopes

Antoine Labeyrie attains the first direct detection of interference fringes with separated telescopes. The telescopes are 12 m apart, and fringes are seen from Vega in the 500-750 nm wave band (Labeyrie, 1975). Optical path compensation is performed by means of a movable optical bench that runs on a track in line

with the Coudé path of both telescopes. The beams are superimposed in the image plane, and detected by a TV camera. A prism is used to disperse the fringes and improve the visibility at each wavelength for finding the zero OPD position.

The success of the instrument leads Labeyrie to relocate it from Nice to Plateau de Calern where it is used for both visible and near-IR (K band) work on numerous stellar diameters. During this time the instrument was named the Interféromètre à 2 Télescopes (I2T).

- 1979: The Mark I Interferometer

Shao and Staelin obtain first fringes on Polaris with the first fringe tracking interferometer, the Mark I Interferometer, at Mount Wilson Observatory (Shao and Staelin, 1980). Fringe tracking is the process by which an interferometer constantly monitors the fringe position and adjusts its optical delay to keep the fringes at a desired position. The Mark I is a pair of 5 cm siderostats, 1.5 m apart, which steer their beams into a Michelson-type pupil plane beam combiner. The OPD is modulated by a mirror mounted to a movable actuator system. The mirror is scanned through one wavelength of delay while photomultiplier tubes detect the interfering starlight from 400 to 900 nm. The delay sweep is broken up into four equal segments, and the light intensity detected during the segments noted as A, B, C , and D . The phase ϕ of the fringe is found by the formula

$$\phi = \tan^{-1} \left[\frac{A - C}{B - D} \right] ,$$

which provides an error signal for tracking. The system was designed only to

point at Polaris, as it was a testbed for the fringe tracking technique.

- 1979-1993: SOIR D'ETE

SOIR D'ETE operates at Plateau de Calern. It is a heterodyne interferometer, mixing the starlight with light from a CO₂ laser and using radio interferometry methods to measure the correlation. It operates at 11 μm , and has a resolution of 70 milli-arc seconds (mas) (Assus et al., 1979).

- 1982-1984: The Mark II Interferometer

The Mark II interferometer is built and operated at the Mount Wilson Observatory as a testbed for improvements on the Mark I design (Shao et al., 1986). The Mark II has a 3.2 m baseline, and movable siderostats designed to observe stars within 15° of the zenith. The increased sky coverage makes it necessary for the Mark II to have a delay line, the first ever monitored by its own laser interferometer. The delay line is contained within an evacuated pipe in order to avoid problems due to longitudinal dispersion in air.

The Mark II is intended as a testbed for two new techniques: First, it detects fringes at red and blue wavelengths simultaneously, and uses the difference between the two measurements to subtract out some of the atmospherically induced position error. Second, it is for testing out the as yet untried technique of measuring the angles between stars via the amount of delay between their fringe packets.

- 1985: First direct detection of optical closure phases

Baldwin, Haniff et al. demonstrate the first closure phases measured by direct

fringe detection at optical wavelengths. By use of an aperture mask with a non-redundant hole spacing, fringe amplitudes and phases may be measured that are equivalent to an those of interferometer array with apertures in the same pattern as the aperture mask holes. The fringe phases are prone to atmospheric perturbations, so much so that all phase information from the source is lost. If the phases from baselines of 3 apertures in the mask are added together, the atmospheric perturbations cancel out and one is left with a closure phase which is independent of the atmospheric turbulence (Baldwin et al., 1986).

- 1985-1988: The 11.4 m Prototype Interferometer

The 11.4 m Prototype Interferometer operates in the suburbs outside of Sydney. It is a testbed for several design features intended for a much larger interferometer to follow (Davis and Tango, 1985) . It is composed of a pair of mirrors that steer their beams into a central building where the beams pass through a pair of beam reducing telescopes (BRTs) and into a pair of delay lines monitored by laser metrology. The delay lines are in the open air, so the surrounding structure is built as a building within a building, with climate control of the intervening space. This allows the temperature of the inner chamber to be regulated without inducing turbulent eddies into the beams.

After OPD compensation, the beams are combined in the pupil plane via incidence on opposite sides of a beam splitter. Both splitter outputs are dispersed spectrally via a prism and imaged onto a detector. The operating wavelength of the system is modified by means of a slit in front of the detector. The wavelength is changed by rotating the prism, and the bandwidth is controlled by the

slit width. The wavelength range is 430-500 nm.

- 1985-Present: The Grand Interféromètre à 2 Télescopes

First fringes on Vega are recorded at the Grand Interféromètre à 2 Télescopes (GI2T) built at Plateau de Calern near I2T (Labeyrie et al., 1986; Mourard et al., 1994). GI2T is the same design as I2T, but with improved beam combining optics and 1.5 m telescopes movable up to 70 m apart. The beams are combined in the image plane, and are spectrally dispersed to improve the coherence length of the fringe packet in each spectral channel. GI2T remains in use until 1996, when it is re-tooled and goes back into operation with the REGAIN beam combiner in 1999. With high spectral resolution, GI2T is able to measure interference fringes in the $H\alpha$ emission line.

- 1986-1993: The Mark III Interferometer

The Mark III Interferometer operates at Mt. Wilson Observatory. It is a general improvement upon the Mark II, with 3 siderostats in a triangle whose baselines can be repositioned for a range from 9 to 20 m (Shao et al., 1988).

The light from the siderostats is transported to the central beam combination building inside of evacuated light pipes to reduce wavefront distortion. The delay lines implement a cat's eye reflector, a telescope with a flat secondary at the primary focus, which returns the beams with a smaller sensitivity to misalignment to the delay line than previous rooftop retroreflector designs. The delay lines are each encased in a 25 m long vacuum vessel, which avoids differential dispersion effects induced by air-filled delay lines.

The light is combined in the pupil plane via a beam splitter and imaged onto

photomultiplier tubes. The interferometer is operated at various wavelength bands across the visible light range, and can record fringes in two color simultaneously for the removal of atmospheric jitter from the measured position of the star. The system tracks on the fringe phase using the same 4-bin phase quadrature algorithm used by the Mark I and Mark II.

The Mark III is designed for the measurement of the angles between stars, and as such the positions of the siderostats must be monitored carefully. A 3-beam laser interferometer utilizes a retroreflector placed on the back of each siderostat to measure its absolute position while the stellar interferometer measures the delay between the fringe packets of pairs of stars.

Finally, the Mark III is the first fully-automated interferometer, acquiring the star, finding fringes, and recording them without intervention by the observer. This enables a high-throughput observing program, and the Mark III produces a large body of scientific publications.

- 1988-Present: The Infrared Spatial Interferometer

The Berkeley Infrared Spatial Interferometer (ISI), a 2-telescope heterodyne interferometer located at the Mount Wilson Observatory, mixes light near $10\ \mu\text{m}$ with a CO_2 laser and detects the beat frequency the laser-starlight frequency difference (Hale et al., 2000). The signals detected at the telescopes are processed by a correlator after the radio interferometry fashion, and the fringe visibility is measured.

ISI is composed of two movable trailers, later upgraded to 3, each housing a 2 m flat siderostat that feeds a 1.65 m vertically mounted parabola, which in

turn feeds a smaller set of mirrors that extend the parabola's focal length. The gradually focusing beam is combined with a laser and comes to a focus on a photodiode.

The ISI telescope trailers can be repositioned to several concrete pads distributed across the site, allowing for baselines from 4 to 80 m. The laser signals that combine with the starlight at each trailer must be in phase with each other. A portion of the laser is projected back and forth between the trailers, and a secondary heterodyne interferometer monitors the phase of the beat frequency between the two lasers with respect to a stable oscillator. The beat frequency phase is used as an error signal to tune one of the lasers, thus keeping both lasers in phase.

After 2000, the third ISI telescope was pressed into service, making phase closure measurements possible with all three telescopes run simultaneously.

- 1990-1992: The InfraRed Michelson Array

The InfraRed Michelson Array (IRMA), built by the University of Wyoming and the National Optical Astronomy Observatory, combines light from two 20 cm siderostats on a North-South baseline which is adjustable from 2.5 m to 19.5 m in 0.5 m increments (Dyck et al., 1993). It is designed to be a very low-cost instrument, with very little automation, not even automated star guiding. It consists of some very simple delay lines, a beam splitter, and a few dichroic splitters which feed visible light to intensified CCD TV cameras for manual star tracking. The fringes are detected in K band ($2.2\ \mu\text{m}$) with two InSb detectors, one on each output of the beam splitter. The two detector signals

are combined as their difference over their sum, which is less sensitive to noise sources common to both arms of the interferometer.

The delay lines have no closed-loop servo control; the change in OPD caused by the rotation of the Earth is used to scan through the fringes. Typically, the delay line is manually set just ahead of the fringe position and set into a fixed position. Earth's rotation causes a change in the projected baseline as seen by the star, and soon the fringes sweep by. The delay line is manually moved ahead again and the process is repeated. The time spent at each delay line position is typically 15-30 seconds, less than 0.2 seconds of which is actually spent inside the fringe packet.

After two years the interferometer is dismantled. The beam combiner hardware and detectors are incorporated into the IOTA interferometer, while parts of the delay lines go to the FLUOR experiment. Later, the detectors move on to the CHARA Array, where they are used to detect first fringes in K band.

- 1990 - Present: HST FGS

The Hubble Space Telescope's (HST) Fine Guidance Sensors (FGS) are a single-aperture shearing interferometer (the HST is the aperture) which can be used for astrometric measurements of angles between celestial objects, as well as diameters of objects. The first of a multitude of papers is Holfeltz and Taff (1992).

- 1991 - Present: The Fiber-Linked Unit for Optical Recombination

The Fiber-Linked Unit for Optical Recombination (FLUOR) injects light from two telescopes into single mode, polarization-preserving optical fibers and com-

bines them via an x-shaped fiber splice, which works as a 50-50 beam splitter. This is the first time fibers are used for beam combination. Each fiber is also split before combination to provide a separate photometric signal for each input beam. The fibers act as spatial filters at the point of injection, which severely reduces the amount of light detected, but allows only the perfectly flat wavefront mode into the fiber, which greatly improves the system visibility. FLUOR enables very accurate visibility squared measurements on bright sources (DuForesto and Ridgway, 1992).

FLUOR is used at the IOTA interferometer beginning in 1994 (Coude Du Foresto et al., 1998), and later moves to the CHARA array (Coude du Foresto et al., 2003).

- 1991 - Present: The Cambridge Optical Aperture Synthesis Telescope

The Cambridge Optical Aperture Synthesis Telescope (COAST) is equipped with four 0.40 m apertures, later upgraded to 5. The available baselines run up to 40 m, with a future upgrade to 100 m in the works.

The delay lines are roof mirrors which are mounted on a cart which runs along a steel track. The cart is vibrationally isolated from its drive unit by a voice coil. The position of the cart is measured with a 552.7 nm laser interferometer.

The beam combiner is a series of beam splitters and mirrors that combines each of the 4 input beams with two other inputs, the result is 4 output beams, each with light from 3 of the input beams. The output beams are each focused onto an avalanche photodiode (APD), which counts individual photons with a 45% quantum efficiency and is sensitive from 400-1000 nm (Cox, 1994). The

fringes are measured by scanning through the fringe envelope, a technique that is improved upon in 2002 when a fringe envelope tracker is developed using this beam combiner and detectors (Thureau et al., 2003).

In 1995 the first optical synthesis image with separated telescopes is made at COAST at 830 nm (Baldwin et al., 1996). In 1998, a near-IR image plane beam combiner was developed. After 2002, COAST changes its focus from an astronomical instrument to a testbed for prototype systems for other larger interferometers such as MROI and VLTI. In 2004 COAST utilizes an electron multiplying CCD which operates in photon counting mode, with effectively zero read noise. The CCD is used as the detector in a 500-channel spectroscope (Basden et al., 2004).

- 1991- Present: The Sydney University Stellar Interferometer

The Sydney University Stellar Interferometer (SUSI), successor to the 11.4 m Prototype Interferometer, uses pairs of siderostats with 0.14 m effective apertures, with two arms running north and south from a main beam combining facility (Davis et al., 1999). 6 siderostat stations are placed on each arm such that they create a geometrically increasing series of available baselines which range from 5 m to 640 m. The longest operational baseline at present is 160 m.

The beams are transported in evacuated pipes to the central beam building, where a beam reducing telescope shrinks them to 50 mm in diameter, and magnifies the angle of any incoming light by 3.

The delay lines and beam combination area are housed in a nested structure similar to the 11.4 m Prototype Interferometer. The space between the inner

and outer enclosures is climate controlled, allowing for a stable temperature in the inner enclosure without any air currents.

The delay lines are in the form of two cat's eye reflectors mounted back to back on a single carriage. Each reflector serves as the delay line for a particular arm of the interferometer, with the light from each arm reaching the cart from opposite directions. Whenever the cart is moved, whatever delay is subtracted from one arm of the interferometer is simultaneously added to the other arm.

Atmospheric dispersion is quite significant in the visible wavelength regime, particularly in the blue. A pair of Risley prisms is used in each beam to remove angular dispersion caused by the starlight entering the atmosphere at an angle, which acts like a weak prism.

Because the delay lines are in air, longitudinal dispersion is a significant factor as well. The geometric delay caused by the light reaching the collectors at different times occurs in the vacuum of space, because it may be assumed that the two beams pass through the same amount of atmosphere (except for small fluctuations in due to atmospheric turbulence). When that delay is compensated in air, the two beams have passed through different amounts of air. The index of refraction of air increases as λ gets smaller, so the position of zero OPD is wavelength dependent, causing the fringes to smear out if the bandwidth of the instrument is too wide. By introducing a certain thickness of glass into the beam with a shorter air path, the dispersion may be approximately equalized in each beam. The Longitudinal Dispersion Compensator (LDC) is a series of crown and flint glass blocks placed into each beam, with a pair of sliding wedges

of each glass type providing a small amount of variable glass thickness.

SUSI is designed for operation in the 400-900 nm wavelength range. A 50-50 beam splitter that works over the entire range would be difficult to make, so the instrument is designed for two beam combiners, one for 400-550 nm (blue) and the other for 600 nm-900 nm (red). A dichroic splitter placed after the LDC separates the red and blue light and sends them to their respective beam combiners. The blue light combiner was formerly the main system at SUSI, with the red system now serving as the main science engine.

The blue light is combined in the pupil plane on a 50-50 beam splitter. Each output passes through a prism-and-slit monochromator of the same design as the 11.4 m Prototype Interferometer. By rotating the prism, one selects the central wavelength of the operating band. The light then strikes the slit, the width of which determines the bandwidth. The light allowed through the monochromator is then brought to a focus on a photomultiplier and detected.

In 2002 SUSI obtains first fringes with the red system. The red beam combiner is preceded by two Gregorian beam reducers (each is a pair of confocal mirrors), which reduce the beam to 12.5 mm and provide an intermediate image plane where field stops may be used. The North beam reflects off of a dither mirror mounted on a PZT stage, which enables OPD scanning independent of the blue system. The beams are combined in the pupil plane on a 50-50 beam splitter and focused onto multimode optical fibers. The fibers transport the light to avalanche photodiodes (APDs), which detect the photons. The APDs have a Peltier cooling system that gives off a lot of heat, which would be detrimental

to the seeing conditions in the beam combining lab. For this reason, the APDs are located outside the lab, which is made simple by an optical fiber feed. The operating wavelength of the red system is controlled by interference filters, mounted in a filter wheel.

The red system represents a major improvement in SUSI's sensitivity. It employs a fringe envelope tracking scheme, which uses a Henkel transform (Bracewell, 1965) and a Fourier space frequency filter to extract the fringe envelope location from each dither scan. A leaky memory buffer keeps track of the recent history of the fringe envelope position. By adding the fringe envelopes in a shift-and-add scheme, a high signal-to-noise model of the fringe envelope can be built, the peak of which, after proper calibration, can give the squared visibility of the fringes (Tuthill et al., 2004).

- 1994 - Present: The Infrared Optical Telescope Array

The Infrared Optical Telescope Array (IOTA), on Mount Hopkins, Arizona, initially uses two repositionable 0.45 m siderostats, later upgraded to 3, at baselines up to 35 m. IOTA goes through many iterations, serving as the testbed for several experimental beam combiners operating in the visible, near-IR, and at 4 μm . Among the various beam combiners are FLUOR, the first fiber-based beam combiner (Coude Du Foresto et al., 1997), and IONIC-3, the first integrated optics combiner for 3 telescopes (Berger et al., 2003). With 3 apertures in a triangle, IOTA focuses on imaging experiments and closure phase measurements, producing a large body of published work (Traub, 1998; Monnier et al., 2004b).

- 1994 - Present: The Navy Prototype Optical Interferometer

The Navy Prototype Optical Interferometer is a Y-shaped array on Anderson Mesa, southeast of Flagstaff, Arizona, with two 0.12 m siderostats on each arm. The inner siderostat positions are monitored by a laser metrology system, giving NPOI the capacity for accurate angle measurements between stars. The outer siderostats can be repositioned to several stations, with the longest available baseline at 250 m, eventually planned to extend to 437 m. The beams are transported to the central building inside evacuated light pipes.

The optical path is controlled for each arm via two evacuated delay systems: one introduces delay in discrete amounts using mirrors that are slid into the beam at regularly spaced stations. The second is a continuous delay line in the form of a cat's eye reflector on a cart, which moves inside an evacuated tube.

Beam combination is in the pupil plane, with a 3-way pairwise combiner of beam splitters and dither mirrors, such that each beam is combined uniquely with the other two, and the OPD for each beam pair is modulated by a separate dither mirror placed before the beam splitters.

NPOI operates in the visible band, with the light spectrally dispersed and injected into multimode optical fibers that feed APD detectors. Fringe detection and tracking is done in the manner of the Mark I-III interferometers, by measuring the intensity at 4 positions along the fringe and solving for the fringe amplitude and phase. The timing of the APDs varies according to each spectral channel so that as the dither mirror sweeps continuously through the fringe, each APD's data is binned into 4 segments, of $\frac{1}{4}$ wavelength.

The central fringe of the packet is identified by comparing the fringe phase across the spectral channels. At the central fringe, the phase is the same across all channels. As the system moves away from zero OPD, the fringe phase gradient across the spectral channels increases proportionally.

NPOI published a long list of stellar diameters, and performs many aperture synthesis imaging experiments (Armstrong et al., 1998).

- 1995 - Present: The Palomar Testbed Interferometer

The Palomar Testbed Interferometer (PTI) continues the development path of the Mark I-III interferometers with 3 40 cm siderostats fixed at baselines of 110, 86, and 87 m in a triangle. PTI uses evacuated light pipes to transport the beams to the central beam combination building, where non-evacuated delay lines equalize the OPD. The beams are combined in the pupil plane and the 2.0-2.4 μm wave band is used for fringe tracking, while a spectrometer is also usable for fringe measurements across the 1.5-2.4 μm wave band. Air delay lines are acceptable because atmospheric dispersion is quite small in the near IR.

PTI adds a new functionality to the Mark III design; a secondary beam in each of the delay lines that leads to its own beam combiner and fringe tracker. The incoming beam from each siderostat is brought to a focus on a mirror with a hole in the center, called a field separator. If a star image is centered on the hole, the light passes through, is re-collimated, and enters the primary delay beam and fringe tracker. If the star image is not centered on the hole, the light reflects off the mirror, is re-collimated, and sent into the secondary beam and fringe tracker. If two stars are close enough together on the sky, both of their

images will be incident on the field separator. The siderostats may be guided so that one star image passes through the field separator while the other reflects. The result is the ability to measure fringes simultaneously from two stars. A beam splitter may also be used as a field separator, as other steerable optics in the light path may be used to select which part of the beam splitter is used as the input for the secondary beam.

Both the primary and secondary beams pass through the same delay line cart, so both stars are given the same delay compensation. In addition, the primary beams pass through a second, smaller delay line which is used to introduce a delay relative to the secondary star. The delay lines are in air, and so the beam combining area and the delay lines are housed in an enclosure built inside a larger building, much like the design of SUSI and the 11.4 m Prototype Interferometer.

Because the two stars are close together on the sky, their atmospheric phase perturbations are highly correlated. Fringes of faint stars that are in close proximity to brighter ones may be measured by using the brighter star as a phase reference in the primary beam to remove atmospheric jitter with the main delay line, while the secondary delay line is used to modulate OPD with respect to the bright star.

The phase referencing technique also provides accurate measurements of the angle between the two stars measured in the direction of the baseline. An accurate determination of the delay in the secondary delay line is equal to the baseline multiplied by the angle in radians. This delay may be measured very

accurately because the main delay line removes the atmospheric jitter for both stars at the same time. In order to get the angle right, both the baseline and the internal optical path of the instrument are monitored with laser metrology systems.

PTI, like the Mark III before it, is highly automated, and as a result is one of the most scientifically prolific interferometers to date.

- 1998-Present: The Mikata InfraRed and optical Array

Mikata InfraRed and optical Array (MIRA) is a 2-aperture interferometer with reconfigurable baselines up to 30 m, with further baseline extensions pending. MIRA is a series of prototype interferometers, representing the Japanese interferometry community's path to the development of a large aperture, long baseline interferometer. Each phase of the project carries the MIRA name.

MIRA-I.1 is the first iteration of the MIRA project. Built as a proof-of-concept instrument, it has 0.13 m siderostats, initially on a 4 m baseline, and operates at 800 nm. It has light pipes and a 4 m continuous delay line that are designed to be evacuated. In addition to the continuous delay line is a coarse delay line that provides delay in a few discrete lengths. Beam combination is in the pupil plane via a beam splitter. MIRA-I.1 obtains stellar fringes on 9 stars, measures some of their diameters, and conducts fringe tracking experiments during its short operational lifetime of October 1998 to March 1999. (Nishikawa et al., 2000)

Mira-I.2 has 30 cm siderostats on a 30 cm baseline. It uses polarization preserving single mode optical fibers instead of the light pipes. Longitudinal dispersion

and polarization compensation systems are developed for the optical fibers. The continuous delay line is evacuated and lengthened to 8 m. Beam combination is again in the pupil plane via a beam splitter. Operating at 800 nm with an APD detector, MIRA-I.2 finds first fringes on Vega in 2002. A 3-color fringe detection system in the 600 nm - 1000 nm range is under development (Kotani et al., 2003; Nishikawa et al., 2004).

- 1999 - Present: The CHARA Array

Georgia State University's Center for High Angular Resolution Astronomy (CHARA) Array utilizes six 1 m telescopes in a Y-shaped array with baselines of up to 340 m (ten Brummelaar et al., 2005). CHARA operates in the near-IR (mostly K band, with some H band development) with a new visible to near-IR (600 nm-1000 nm) capacity, described in this dissertation. See section 1.3 for a complete description of the CHARA array.

- 2001-Present: The Keck Interferometer

The two 10-meter Keck telescopes on Mauna Kea are combined to make the Keck Interferometer (Keck-I) with a baseline of 80 m and a massively improved light gathering power over the previous generation of interferometers (Colavita et al., 2004).

The Keck-I is designed with three separate functions in three wavelength regimes. Fringe visibility measurements are made at 2.2μ using a pupil plane beam combiner that injects the output beams into single mode fibers which in turn feed the light into the camera. One output of the beam combiner is used as a white light fringe, while the other is spectrally dispersed over 4 pixels. The fringes are

phase locked and their visibility measured via a 4-bin algorithm based on the PTI and MarkI-III models. Another detector is used in the 1.5-6 μm regime to measure the difference in fringe phase with wavelength, which has the possibility of detecting ‘hot’ Jupiters, with surface temperatures around a few hundred degrees Kelvin. The ratio of the planet’s brightness to its parent star is higher in the infrared, near the peak of the planet’s black body emission curve. In the 10-12 μm range, a third nulling system is used to detect exozodiacal emission around nearby stars.

First fringes are found with the Keck telescopes in 2001, and the first scientific results published in 2003 (Colavita et al., 2003).

A potential extension of the Keck-I is under development in the form of four 1.8-meter outrigger telescopes that will be positioned around the Keck building to extend the u, v plane coverage and provide more dedicated observing time for the interferometer. The proposed outrigger telescopes will provide baselines up to 140 m.

- 2001 - Present: The Very Large Telescope Interferometer

The Very Large Telescope Interferometer begins operation with four 8 m Unit Telescopes (UTs) and baselines of up to 130 m at Cerro Parnal, Chile (Glinemann et al., 2004). Three Auxiliary Telescopes (ATs) will be added later, starting in 2004, which may be repositioned to 30 stations around the site, providing dense u, v plane coverage at baselines up to 202 m.

The beams from the telescopes are transported through underground tunnels to a delay line tunnel and beam combining laboratory, also underground. The

beam combining lab is home to many independent beam combiners and detectors, listed below.

The commissioning combiner, VINCI, is a fiber combiner patterned after FLUOR. Light is injected into two single mode fibers. Each splits to provide a photometric output and a combiner input. The two combiner input fibers cross in a X junction which acts as a 50-50 beam splitter. The combiner outputs are then sent into a K-band ($2.2\ \mu\text{m}$) detector. First fringes are found with VINCI in 2001.

MIDI, a second combiner, operates at N band ($8\text{-}12\ \mu\text{m}$), and combines 2 beams in a spectrograph with a resolution of 200. In 2003, the MIDI makes the first observation of an extragalactic source, which is followed by K band observations with VINCI (Wittkowski et al., 2004). MIDI may have its operating wavelength extended to Q band, $17\text{-}25\ \mu\text{m}$.

AMBER is a 3-beam combiner which operates in the J, H, and K bands, from 1 to $2.5\ \mu\text{m}$, with a spectral resolution of up to 10000. The 3-beam capacity makes it the first detector at VLTI which may be used for measuring closure phases. AMBER is expected to reach $K=14$, with the aid of a fringe tracker, and $K=20$ with a dual feed phase reference system and bright nearby star.

PRIMA is the dual feed phase referencing system designed similar to PTI's primary and secondary beam system with a field separator. With a guide star brighter than 12th magnitude in H band ($1.65\ \mu\text{m}$) inside the same isoplanatic patch of sky as the object, the guide star can provide a fringe tracking signal for removing atmospheric piston errors from the object. PRIMA includes two fringe

tracking sensors. One is for the guide star, and the other may be used on the object star instead of AMBER or VINCI to measure very accurate relative delay between the two stars, and hence their separation angle. PRIMA undergoes implementation in 2005.

FINITO is a fiber-fed 3-beam fringe tracker that works in H band ($1.65\ \mu\text{m}$). It drives the PZT-mounted secondary on the delay line cat's eye retroreflectors, enabling it to provide fringe tracking for any of the combiners.

MACAO is the adaptive optics system, a visible band curvature sensor and 60-actuator bimorph mirror in the Coudé path.

In 2004, the integrated optics combiner IONIC moves from IOTA to VLTI.

In 2004 VLTI begins to produce scientific results. The large amount of resources furnished to the project begins to pay off as many papers are published throughout 2005.

Interferometers under development

- The Magdalena Ridge Interferometer

The Magdalena Ridge Interferometer (MRI) on Magdalena Ridge, outside of Socorro, New Mexico, is planned to be an imaging array, a larger, more extensive version of the CHARA array with 10 telescopes of 1.5 m aperture and baselines from 8 to 400 m.

The instrument throughput is being stressed in the design. Extensive work has been done at COAST to develop high quality optical coatings that will boost throughput via improved efficiency.

The beam train has only 15 reflections (CHARA has 23), made possible by an ALT-ALT telescope design which produces a stationary Coudé output beam after only 3 mirror surfaces. An ALT-AZ design requires 6 reflections.

Three more reflections may be saved by forcing all optical delay compensation into one long delay line, rather than delay done in coarse and continuous lines like at most facilities. The delay line is designed to run on the inside of an evacuated pipe, with radio communication and inductive power transfer so that no cables hamper its movement.

Starlight will be separated into visible J, H, and K bands, each sent to its own beam combining optical table. Fringe tracking will be performed in H or K band, while science observations may be performed in any band. Beam combiners made of solid pieces of glass cemented together will give the system fewer degrees of freedom, reducing vibration sensitivity and simplifying alignment of the system.

The visible camera will likely be an E2V back illuminated CCD87 chip, which has a multiplication readout register and may be used in a photon counting mode with effectively zero read noise. The IR detectors are still under development. First fringes are anticipated by 2007, with first closure phase by 2008 (Creech-Eakman et al., 2004).

- The Large Binocular Telescope

The Large Binocular Telescope (LBT) in Arizona will consist of two 8.4 m telescopes, mounted side by side on a single monolithic altitude-azimuth mount. The two apertures will be spaced with their centers 14.4 m apart, giving the

instrument a 22.8 m baseline edge-to-edge and a continuous u, v plane coverage within that distance.

The co-mounted telescopes always keep the aperture plane perpendicular to the star, which enables the instrument to image over a field the size of the isoplanatic angle, provided that the wavefronts from both apertures are adaptively smoothed and cophased.

The beams combine in the image plane, and fringes across the Airy disk provide some extra resolution in the direction of the baseline. As the Earth rotates, the baseline is rotated with respect to the star, rotating the u, v coverage of the telescope.

The LBT will operate in a broad wavelength range, from 400 nm to 40 μm . The instrument should have first light on both primary mirrors by 2006 (Hill and Salinari, 2004).

- The Optical Hawaiian Array for Nanoradian Astronomy

The Optical Hawaiian Array for Nanoradian Astronomy (OHANA) is a project to link 6 of the telescopes already in operation on Mauna Kea via single mode, polarization preserving optical fibers. The telescopes range in aperture size from 3.5 m to 10 m, and baselines of up to ~ 1 km.

The fibers will transport the light to bulk optic delay lines, then beam combination and detection will occur in the near-IR (Perrin et al., 2000, 2004).

- The Space Interferometry Mission

The Space Interferometry Mission (SIM) will be launched into orbit in 2012 or

later. It is a pair of flat 7 cm mirrors mounted on a boom, with a baseline of 9 m that can fill a circle in the u, v plane by tilting and rotating the boom with respect to the star.

SIM will conduct high precision astrometry, measuring star positions with respect to a carefully selected reference grid of background stars. SIM will search for extrasolar planets by detecting the wobble induced in a star's motion by its satellites (Shao, 2004).

- The Terrestrial Planet Finder

The Terrestrial Planet Finder (TPF), is the successor to SIM, and will expand extrasolar planet research. TPF is actually two projects. TPF-C is a nulling coronagraph designed to find planets by cancelling out the light from a star and detecting companions just outside the nulling window (Ford et al., 2004). TPF-I is a formation flying array of four 4 m telescopes (Miller and Fischer, 2004). TPF-C is nominally scheduled for launch in 2014, while TPF-I is set for 2020.

- The European Space Agency's mission to find extrasolar planets is called Darwin. Several designs are in development, including a coronagraph and a formation flying interferometer.

1.3 The CHARA Array

The CHARA Array sits atop Mount Wilson, sprawled around the 100-inch telescope dome in a Y shape, with two 1-meter telescopes in each of its arms. The telescopes are named after the direction of the arm which they occupy, in order from the outside

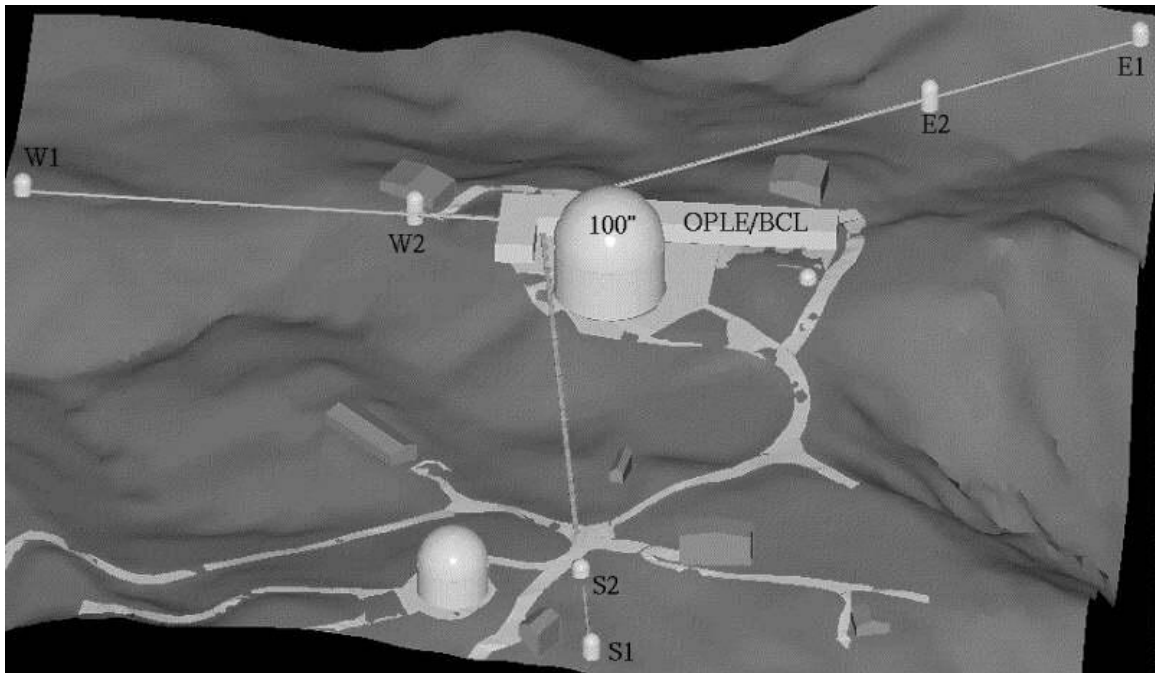


Figure 1.5: A CAD rendering of a 3D model of the CHARA Array and the surrounding grounds of the Mount Wilson Observatory. The OPLE/BCL building and telescopes are labeled.

in, ie., S1 is the end telescope on the South line, with S2 closer to the center of the Array.

The various components of the Array are described below, in the order which the incoming starlight encounters them, telescopes, light pipes and turning boxes, delay lines, beam reducers, beam switchyard and beam combining lab being the main components.

An array with n collectors has $\frac{(n)(n-1)}{2}$ baseline combinations, so the density of the u, v plane coverage increases dramatically with each telescope added. While 6 telescopes is certainly a very large number to control simultaneously, the resulting

15 baselines barely provide a dense enough u, v plane coverage to be used for model-independent imaging. Earth rotation helps to fill in the u, v plane by changing the projection of the array baselines on the sky. Observations at different wavelengths may also be used together, provided the source has the same intensity distribution in all wavelengths. This increases u, v plane coverage because the length of the baseline vector in u, v space is $\frac{B_{\text{proj}}}{\lambda}$, where B_{proj} is the projected baseline.

Without earth rotation and use of multiple wavelengths, CHARA still has sufficient u, v coverage to adequately constrain simple source morphology models. In particular, CHARA is ideal for stellar diameter measurements and binary star orbits, as the available baselines can tie down the parameters for such models in two dimensions with a single observation with all 6 telescopes.

Before 2005, CHARA was operated primarily in the K' band ($2.13 \mu\text{m}$), one pair of telescopes at a time. As the array has been increasingly automated over the years, it has become possible to operate more than two telescopes simultaneously. During the 2005 observing season, the MIRC fiber-based combiner recorded data on 4 telescopes simultaneously, measuring closure phases on 4 different baseline triangles and 3 closure amplitudes. This represents CHARA's first step into the domain of imaging, rather than just measuring squared visibilities one baseline at a time. With this new capability, CHARA's scientific productivity should increase dramatically.

To date three scientific papers have been published in refereed journals from data taken at CHARA, with another two in the review process, plus an instrument paper and a catalog of binary stars to serve as targets. They are presented in chronological order:

1. *The CHARA Catalog of Orbital Elements of Spectroscopic Binary Stars*, is an updated catalog of binary stars from which CHARA targets are chosen (Taylor et al., 2003).
2. *First Results from the CHARA Array. I. An Interferometric and Spectroscopic Study of the Fast Rotator alpha Leonis (Regulus)*. Interferometric squared visibility measurements coupled with spectroscopic data are used to measure Regulus. Its oblateness from rotation, its orientation, rotational velocity and period, temperature at equator and poles, and gravity darkening are obtained. This represents the first measurement of gravity darkening on a rapid rotator, and the first gravity darkening measurement from a star which is not an eclipsing binary (McAlister et al., 2005).
3. *First Results from the CHARA Array. II. A Description of the Instrument*. The CHARA array and its data analysis methods are described in detail (ten Brummelaar et al., 2005).
4. *The Projection Factor of delta Cephei: A Calibration of the Baade-Wesselink Method Using the CHARA Array*. Using the FLUOR fiber beam combiner, the angular size of δ Cephei is measured as it pulsates. When coupled with a parallax measurement, this yields a physical diameter, enabling a calibration of the luminosity, and hence the Cepheid Period-Luminosity relation. The Period-Luminosity relation is used to determine distances to other galaxies, where geometric parallaxes are impossible to obtain. This is an independent calibration of the Period-Luminosity relation's zero crossing, and agrees well with recent model predictions (M  rand et al., 2005).

5. *First Results from the CHARA Array. III. Oblateness, Rotational Velocity and Gravity Darkening of Alderamin* Angular diameter measurements of Alderamin show that it is oblate, with a brighter spot near the pole. As with the Regulus paper, the visibility data are fit to a model, and a best fit value is found for oblateness, inclination, surface temperature distribution, and gravity darkening. Parallax data give the star's physical size, and radial velocity data give its true rotational velocity which is $\sim 85\%$ of breakup speed (van Belle et al., 2005).
6. *First Results from the CHARA Array. IV. The Interferometric Radii of Low-Mass Stars* The low-mass end of the main sequence has very few stars whose diameters are known, due to their low luminosity. The diameters of 6 low-mass main sequence stars are measured, which when coupled with a parallax gives a physical diameter which is used to calibrate their luminosity (Berger et al., 2005).
7. *First Results from the CHARA Array V. Binary Star Astrometry: the Case of 12 Persei* Binary star separations may be measured by the separation of the two components' fringe packets in the same dither scan, when the delay between the fringe packets is larger than the width of each packet. An orbit for the resolved, double-lined spectroscopic binary 12 Persei is presented, with much improved precision over previous orbits. An intensity ratio is given for the two components, as well their masses (Baguolo et al., 2005).

Several more papers based on data taken during the 2004 and early 2005 observing seasons are currently in the submission process, and the body of CHARA published science should double within a year. As CHARA begins to mature into a true observ-

ing facility, the demand for observing time on the array has skyrocketed, forcing the Time Allocation Committee to choose very carefully among the various proposals. With the array in constant use throughout the year by so many investigators, the CHARA Array promises to amass a large body of scientific publications.

1.3.1 The Telescopes

Each CHARA telescope reduces incoming starlight from a 1 m beam to a 125 mm beam. Each reflective surface that the beam encounters is named after its order in the optical train. The telescope primary mirror is M1, and the secondary is M2. The beam is transferred by means of four fold mirrors through the elevation and azimuth axes of the scope, and down to the Coudé box at the base of the telescope pier. M3 catches the reduced beam from the secondary and folds it out through the telescope elevation axis. M4 sends the beam down the side of the fork mount, M5 sends the beam horizontally to the telescope azimuth axis, and M6 sends the beam straight down through the azimuth axis to the Coudé box at the base of the telescope pier.

Tip/tilt of the starlight beam is controlled by tilting the secondary mirror, which is mounted on three actuators. The tip/tilt detectors are in the Beam Combining Laboratory.

Each telescope is equipped with TEMA (the TElescope MANager), a video multiplexer and servo control system which communicates with the control computers. TEMA handles video signals from the acquisition and guide cameras, as well as two cameras that image the scope itself and are used to check its status remotely. In addition, TEMA controls the various mirror covers, enabling the observer to open up the telescopes and close them down without walking out to them, all while watching from

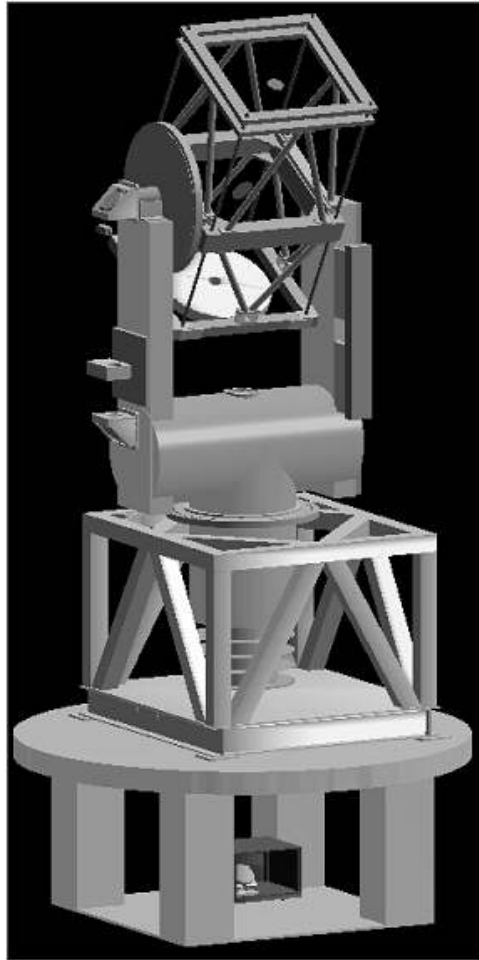


Figure 1.6: A CAD rendering of a CHARA telescope 3D model. The primary and secondary (M1 and M2) form an afocal beam reducer. A flat tertiary (M3) folds the beam out through the center of the altitude axis. The beam is steered down the side of the fork by M4, to the azimuth axis by M5, and down through the center of the azimuth axis by M6.

the remote cameras on the scope. TEMA has greatly increased CHARA's observing efficiency by making many aspects of the alignment routine remotely controllable.

1.3.2 The Light Pipes

One or more fold mirrors in the Coudé box steer the beam through a window and into an evacuated light pipe which leads to the central building.

Polarization effects are kept to a minimum by making sure that the beam from each telescope experiences the same reflection angles in the same order (Traub, 1988). Because the array is ‘Y’-shaped, the beams from telescopes in the different arms come toward the central facility from completely different angles. A single fold mirror in each Coudé box would introduce a different reflection angle for each arm of the array.

Because there are three arms to the array, the answer lies in using three mirrors . The three mirrors are positioned in a triangle, such that the beam direction in each leg of the triangle is parallel to one of the arms of the array, as shown in Figure 1.7. M7 sends the beam parallel to the East arm, M8 sends the beam parallel to the West arm, and M9 sends the beam parallel to the South arm. The triangle is split between the Coudé box and the turning box at the other end of the pipe, in the central building.

S1 & S2 have M7-9 the Coudé box, and M10 in the turning box, so the long side of the triangle transports the beam from the Coudé box to the turning box. Similarly, E1 & E2 have only M7 in the Coudé box, while W1 & W2 have M7 & M8 in the Coudé box. After the beam has left M9, it hits M10, the last mirror in the turning box. M10 steers the beam down the POP pipe.

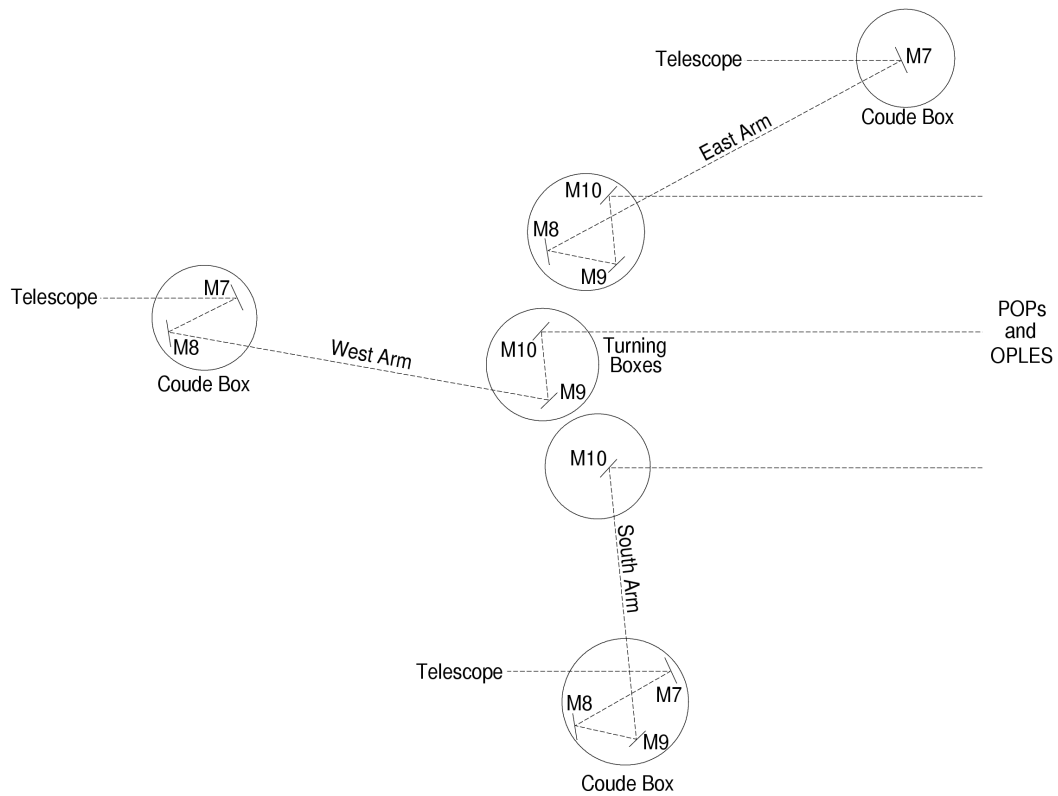


Figure 1.7: The M7-M10 configuration. M7 receives the beam at the bottom of the telescope's azimuth axis, and M10 sends the beam into the POP delay lines. The beams from each telescope must go through the same reflection angles in the same order to prevent differences arising in polarization of the starlight. The light path from M7 to M8 is parallel to the East arm, M8 to M9 is parallel to the West arm, and M9 to M10 is parallel to the south arm. For each arm of the interferometer, the distance is lengthened for the mirror pair which is parallel to that arm, while the other two distances are kept short, inside either the Coudé or turning boxes. The distances shown are not to scale, in order to more readily show the M7-M10 triangles.

1.3.3 The POPs

The POPs are the part of the system which introduces discrete amounts of optical delay to the starlight beams. The POP acronym is short for "Pipes of Pan," as an early concept of the system called for several pipes of various lengths.



Figure 1.8: The POPs, periscope, and OPLE, as seen looking East. The beam from the telescope travels East down the POP pipes (bottom left) until they reach whichever POP station has its mirror in the beam. The beam reflects at the appropriate POP station, then returns westward and enters the periscope (far right). The periscope is a set of two mirrors which relay the beam upward, out of the evacuated POP pipe through a window, then Eastward to the OPLE cart (top center).

The POPs are evacuated pipes, large enough for two beams to fit side by side, with 6 mirror stations, one every 15 meters from the beginning of the POP pipe. Each POP mirror (M11) is mounted on a swinging arm. When activated, the mirror swings into the beam, sending it back to the start of the POP pipe, where it hits

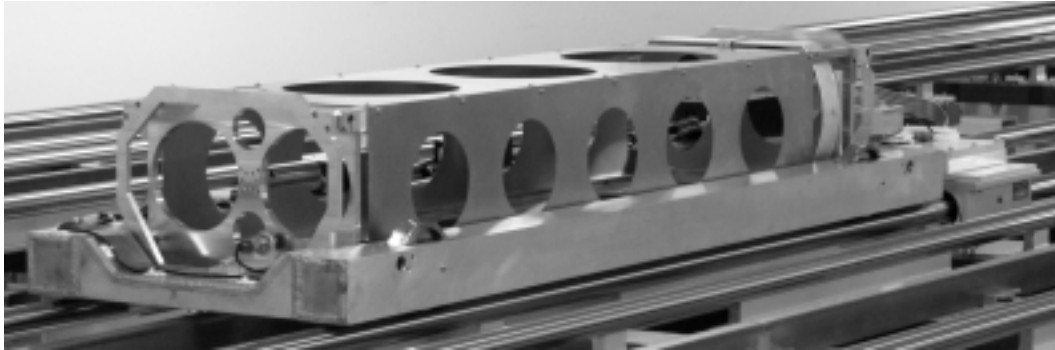


Figure 1.9: The OPLE cart.

M12, the lower periscope mirror. The POPs are used to introduce optical delay to the beams in discrete 30 meter segments.

The beam going down the POP is not parallel to the return beam, because M10 is at a different location than M12. This means that either M10, M12, or both need to change their angle when the POP mirror in use is changed. M10 is placed such that it steers the beam parallel to the axis of the POP pipe, so that each POP swings into the beam without any adjustment in M10's angle. As a result, the lines connecting M12 to each of the POP mirrors are at slightly different angles. M12 is mounted on an automated turntable to enable adjustment of its azimuthal angle. As POPs are switched, M12 rotates slightly to catch the beam from the appropriate POP mirror. After hitting M12, the beam reflects straight upwards, out of the vacuum system via another window, and hits M13, the upper periscope mirror. M13 sits just above the POP pipe, in line with the OPLE rails.

1.3.4 The OPLEs

The OPLE, or Optical Path Length Equalizer, is the system for introducing continuous amounts of optical delay into the beam. The OPLE system consists of 6 carts, which ride on rails mounted directly above the POP pipes. The OPLE carts are movable cat's-eye reflectors; each cart consists of a parabolic primary mirror and a flat secondary, placed at the focus of the primary. A beam which hits the primary will be focused onto the secondary, back to the primary on the other side, and leave the OPLE cart at the same angle that it entered. The cat's eye is a good configuration for a movable delay optic because it is insensitive to angular errors. The OPLE primary acts as M14 & M16, because it gets hit twice by the beam, while the OPLE secondary is M15.

The OPLE carts have a 3-stage architecture for controlling optical delay:

1. The rear portion of the cart is a separate piece which contains the drive motor, and is coupled to the main cart via a voice coil. The drive motor runs a small wheel which is held against the side of a flat drive rail which runs down center of the OPLE rail line, controlling delay in the 2 Hz frequency range. The voice coil enables the motor segment to move the cart while filtering out mechanical vibrations from the motor, and handles delay changes in the 20 Hz range.
2. The retroreflector optics are mounted in a cage which swings from a four-bar mechanism, keeping the cage at the same angle relative to the cart body at any position in its swing. The cage is coupled to the cart body via a second voice coil, which filters out mechanical vibrations due to the cart body rolling along the rails. This voice coil handles the 200 Hz frequency regime of delay changes.

3. The cat's-eye secondary mirror is mounted on a piezoelectric transducer (PZT) stage. By varying the voltage to the PZT, the secondary may be moved a small amount toward or away from the primary mirror. This introduces a small amount of optical delay, while the focus of the cat's eye return beam remains unaffected if the PZT displacement is small. The PZT has a very fast mechanical response time to voltage changes, so it is useful for handling high-frequency (5 kHz) OPD changes on the order of 100 μm .

The main cart body never contacts the drive rail. It is supported by three stainless steel wheels which run on two round rails on either side of the drive rail. Three wheels provide three points to define the orientation plane of the cart body. The cart cannot have a suspension system, as this would complicate defining the cart orientation. Four wheels held in rigid position would define four contact points. If those points are not exactly coplanar, the result is the definition of two orientation planes, and the cart will rock back and forth between them, only ever resting on a maximum of three wheels at a time.

The southern side of the cart body is supported by two wheels with dovetailed contact surfaces. The north side of the cart body is supported by a single flat-surfaced wheel. The dovetailed wheels force the cart to follow the position of the southernmost rail in the plane perpendicular to the rail line. For this reason the southern rail is called the guide rail. The flat wheel on the north side allows the north rail, called the support rail, to determine the tilt of the cart but not its position. Working together, the guide and support rails determine the position and angle of the cart without over-defining it.

The OPLE carts' position is monitored by a metrology system which fires a $1.3\ \mu\text{m}$ laser from the BRT tables, through the OPLE cart, and back to the BRT table where it hits a mirror which sends the laser beam back the way it came, a total of 4 passes along the delay line. The metrology system detects a Doppler shift in the return laser beam when the carts move, thus measuring cart's velocity very accurately. By integrating the cart velocity, the metrology system keeps track of the cart position to within 2 nm resolution.

The OPLEs are used to maintain tight control on the relative optical delay between pairs of telescopes. They provide the continuous delay tracking necessary to compensate for constantly changing delay requirements caused by targets moving across the sky as the Earth rotates.

Because the OPLE carts introduce delay in air, it is imperative that they be in a thermally stable environment with as little air turbulence as possible. As per the design of previous interferometers with air delay lines (the 11.4 m Prototype Interferometer, SUSI, and PTI) the OPLE lines and beam combination lab are built in a nested building-within-a-building. All climate control is performed in the intervening space between the inner and outer layers of the building, keeping the OPLE and BCL thermally stable without inducing air currents.

1.3.5 The BRTs

After passing through the POPs and OPLEs, each 125 mm beam is shrunk to 19 mm by its Beam Reducing Telescope. Each telescope is an afocal Cassegrain. The beam strikes the primary mirror to one side of the central hole, is reduced by a factor of 6.6 as it propagates to the secondary, and reflects off the secondary and passes through

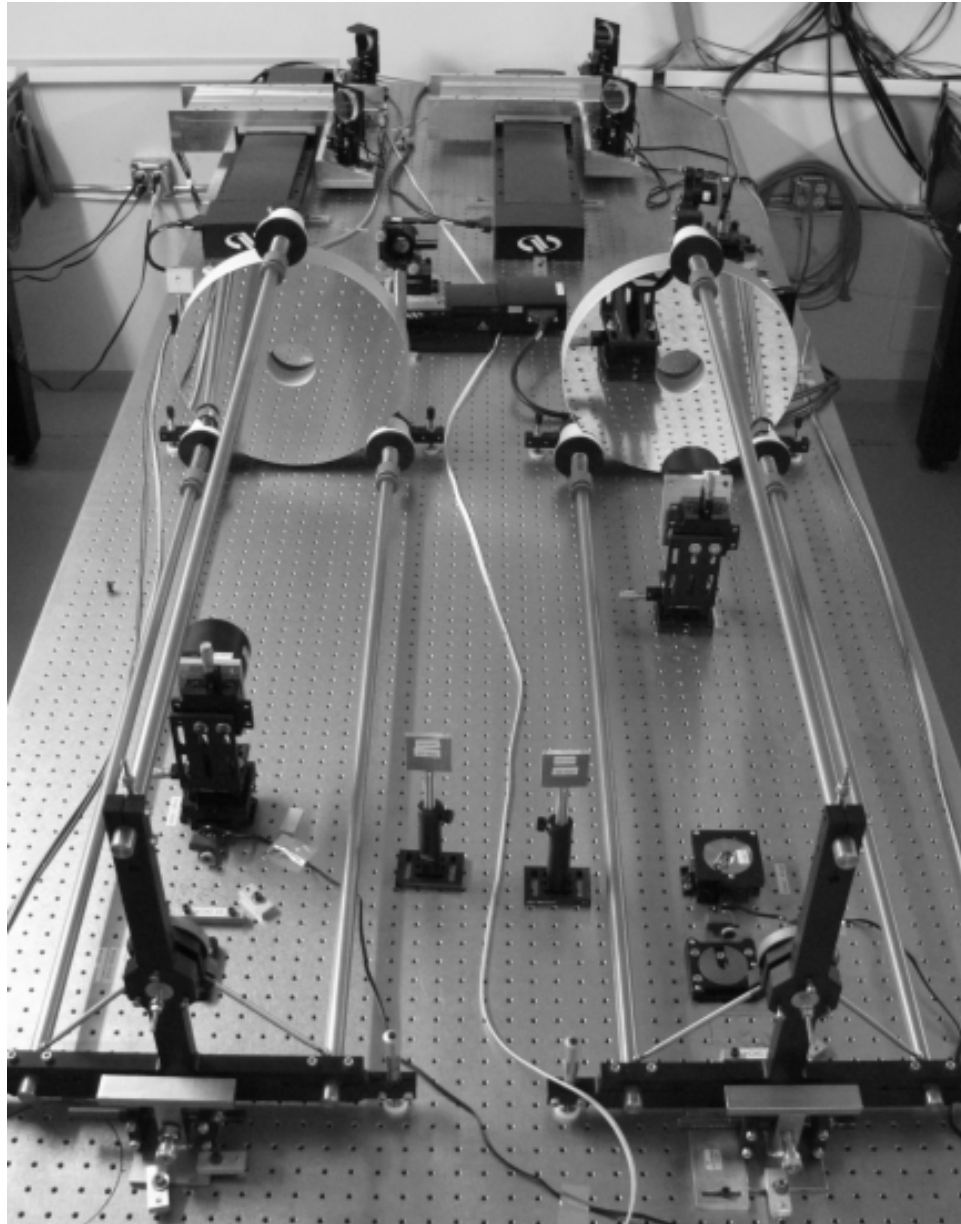


Figure 1.10: The Beam Reducing Telescopes and Beam Switchers.

the central hole in the primary as a collimated beam. As the beam is shrunk by 6.6, the angle of any incident light rays with respect to the optical axis is magnified by 6.6.

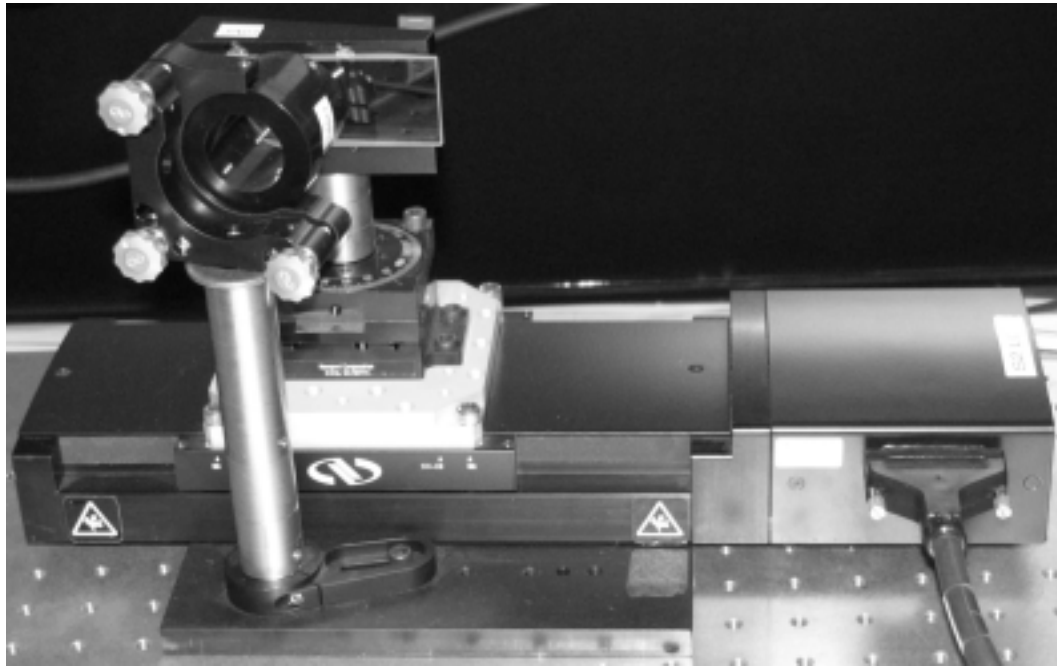


Figure 1.11: The Longitudinal Dispersion Compensators. The long wedge slides past the shorter, fixed wedge to control the thickness of the glass in the beam. The glass is used to introduce dispersion into the line with less in-air delay, so that both beams have the same amount of dispersion when they reach the beam combiner.

1.3.6 The LDCs

The Longitudinal Dispersion Compensators (LDCs) were designed and built by David Berger (2003) to compensate for differential dispersion introduced into the beams as a result of using delay lines in air.

When the telescope baseline is not perpendicular to the light coming from a star, the wavefront reaches one telescope before the other. Figure 1.3 was used in section 1.1.2 to illustrate this situation. Both beams have passed through the same amount of atmosphere (except for small fluctuations due to atmospheric turbulence)

before they reached the telescopes, so the delay between the two telescopes occurred in the vacuum of space. This delay is referred to as *geometric delay*, as it is a result of the baseline-star geometry.

When air-filled delay lines are used to compensate geometric delay, the line with more delay added has traveled through a longer path in air. The index of refraction of air increases toward shorter wavelengths, so the blue wavelengths are retarded with respect to the red. The zero optical path position now has a wavelength-dependent position.

To understand what this will do to the fringes, consider that the white light fringe packet is made of the sum of narrow-band fringe packets added together across the entire wave band. The phase of all of those fringe packets is identical at the central fringe, so the fringes across all wavelengths interfere constructively there. If the zero OPD position is wavelength dependent, the fringe packets for each wavelength are shifted a little with respect to neighboring wavelengths. If the white light bandwidth is wide enough, and the uncompensated dispersion severe enough, the blue fringes may be shifted so far from the red ones that they tend to blur out when added together. Berger showed in his dissertation that the result of uncompensated dispersion is a reduction in fringe visibility and distortion of the fringe envelope.

The mismatched air paths may be compensated for by introducing the right thickness of glass into the beam with the shorter air path. If the index of refraction curve of the glass with respect to λ is proportional to that of air, then a given amount of glass introduces longitudinal dispersion equivalent to a given amount of air.

Dispersion is added to the short air-path beam until its air+glass dispersion is equal to the air dispersion of the other beam. Note that the blue wavelengths are still

retarded with respect to the red ones, but they are retarded equally in both beams, so the coherence of the light at the beam combiner will remain unaffected.

In practice, the glass is introduced into both beams in the form of a pair of sliding prisms. Their outer surfaces are parallel so that the starlight beams are not deviated as they pass through. The thickness of the glass is controlled by sliding one prism past the other on a motorized, computer controlled translation stage, with the axis of translation parallel to the plane of the interface between the prisms. Both beams have glass in them, but the beam with less air delay is simply given more glass. It is the difference in glass thickness between the two beams which is important.

The effects of longitudinal dispersion are weak enough in K' band ($2.13\ \mu\text{m}$) that the LDCs are not necessary. However, in the VIS system they become significant. The VIS system spectral bands are narrow enough that the fringe visibility is not significantly reduced, but the fringe position of the VIS band fringes is offset from the K' band fringes by an amount which depends on the baseline and zenith angle, due to the amounts of air delay they require. Even with delay matched to within 20 m in the evacuated POPs, the remaining OPLE air delay can be enough to offset the VIS fringes by half of the dither sweep, which makes finding fringes difficult when the K' system is being used to locate the fringes for the VIS system. The LDCs eliminate the differential delay between K' and the VIS band, greatly simplifying their use with large amounts of OPLE delay.

1.3.7 The Beam Sampling System

The Beam Sampling System (BSS) is used to direct the beam from any telescope into any beam combiner input. A dichroic splitter and a flat steering mirror are mounted



Figure 1.12: The Beam Sampling System. As light leaves the BRTs and LDCs (left) it reaches a dichroic splitter and fold mirror which are mounted at a fixed distance apart on a computer-controlled translation stage (pictured at right). The dichroic reflects wavelengths short of $1\ \mu\text{m}$ into the VIS table input beams, while the fold mirror steers the IR light into the corresponding IR input beam. The translation stage is moved to select which input beams a particular telescope uses. The VIS and NII tables are visible in the background.

on a small table on a computer-controlled translation stage. The dichroic reflects wavelengths short of $1\ \mu\text{m}$ into the VIS table input beams, while the mirror folds the IR light which was transmitted through the dichroic into the IR input beams.

All of the input beams for both the visible and IR systems are parallel, and the IR beams are a fixed distance from their visible counterparts. The translation stage axis is parallel to the light path coming out of the BRT, so moving the Beam Sampler changes which input beam the telescope's light enters.

The Beam Sampling System gives CHARA the flexibility to select whatever telescopes the observer requires for their desired baselines.

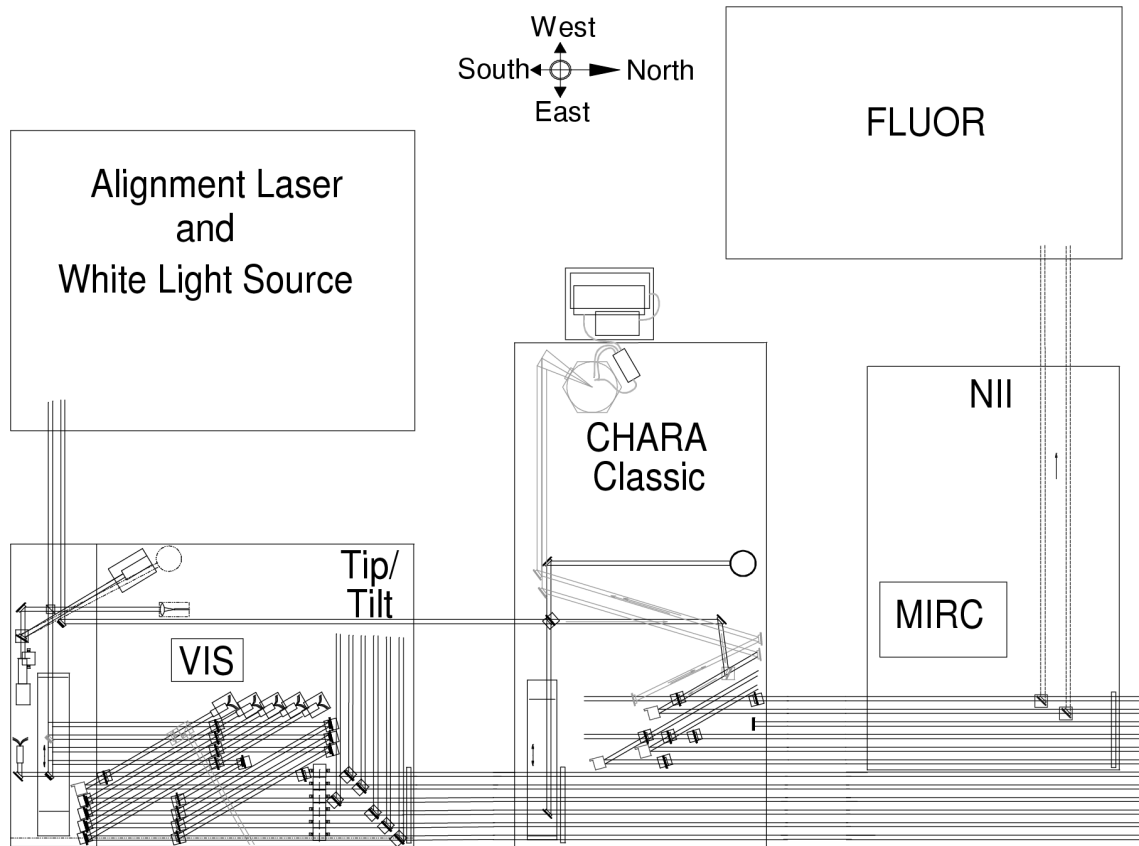


Figure 1.13: The Beam Combining Laboratory layout. The 6 VIS input beams are at the lower right, with the 6 IR input beams west of them. The beams are numbered VIS and IR 1-6, with the #1 beams to the east, and beam number increasing westward. The various optical tables are described in the text.

1.3.8 The Beam Combining Laboratory

After leaving the Beam Sampling System, the visible and IR beams enter the Beam Combining Laboratory (BCL) from the southeast corner, heading due south. The purpose of each optical table is described below, while the layout of the lab is shown in Figure 1.13.

NII

The first table which the IR beams reach is the Near Infrared Imaging (NII) table. The NII table currently holds the MIRC beam combiner, described below, and pick-off mirrors which send two beams to the FLUOR table when FLUOR is in use. MIRC uses pick-off mirrors as well, so when they are removed from the beams, the IR input beams can continue on to the CHARA Classic table.

CHARA Classic

The CHARA Classic table is just to the south of the NII table, in the center of figure 1.13. This table is where the first CHARA beam combiner was constructed, using four detectors, two in H band and two in K band. Since then CHARA has switched to using the NIRO camera, which is now placed on this table as well, along with the most recent iteration of the CHARA default IR beam combiner. With several beam combiners available to choose from, the default combiner has acquired the name “CHARA Classic,” a reference to its status as the workhorse of the Array, and to the fierce loyalty of the people of Atlanta to certain brand names with which they are comfortable and familiar.

A secondary laser alignment beam, called the “engineering beam,” is sent directly to the CHARA Classic table, circumventing the VIS beam combiner, where the main alignment beam is injected into the back end of the CHARA beam train. The engineering beam may be introduced into one of the outputs of the CHARA Classic combiner as a way of aligning from the back end of the IR system rather than the visible. A fold mirror on a computer-controlled translation stage may be used to

introduce the engineering beam directly into any of the VIS input beams, heading outwards toward the telescopes. This provides a way to circumvent the VIS beam combiner, so that one person may send the engineering beam to a particular telescope while other telescopes are used for observing.

FLUOR

The aptly named FLUOR table holds the FLUOR combiner. It is west of the NII table, upper right in Figure 1.13.

VIS

In the southeast corner, the VIS table holds the 6-way VIS beam combiner, described in Chapter 3. It also holds the tip-tilt detectors, an intensified CCD used to align the tip-tilt detectors to the beam train, a small theodolite used for co-alignment of the laser and white light source, and a low resolution prism spectrograph, separate from the VIS spectrograph, which is used to locate the white light fringes.

Laser / White Light

Just to the west of the VIS table, this table holds the laser and white light sources. They are combined by means of a beam splitter, so that when they are aligned properly both beam splitter outputs contain coaligned white light and laser beams. One output goes to the back end of the VIS combiner, where it is introduced into the system for alignment, and the other output is the engineering beam.

A medium resolution fiber-fed grating spectrograph is also housed on the Laser / White Light table. It is compatible with the VIS system fiber injectors, so it may

be used for interferometric measurements as long as the VIS system is slaved to a phase-locking fringe tracker.

1.3.9 Tip-tilt Compensation Systems

PMT-based

The original tip-tilt system for CHARA, this tip-tilt system uses quad cell photomultiplier tubes (PMTs) which are very blue sensitive. A long focal length camera lens images the star onto the quad cell. They are aligned such that when the image of the star falls on the junction between all 4 quadrants, the signal in the 4 channels is even and the star light is exactly aligned with the optical axis of the entire CHARA system.

When the wavefront at the telescopes has a tilt component, the image of the star is displaced, and the signal in the quadrant where the image went is much stronger. The tip-tilt servo simply changes the angle of the telescope secondary to keep the star image at the center of the quad cell.

The light for this tip-tilt system is separated from the visible input beams by dichroic splitters which pick off wavelengths shorter than 600 nm and send them into the tip-tilt detectors.

The PMT-based tip-tilt system was used at CHARA exclusively until the end of the 2005 observing season, when the more sensitive CCD-based tip-tilt system (described below) came on line. All data taken for this dissertation utilized the older PMT-based tip-tilt system.

The extreme blue operating wavelength of the PMT tip-tilt system is problematic

when observing cool, dim stars, as the limiting magnitude of the tip-tilt system is much brighter than that of CHARA Classic.

CCD-based

In September of 2005, a tip-tilt system designed and built by Lazslo Sturmann was tried out on the sky, and immediately worked well. It uses the second of CHARA's ARC CCD cameras, its twin being the detector for the VIS spectrograph, described in Chapter 3. Up to 9 input beams are imaged onto regions of the CCD, each of which behaves like the quad cell in the PZT tip-tilt system. An image centroid is calculated for the star, and its offset from the desired position is the error signal.

The CCD-based tip-tilt system is fed by 50-50 beam splitters that pick off light from the VIS input beams when they first reach the VIS table. The CCD has some sensitivity even out to 1000 nm, so it will be able to track on cool, dim stars much more effectively than the PMT tip-tilt.

A down side to the 50-50 gray split is that the VIS system will only get half as much light as it did previously. On the other hand, it opens up the wavelength range short of 600 nm, should the VIS system be able to acquire fringes in that range.

1.3.10 Beam Combiners

CHARA Classic

The first CHARA beam combiner was a 2-beam pupil plane combiner with single pixel detectors (used previously at IRMA & IOTA) in H and K band at the combiner outputs. The NIRO camera was developed as the main CHARA IR detector, and

replaced the old detectors as soon as it was functional. NIRO can operate at several wave bands; and a filter wheel inside the dewar makes switching between them simple.

CHARA Classic has gone through a few modifications since then, but functionally remains the same 2-beam combiner, using IR input beams 5 and 6. A 3-beam combiner is under development for beams 1-3.

CHARA Classic has mostly been used for K' band observations, but work in the H band is possible, especially since some recent modifications in NIRO have improved its sensitivity and noise characteristics.

FLUOR

The Fiber-Linked Unit for Optical Recombination (FLUOR) is a 2-beam fiber-based combiner. A complete description of its design may be found in Coude Du Foresto et al. (1997). Two beams are transported to the FLUOR table where a small amount of optical delay may introduced by means of a dither mirror. The beams are focused onto the tips of polarization preserving single mode optical fibers. The single mode fibers act as a spatial filter, rejecting all components of the wavefront which are not flat.

The fibers each split into two, with one fiber acting as a photometric calibration channel, and the other leading to the point of beam combination. The actual combiner is an 'X'-shaped junction where the fibers are spliced together and then separate. The junction works just like a 50-50 beam splitter.

The interference and photometric fiber tips are imaged onto the NIRO detector, which is moved to the FLUOR table for each of their runs. The photometric channels and spatial filtering make the FLUOR instrument able to measure visibilities very

accurately, though it has a severe magnitude limit.

FLUOR was operated for years at the IOTA interferometer before coming to CHARA in 2002.

Considerable effort has gone into making FLUOR remotely operable (Merand et al., 2004). A dedicated remote observing facility has been assembled at Meudon, as well as one at Georgia State University, and the control software may be run via a virtual private network at either of these locations. The CHARA Array remains a complex instrument, however, and has never been run without a person on the mountain on call, monitoring the observing.

MIRC

The Michigan InfraRed Combiner (MIRC) is fiber-fed, much like FLUOR. However, MIRC is an image plane combiner (Monnier et al., 2004a). The heart of MIRC is a linear array of polarization-preserving single mode fibers arranged in v-grooves etched into a silicon chip. The fiber tips act as apertures in a double (or multiple) slit mask. The light coming out of the fibers is brought to a focus on a detector. Light from pairs of fiber interferes, forming fringes on the image. By selecting a set of fibers in the array with non-redundant spacings, one can insure that the fringes for each fiber pair have a unique spatial frequency on the detector. A power spectrum of the detector image will show discrete peaks for each baseline, and the area under the peak is proportional to the visibility squared.

MIRC is meant to handle 6 beams, though it currently picks the light off of IR beams 1-4. It was tried out in September 2005 with 3 beams, and fringes were indeed found, measuring MIRC's first on-sky closure phase. The next evening fringes were

found with 4 fibers, measuring 4 closure phases and 3 closure amplitudes. This will allow for the complete imaging of simple sources like binary stars in a single night, if not in a single observation. For more complex sources, MIRC could gather enough data in a few nights to maximize u, v plane coverage of the data. The multi-beam capacity of MIRC is a big step toward the full utilization of the CHARA Array's potential.

MIRC has the severe magnitude limit characteristic of fiber combiners, as it rejects the components of the wavefront which would distort it and reduce the visibility anyway. In order to boost the sensitivity, the image on the detector is collapsed with a cylindrical lens in the direction parallel to the fringes, bringing the light along that dimension into one pixel. The image is then spectrally dispersed, resulting in a two-dimensional fringe pattern which plots delay on one axis and $\frac{1}{\lambda}$ on the other. The spectral dispersion is to make the fringes easier to see. If the system bandwidth $\Delta\kappa$ is spread over n spectral rows, then each spectral row has a bandwidth $\frac{\Delta\kappa}{n}$. Recall that the coherence length is the inverse of the bandwidth, so each spectral channel has a coherence length n times longer than that of the full system bandwidth. An IR fringe tracker is currently under development for MIRC, which will phase lock, artificially extending τ_0 and allowing MIRC to greatly increase its sensitivity.

VIS

The Visible to near-Infrared Spectrograph (VIS) system beam combiner is a 6 beam pairwise pupil-plane combiner, described more fully in Chapter 3.

It is currently set up as a 2-beam combiner on beams 5 and 6, which were used for the first fringes on the VIS spectrograph, and a 4-beam pairwise combiner on beams

1-4. The beam 1-4 portion is used to introduce alignment beams into beams 1-4 of the entire CHARA beam train, necessary for the alignment of MIRC. Beams 5 & 6 are used to introduce alignment beams to CHARA Classic and FLUOR. When aligning the system, a corner cube placed at the BRTs reflects the outgoing VIS beams, and some of the returning light leaks through the dichroic, passing the alignment beams into the IR input beams, allowing for co-alignment of the IR systems..

Each north output beam of the VIS combiner is broken up into 7 sub-apertures and injected into multimode fibers. Currently only the beam 5-6 output is equipped with a fiber injector, but the others may be installed whenever they are needed. The injector fibers are patched to a 36-fiber linear array, which acts as the slit of a low-resolution spectrograph. The whole arrangement is designed for the 600-1000 nm wave band. Use of the CCD tip-tilt system will cut the VIS system throughput in half due to the 50-50 gray split for the CCD tip-tilt vs. the dichroic split used for the PMT tip-tilt, but the VIS system's sensitivity will be extended to wavelengths shorter than 600 nm.

See Chapter 3 for a full description of the VIS system, and Chapter 4 for throughput and system visibility estimates.

The best material model for a cat is another, or preferably the same, cat.

— A. Rosenbluth

Chapter 2

CHARA Array Diffraction Effects in the Visible to Near-Infrared

This chapter addresses the effects of diffraction on the wavefront as it propagates through the CHARA beam path. When the telescope pupil is an order of magnitude larger than r_o , and no adaptive optical correction beyond tip-tilt is available, it is desirable to break the aperture up into sub-apertures and process them separately. Because the CHARA beam path is so long, the effect taking sub-apertures of the diffracted beam at the back end is shown.

The optimum aperture diameter for an interferometer with tip-tilt correction is about $3r_o$. Buscher (1988a) found that the signal to noise ratio (SNR) peaks at this aperture size for a group delay fringe tracker in the photon noise limited regime. As aperture size increases past $3r_o$, the SNR settles to $\frac{1}{2}$ the peak value. The CHARA Array's 1-meter primary mirrors have a diameter of about $3r_o$ in K' band ($2.1315 \mu\text{m}$) for fair seeing, $r_o(550 \text{ nm})$ of 6.5 cm.

r_o scales as $\lambda^{\frac{6}{5}}$, so at shorter wavelengths, the wavefront is more perturbed. If the primary diameter is $3r_o$ in K' band, it is $13r_o$ at $\lambda = 600 \text{ nm}$, the peak of the R band filter (Bessell, 1990). If the system is used at visible wavelengths with the full aperture, the SNR is about half that of a $3r_o$ aperture.

The most obvious method to increase the SNR would be to stop down the telescope

apertures to $3r_o$. This option would require either a large iris or set of masks that could be inserted into the beam, possibly in the Coudé train of the telescope. The infrastructure needed to implement such an approach is extensive enough to merit a search for another method.

An alternative method would be to stop down the beam after the beam combining splitter. A small array of lenses could be used to break the combiner output beam into sub-apertures closer to $3r_o$ in size, then each sub-aperture could be processed separately.

The back-end masking approach relies upon the assumption that an aperture at the beam combiner is equivalent to one at the telescope. Diffraction modifies the pupil plane as it propagates through the system, such that the pupil plane at the beam combiner may have very little resemblance to the telescope pupil. A study of diffraction through the system was necessary to determine whether back-end masking could be implemented effectively.

Bagnuolo (1993) modeled the diffraction of atmospherically perturbed wavefronts through the CHARA system at both $2.2\ \mu\text{m}$ and $700\ \text{nm}$ wavelengths. The $700\ \text{nm}$ model stopped down the apertures at the telescopes, and assumed two cases: with and without an Adaptive Optics system to partially correct the wavefront. The current CHARA system has only tip-tilt compensation. At the time, CHARA was still in the design stages, and the internal propagation distances had to be estimated. By the time the array was built, the actual propagation distances turned out to be more than twice the estimate. This was mostly due to a longer propagation distance than planned from the BRT to the beam combiner, whose effect will be described in Section 2.1.2. Because the previous model did not address longer propagation

distances, tip-tilt correction only, and stopping down the aperture at the back end, a new model was needed.

There are many different approaches to modeling diffraction, some very simple and approximate, and others very complicated but exact. Hrynevych (1992) found that for the ratio of beam size to propagation distance typical of most ground-based interferometers, that both the Fraunhofer and Rayleigh-Somerfeld formulations are accurate enough to use for diffraction modeling. This makes life easier for the would-be diffraction modeler, because they are the two simplest diffraction formulations, differing only slightly in form and boundary conditions. They are also the two methods used in this chapter. For a full description of each, refer to a standard optics text such as Born and Wolf (1998).

The diffraction formulations used here assume that the electric field is a scalar. By ignoring the vectoral nature of the electric field, we assume that polarization effects are minimal, or at least consistent between any two telescope lines.

2.1 Fresnel Zones and Diffraction

The CHARA telescopes and BRTs act as beam reducers; they shrink a collimated input beam to a smaller collimated output beam. A screen placed in the output beam shows that it is a smaller copy of the input pupil. Diffraction effects aside, this means that the pupil of the BRT output is simply a re-scaled version of the telescope pupil. It would be convenient if one could stop down the beam at the combiner and assume that it had the same effect as stopping down the primary mirror.

The assumption that the beam combiner pupil is equivalent to the telescope pupil

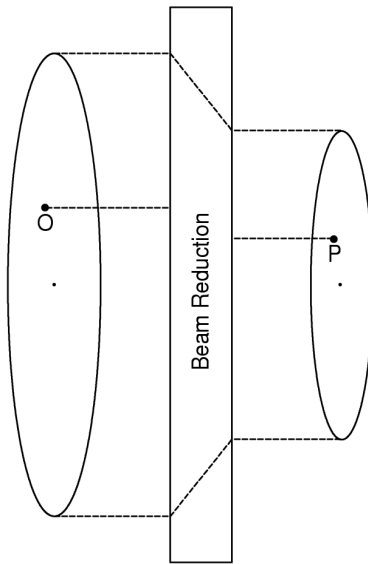


Figure 2.1: Point P in the beam combiner pupil plane corresponds to point O in the telescope pupil plane if diffraction effects can be neglected.

depends upon the following: the field at some point P in the combiner pupil is determined solely by the field at the corresponding point O in the telescope pupil (Figure 2.1).

The effect of diffraction is that the field at P is no longer determined solely by point O , but rather by a region surrounding O . The further the propagation distance between O and P , the larger the area around O becomes which contributes significantly to the field at P . Unfortunately, the propagation distance from the primary mirror to the beam combiner is too long to dismiss the effects of diffraction. We must find out how large the region is around O that contributes to P . The size of that region convolved with the size of the aperture stop will be closer to the effective size of the aperture stop.

We can use the Huygens-Fresnel principle to quantify the field at P . Chris-

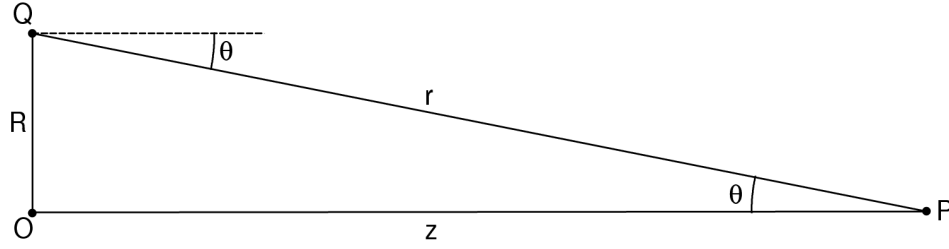


Figure 2.2: The geometry for determining the field at a point behind an aperture. P is the point whose field we wish to determine. \vec{OP} is parallel to the optical axis. dS is an infinitesimal area of the aperture, located at point Q . θ is the angle between the normal of the wave front (shown by the dotted line at Q) and \vec{QP} , which we shall call r .

tian Huygens (1690) put forward the principle that a wave front can be broken down into infinitesimal sections, and each of those sections can be treated as the source of a new secondary spherical wave. The waves from each of these secondary sources add together to determine the ensuing shape of the wave front as a whole. Augustin Fresnel (1816) added Young's idea that the secondary waves must interfere, and this (now called the Huygens-Fresnel Principle) formed the basis for his mathematical treatment of diffraction.

Using this principle, the field at P may be found simply by summing the contribution from each part of the telescope pupil. At this stage we will neglect the effects of beam reduction, and take them into account later. Figure 2.2 shows the geometry we will use, with point P at the beam combiner pupil, and corresponding point O in the telescope pupil. Point Q is the location of the infinitesimal area dS whose contribution to the field at point P we want to quantify. Born and Wolf (1998, Section 8.2) contains a derivation of the field contribution, but it is done for the case

where the incident wave is a spherical one emanating from a point source. For our purposes, we will follow Born and Wolf's derivation, but modified for the case of an incident plane wave.

2.1.1 The Derivation of Fresnel Zones

The contribution to the field at point P by the infinitesimal area dS is given by the equation

$$d\mathbf{U}(P) = K(\theta) \frac{\mathbf{A}e^{ikr}}{r} dS, \quad (2.1)$$

where θ is the angle between the incident wave's normal (the dotted line in Figure 2.2) and r . $K(\theta)$ is an obliquity factor

$$K(\theta) = -\frac{i}{2\lambda}(1 + \cos \theta), \quad (2.2)$$

k is $\frac{2\pi}{\lambda}$, and \mathbf{A} is the complex amplitude at point Q. For an incident plane wave we may assume that \mathbf{A} is a constant over the entire aperture.

The total field U at P is simply the integral of 2.1 over the entire telescope pupil.

$$\mathbf{U}(P) = \mathbf{A} \iint_S \frac{e^{ikr}}{r} K(\theta) dS \quad (2.3)$$

We will find it useful if we divide the aperture into zones (commonly referred to as Fresnel zones) where r differs by a half wavelength from one zone to the next. Fresnel zones form concentric circles centered on O with

$$r = z, \quad z + \frac{\lambda}{2}, \quad z + \frac{2\lambda}{2}, \quad \dots, \quad z + \frac{j\lambda}{2}$$

where $j = 1 \dots \infty$ for an infinite aperture.

The radius R of zone j can be found using

$$r = \sqrt{z^2 + R^2}$$

and if $R \ll z$ then r can be approximated as

$$r \approx z + \frac{R^2}{2z} \quad (2.4)$$

using $f(x_0 + \delta x) \approx f(x_0) + f'(x_0)\delta x$ where z acts as x_0 and R as δx . Setting 2.4 equal to $z + \frac{j\lambda}{2}$ and simplifying gives us

$$R_j = \sqrt{j\lambda z} . \quad (2.5)$$

$z \gg \lambda$ so we can assume that r changes very little from zone j to $j + 1$. If we substitute $\cos \theta = \frac{z}{r}$ into 2.2, we get

$$K_j = -\frac{i}{2\lambda} \left(1 + \frac{z}{z + \frac{j\lambda}{2}} \right) . \quad (2.6)$$

K_j is effectively constant over zone j because r does not change significantly.

We simplify (2.3) by changing the variable of integration to dr . First,

$$dS = R dR d\phi \quad (2.7)$$

where ϕ is the azimuthal angle measured in the aperture plane. From Figure 2.2 one

can see that

$$r^2 = R^2 + z^2 ,$$

so holding z constant and taking the derivative of both sides gives

$$r dr = R dR . \quad (2.8)$$

Substituting (2.8) into (2.7) will give

$$dS = r dr d\phi . \quad (2.9)$$

Integrate (2.3) over just zone j to get the contribution of zone j to the total field at P :

$$\mathbf{U}_j(P) = K_j \mathbf{A} \int_0^{2\pi} d\phi \int_{z+\frac{(j-1)\lambda}{2}}^{z+\frac{j\lambda}{2}} \frac{e^{ikr}}{r} r dr ,$$

which simplifies to

$$\mathbf{U}_j(P) = 2\pi K_j \mathbf{A} \int_{z+\frac{(j-1)\lambda}{2}}^{z+\frac{j\lambda}{2}} e^{ikr} dr . \quad (2.10)$$

Performing the integration yields

$$\begin{aligned} \mathbf{U}_j(P) &= 2\pi K_j \mathbf{A} \left[\frac{1}{ik} e^{ikr} \right]_{z+\frac{(j-1)\lambda}{2}}^{z+\frac{j\lambda}{2}} \\ &= 2\pi K_j \mathbf{A} \frac{1}{ik} \left(e^{ik(z+\frac{j\lambda}{2})} - e^{ik(z+\frac{(j-1)\lambda}{2})} \right) \\ &= -\frac{2\pi i}{k} K_j \mathbf{A} e^{ikz} e^{ij\frac{k\lambda}{2}} \left(1 - e^{-i\frac{k\lambda}{2}} \right) . \end{aligned}$$

Substituting $k\lambda = 2\pi$ into the exponential terms and remembering that $e^{i\pi} = -1$

simplifies them greatly:

$$e^{ij\frac{k\lambda}{2}}(1 - e^{-i\frac{k\lambda}{2}}) = e^{i\pi j}(1 - e^{-i\pi}) = (-1)^j(1 - (-1)) = 2(-1)^j,$$

so that now

$$\mathbf{U}_j(P) = \frac{-2\pi i}{k} 2(-1)^j K_j \mathbf{A} e^{ikz}.$$

A last little tweak is performed by substituting $\lambda = \frac{2\pi}{k}$ and incorporating the $-$ out front into the exponent on the (-1) so that

$$\mathbf{U}_j(P) = 2i \lambda (-1)^{j+1} K_j \mathbf{A} e^{ikz}, \quad (2.11)$$

and

$$\mathbf{U}(P) = 2i \lambda \mathbf{A} e^{ikz} \sum_{j=1}^n (-1)^{j+1} K_j. \quad (2.12)$$

$\mathbf{U}_j(P)$ is positive where j is odd, and $\mathbf{U}_j(P)$ is negative where j is even. Because K_j changes very little in adjacent zones, but the sign flips, $K_j \approx -\frac{1}{2}(K_{j-1} + K_{j+1})$. This will tend to cancel out the contribution of all zones except the first and last.

Born and Wolf show that the sum in (2.12) simplifies to

$$\mathbf{U}(P) = \frac{1}{2}[\mathbf{U}_1(P) + \mathbf{U}_n(P)], \quad (2.13)$$

When the aperture is unbounded, the angle θ to zone n approaches 90° . $\mathbf{U}_n(P)$ goes to zero, leaving

$$\mathbf{U}(P) = \frac{1}{2}\mathbf{U}_1(P). \quad (2.14)$$

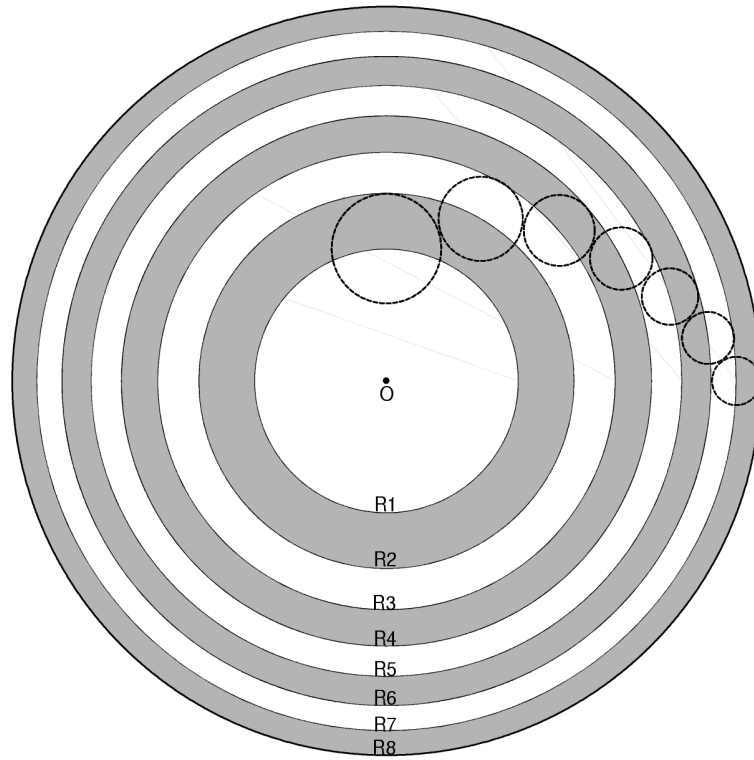


Figure 2.3: A seeing cell (dashed circles) will not contribute to the field at point P if its area is evenly divided between positive and negative Fresnel zones. Fresnel zones whose contribution to $\mathbf{U}(P)$ is negative are shown in gray, while zones with a positive contribution are shown in white. Seeing cells (dashed circles) whose diameters are larger than $2(R_{j+1} - R_j)$ cancel themselves out if their centers are outside of R_j .

This concludes the review of the Born & Wolf derivation of the role of Fresnel zones in diffraction calculations.

The derivation up to this point has assumed a planar wave front where \mathbf{A} is constant over the entire aperture. We are interested in a wave front deformed by turbulence, but we can still establish some reasonable assumptions using Fresnel zones.

Let us consider an approximation to the deformed wave front, broken up into circular cells of diameter r_o . Let each cell have a flat wave front, but with a phase offset between cells. \mathbf{A} , the complex amplitude in the aperture, is constant over each individual seeing cell. Figure 2.3 shows the first 8 Fresnel zones in the aperture plane.

Consider a seeing cell whose center is at radius R_j from point O . If the cell's diameter is twice $R_{j+1} - R_j$, then the cell is approximately spread across two adjacent zones (Figure 2.3 shows appropriately sized seeing cells for radii R_1 through R_7). The field contributions from the two halves of the cell will approximately cancel out. If the cell's diameter is larger than $R_{j+1} - R_j$ then the cell will still be spread across approximately equal areas of adjacent zones, and contribute negligibly to the field at point P . This justifies a rule of thumb: *The only portion of the aperture plane that contributes significantly to the field at P is within R_j of point O when*

$$R_{j+1} - R_j \leq \frac{r_o}{2} \quad (2.15)$$

for the smallest possible value of j .

Because the phase is not actually constant over a given Fresnel zone, it is better to look at this in terms of the phase change as R changes. If Φ is the phase from point Q to point P in Figure 2.2,

$$\Phi(R) = \frac{2\pi}{\lambda} r = \frac{2\pi}{\lambda} \sqrt{R^2 + z^2}. \quad (2.16)$$

Let $u = R^2 + z^2$, then $\frac{\partial u}{\partial R} = 2R$. Taking the partial derivative $\frac{\partial \Phi}{\partial R}$:

$$\begin{aligned} \frac{\partial \Phi}{\partial R} &= \frac{2\pi}{\lambda} \frac{\partial}{\partial R} \sqrt{R^2 + z^2} \\ &= \frac{2\pi}{\lambda} \frac{\partial}{\partial u} u^{\frac{1}{2}} \frac{\partial u}{\partial R} = \frac{2\pi}{\lambda} \frac{1}{2} u^{-\frac{1}{2}} \frac{\partial u}{\partial R} \\ &= \frac{2\pi}{\lambda} \frac{\frac{1}{2} 2R}{\sqrt{R^2 + z^2}}; \end{aligned}$$

so

$$\frac{\partial \Phi}{\partial R} = \frac{2\pi \frac{R}{\lambda}}{\sqrt{R^2 + z^2}}. \quad (2.17)$$

As a rough assumption, let $R \gg r_o$ so that the curvature of the Fresnel zone edge across the seeing cell can be neglected, and $\frac{\partial \Phi}{\partial R}$ is approximately constant over the seeing cell (which is assumed to have a constant phase.) If $\frac{\partial \Phi}{\partial R} \geq 2\pi$ over the width of the seeing cell, then the field across the cell should cancel out at P .

$$\frac{2\pi}{r_o} = \frac{2\pi \frac{R}{\lambda}}{\sqrt{R^2 + z^2}}$$

Isolate the radical and square both sides:

$$R^2 + z^2 = R^2 \left(\frac{r_o}{\lambda} \right)^2$$

Isolate R :

$$R^2 = \frac{-z^2}{1 - \left(\frac{r_o}{\lambda} \right)^2} = \frac{z^2}{\left(\frac{r_o}{\lambda} \right)^2 - 1}$$

So, the Fresnel radius $R_{2\pi}$ where an atmospheric seeing cell of diameter r_o would

contribute nothing to the field at P is

$$R_{2\pi} = \frac{z}{\sqrt{\left(\frac{r_o}{\lambda}\right)^2 - 1}} . \quad (2.18)$$

This only works when $R \gg r_o$, because we assumed that $\frac{\partial\Phi}{\partial R}$ is constant across the seeing cell. The case that is of most interest to us is where R and r_o are the same order of magnitude. In this case, a much more complicated integral of (2.3) must be performed over the seeing cell, which must be numerically modeled.

In the case of large seeing cells that partially enclose the first Fresnel zone, the contribution of everything outside the first zone will tend to cancel out. This agrees with 2.14 which would be the limiting case for a cell with $r_o = \infty$.

Small seeing cells are a real source of trouble. As r_o decreases in size, one must take into account a larger area of the aperture to find the field at P . This larger area will contain more seeing cells, so the interference fringe signal-to-noise will be reduced. We could expect the effects of diffraction to cause an extra decline in visibility as r_o becomes small, because the footprint on the primary that each point at the beam combiner sees gets larger.

However, there is another effect of diffraction that may counter this. For long propagation distances, propagation itself acts as a spatial filter, which will clean up the wave front and improve visibility. Ultimately, a model is required to determine which effect will win out. Section 2.2 will describe the spatial filtering property of propagation further.

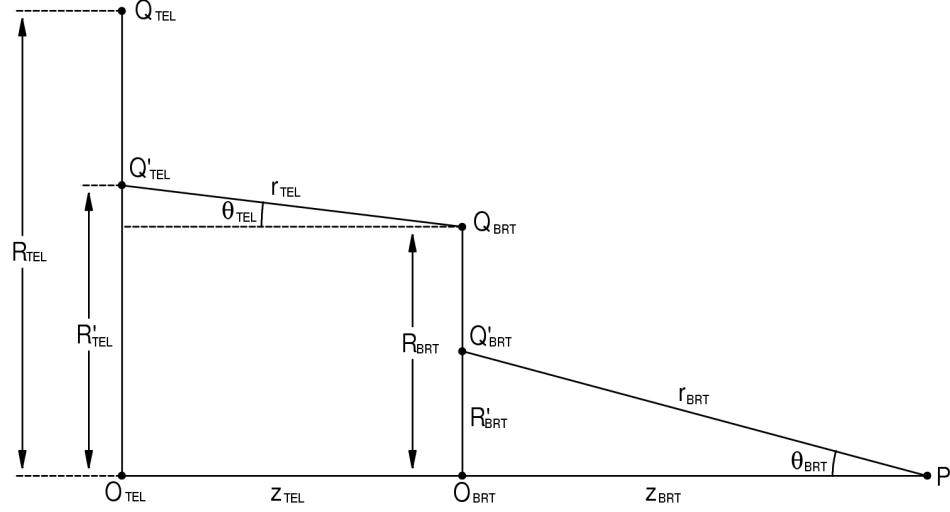


Figure 2.4: The geometry for determining the field at a point at the CHARA beam combiner. The Beam Reducing Telescope at O_{brt} shrinks the beam by a factor of m_{brt} , while it magnifies the beam angle by the same amount. $R_{j\text{ tel}}$, the radius of Fresnel zone j at the telescope input aperture, can still be found by $r_{\text{tel}} + r_{\text{brt}} = z_{\text{tel}} + z_{\text{brt}} + j \frac{\lambda}{2}$.

2.1.2 Taking Beam Reduction Into Account

The derivation in the previous section was based on a simple geometry represented by Figure 2.2. The effects of beam reduction will complicate this somewhat. Figure 2.4 shows the effect of placing beam reducers at O_{tel} and O_{brt} .

If a ray projects backward through the system from point P at angle θ_{brt} from $\overline{O_{\text{tel}}P}$ (which is parallel to the optical axis), it reaches the BRT secondary at point Q'_{brt} , R'_{brt} from $\overline{O_{\text{tel}}P}$. Assume that the internal path lengths of the beam reducing telescopes are very small compared to the distances z_{tel} and z_{brt} , or are accounted for in their calculation. The BRT re-maps the ray to point Q_{brt} on the BRT primary, at angle θ_{tel} and R_{brt} from $\overline{O_{\text{tel}}P}$. The ray reaches the telescope secondary at point

$Q'_{\text{tel}}, R'_{\text{tel}}$ from $\overline{O_{\text{tel}}P}$. The telescope re-maps the ray to point Q_{tel} on the telescope primary, R'_{tel} from $\overline{O_{\text{tel}}P}$.

Equation 2.5 was built by breaking the telescope aperture up into zones where $r = z + j \frac{\lambda}{2}$. The underlying structure of the Fresnel zones is that the distance from P to the aperture is $j \frac{\lambda}{2}$ greater than z at R_j . By breaking these distances into two segments, one can replace r by $r_{\text{tel}} + r_{\text{brt}}$ and z by $z_{\text{tel}} + z_{\text{brt}}$:

$$r_{\text{tel}} + r_{\text{brt}} = z_{\text{tel}} + z_{\text{brt}} + j \frac{\lambda}{2} . \quad (2.19)$$

Use the Pythagorean relation for

$$r_{\text{brt}} = \sqrt{z_{\text{brt}}^2 + R_{\text{brt}}'^2} , \quad r_{\text{tel}} = \sqrt{z_{\text{tel}}^2 + (R'_{\text{tel}} - R_{\text{brt}})^2} . \quad (2.20)$$

Because $z_{\text{brt}} \gg R_{\text{brt}}'$ and $z_{\text{tel}} \gg R'_{\text{tel}} - R_{\text{brt}}$, (2.20) can be approximated via Taylor series as

$$r_{\text{brt}} = z_{\text{brt}} + \frac{R_{\text{brt}}'^2}{2z_{\text{brt}}} , \quad r_{\text{tel}} = z_{\text{tel}} + \frac{(R'_{\text{tel}} - R_{\text{brt}})^2}{2z_{\text{tel}}} . \quad (2.21)$$

Substituting (2.21) into (2.19) gives

$$z_{\text{brt}} + \frac{R_{\text{brt}}'^2}{2z_{\text{brt}}} + z_{\text{tel}} + \frac{(R'_{\text{tel}} - R_{\text{brt}})^2}{2z_{\text{tel}}} = z_{\text{tel}} + z_{\text{brt}} + j \frac{\lambda}{2} ,$$

which simplifies to

$$\frac{R_{\text{brt}}'^2}{z_{\text{brt}}} + \frac{(R'_{\text{tel}} - R_{\text{brt}})^2}{z_{\text{tel}}} = j \lambda . \quad (2.22)$$

(2.22) may be further simplified by establishing the relationship between the various R s. The BRT will provide the needed connection. The beam reduction is de-

scribed by

$$R_{\text{brt}} = m_{\text{brt}} R'_{\text{brt}} \quad (2.23)$$

and

$$\theta_{\text{tel}} = \frac{\theta_{\text{brt}}}{m_{\text{brt}}} , \quad (2.24)$$

where m_{brt} is the factor by which the beam size is reduced.

Because θ_{brt} and θ_{tel} are both very small angles,

$$\theta_{\text{brt}} \approx \tan \theta_{\text{brt}} = \frac{R'_{\text{brt}}}{z_{\text{brt}}}$$

and

$$\theta_{\text{tel}} \approx \tan \theta_{\text{tel}} = \frac{R'_{\text{tel}} - R_{\text{brt}}}{z_{\text{tel}}} .$$

Substituting these into (2.24) and using (2.23) to eliminate R_{brt} gives

$$\frac{R'_{\text{tel}} - m_{\text{brt}} R'_{\text{brt}}}{z_{\text{tel}}} = \frac{R'_{\text{brt}}}{m_{\text{brt}} z_{\text{brt}}} .$$

Multiply both sides by z_{tel} and gather the R'_{brt} terms to one side to get

$$R'_{\text{tel}} = R'_{\text{brt}} \left(\frac{z_{\text{tel}}}{m_{\text{brt}} z_{\text{brt}}} + m_{\text{brt}} \right) = R'_{\text{brt}} \frac{1}{m_{\text{brt}}} \left(\frac{z_{\text{tel}}}{z_{\text{brt}}} + m_{\text{brt}}^2 \right) .$$

Define

$$\alpha = \frac{m_{\text{brt}}}{\frac{z_{\text{tel}}}{z_{\text{brt}}} + m_{\text{brt}}^2} , \quad (2.25)$$

then

$$R'_{\text{brt}} = \alpha R'_{\text{tel}} . \quad (2.26)$$

Substituting (2.26) and (2.23) into (2.22) yields

$$\frac{(\alpha R'_{\text{tel}})^2}{z_{\text{brt}}} + \frac{(R'_{\text{tel}} - m_{\text{brt}} \alpha R'_{\text{tel}})^2}{z_{\text{tel}}} = j \lambda ,$$

which simplifies to

$$R'^2_{\text{tel}} = j \lambda \left[\frac{\alpha^2}{z_{\text{brt}}} + \frac{(1 - m_{\text{brt}} \alpha)^2}{z_{\text{tel}}} \right]^{-1} .$$

Define

$$\beta = \left[\frac{\alpha^2}{z_{\text{brt}}} + \frac{(1 - m_{\text{brt}} \alpha)^2}{z_{\text{tel}}} \right]^{-1} , \quad (2.27)$$

and take the square root of both sides to get

$$R'_{\text{tel}} = \sqrt{j \lambda \beta} . \quad (2.28)$$

Equation 2.28 has the same form as equation 2.5, which shows that β is the effective distance from O_{tel} to P if the beam were never reduced at the BRT.

One last step remains. R'_{tel} is the size of R_j on the telescope secondary. The telescope is a beam reducer, so

$$R_{\text{tel}} = m_{\text{tel}} R'_{\text{tel}} . \quad (2.29)$$

Substituting (2.29) into (2.28) and isolating R_{tel} gives

$$R_{\text{tel}} = \sqrt{j \lambda m_{\text{tel}}^2 \beta} .$$

Define

$$\gamma = m_{\text{tel}}^2 \beta , \quad (2.30)$$

then

$$R_{\text{tel}} = \sqrt{j \lambda \gamma} . \quad (2.31)$$

As with equation 2.28, γ is the effective distance from O_{tel} to P with no beam reduction whatsoever. That is, if instead of a telescope there was only an aperture stop at the telescope's position, γ is the distance that the beam would propagate at full size to experience the same diffraction effects that are observed at the beam combiner with both beam reducers.

In general, (2.30) can be applied at the BRT as well. Multiply the propagation distance from the BRT to the beam combiner, z_{brt} , by m_{brt}^2 , to obtain the effective propagation distance $z_{\text{brt eff}}$ as if the beam at the BRT had never been reduced. That is to say:

$$\beta = z_{\text{tel}} + m_{\text{brt}}^2 z_{\text{brt}} , \quad (2.32)$$

and direct calculations of α are unnecessary. As a sanity check, values of β were calculated using both 2.27 and 2.32, yielding identical results.

The fact that the effective distance after a beam reducer is $m^2 z$ is commonly known. A derivation justifying its use is appropriate, especially to be sure that it is still usable with nested beam reducers.

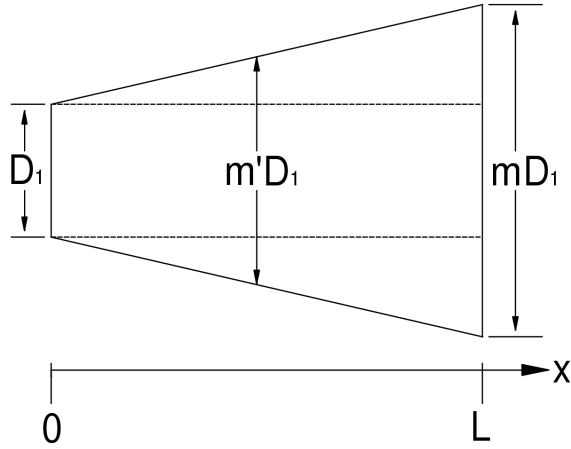


Figure 2.5: A simple beam expander with input beam diameter D_1 , and output beam diameter $m D_1$ at $x = L$. The beam diameter at x is $m' D_1$.

Beam Reducer Effective Internal Path

Another refinement of the effective propagation distances is possible. Up to this point, we have neglected the internal paths of the beam reducers themselves: the distance from the telescope or BRT primary to the secondary. It may seem troublesome, because the beams are converging along this length, and really are not at either the input or output scale.

Consider the case of a beam expander. The beam diameter at $x = 0$ is D_1 , and at $x = L$ it is $m D_1$. In the region $x = 0..L$, the beam diameter grows linearly (see Figure 2.5.) The slope of this line is

$$\frac{y_L - y_0}{x_L - x_0} = \frac{m D_1 - D_1}{L} = \frac{(m - 1) D_1}{L}.$$

The diameter of the beam can then be described by the equation

$$y = \frac{(m - 1) D_1}{L} x + D_1 . \quad (2.33)$$

At any point x , define the magnification m' at that point such that

$$y = m' D_1 .$$

Substituting this for y in equation 2.33 and dividing by D_1 yields the equation for m' as a function of x :

$$m' = \frac{m - 1}{L} x + 1 . \quad (2.34)$$

Now consider equation 2.30 once again. It is used to describe the effective distance after a beam *reducer*, but the value of m we have been using is > 1 . To be generally correct, change the form to

$$\gamma = \frac{x}{m^2} \quad (2.35)$$

where $m > 1$ for a beam expander and $m < 1$ for a beam reducer. This will still produce the same results as before. For example, if the BRTs reduce the beam by 6.6, the magnification would be $\frac{1}{6.6}$, giving an effective propagation distance after the BRT of $6.6^2 x$.

Using this definition of γ , consider the effective propagation distance of an infinitesimal length dx at position x along the beam expander's internal path:

$$d\gamma = \frac{dx}{m'^2}$$

Substituting in equation 2.34 for m' and integrating from 0 to L gives

$$\gamma_{\text{int}} = \int d\gamma = \int_0^L \left(\frac{(m-1)x}{L} + 1 \right)^{-2} dx .$$

A change of variables will simplify the integral. Let $u = \frac{m-1}{L}x + 1$, then $du = \frac{m-1}{L} dx$, $u(0) = 1$, and $u(L) = m$.

$$\begin{aligned} \gamma_{\text{int}} &= \frac{L}{m-1} \int_0^L \left(\frac{m-1}{L}x + 1 \right)^{-2} \frac{m-1}{L} dx \\ &= \frac{L}{m-1} \int_1^m u^{-2} du = \frac{L}{m-1} \left[-u^{-1} \right]_1^m \\ &= \frac{L}{m-1} \left[-\frac{1}{m} + 1 \right] = \frac{L}{m-1} \frac{m-1}{m} = \frac{L}{m} , \end{aligned}$$

so

$$\gamma_{\text{int}} = \frac{L}{m} . \tag{2.36}$$

Care must be taken to make sure the correct value of m is used, but (2.36) works for both beam expansion and reduction, and can give equivalent propagation distances on both the input and output beam scales.

The telescopes reduce a 1 meter beam to 12.5 cm, which makes $m_{\text{tel}} = 8$. The BRTs reduce the beam further to 18.9 mm ($\frac{3}{4}$ inch), so $m_{\text{brt}} = 6.6$. In order to simplify calculations, it is best to find γ_{int} for both the telescopes and the BRTs at the 12.5 cm beam scale. To find the γ_{int} for a particular scale, view the problem as if that scale were the input end of the beam expander/reducer, and choose m accordingly.

The 12.5 cm beam sees the telescope as a beam expander. The distance from the telescope primary to the secondary is 2.188 m. With $L = 2.188$ m and $m = 8$,

Table 2.1: CHARA array values for z_{tel} and z_{brt}

	S1	S2	E1	E2	W1	W2
With combiner on VIS table:						
z_{brt}	13.67	14.25	17.90	17.32	15.49	16.08
With combiner on NII table:						
z_{brt}	7.94	8.52	12.18	11.60	9.77	10.36
Distance is for beam 5.						
$z_{\text{tel min}}$	234.0	201.4	245.3	183.5	187.9	79.8

$z_{\text{tel min}}$ settings: POP: 1, OPLE Delay: 0.

POPs 2-4 add 36.58 m each, End POP adds 33.32 m.

OPLEs add up to 90 m.

$$\gamma_{\text{int tel}} = 0.2735 \text{ m.}$$

In contrast, the 12.5 cm beam sees the BRT as a beam reducer. In this case use $\frac{1}{6.6}$ for m . The distance from the BRT primary to the secondary is 1.080 m. These give a value of $\gamma_{\text{int brt}} = 7.125 \text{ m.}$

When scaling the beam down, expect the effective internal path of the BRT to be longer than L , because diffraction effects are magnified as the beam shrinks. The opposite is true for a beam expander. The diffraction effects will be lessened as the beam grows, so the effective internal path will be smaller than L .

CHARA Array Values of z , β , γ , and R_j

Let us plug in some numbers and see how big the Fresnel zones are on the primary mirror.

z_{tel} , the distance from the telescope to the BRT, varies because the beam passes through the POPs and the OPLE delay lines. Taking into account that some telescopes are closer to the delay line building than others, z_{tel} can have values from 80 to

Table 2.2: CHARA Array values for α , β , and γ

α_{long}	0.0939
α_{typical}	0.122
α_{short}	0.134
Effective total propagation distance at 12.5 cm beam scale	
β_{long}	1.26 km
β_{typical}	947 m
β_{short}	778 m
Effective total propagation distance at 1 m beam scale	
γ_{long}	80.5 km
γ_{typical}	60.6 km
γ_{short}	49.8 km

Table 2.3: CHARA Array typical values for r_{\circ} , $R_{1 \text{ tel}}$, $R_{2 \text{ tel}}$, $2\Delta R_{2 \text{ tel}}$

λ (nm)	Good r_{\circ} (cm)	Excellent r_{\circ} (cm)	$R_{1 \text{ tel}}$ (cm)	$R_{2 \text{ tel}}$ (cm)	$2\Delta R_{2 \text{ tel}}$ (cm)
550	10.0	15.0	18.3	25.8	15.1
600	10.9	16.7	19.1	27.0	15.8
750	14.5	21.8	21.3	30.1	17.7
900	16.4	27.1	23.4	33.0	19.3
1000	18.2	30.7	24.6	34.8	20.4
2200	40.0	60.0	36.5	51.6	30.2

478 m (see Table 2.1). The effective internal paths $\gamma_{\text{int tel}}$ and $\gamma_{\text{int brt}}$ at the 12.5 cm beam scale have been included in z_{tel} .

z_{brt} , the distance from the BRT to the beam combiner, varies from 13.7 to 17.9 m, because the BRT tables vary in distance from visible band beam combiner.

Using these values in (2.25), (2.27), and (2.30) yields a range of values given in Table 2.2. Again, α is dimensionless, β is the effective total propagation distance at the 12.5 cm beam scale, and γ is the effective total propagation distance at the 1 m beam scale.

Table 2.3 shows the radii of the 1st ($R_{1\text{ tel}}$) and 2nd ($R_{2\text{ tel}}$) Fresnel zones projected onto the telescope primary from a point at the beam combiner. γ_{typical} from Table 2.2 was used as the effective total propagation distance in Equation 2.31.

$2\Delta R_{2\text{ tel}}$ is twice the width of the 2nd Fresnel zone. According to Equation 2.15, this is the minimum size r_o for which only the 1st Fresnel zone contributes to the field at P . If r_o is smaller than $2\Delta R_{2\text{ tel}}$, then more Fresnel zones must be taken into account.

Table 2.3 shows that, depending upon wavelength, an r_o of up to 15 cm at λ of 550 nm is required in order to limit the diffraction contribution at P to the 1st Fresnel zone. The effective size of a sub-aperture of radius r at the beam combiner projected out to the primary would be a circle of radius $m_{\text{tel}} m_{\text{brt}} r$ convolved with a circle of radius R_j from (2.15.)

This approach is rather pessimistic, and predicts that the fringe visibility will rapidly deteriorate as r_o shrinks, because R_j will increase and the effective size of the sub-aperture will increase. With a larger effective sub-aperture, yet smaller r_o , the number of seeing cells across the sub-aperture increases even faster than if we neglected the effects of diffraction.

We will see that such a pessimistic model is incorrect in Section 2.2, but this approach has been retained because of the derivations of the effective propagation distances β and γ to which it led. The description of Fresnel zones contained in this section is also useful, as the next section will show that they have an analog in frequency space.

2.2 An Angular Spectrum Diffraction Model

In order to create a quantitative description of wave front behavior through the system, it is necessary to build a model of the effects of atmospheric turbulence and diffraction. The most most straightforward method may seem to be by calculating the integral in Equation 2.3. However, the phase of dU can change rapidly from one aperture point to the next in a numerical model, which makes adequate sampling of the aperture a problem.

For example, let us impose the constraint that the phase cannot change by more than π radians from one sampling point to the next across the aperture. This translates to a maximum path length difference of $\frac{\lambda}{2}$ from one point to the next. This should sound familiar from Section 2.1, because it is the same condition used to determine the radius of the 1st Fresnel Zone. Using $j = 1$ in equation 2.5 gives the constraint

$$\Delta y = \sqrt{\lambda \gamma}$$

where Δy is the distance between sampling points at the aperture.

This is an optimistic assumption, because the phase will change more rapidly as we move away from the optical axis, just as higher order Fresnel zone radii pack more closely together. One can refine it by finding the Fresnel order j_{apt} of the edge of the aperture R_{apt} , and requiring that Δy be smaller than $R_{j_{\text{apt}}} - R_{j_{\text{apt}}-1}$. This guarantees that the phase difference Δy at the edge of the aperture is less than π .

Plug R_{apt} into the definition of the Fresnel zone radius (equation 2.5), and solve for j to obtain

$$j_{\text{apt}} = \frac{R_{\text{apt}}^2}{\lambda z} . \quad (2.37)$$

Again using 2.5 for the radius of Fresnel zone $j_{\text{apt}} - 1$,

$$\Delta y = R_{\text{apt}} - \sqrt{(j_{\text{apt}} - 1) \lambda z}$$

then plug in 2.37 and simplify:

$$\Delta y = R_{\text{apt}} - \sqrt{R_{\text{apt}}^2 - \lambda z} . \quad (2.38)$$

For long propagation distances, this sampling requirement may be quite modest. The shortest effective z_{tel} in the CHARA array is $m_{\text{tel}}^2 \times 79.8$ m, or 5107 m. Using this as z and 0.5 m as R_{apt} in 2.38 yields a Δy of 3 mm. A grid of 333 x 333 points would be adequate to sample the aperture for numerically evaluating equation 2.3. As z increases, the minimum Δy will become larger and a more sparse grid of aperture sampling points possible.

In general, propagation distances tend to be much shorter, and the sampling requirement much too high to be computationally efficient. There is another approach to modeling diffraction whose sampling requirements are generally less stringent and are independent of the propagation distance.

2.2.1 The Angular Spectrum Approach to Diffraction

In his book *Introduction to Fourier Optics* (Goodman, 1968, Section 3.7), Goodman explains that the field $\mathbf{U}(x, y, 0)$ across an aperture may be equivalently described by

its Fourier transform

$$\mathbf{A}_o(f_x, f_y) = \iint_{-\infty}^{\infty} \mathbf{U}(x, y, 0) e^{-i 2\pi (f_x x + f_y y)} dx dy , \quad (2.39)$$

and the inverse relationship

$$\mathbf{U}(x, y, 0) = \iint_{-\infty}^{\infty} \mathbf{A}_o(f_x, f_y) e^{i 2\pi (f_x x + f_y y)} df_x df_y . \quad (2.40)$$

The Angular Spectrum approach was first put forth by Ratcliffe (1956). Sherman (1967) shows that this approach is equivalent to the first solution of the Rayleigh-Sommerfeld formulation of diffraction.

In order to understand the physical significance of (2.40), consider a plane wave with unit amplitude, propagating such that its direction of travel is at an angle θ from the x axis toward the positive y axis, and angle ϕ from the z axis (see Figure 2.6).

The equation for the plane wave is given by

$$\mathbf{B}(x, y, z) = e^{i \bar{\mathbf{k}} \cdot \mathbf{r}}$$

where $\bar{\mathbf{k}} = \frac{2\pi}{\lambda} \hat{\mathbf{k}}$, and $\hat{\mathbf{k}}$ is a unit vector in the direction of the plane wave's propagation.

Breaking $\hat{\mathbf{k}}$ down into its components parallel to the x , y , and z axes:

$$\hat{\mathbf{k}} = \cos \theta \sin \phi \hat{\mathbf{x}} + \sin \theta \sin \phi \hat{\mathbf{y}} + \cos \phi \hat{\mathbf{z}}$$

Let $\alpha = \cos \theta \sin \phi$, $\beta = \sin \theta \sin \phi$, and $\gamma = \cos \phi$. These are often referred to as the plane wave's *direction cosines*, because they are the projection of $\hat{\mathbf{k}}$ onto each axis.

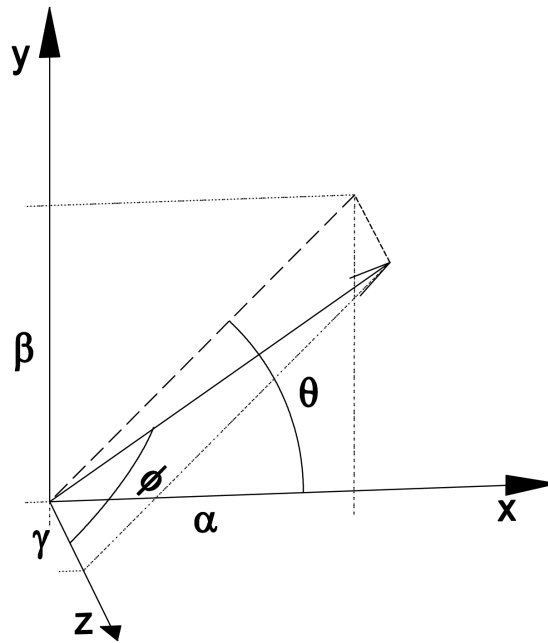


Figure 2.6: Unit vector $\hat{\mathbf{k}}$ with angle ϕ from the z axis and angle θ from the x axis in the xy plane. $\alpha = \cos \theta$, $\beta = \sin \theta$, and $\gamma = \cos \phi$.

Now

$$\hat{\mathbf{k}} = \alpha \hat{\mathbf{x}} + \beta \hat{\mathbf{y}} + \gamma \hat{\mathbf{z}} .$$

Similarly,

$$\bar{\mathbf{r}} = x \hat{\mathbf{x}} + y \hat{\mathbf{y}} + z \hat{\mathbf{z}} .$$

Thus, the equation for the plane wave becomes

$$\mathbf{B}(x, y, z) = e^{i \frac{2\pi}{\lambda} (\alpha x + \beta y + \gamma z)} . \quad (2.41)$$

Because $\hat{\mathbf{k}}$ is a unit vector, then

$$\gamma = \sqrt{1 - \alpha^2 - \beta^2} . \quad (2.42)$$

Let

$$\alpha = \lambda f_x , \quad \beta = \lambda f_y , \quad \gamma = \sqrt{1 - (\lambda f_x)^2 - (\lambda f_y)^2} \quad (2.43)$$

and substitute these in to (2.41) at $z = 0$. The resulting equation for the plane wave,

$$\mathbf{B}(f_x, f_y; 0) = e^{i 2\pi (f_x x + f_y y)} , \quad (2.44)$$

is identical to the exponential part of (2.40).

This means that $\mathbf{U}(x, y, 0)$ may be expressed as a superposition of plane waves with complex amplitude $\mathbf{A}_o(f_x, f_y)$, propagating at angles

$$\cos \theta \sin \phi = \alpha = \lambda f_x$$

$$\sin \theta \sin \phi = \beta = \lambda f_y$$

with the x axis, and angle

$$\cos \phi = \gamma = \sqrt{1 - (\lambda f_x)^2 - (\lambda f_y)^2}$$

with the z axis. $\mathbf{A}_o(f_x, f_y)$ is called the *angular spectrum* of $\mathbf{U}(x, y, 0)$ because it gives the complex amplitude of the distribution of plane waves across the spectrum of incoming angles.

2.2.2 The Propagation Transfer Function

The utility of using the angular spectrum to describe a wave front is that in frequency space, propagation of the wave front is described by a relatively simple transfer function. The transfer function $\mathbf{T}(f_x, f_y; z)$ is defined such that at distance z ,

$$\mathbf{A}_z(f_x, f_y) = \mathbf{T}(f_x, f_y; z) \mathbf{A}_o(f_x, f_y) .$$

Goodman finds the propagation transfer function by solving the Helmholtz equation

$$\nabla^2 \mathbf{U} + k^2 \mathbf{U} = 0$$

with $\mathbf{U}(x, y, z)$ from (2.40). However, it may also be reasoned out geometrically.

Figure 2.7 represents a plane wave starting at $z = 0$ and traveling at angle ϕ from the z axis. We want to quantify how this wave changes by the time it reaches a point on the z axis, at a distance z from the origin. The wave need only travel a distance of $z \cos \phi$ before some portion of the wave front reaches the observation point at z . The wave's amplitude will remain unchanged, but its phase will advance by 2π times the number of wavelengths it traveled, $\frac{z \cos \phi}{\lambda}$.

The plane wave's complex amplitude at $z = 0$ is $\mathbf{A}_o(f_x, f_y)$, so its complex amplitude at z is

$$\mathbf{A}(f_x, f_y; z) = \mathbf{A}_o(f_x, f_y) e^{i 2\pi \frac{z}{\lambda} \cos \phi} .$$

From this it is apparent that the transfer function is simply

$$\mathbf{T}(f_x, f_y; z) = e^{i \frac{2\pi z}{\lambda} \cos \phi}$$

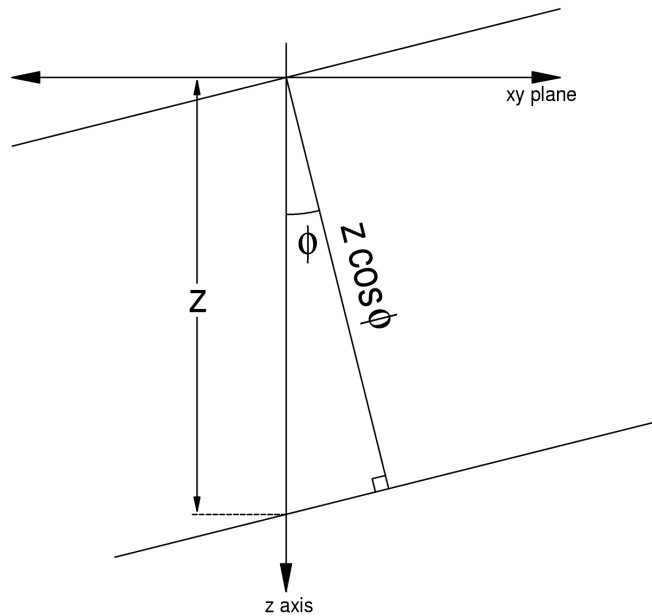


Figure 2.7: A plane wave traveling at an angle ϕ from the z axis only travels a distance of $z \cos \phi$ before the wave front reaches distance z on the z axis.

$$= e^{i 2\pi z \sqrt{\left(\frac{1}{\lambda}\right)^2 - (f_x^2 + f_y^2)}} . \quad (2.45)$$

The latter form is obtained by using (2.43) and moving λ inside the radical. Because $f_x^2 + f_y^2$ will show up quite often, it is simpler to use

$$\rho^2 = f_x^2 + f_y^2 . \quad (2.46)$$

ρ is the distance from the origin on the f_x, f_y plane.

Propagation as a Spatial Filter

The phase shift in (2.45) is proportional to γ . Plane waves traveling directly along the z axis will experience the greatest phase shift. If a wave is traveling parallel to

the x, y plane, then $\gamma = 0$ and the wave contributes no power to the integral in (2.40).

This is the case when $\rho = \frac{1}{\lambda}$. Therefore ρ will be limited to $\rho < \frac{1}{\lambda}$.

When $\rho > \frac{1}{\lambda}$, then γ becomes negative, and the transfer function becomes a decaying exponential. This situation describes evanescent waves, which fall off rapidly in intensity as z increases. As long as z is greater than a few wavelengths, evanescent waves may be ignored. This concludes the review of Goodman's angular spectrum presentation.

The maximum phase shift for $\gamma = 1$ (and hence, $\rho = 0$) is $2\pi \frac{z}{\lambda}$. As ρ increases, γ decreases and the phase shift lessens.

The f_x, f_y plane may be divided into Fresnel-type zones. If the phase shift Φ is $2\pi \frac{z}{\lambda} \gamma$, then the difference in phase from the center of the f_x, f_y plane to a radius ρ is

$$\Phi(0) - \Phi(\rho) = 2\pi \frac{z}{\lambda} - 2\pi z \sqrt{\left(\frac{1}{\lambda}\right)^2 - \rho^2}.$$

If this phase difference is set to equal $j\pi$, then one may solve for ρ_j as a frequency plane analog to the Fresnel zone radius R_j .

$$j\pi = 2\pi \frac{z}{\lambda} - 2\pi z \sqrt{\left(\frac{1}{\lambda}\right)^2 - \rho^2}$$

Isolate the radical and simplify to obtain

$$\sqrt{\left(\frac{1}{\lambda}\right)^2 - \rho^2} = \frac{1}{\lambda} - \frac{j}{2z},$$

then square both sides and isolate ρ for

$$\rho^2 = \left(\frac{1}{\lambda}\right)^2 - \left(\frac{1}{\lambda} - \frac{j}{2z}\right)^2.$$

The right side may be expanded and then simplified:

$$\begin{aligned} \left(\frac{1}{\lambda}\right)^2 - \left(\frac{1}{\lambda} - \frac{j}{2z}\right)^2 &= \left(\frac{1}{\lambda}\right)^2 - \left(\left(\frac{1}{\lambda}\right)^2 - \frac{2j}{2\lambda z} + \left(\frac{j}{2z}\right)^2\right) \\ &= \frac{j}{\lambda z} - \left(\frac{j}{2z}\right)^2 \end{aligned}$$

Square both sides and factor out $\frac{j}{z}$ to obtain

$$\rho_j = \sqrt{\frac{j}{z} \left(\frac{1}{\lambda} - \frac{j}{2z}\right)}. \quad (2.47)$$

For most optical applications, especially at CHARA, $z \gg \lambda$. In that case, the second term in the radical is much smaller than the first, and

$$\rho_j = \sqrt{\frac{j}{z\lambda}}. \quad (2.48)$$

This is very similar in form to (2.5), $R_j = \sqrt{j\lambda z}$. In fact,

$$\rho_j = \frac{j}{R_j} \quad (2.49)$$

which is a handy relation for those who are much more familiar with thinking about the x, y aperture plane than the f_x, f_y frequency plane.

By following the same steps as were taken to derive $\frac{\partial \Phi}{\partial R}$, using $\Phi = 2\pi \frac{z}{\lambda} \gamma$:

$$\frac{\partial \Phi}{\partial \rho} = 2\pi z \frac{\partial}{\partial \rho} \sqrt{\left(\frac{1}{\lambda}\right)^2 - \rho^2}.$$

Let $u = \frac{1}{\lambda}^2 - \rho^2$, and $\frac{\partial u}{\partial \lambda} = -2\rho$.

$$\frac{\partial \Phi}{\partial \rho} = 2\pi z \frac{\partial}{\partial u} u^{\frac{1}{2}} \frac{\partial u}{\partial \rho} = 2\pi z \frac{1}{2} u^{-\frac{1}{2}} \frac{\partial u}{\partial \rho} = 2\pi z \frac{1}{2} \frac{-2\rho}{\sqrt{\left(\frac{1}{\lambda}\right)^2 - \rho^2}},$$

so

$$\frac{\partial \Phi}{\partial \rho} = -\frac{2\pi z \rho}{\sqrt{\left(\frac{1}{\lambda}\right)^2 - \rho^2}}. \quad (2.50)$$

Consider a feature on the f_x, f_y plane with diameter d_f with equal phase and intensity across its diameter. Figure 2.3 readily depicts this situation, with the substitution of d_f for r_o and ρ_j for R_j . If $\rho \gg d_f$, it will have a net zero contribution to $\mathbf{U}(x, y, z)$ when Φ changes by 2π across d_f :

$$\frac{\partial \Phi}{\partial \rho} = -\frac{2\pi}{d_f} = -\frac{2\pi z \rho}{\sqrt{\left(\frac{1}{\lambda}\right)^2 - \rho^2}}$$

Isolate the radical and square both sides to obtain

$$\left(\frac{1}{\lambda}\right)^2 - \rho^2 = (d_f z \rho)^2,$$

then isolate ρ for

$$\rho^2 = \frac{\left(\frac{1}{\lambda}\right)^2}{(d_f z)^2 + 1}.$$

Take the square root of both sides:

$$\rho_{2\pi} = \frac{\frac{1}{\lambda}}{\sqrt{(d_f z)^2 + 1}}. \quad (2.51)$$

Note that the equations for ρ_j (2.48), $\frac{\partial\Phi}{\partial\rho}$ (2.50), and $\rho_{2\pi}$ (2.51) are those for R_j (2.5), $\frac{\partial\Phi}{\partial R}$ (2.17), and $R_{2\pi}$ (2.18) with $\frac{1}{\lambda}$ exchanged for z , and ρ substituted for R . $\frac{\partial\Phi}{\partial\rho}$ and $\rho_{2\pi}$ have sign changes as well, stemming from the fact that $r = \sqrt{R^2 + z^2}$ while $\gamma = \sqrt{1 - (\lambda\rho)^2}$.

Just as with $R_{2\pi}$ (2.18), $\rho_{2\pi}$ (2.51) only works when the feature diameter is much smaller than the Fresnel radius. We are more interested in the case where ρ and d_f are the same order of magnitude, but this calls for a numerical evaluation of (2.40) over the region of interest in the spatial frequency plane. One might as well just do the same calculations over the entire frequency plane and model what exactly to expect at the back end of the system.

A rough understanding of propagation as a spatial filter may be gleaned by using (2.51). Phase fluctuations at the aperture plane cause the formation of speckles in the spatial frequency plane. Each speckle may be thought of as a nearly diffraction limited image, an Airy disk convolved with a delta function at the spatial frequency point corresponding to the incoming angle.

As z increases, $\rho_{2\pi}$ shrinks. Once a speckle is outside of $\rho_{2\pi}$, the phase difference across it will nearly cancel out its contribution to $\mathbf{U}(x, y, z)$. Because the speckles furthest from the f_x, f_y origin are due to the most tilted portions of the wave front, a shrinking $\rho_{2\pi}$ will filter them out first. The resulting wave front is smoother, because the maximum tilt allowed at any point has been reduced.

2.2.3 Execution of the Model

Let us define arbitrary complex functions $\mathbf{f}(x, y), \mathbf{g}(x, y), \mathbf{h}(x, y)$ and their transforms $\mathbf{F}(f_x, f_y), \mathbf{G}(f_x, f_y), \mathbf{H}(f_x, f_y)$. It is a fundamental property of Fourier transforms that if

$$\mathbf{h} = \mathbf{f} \mathbf{g}, \quad \mathbf{H} = \mathbf{F} * \mathbf{G}$$

and if

$$\mathbf{h} = \mathbf{f} * \mathbf{g}, \quad \mathbf{H} = \mathbf{F} \mathbf{G}.$$

* represents the convolution operator:

$$\mathbf{f} * \mathbf{g} = \iint_{-\infty}^{\infty} \mathbf{f}(\xi, \eta) \mathbf{g}(x - \xi, y - \eta) d\xi d\eta$$

(Goodman, 1968, or any text on Fourier transforms).

Modeling of diffraction may be greatly simplified by switching back and forth between $\mathbf{U}(x, y, z)$ and its angular spectrum $\mathbf{A}(f_x, f_y; z)$ such that convolution operations are avoided and replaced by the multiplication.

For instance, as a wave front reaches the input aperture of a telescope, the simplest approach is to apply the telescope's aperture function in x, y space. It's a multiplication of the $\mathbf{U}(x, y, z)$ by the transmission of the aperture at each point $\mathbf{ap}(x, y)$. The more difficult approach would be to convolve $\mathbf{A}(f_x, f_y; z)$ with the Fourier transform of the aperture function, $\mathbf{AP}(f_x, f_y)$.

On the other hand, propagation is simpler in f_x, f_y space. $\mathbf{A}(f_x, f_y; z) \mathbf{T}(f_x, f_y; \Delta z)$ is much simpler than convolving $\mathbf{U}(x, y, z)$ with the Fourier transform of the propagation transfer function, $\mathbf{t}(x, y, \Delta z)$.

The CHARA diffraction model was written as a series of IDL procedures. IDL was chosen primarily because it was written to handle arrays efficiently, and already has a large library of astronomical functions available. The model generates two independent wave fronts, one for each arm of a 2-element interferometer. The model keeps track of $\mathbf{U}(x, y, z)$ and $\mathbf{A}(f_x, f_y; z)$ for each beam in separate arrays. Each array point is a structure with data fields for x, y coordinates (or f_x, f_y), a complex vector ($\mathbf{U}(x, y)$ or $\mathbf{A}(f_x, f_y)$), and dx, dy values for the grid spacing. Such a structure may be applied equally well to f_x, f_y or x, y planes.

If the model makes a change to either \mathbf{U} or \mathbf{A} , it takes a Fourier transform of the altered array and updates the other. During the Fourier transform routine, the model ensures that the total power in the x, y and f_x, f_y planes are equal.

The sequence of operations performed by the model is as follows:

1. Generate a turbulent wave front. This is most easily accomplished in f_x, f_y space, as per previous models (Buscher, 1988a; Bagnuolo, 1993; Horton et al., 2001).

$\mathbf{A}(f_x, f_y)$ at each point in the array is independently generated as a random complex vector. The phase ranges from 0 to 2π , and the magnitude from 0 to $3^{\frac{1}{3}}$. With this magnitude range, $\langle |\mathbf{A}|^2 \rangle = 1$. This can be shown by

$$\langle x^2 \rangle = \int_0^{A_{max}} x^2 dx = \frac{A_{max}^3}{3},$$

set this equal to 1 and solve for $A_{max} = 3^{\frac{1}{3}}$. At this point \mathbf{A} is simply white noise.

The model filters the noise by multiplying it by the square root of the phase perturbation power spectrum (equation 1.39):

$$\Phi(\rho) = 0.229 r_o^{\frac{5}{3}} \rho^{-\frac{11}{3}} .$$

Now the average power spectrum of \mathbf{A} is

$$\langle \mathbf{A} \mathbf{A}^* \rangle = \langle (|\mathbf{A}_{noise}| \sqrt{\Phi(\rho)})^2 \rangle = \Phi(\rho) \langle |\mathbf{A}_{noise}|^2 \rangle = \Phi(\rho) .$$

The Fourier transform of \mathbf{A} is the phase shift imparted by atmospheric turbulence in the x, y plane. Only the real component is used, as an imaginary phase shift causes an exponential decay in amplitude.

In order to verify that the turbulent phase shift was generated correctly, its structure function was calculated. The structure function $\langle |\phi(r_1 + r_2) - \phi(r_1)|^2 \rangle$ was measured by finding the phase difference between the origin and points at various distances along the x axis. This phase difference was squared, and then averaged over 1000 independent cases. The form of $\Phi(\rho)$ was modified until the phase structure function was as close a match as possible to (1.37):

$$D_\phi(r) = 6.88 \left(\frac{r}{r_o} \right)^{\frac{5}{3}} .$$

The final form of $\Phi(\rho)$ was

$$\Phi(\rho) = 0.257 r_o^{\frac{5}{3}} \rho^{-4} . \tag{2.52}$$

The fit was considered with turbulent phases generated with r_o ranging from 5 to 60 cm, the upper limit representing the size of seeing cells in the K band when seeing is very good ($\lambda(550 \text{ nm}) \approx 11 \text{ cm.}$)

Buscher (1988a, Appendix B) has pointed out that sampling in the f_x, f_y plane must at least satisfy

$$\rho_{min} < \frac{1}{1.08 r_o}$$

$$\rho_{max} > \frac{1}{6 D}$$

where D is the telescope diameter. The latter requirement is valid only if the telescopes have tip/tilt correction, otherwise ρ_{max} must be much larger. With the smallest r_o at 5 cm, and D at 1 meter, a grid of at least 256 x 256 points, 8 m in span on the x, y plane will satisfy both conditions. In some cases the model was run with the same grid span, but a larger number of points per side, up to 1024 x 1024 when generating images of characteristic diffraction patterns.

2. Remove tip/tilt from the atmospheric phase shift. A built-in IDL routine finds the best fit plane to the atmospheric phase shift within a square sub array $\sqrt{\frac{D}{2}}$ m to a side, centered on the x, y origin. The fitting routine is unable to ignore points within an array, making an exact fit over a round aperture difficult. Because a square array must be used, the logical choice is one with an area that is the same as a circle of diameter D . The square array will be slightly too large on the diagonals, but slightly too short on the x, y axes, so a plane fit to its atmospheric phase shift should be very nearly the same as one fit to the aperture itself. A square that is $r\sqrt{\pi}$ to a side has an area of πr^2 ,

hence the size of the sub array.

The best-fit plane is subtracted from the entire atmospheric phase shift array, then the phase shift is applied to an array of complex vectors with a magnitude of 1 and phase of 0. The resulting array is $\mathbf{U}(x, y, 0)$, the atmospherically perturbed wave front just before it enters the telescope. The amplitude of \mathbf{U} is still 1 at every point, but the phases have been modified by the atmosphere. This is consistent with the assumption that the atmospheric phase perturbations happen close enough to the ground that amplitude fluctuations may be neglected and only phase fluctuations considered.

The model accounts for the efficiency of the tip/tilt system by multiplying the best fit plane by a scale factor before subtracting it from the phase shift of the wave front. A typical efficiency for CHARA's tip/tilt system is 90% (ten Brummellaar, personal communication).

3. Multiply the wavefront function by the aperture function of the telescope in x, y space. The CHARA telescope primary has a radius of 0.5 m. The secondary and tertiary mirrors (M2 & M3) form a central obscuration that is within a few mm of circular, and is offset by a few mm. Because these offsets are so small compared to the size of the central obscuration, a uniform, centered radius of 12.5 cm was used. Any point in the $\mathbf{U}(x, y)$ array that lies outside the mirror edge or inside the central obscuration has its \mathbf{U} vector set to zero.

Other mirrors in the beam have a small hole in the center about 5 mm in diameter. These holes have been neglected in the model.

4. Reduce the beam at the telescope by m_{tel} , while the angular spectrum is ex-

panded by m_{tel} . The values of x , y , dx , and dy in the $\mathbf{U}(x, y)$ array are divided by m_{tel} . $\mathbf{U}(x, y)$ itself is multiplied by m_{tel} . The power is a sum over the entire array:

$$\sum_{i,j} |\mathbf{U}_{i,j}|^2 m_{\text{tel}}^2 \frac{dx}{m_{\text{tel}}} \frac{dy}{m_{\text{tel}}} ,$$

so the magnification factors will cancel out, ensuring that the power in the x, y plane remains unchanged.

Similarly, f_x , f_y , df_x and df_y in the $\mathbf{A}(f_x, f_y)$ array are multiplied by m_{tel} , while $\mathbf{A}(f_x, f_y)$ is divided by m_{tel} .

Beam reduction could be avoided by using effective propagation distances, such as those in Table 2.2. However, because beam reduction is such a trivial step, the model uses actual distances and takes beam reductions into account.

5. Propagate to the BRTs. Multiply $\mathbf{A}(f_x, f_y)$ by the propagation transfer function $\mathbf{T}(f_x, f_y; z_{\text{tel}})$. The distance from each telescope to the BRT, z_{tel} , may be found in Table 2.1.
6. Reduce the beam $\mathbf{U}(x, y)$ at the BRT by m_{brt} , and expand the angular spectrum $\mathbf{A}(f_x, f_y)$ by the same factor. This step is identical to the beam reduction at the telescope, with the substitution of m_{brt} .
7. Propagate to the beam combiner. Multiply $\mathbf{A}(f_x, f_y)$ by $\mathbf{T}(f_x, f_y; z_{\text{brt}})$. The distance from the BRT to the beam splitter, z_{brt} may be found in Table 2.1.
8. Integrate $\mathbf{U}(x, y, z)$ from both beams over a sub-aperture at the beam combiner to find the flux, visibility, and signal-to-noise ratio. The definitions of these

quantities follow, and are consistent with Horton's recent model (Horton et al., 2001).

When two beams with fields $\mathbf{U}_1(x, y)$ and $\mathbf{U}_2(x, y)$ are combined by superimposing them, the total intensity is

$$I(\phi) = \iint_{ap} |\mathbf{U}_1(x, y) + \mathbf{U}_2(x, y) e^{i\phi}|^2 dx dy \quad (2.53)$$

where ϕ is a relative phase difference between the two beams, and the integration is performed over the aperture. Expanding the square and separating out terms without ϕ dependence gives

$$\begin{aligned} I(\phi) = & \iint_{ap} [|\mathbf{U}_1(x, y)|^2 + |\mathbf{U}_2(x, y)|^2] dx dy \\ & + 2 \Re \left[e^{-i\phi} \iint_{ap} \mathbf{U}_1(x, y) \mathbf{U}_2^*(x, y) dx dy \right]. \end{aligned}$$

The first term is simply the flux F from both beams defined as

$$F = \iint_{ap} [|\mathbf{U}_1(x, y)|^2 + |\mathbf{U}_2(x, y)|^2] dx dy. \quad (2.54)$$

The integral in the second term gives the magnitude of the intensity fluctuations due to interference. This quantity X is defined as

$$X = \iint_{ap} \mathbf{U}_1(x, y) \mathbf{U}_2^*(x, y) dx dy. \quad (2.55)$$

Now

$$I(\phi) = F + 2 \Re [e^{-i\phi} X] \ .$$

Twice the magnitude of X is the total strength of the intensity fluctuation. Because the fringe visibility is defined as the magnitude of the oscillation divided by the mean intensity, the expression for visibility is

$$V = \frac{2|X|}{F} \ . \quad (2.56)$$

Several independent cases must be considered, each with independently generated turbulent wavefronts propagated through the system, and a root-mean-square value of V found which shows an averaged behavior for V . Horton et al. (2001) point out that both X and F fluctuate randomly with time, and so should be averaged separately. With this in mind, we use the following estimator for the rms visibility:

$$V_{\text{rms}} = 2 \sqrt{\frac{\langle |X|^2 \rangle}{\langle F \rangle^2}} \ . \quad (2.57)$$

In his Ph.D. thesis, Buscher (1988b) showed that the SNR for a group-delay fringe tracker that finds the frequency peak for fringes in a channel spectrum dominated by photon noise is

$$SNR \propto \frac{\langle |X|^2 \rangle}{\langle F \rangle^2} = \frac{\langle F \rangle}{4} V_{\text{rms}}^2 \ . \quad (2.58)$$

In order to find how V and SNR change with sub-aperture size, the model calculates F and X over several different sub-aperture sizes and stores these in

an array. To get $\langle |X|^2 \rangle$ and $\langle F \rangle$, the model generates 1000 independent cases of interfering beams. For each sub-aperture size, the model uses IDL's MOMENT function to find the average of $|X|^2$ and F over all 1000 cases. V_{rms} and SNR as functions of sub-aperture size are then calculated.

2.3 Model Predictions

Hrynevych (1992) found that the worst visibility losses due to diffraction happen when propagation distances are grossly mismatched. If two identical beams propagate the same distance, their diffraction patterns will be the same, and so there will be no visibility loss. However, if the propagation distances are very different, the bright rings in each diffraction pattern may not line up very well. The areas of strong field for the two beams do not coincide well, so the interference is weaker than that of identical beams.

Because the coupling of atmospheric turbulence with the spatial filtering aspect of propagation may result in an unpredictable visibility loss, two main cases were modeled:

1. Unequal propagation distances. Various POP and OPLE delay configurations were checked out with the chara_plan observation planning program until a worst-case scenario was found. When an object is rising in the East, the E1-W2 baseline provides the worst path mismatch. E1 is at the end POP, with up to 90 m of OPLE delay, while W2 is at POP 1 with zero OPLE delay. Another case that comes close is S1-E1, also when an object is rising in the East. In this case most of the path mismatch comes from the fact that E1 is the furthest

BRT from the beam combiner, while S1 is the closest.

2. Equal propagation distances. While no visibility loss is expected from plane wave beams that have matched propagation distances, atmospherically perturbed wavefronts may be another story completely. The “typical” case for the VIS system was chosen, which is the S1-S2 baseline, looking at an object above 60° altitude. In general, the POP and OPLE configurations will work out such that z_{tel} for both telescopes will be equal to the greater of the pair. That is, the scope with the shorter propagation distance tends to add enough OPLE and POP delay to bring it up to the longer propagation distance. This is mainly because S1-S2 is such a short baseline that the OPLE delay required due to the star’s motion during the night is less than 20 m, which is small compared to the 250 m total propagation distance.

2.3.1 Unequal Paths

The worst case of mismatched propagation distances is E1-W2 looking at an object rising in the east. At this position, E1 uses the end POP and up to 90 m of OPLE delay, totaling a z_{tel} of 478 m (See Table 2.1). W2 uses POP 1 and no OPLE delay, and so has a very short z_{tel} of 80 m. Also from Table 2.1, E1 has a z_{brt} to the VIS table combiner of 17.90 m, while that of W2 is 16.08 m. If the beam combiner were placed on the closest table to the BRTs (the Near Infrared Imaging table), those distances would be reduced to E1 z_{brt} : 12.18 m, W2 z_{brt} : 10.36 m.

The effective propagation distance β gives a direct feel for how badly the E1-W2 distances are mismatched. As a reminder, β is the distance from the telescope to the

beam combiner that would produce the same diffraction effects without the BRTs as the normal CHARA configuration with the BRTs. Equation 2.32 states that

$$\beta = z_{\text{tel}} + m_{\text{brt}}^2 z_{\text{brt}} .$$

With the beam combiner on the VIS table, E1 β is 1258 m, and W2 β is 780 m. If the beam combiner were moved to the NII table, E1 β is 1009 m, and W2 β is 451 m. Beam combination at the BRT is the worst possible case of mismatched paths, with the E1 beam traveling almost 6 times further than the W2 beam. The propagation distances after the BRT rapidly reduce this ratio, because the m_{brt}^2 term in (2.32) quickly dominates.

Beam Profiles

Figure 2.8 depicts an example of a single iteration of the model. In this case, the model was run at 750 nm, where the VIS is most sensitive, assuming an r_o of 10 cm at 550 nm. The images are the central 256 x 256 points of a 1024 x 1024 array, representing a 2 m x 2 m square at the telescope aperture. The array size was chosen to show detail in the beams. The figure shows the two telescope beam intensities, and the phase across each beam. Beam *a* corresponds to the E1 line, and beam *b* to the W2 line.

The first row is at the telescope aperture. The beam intensities are both 1, and the phase perturbation due to the atmosphere clearly visible in each beam.

In the second row, after the beams have propagated to the BRTs, the intensity of the E1 beam is clearly more degraded due to diffraction than the W2 beam. The

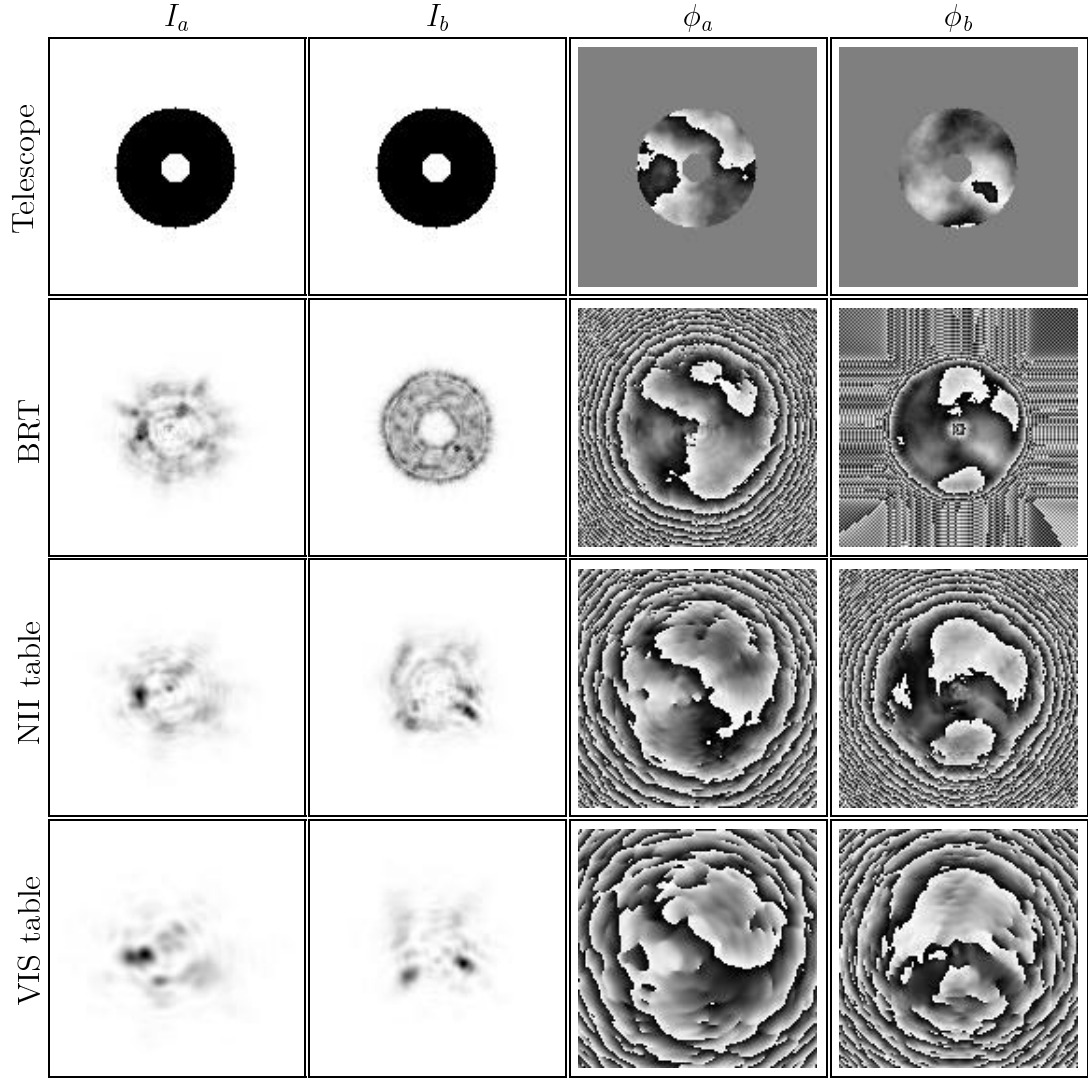


Figure 2.8: E1-W2 beams a and b vs. propagation distance. λ : 750 nm. $r_o(550nm)$: 10 cm. The columns are: I_a , I_b , ϕ_a , ϕ_b . The rows are the locations: Telescope , BRT , NII table, VIS table. Intensities are 0 (white) to 1 (black), for the sake of printing clarity. Phases are $-\pi$ (black) to π (white).

phase structure of each beam remains largely unchanged, with a slight blurring that increases with propagation distance.

The third row represents the beams at the combiner, if it were placed on the NII

table, the closest table to the BRTs. The beam intensities have degraded significantly since the BRTs. Because the equivalent propagation distance after the BRT is $m_{\text{brt}}^2 z$, propagating 10 m after the BRT is equivalent to propagating 435.6 m before the BRT.

Row 4 shows the beams at the combiner on the VIS table. The beam intensities bear little resemblance to those at the BRTs. The spatial filtering effect of propagation is manifest in the lack of small features in the beam intensity. The wavefront has been smoothed, removing the smaller structures.

The most physically intuitive way to visualize the effect of diffraction may be to consider the two beams separately. However, this does not lend itself very well to showing how fringe visibility itself is affected. A more useful set of plots are in Figure 2.9. These show the combined beam intensity F , $|X|$, and the phase of X , which shall be referred to as ϕ_X .

Equation 2.56 gives the definition of V for a single instant as $\frac{2|X|}{F}$, where X and F are both integrated over the region of interest in the beam. ϕ_X is important because $|X|$ is taken *after* the integral. ϕ_X may vary over the sub aperture and cause the integral of X to be zero. Figure 2.9 shows the value of $|X|$ and ϕ_X at each point in the combined beam. In order for V to be high over a sub-aperture, ϕ_X should change minimally over the areas where $|X|$ is high. As ϕ_X and $|X|$ become corrugated, V within a given sub-aperture will drop.

Figure 2.10 shows how changing r_o affects the combined beam. For a given sub-aperture size, V is much worse for small r_o . Similarly, Figure 2.11 shows how F , $|X|$, and ϕ_X change with wavelength. The first three rows represent the VIS system's approximate sensitivity range, with 750 nm near the peak sensitivity. One may readily see why interference fringes are much easier to obtain near the K' band (2.1315 μm)

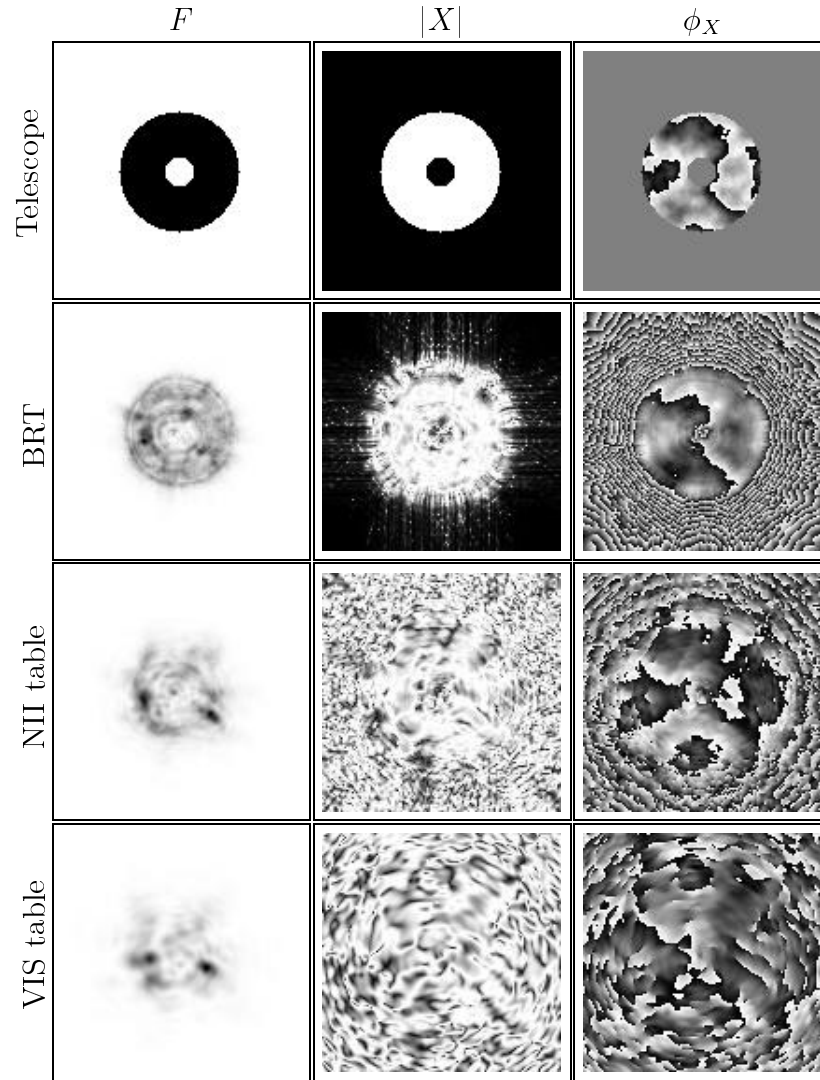


Figure 2.9: E1-W2 combined quantities vs. propagation distance. λ : 750 nm. $r_o(550 \text{ nm})$: 10 cm. Columns are: F , $|X|$, ϕ_X . Rows are the locations: Telescope, BRT, NII table, VIS table.

than in the visible, as X is much more smooth.

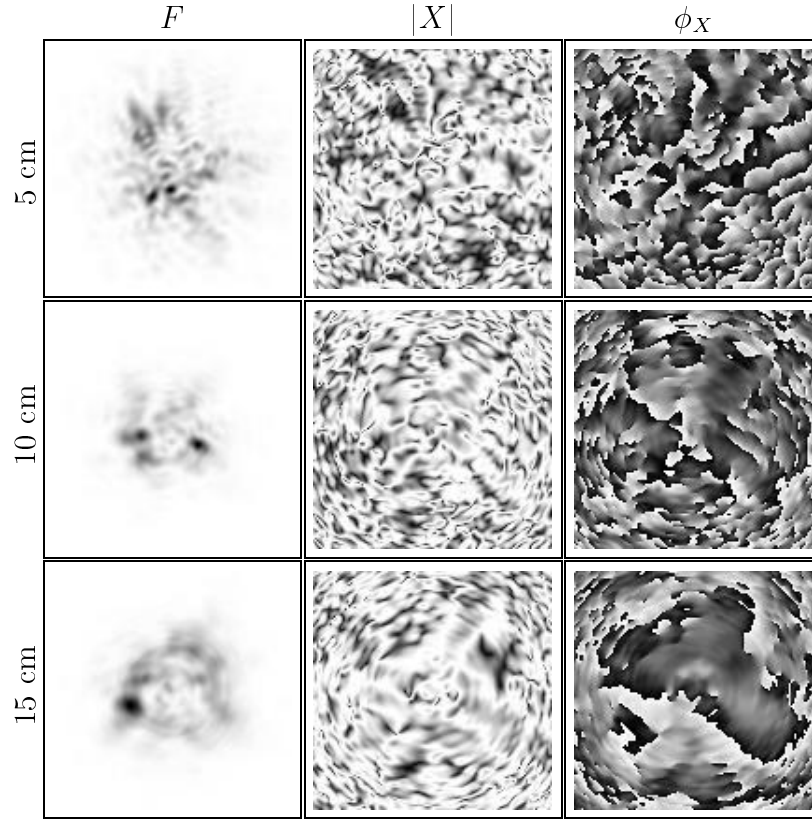


Figure 2.10: r_o dependence of E1-W2 combined quantities at the VIS table beam combiner. λ : 750 nm. Columns are: F , $|X|$, ϕ_X . Rows are r_o (550 nm): 5 cm, 10 cm, 15 cm.

V and SNR integrated over variable sub-apertures

The most important quantities to be considered are V and SNR integrated over the beam. The main purpose of this diffraction model was to investigate how these quantities change with the size of a sub-aperture. The most probable sub-aperture configuration is 7 circular apertures, each $\frac{1}{3}$ the diameter of the beam. Figure 2.12 illustrates this configuration.

The model integrates X and F over a sub-aperture that is centered on a point $\frac{1}{3}$

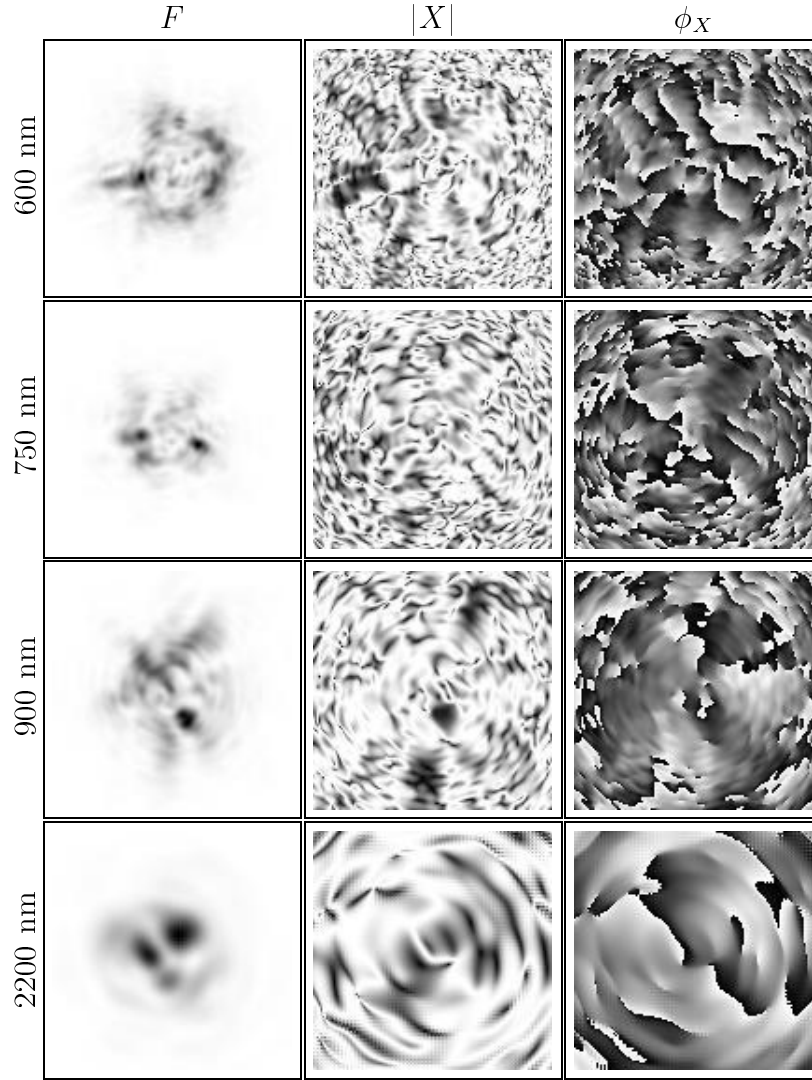


Figure 2.11: λ dependence of E1-W2 combined quantities at the VIS table beam combiner. $r_o(550 \text{ nm})$: 10 cm. Columns are: F , $|X|$, ϕ_X . Rows are wavelengths: 600 nm, 750 nm, 900 nm, 2200 nm.

of the beam diameter away from the beam center. This corresponds to the center of any of the outer sub-apertures in Figure 2.12. The integration is carried out over a range of sub-aperture radii, from $\frac{1}{18}$ of the beam radius up to twice the beam radius, which is enough to contain the entire beam. After all iterations of the model are

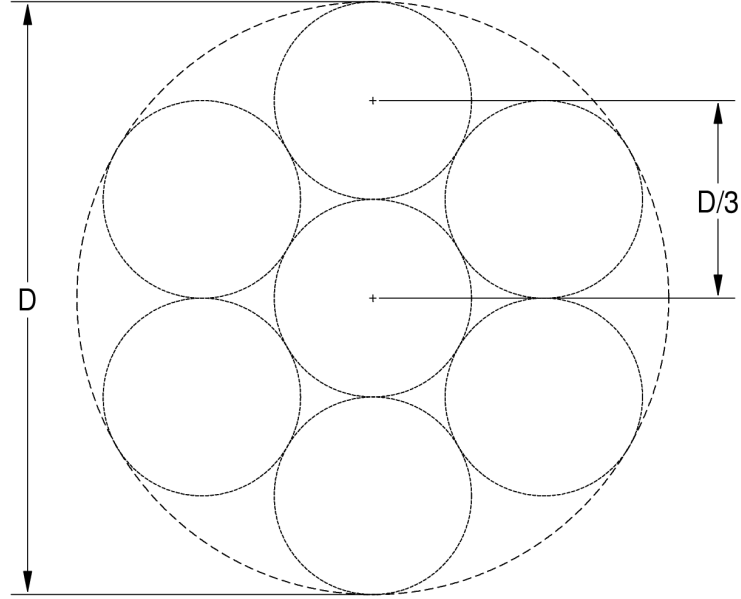


Figure 2.12: Sub-apertures taken at the beam combiner. Each sub-aperture is $\frac{1}{3}$ of the beam diameter, with the outer sub-aperture centers $\frac{1}{3}$ of the beam diameter away from the beam center.

complete, $\langle |X|^2 \rangle$ and $\langle F \rangle$ are calculated for each sub-aperture radius. V_{rms} and SNR are calculated according to (2.57) and (2.58).

Figure 2.13 depicts V_{rms} and SNR for poor seeing, an $r_o(550 \text{ nm})$ of 5 cm. The model generated 1000 iterations, each a 256×256 array spanning 8 m at the telescope aperture. The sampling across the innermost 1m containing the telescope primary is 32×32 .

Several curves are plotted, representing beam combination at different positions. The first position is at the telescope aperture. This is the behavior that would be manifest if diffraction had no effect on the beam whatsoever, and one could assume that a sub-aperture taken anywhere in the system is equivalent to one taken at the

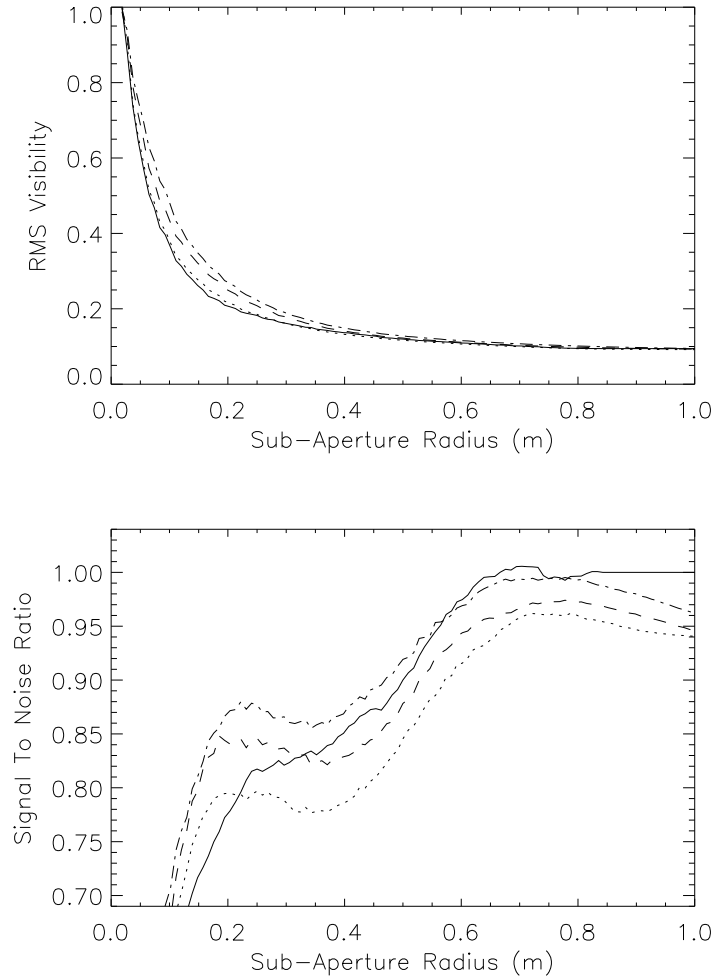


Figure 2.13: E1-W2 V_{rms} (top) and SNR (bottom) for poor seeing. $r_o(550 \text{ nm})$: 5 cm. λ : 750 nm. The lines are locations: Telescope - solid, BRT - dotted, NII table - dashed, and VIS table - dash-dot. The sub-aperture radius is measured at the scale of the telescope aperture. SNR is normalized to 1 for the full beam at the telescope aperture.

telescope. The remaining curves represent beam combination at the BRTs, the NII table, and the VIS table.

The visibility drops more gradually with sub-aperture radius as the propagation

distance to the beam combiner increases. There are two factors that contribute to this increase in visibility with propagation distance: the spatial filtering effect begins to dominate, and the mismatch in propagation distance between the two beams lessens.

The SNR plot shows that at either the VIS table or NII table beam combiners, the SNR actually is higher than at the telescope aperture for a sub-aperture radius under 0.35 m projected onto the primary. Because the propagation distance at the telescopes is effectively equal, the excess SNR at the VIS and NII tables represents the extra visibility boost caused by propagation-induced spatial filtering. The peak in SNR in this region is around 0.20 to 0.25 meter sub-aperture radius, depending upon the propagation distance to the beam combiner. An r_o of 5 cm at 550 nm becomes 7.25 cm at 750 nm. The peak position corresponds to a sub-aperture diameter of about $7r_o$.

The SNR at this sub-aperture diameter is about 88% that of the full beam at the telescope. This is the driving reason behind using multiple sub-apertures. If n sub-apertures are used simultaneously, the SNR will be \sqrt{n} times the SNR of one sub-aperture alone. If 7 sub-apertures are used, each is $\frac{1}{3}$ of the beam diameter, which gives each a radius of 0.1667 m projected onto the telescope aperture. The SNR for that radius is approximately 92% of the full beam SNR at the VIS table combiner. The combined SNR for all 7 sub-apertures is approximately 2.4 times the full beam SNR at the VIS table beam combiner. Even if only 2 sub-apertures are used, the SNR is still higher than that of the full beam.

As r_o increases, the behavior of the SNR with sub-aperture size changes. Figure 2.14 shows V_{rms} and SNR for good seeing, an r_o (550 nm) of 10 cm. The visibility is higher than the 5 cm r_o case, but the difference between the visibility at the tele-

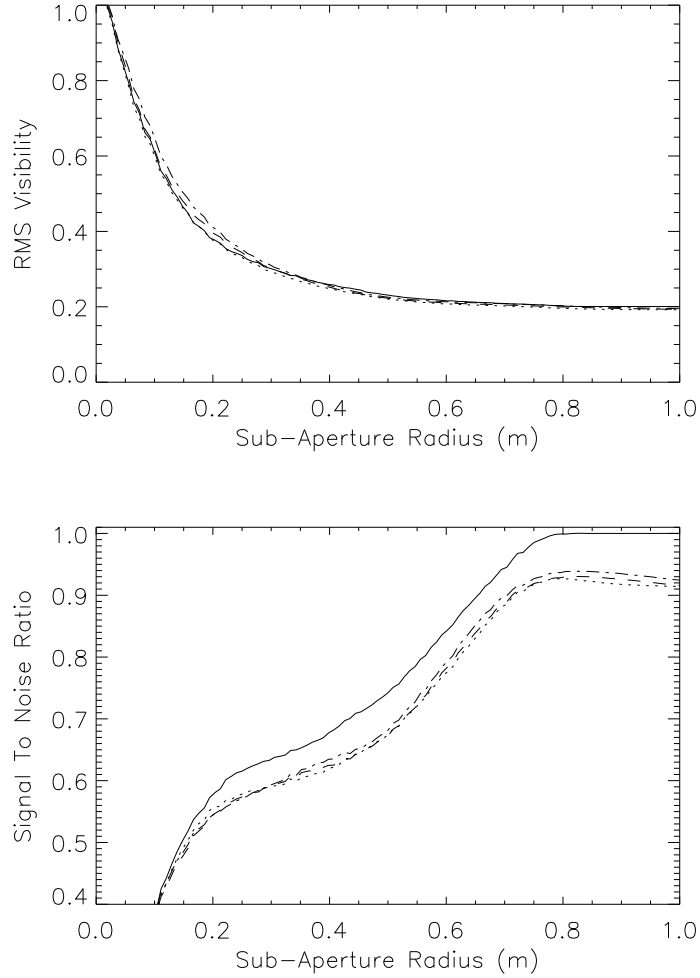


Figure 2.14: E1-W2 V_{rms} (top) and SNR (bottom) for good seeing. $r_o(550 \text{ nm})$: 10 cm. λ : 750 nm. The lines are locations: Telescope - solid, BRT - dotted, NII table - dashed, and VIS table - dash-dot. The sub-aperture radius is measured at the scale of the telescope aperture. SNR is normalized to 1 for the full beam at the telescope aperture.

scope and at the VIS table beam combiner is smaller. Because the wavefront is smoother, propagation-induced spatial filtering does not affect it as much, producing less of a visibility boost. The knee in the SNR curve is less pronounced, as r_o at

750 nm is 14.5 cm, so the knee is diminished to a local minimum in slope at a sub-aperture diameter of 0.30 m, which gives a sub-aperture diameter of $4 r_o$. The knee SNR at the VIS table beam combiner is no longer better than at the telescope, so the wave front must be too smooth for propagation-induced spatial filtering to boost the visibility significantly.

Figure 2.15 shows V_{rms} and SNR for excellent seeing, an $r_o(550 \text{ nm})$ of 15 cm. The spread in visibility with propagation distance is about $\frac{1}{3}$ that of the 5 cm r_o case. The SNR curve has a much less prominent knee, now just a local minimum in the slope near sub-aperture radius 0.35 m. At 750 nm r_o is 21.8 cm, so the knee is at a sub-aperture diameter of $3.2 r_o$. The SNR for a sub-aperture $\frac{1}{3}$ of the beam diameter is only 39% of the full beam SNR. Under these conditions, all 7 simultaneous sub-apertures are needed to obtain an SNR equal to that of the full beam. In this case, it is no longer advantageous to break the beam into sub-apertures. The wave front is smooth enough to use the full beam.

2.3.2 Comparison of Unequal With Equal Paths

A typical case of nearly equal propagation distances is S1-S2 looking at an object near the zenith. At this position, S2 uses POP 2 and S1 uses POP 1. This is enough delay to make the propagation distances from the telescopes to the BRTs about the same in both lines. Only a small amount of OPLE delay is required, typically less than 20 m over the portion of the sky above 45° elevation. Overall, z_{tel} for both S1 and S2 may be estimated as 250 m (See Table 2.1). Also from Table 2.1, S1 has a z_{brt} to the VIS table combiner distance of 13.67 m, while that of S2 is 14.25 m. If the beam combiner were placed on the closest table to the BRTs (the Near Infrared

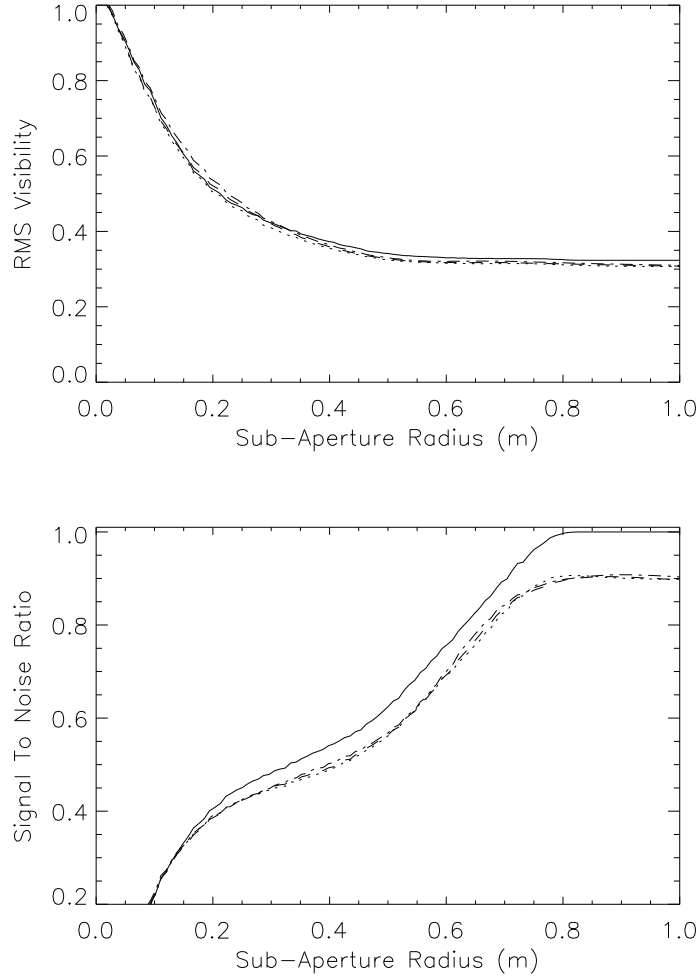


Figure 2.15: E1-W2 V_{rms} (top) and SNR (bottom) for excellent seeing. $r_o(550 \text{ nm})$: 15 cm. λ : 750 nm. The lines are locations: Telescope - solid, BRT - dotted, NII table - dashed, and VIS table - dash-dot. The sub-aperture radius is measured at the scale of the telescope aperture. SNR is normalized to 1 for the full beam at the telescope aperture.

Imaging table), those distances would be reduced to S1 z_{brt} : 7.94 m, W2 z_{brt} : 8.52 m.

The equivalent propagation distance β at the 12.5 cm beam scale is calculated

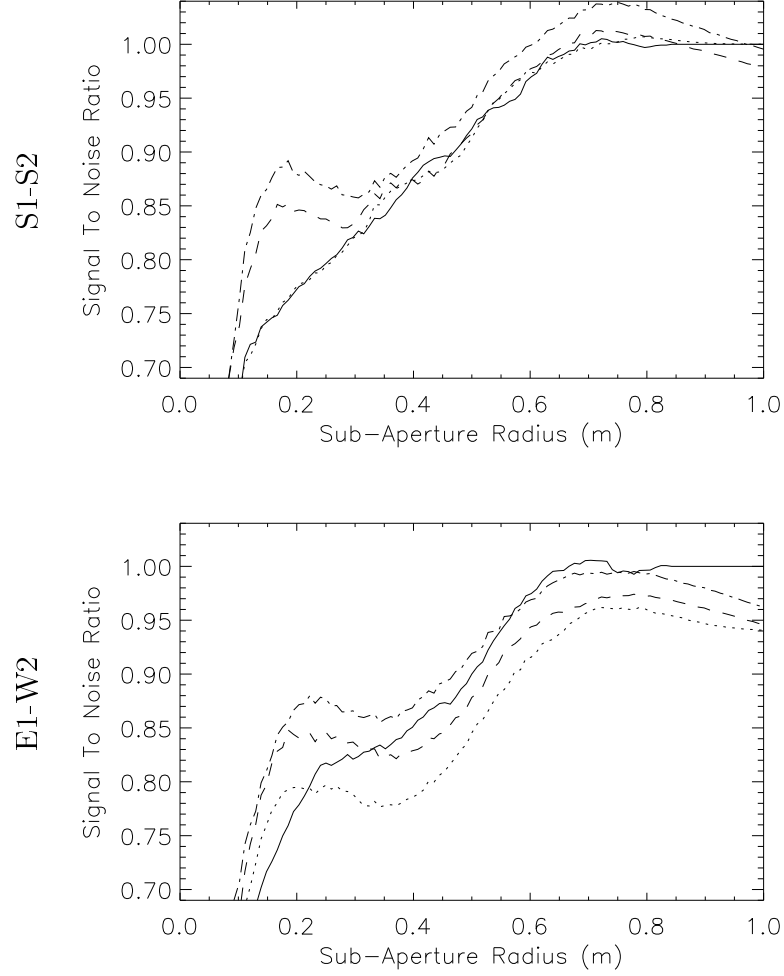


Figure 2.16: S1-S2 SNR (top) compared to E1-W2 SNR (bottom) for poor seeing. $r_o(550 \text{ nm})$: 5 cm. λ : 750 nm. The lines are locations: Telescope - solid, BRT - dotted, NII table - dashed, and VIS table - dash-dot. The sub-aperture radius is measured at the scale of the telescope aperture. SNR is normalized to 1 for the full beam at the telescope aperture.

using Equation 2.32 and the above values of z_{tel} and z_{brt} . With the beam combiner on the VIS table, S1 β is 845 m, and S2 β is 871 m. If the beam combiner were moved to the NII table, S1 β would be 596 m, and S2 β would be 621 m.

The S1-S2 visibility and SNR curves are very similar to those of E1-W2, especially for the larger r_o sizes. Poor seeing is the only case where the S1-S2 data differs enough from E1-E2 to merit a detailed plot. Figure 2.16 shows the SNR curves for S1-S2 vs. sub-aperture size, and compares them to the same curves for E1-E2. In general, the E1-W2 SNR curve is the same shape as the S1-S2 SNR curve, and about 90-95% its amplitude. The SNR curves for all values of r_o are plotted for both the E1-W2 and S1-S2 cases in Figure 2.17.

There are two main differences between the plots. First, the knee present in the E1-W2 plots for beam combination at the Telescope and at the BRT is absent in the S1-S2 data. The knee remains virtually unchanged in the S1-S2 curves for beam combination at the NII and VIS tables. The second difference is manifest where the sub-aperture size is large enough to contain the entire beam. All of the E1-W2 curves level out at a lower value than the S1-S2 curves as they near a sub-aperture radius of 1 m. The visibility is lower here for the E1-W2 case because of the mismatch in propagation distances between the two beams. Note that the E1-W2 curve for beam combination at the BRT is the most severely affected, which is where the propagation distances are most uneven.

SNR and V dependence upon r_o

The SNR figures in the previous sections have been presented each on their own scale, normalized to the SNR of the full beam at the telescope. Now we must establish the scale of the SNR data for all seeing conditions with respect to each other. Figure 2.17 has SNR curves at the VIS table beam combiner plotted for r_o (550 nm) from 5 to 15 cm. The solid curves are S1-W1 data, while E1-W2 is represented by dotted curves.

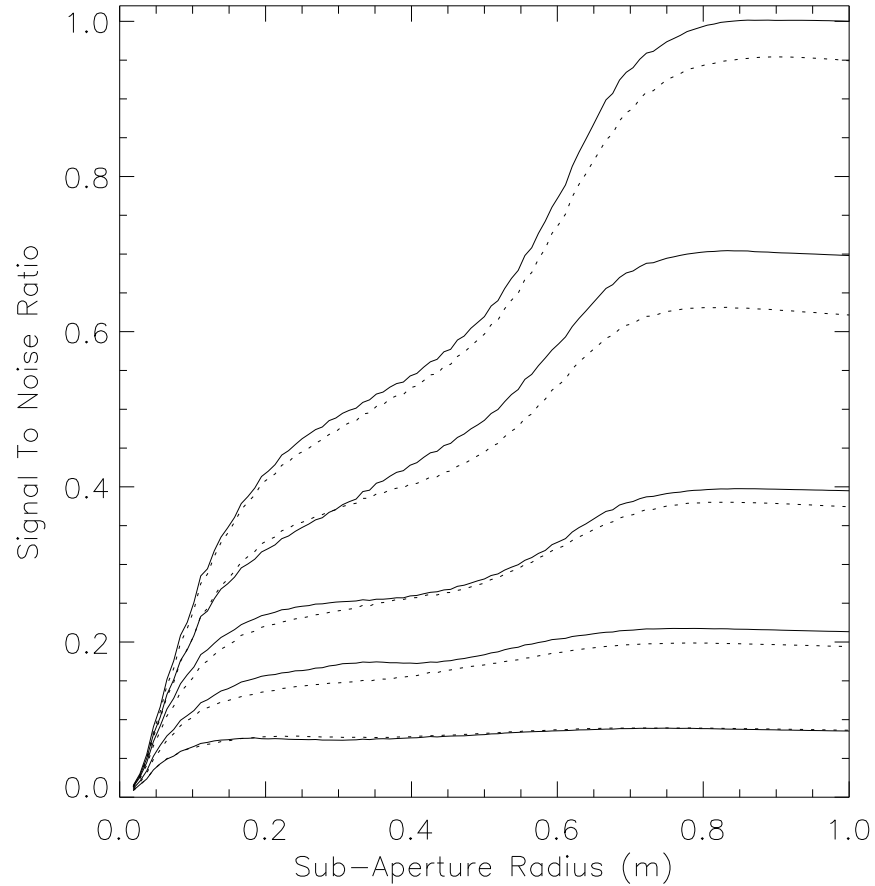


Figure 2.17: E1-W2 and S1-S2 SNR at the VIS table combiner vs. sub-aperture radius over a range of r_o . λ : 750 nm. The solid lines represent the S1-S2 SNR curves, in order of $r_o(550 \text{ nm})$ from the top down: 15 cm, 12.5 cm, 10 cm, 7.5 cm, 5 cm. The dotted lines are the E1-W2 SNR curves, in the same order. The sub-aperture radius is measured at the scale of the telescope aperture. SNR is normalized to 1 for the S1-S2 full beam at the VIS table combiner, with $r_o(550 \text{ nm})$: 15 cm.

All of the data were normalized to the S1-S2 SNR of the full beam at the VIS table beam combiner, with $r_o(550 \text{ nm}) = 15 \text{ cm}$. The curves are in order of seeing quality; the topmost curve is for a 15 cm $r_o(550 \text{ nm})$, down to the lowest curve at 5 cm. The

SNR at 5 cm r_o (550 nm) is $\frac{1}{6}$ of the 15 cm r_o SNR for a 16.67 cm sub-aperture radius, while it is $\frac{1}{12}$ the 15 cm SNR at the full beam width.

While the plots up to this point have covered a continuous range of sub-aperture radius, there are only a few sub-aperture sizes and configurations worth considering. Assuming that we want to put as much of the beam's area into sub-apertures as possible, the best configurations are 3, 4, and 7 sub-apertures. In each of these configurations, the sub-aperture radii are sized such that they touch the outer edge of the (non-diffracted) beam and each adjacent sub-aperture. The 1 m telescope aperture can accommodate 7 sub-apertures of 16.67 cm radius, 4 sub-apertures of 20.71 cm radius, or 3 sub-apertures of 23.21 cm.

Figure 2.18 plots the E1-W2 V_{rms} and SNR for 7, 4, and 3 sub-apertures as well as for the full beam. The SNR curves show that by using 7 sub-apertures, we can obtain a higher SNR than any other configuration. It is apparent that 7 sub-apertures is more effective than 3 or 4, with the full beam SNR matching the 7 sub-aperture SNR at excellent seeing conditions. Both the V_{rms} and the SNR fall off severely with smaller r_o , so although 7 sub-apertures perform better in worse seeing, the best possible seeing conditions should be sought.

The curves for E1-W2 SNR and V were each found to loosely fit an exponential, with the full beam curves fitting the best. For 7 sub-apertures, $\text{SNR} \propto r_o^{1.49}$. For the full beam, $\text{SNR} \propto r_o^{2.19}$. For 7 sub-apertures, $V_{\text{rms}} \approx 0.123 r_o^{0.588}$. For the full beam, $V_{\text{rms}} \approx 0.0163 r_o^{1.08}$.

Figure 2.19 plots the S1-S2 V_{rms} and SNR for 7, 4, and 3 sub-apertures as well as for the full beam. They have almost the same form as the E1-W2 curves, but were different enough to merit a separate plot. For 7 sub-apertures, $\text{SNR} \propto r_o^{1.45}$, less steep

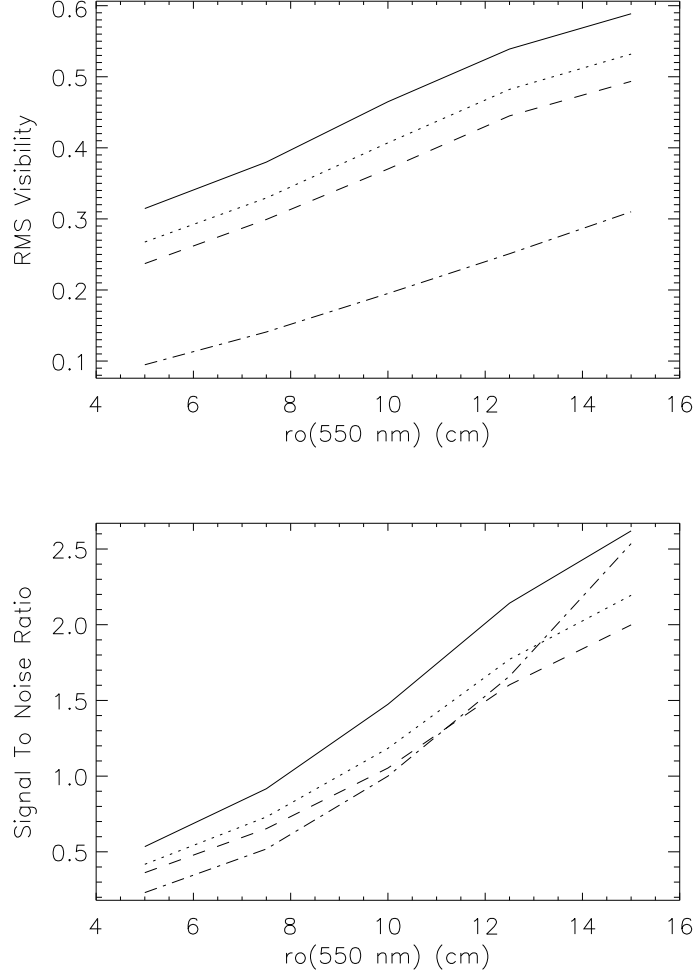


Figure 2.18: E1-W2 V_{rms} (top) and SNR (bottom) vs. $r_o(550 \text{ nm})$ for various sub-aperture sizes. λ : 750 nm. The lines are number of sub-apertures and their radii: 7 (0.1667 m) - solid, 4 (0.2071 m) - dotted, 3 (0.2321 m) - dashed, and 1 (full beam) - dash-dot. The sub-aperture radius is measured at the scale of the telescope aperture. SNR is normalized to 1 for the full beam at the VIS table beam combiner with $r_o(550 \text{ nm})$: 15 cm.

than the E1-W2 7 sub-aperture SNR. For the full beam, $\text{SNR} \propto r_o^{2.26}$, steeper than E1-W2. Note that in the S1-S2 plot, the 7 sub-aperture and full beam SNR curves actually meet at $r_o(550 \text{ nm}) = 15 \text{ cm}$. The 7 sub-aperture $V_{\text{rms}} \approx 0.130 r_o^{0.559}$, while

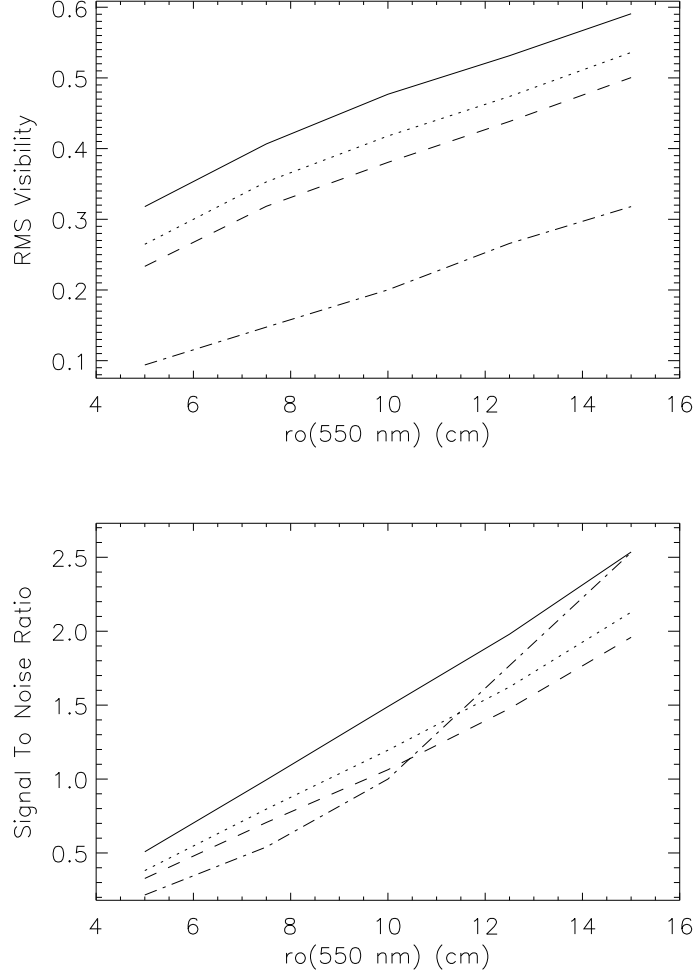


Figure 2.19: S1-S2 V_{rms} (top) and SNR (bottom) vs. $r_o(550 \text{ nm})$ for various sub-aperture sizes. λ : 750 nm. The lines are number of sub-apertures and their radii: 7 (0.1667 m) - solid, 4 (0.2071 m) - dotted, 3 (0.2321 m) - dashed, and 1 (full beam) - dash-dot. The sub-aperture radius is measured at the scale of the telescope aperture. SNR is normalized to 1 for the full beam at the VIS table beam combiner with $r_o(550 \text{ nm})$: 15 cm.

the full beam $V_{\text{rms}} \approx 0.0155 r_o^{1.12}$.

SNR and V Dependence Upon λ

The diffraction model was run at 4 wavelengths. The first three, 600 nm, 750 nm, and 900 nm loosely represent the band pass of the VIS system, with the peak sensitivity at 750 nm. In addition, the model was run at 2200 nm to compare the VIS system with the K band, where the bulk of observations at CHARA are currently taken.

The effect of moving to a shorter wavelength is identical to choosing a smaller r_o as far as the incoming wavefront is concerned, but the proportion of the beam diameter to λ is different, making it necessary to consider wavelength and r_o independently.

The model results did not appear significantly different from those previously presented at varying r_o so all of the results are shown together. Figure 2.20 is a plot of E1-W2 and S1-S2 SNR vs. sub-aperture size for all wavelengths. Again, S1-S2 data is plotted as solid lines, while E1-W2 lines are dotted.

The 2200 nm data has a SNR that is so much higher than the VIS band data that it has been allowed to run off the plot in order to better view the other wavelengths' SNR. All of the SNR data has been normalized to the full beam SNR at the VIS table beam combiner of S1-S2 at $\lambda = 2200$ nm. The 2200 nm data increases almost linearly until it flattens out at the full beam size (a radius of approximately 0.83 m due to the center of the sub-aperture not being at the center of the beam) to a value of 1.0 for S1-S2, and 0.85 for E1-W2.

Figure 2.21 plots the E1-W2 V_{rms} and SNR vs. λ at $r_o(550 \text{ nm}) = 10$ cm for 7, 4, and 3 sub-apertures as well as for the full beam. At $\lambda \geq 1000$ nm, the full beam SNR surpasses the 7 sub-aperture SNR. This is a very rough estimate, as there are no data points between 900 and 2200 nm. It does show, however, that for any near-Infrared

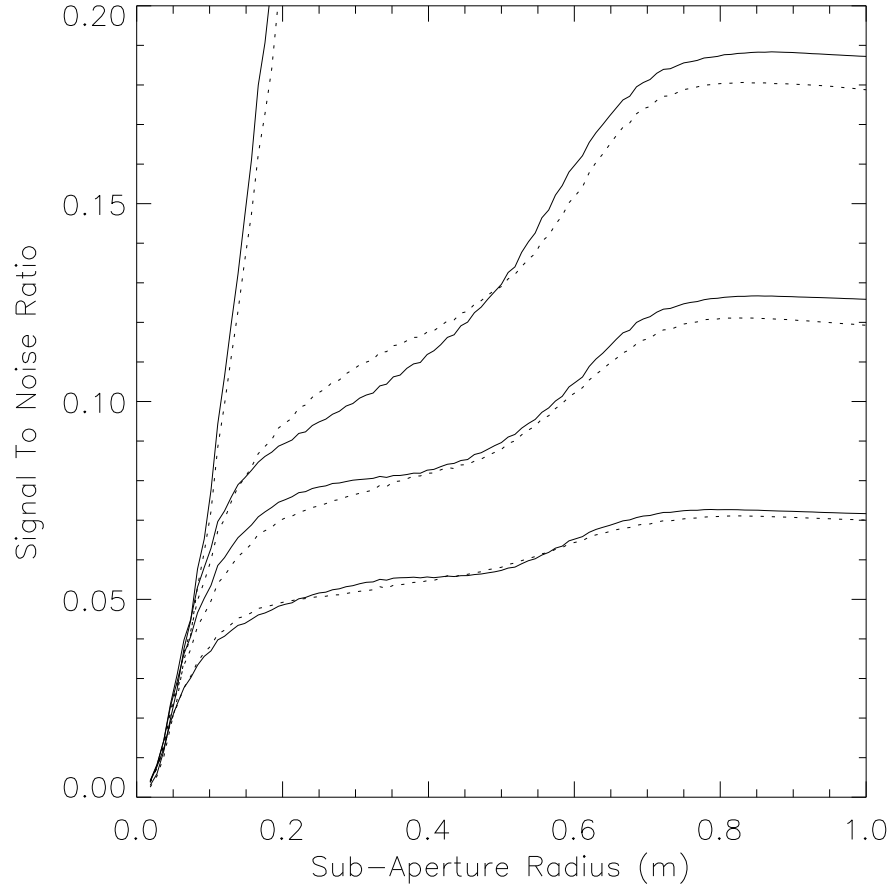


Figure 2.20: E1-W2 and S1-S2 SNR at the VIS table combiner vs. sub-aperture radius over a range of λ . $r_o(550 \text{ nm})$: 10 cm. The solid lines represent the S1-S2 SNR curves, in order of λ from the top down: 2200 nm, 900 nm, 750 nm, 600 nm. The dotted lines are the E1-W2 SNR curves, in the same order. The sub-aperture radius is measured at the scale of the telescope aperture. SNR is normalized to 1 for the S1-S2 full beam at the VIS table combiner, with λ : 2200 nm. The 2200 nm curves climb nearly linearly off the graph to reach 1.0 for S1-S2 and 0.85 for E1-W2.

observations longer than 1000 nm in good seeing, there is negligible benefit, at best, to using sub-apertures. We also see that the full beam SNR in K band is about 7 times the full beam SNR at 750 nm. The SNR drops precipitously with λ , which

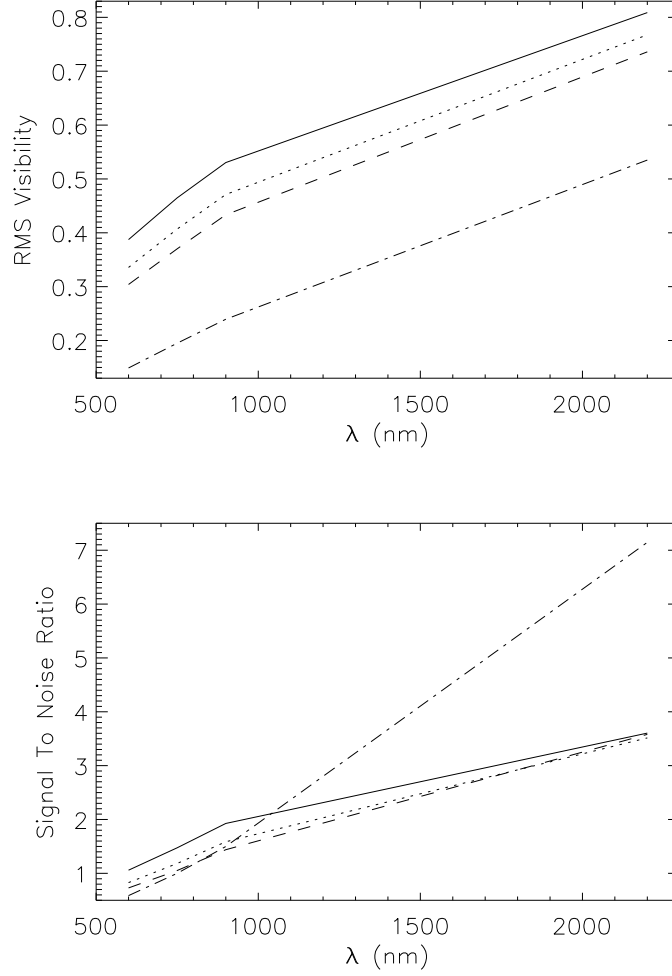


Figure 2.21: E1-W2 V_{rms} (top) and SNR (bottom) vs. λ for various sub-aperture sizes. $r_o(550 \text{ nm})$: 10 cm. The lines are number of sub-apertures and their radii: 7 (0.1667 m) - solid, 4 (0.2071 m) - dotted, 3 (0.2321 m) - dashed, and 1 (full beam) - dash-dot. The sub-aperture radius is measured at the scale of the telescope aperture. SNR is normalized to 1 for the full beam at the VIS table beam combiner with λ : 750 nm.

when combined with shorter τ_o from a smaller r_o , will compound the difficulty of measuring stellar fringes in the visible wavelength regime.

For 7 sub-apertures, $\text{SNR} \propto \lambda^{0.89}$. For the full beam, $\text{SNR} \propto \lambda^{1.88}$. The 7 sub-

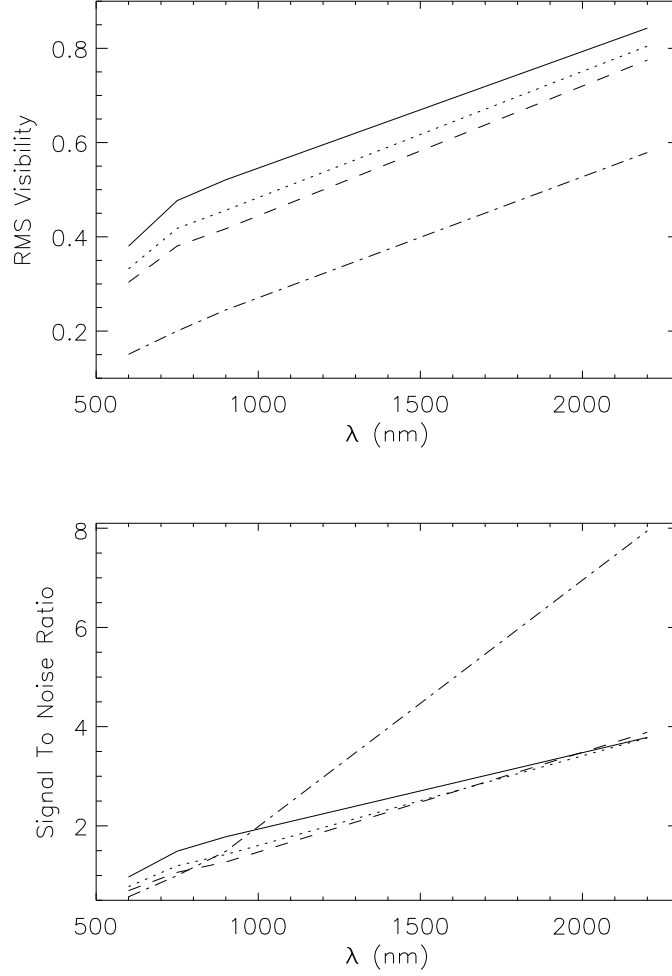


Figure 2.22: S1-S2 V_{rms} (top) and SNR (bottom) vs. λ for various sub-aperture sizes. $r_o(550 \text{ nm})$: 10 cm. The lines are number of sub-apertures and their radii: 7 (0.1667 m) - solid, 4 (0.2071 m) - dotted, 3 (0.2321 m) - dashed, and 1 (full beam) - dash-dot. The sub-aperture radius is measured at the scale of the telescope aperture. SNR is normalized to 1 for the full beam at the VIS table beam combiner with λ : 750 nm.

aperture $V_{\text{rms}} \approx 0.0124 \lambda^{0.545}$, while the full beam $V_{\text{rms}} \approx 3.25 \times 10^{-4} \lambda^{0.965}$.

Figure 2.22 plots the S1-S2 V_{rms} and SNR vs. λ at $r_o(550 \text{ nm}) = 10 \text{ cm}$ for 7, 4, and 3 sub-apertures as well as for the full beam. The plots are nearly identical to

those for E1-W2, except that the full aperture SNR surpasses the 7 sub-aperture SNR at a slightly shorter wavelength, and at 2200 nm it reaches 8 times the full aperture SNR at 750 nm.

For 7 sub-apertures, $\text{SNR} \propto \lambda^{0.98}$. For the full beam, $\text{SNR} \propto \lambda^{1.99}$. The 7 sub-aperture $V_{\text{rms}} \approx 9.60 \times 10^{-3} \lambda^{0.583}$, while the full beam $V_{\text{rms}} \approx 2.35 \times 10^{-4} \lambda^{1.02}$.

2.3.3 Conclusion

The model results show that a cluster of seven sub-apertures offers the highest possible SNR in the 600-1000 nm wavelength regime for standard seeing conditions. When seeing becomes excellent, the option of using the entire beam would be desirable. It must be stressed that if not all seven sub-apertures are used, the the quantities predicted in this section must be scaled by $\sqrt{\frac{n}{7}}$, where n is the number of sub-apertures actually used.

Invention, my dear friends, is 93 percent perspiration, 6 percent electricity, 4 percent evaporation, and 2 percent butterscotch ripple.

— Willy Wonka

Chapter 3

The CHARA VIS System Design

The CHARA Array Visible to Near Infrared Spectrograph (VIS) System is the first CHARA subsystem designed to detect starlight fringes in the 600-1000 nm wavelength regime. The CHARA telescope apertures are much larger than $3r_o$ at these wavelengths, so the beam must be stopped down to improve fringe signal-to-noise. In order to preserve as much starlight as possible, the beams are broken into 7 sub-apertures after combination by an array of lenses that each inject their light into a multimode optical fiber (the original 7-lens concept originated with Bill Bagnuolo during the early design stages of the CHARA Array). The optical fiber makes the task of piping the light from each sub-aperture to the detector much more manageable than any method using bulk optics.

The optical fibers are grouped together into a linear array, which forms the slit of a low-resolution prism spectrograph. Each fiber tip in the slit corresponds to a particular sub-aperture of a particular beam combiner output beam. The light from the fiber slit is collimated, spectrally dispersed, then focused onto a CCD that is read out at high speed. Each fiber in the slit forms a separate spectrum on the detector, so the light from each sub-aperture may be processed independently.

One of the two beams in a given pair reflects off of a dither mirror before it reaches

the beam combining splitter. Starlight fringes are recorded by continuously scanning the dither mirror through a $88.18\ \mu\text{m}$ range, producing a $182.58\ \mu\text{m}$ window of OPD in which to find the stellar fringe packet.

The software runs under Real-Time Linux, a Unix-based operating system for personal computers for which users may write modules to add to the operating system kernel. Kernel modules are necessary for tasks that must be done real-time, such as receiving data from a high-speed camera.

The camera driver places each frame's data in a Direct Memory Access (DMA) buffer, and activates a callback function. The callback function is in a real-time module, which does some processing of the data and places it in a First-In-First-Out (FIFO) buffer accessible by normal non-kernel programs. A separate user-side (non-kernel) program reads the data from the FIFO, displays it for the user, and records the data to a file.

In the following sections, each component of the VIS system is described in detail.

3.1 The VIS Table Layout

Figure 3.1 is an overhead view of the VIS optical table in the Beam Combining Lab. The 6 visible beams enter from the lower right of the diagram. Beams 5 and 6 (drawn with black lines) were the two beams used for the proof-of-concept 2 beam combiner in 2005. Beams 1-4 (drawn in gray) were being used as a 4-way pairwise beam combiner in conjunction with the MIRC system, a 4-6 beam combiner in the infrared. Before this point, the beams 5 and 6 have passed through 2 dichroic splitters. The first dichroic separates out all light with a wavelength longer than $1\ \mu\text{m}$, sending it to the

IR table and the CHARA Classic IR beam combiner. The second dichroic removes all visible light bluer than 600 nm, which is sent into the 2-beam photomultiplier-based tip/tilt system.

When the 6 telescope beams reach the VIS table, beams 1-4 pass through a gray splitter, which reflects 50% of the light in the VIS system wave band. The reflected beams travel west to the newer CCD-based tip/tilt system, built to handle up to 9

beams.

After the tip-tilt splitters, all 6 beams pass through a line of shutters. The shutters are computer-controlled, and used to manage which beams enter the combiner at any given time.

After the shutters, beam 5 reflects off of the dither mirror (see Section 3.2) and through the back side of the beam 5 & 6 combining splitter. The beam combination takes place on the north surface of the splitter. Beam 6 passes through a compensator plate of the same thickness as the combining splitter so that beam 5 and 6 have the same chromatic dispersion when they reach the beam combination surface.

The beam combining splitter is a 50/50 split, so half of the energy from beam 6 passes through the splitter, and the other half reflects. Beam 5 does likewise, so the output beams from the splitter are each half beam 5, half beam 6. The two beams are 180° out of phase, when the north beam has a bright constructive interference fringe, the south beam has a dark destructive interference fringe.

The north output beam leaves the 5 & 6 combining splitter and shines directly into the fiber injector. A cube splitter may be placed into the beam before the injector for alignment of the injector (see Section 3.4.1).

The south beam is used to introduce an alignment laser beam into the system. The laser beam enters the VIS table from the southwest corner. The alignment laser passes through a cube splitter as it enters the VIS table. The cube splitter serves a dual purpose: first, when the beam is traveling east a secondary beam exits the cube to the north and is injected into a multimode optical fiber. The fiber is used as a light source in the fiber injector alignment scheme. The second purpose of the cube splitter comes in to play when the beam is traveling west. In this case a secondary

beam exits the cube splitter to the south. This beam is steered into an intensified CCD used for alignment of tip/tilt to the visible beams 1-6. The beam may also be steered into a small alignment telescope or a theodolite by inserting steering mirrors on kinematic mounts into the beam.

After passing eastward through the cube splitter, the beam reflects off of a steering mirror mounted on a computer controlled stage. The stage makes it possible to introduce the alignment laser into the system through the south output beam of any of the beam 1-6 combining splitters.

The south output beam of the beam 5 & 6 combiner may also be piped into a fiber injector by using a fold mirror to crowd an additional injector onto the extreme south end of the table. This has not been implemented as of August 2005, as that area was taken up by some of the intensified CCD electronics to the west. It may be best in the long run to fit an injector there, however, as this would mean that both sides of the beam 5 & 6 splitter are in use. Because the two outputs of a single pairwise combiner are 180° out of phase, the visibility of the fringes may be recorded by keeping track of the sum and difference of the two channels, which will be less sensitive to noise, and removes any scintillation.

In the event that the south output of the beam 5 & 6 combiner is used, it will need to be aligned by a laser beam introduced to the system from the north side of the combiner. The alignment laser can be steered into the north side of the beam 5 & 6 combiner simply by placing a steering mirror on a kinematic base into the north output beam and moving the linear stage so that the laser beam hits the kinematic steering mirror. This configuration is depicted in the diagram by the movable linear stage mirror in line with the beam 3 & 2 combiner output beam, which happens to

line up with where the steering mirror would be placed into the north output beam of the 5 & 6 combiner.

Beams 1-4 are part of a separate combination scheme in this configuration. The 4 beams are set up in a pairwise combiner in which each beam is split, then combined with its two adjacent neighbors. That is, beam 2 is split and combined with beams 1 and 3, beam 3 with 2 and 4, beam 4 with 3 and 1, etc. The pre-combination split takes place just after the shutters, at the east side of the VIS table. The steering mirrors for the north output beams of the pre-combination splitters are in a different order than those of the south output beams. This makes it possible to combine a given beam with its neighbors. For instance, consider the beam 4 splitter. The north output beam steering mirror is lined up with the beam 1 & 4 combining splitter, while the south output beam steering mirror is lined up with the beam 4 & 3 combining splitter.

As of August 2005, the beam 1-4 combiner was used mostly as a mechanism to introduce alignment laser beams into beams 1-4 of the CHARA system, in order to align the system for MIRC. By replacing either the north or south beam steering mirrors with dither mirrors, the beam 1-4 combiner could be used to record starlight fringes by scanning the dither OPD range and recording the fringe packets.

The north outputs of the beam combining splitters were not in use as of August 2005, but fiber injectors will be placed at each output in the event that the combiner will be used on the sky. The south output beams of the beam 1-4 combiner could also be steered onto fiber injectors, but more optical table real estate would need to be made available. In the current configuration, the beam 1-4 combiner north fiber injectors are in the way of where more steering mirrors would need to go in order

to introduce the alignment laser into the beam 1-4 combiners from the north side to align the south fiber injectors. None of these problems is terribly limiting, but would require a few components to move around in order to implement.

The beam 1-4 combiner was designed by Theo ten Brummelaar and Judit Sturmann, and was implemented during the summer of 2005, because alignment beams 1-4 were needed for MIRC, as well as for an upgrade of CHARA classic to a 3-beam combiner. The beam 5-6 combiner was the first assembled at CHARA in 1999, designed and implemented by Theo ten Brummelaar, Steve Ridgway, and Judit Sturmann. It has been through several iterations, but the layout has remained fundamentally the same for some years. The beam 5-6 combiner has been used for the introduction of alignment beams to CHARA Classic and FLUOR. The fiber injectors and fiber slit were designed by the author and engineers at Fiberguide Industries. The spectrograph was designed and implemented by the author, following the basic prism spectrograph design.

3.2 Dither Mirror

The VIS dither mirror is an Owens-Corning ULE Titanium Silicate glass flat of $\lambda/20$ surface quality with an FS.99-500 protected silver coating by Denton Optical. The mirror is mounted on a Piezosystem Jena PU 100 Piezoelectric Transducer (PZT) stage. A PZT is a quartz crystal that expands when a voltage is applied. Because the PZT has no moving parts and is mounted in a one-piece stage that flexes as the PZT expands, it makes a very reliable positioning device, with a mechanical resolution of 0.18 nm. The actual positioning resolution of the dither mirror depends upon the

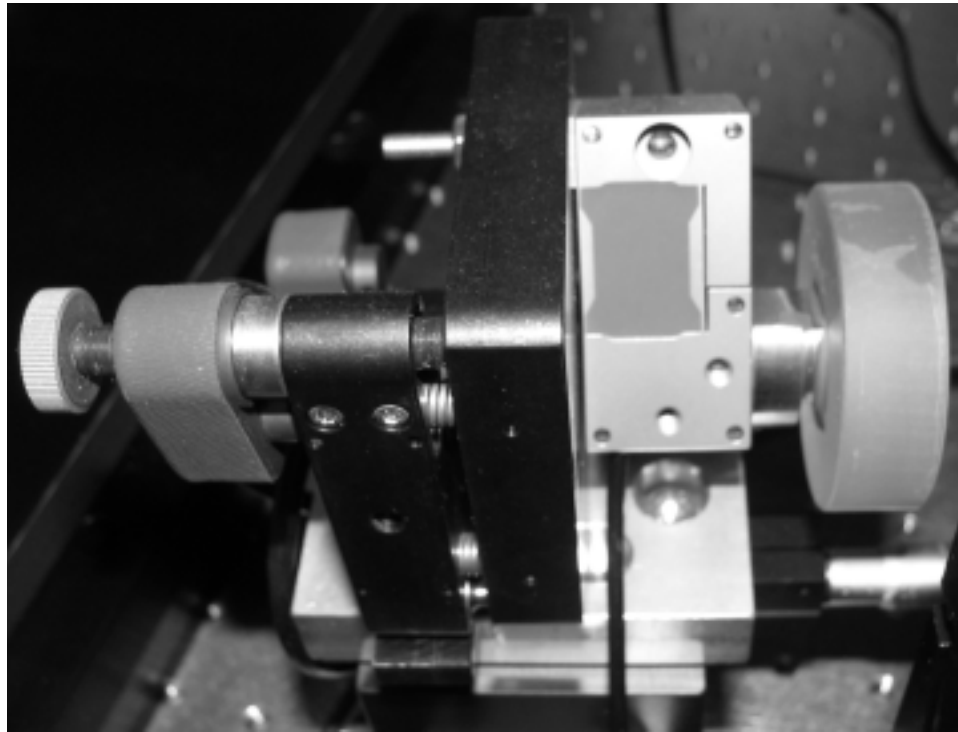


Figure 3.2: The VIS dither mirror. The PZT is inside the large silver block upon which the mirror (far right) is mounted. An increase in voltage to the PZT causes it to expand. The mirror moves closer to the combining splitter, causing a decrease in OPD. Computer-controlled picomotors (far left) change the angle of the dither mirror for alignment of the system.

voltage resolution of the controller that drives the PZT.

The PZT voltage is managed by a PCI card installed in the same machine that controls the VIS CCD camera. The PCI card outputs a voltage in the 0-10V range, which is amplified before being sent to the PZT. The PCI card voltage has 15 bits (32,768 levels) of resolution, which when amplified translate to a 2.69 nm resolution in dither mirror position. The driver for the PCI card was written for CHARA by Tom Schneider of Riptide Realtime (www.riptiderealttime.com). The driver takes the low 15 bits of a 16-bit unsigned integer, and tells the PCI card to output the

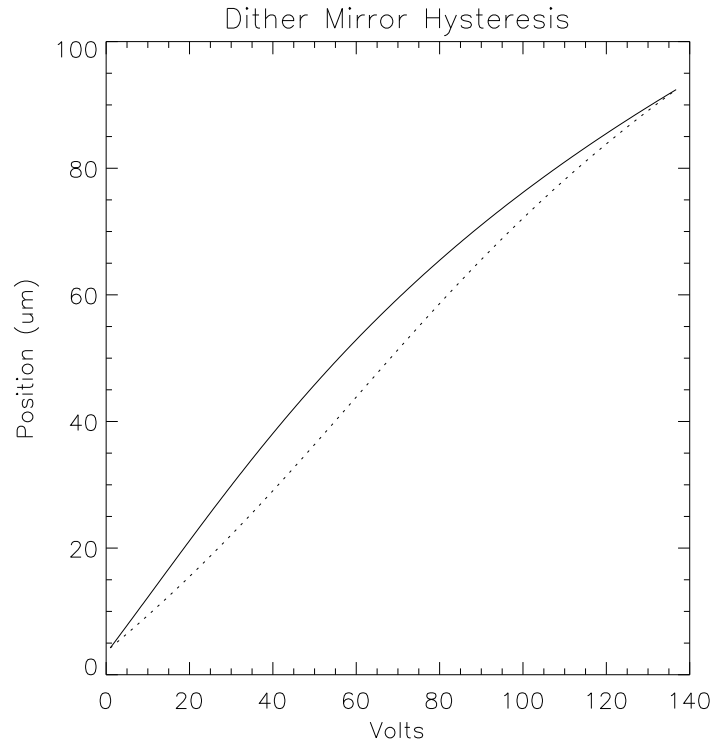


Figure 3.3: The VIS dither mirror hysteresis curve. The dotted line is the mirror position as the voltage increases from 0. As the voltage increases, the PZT expands, moving the dither mirror toward the combining splitter, shortening the path of beam 5. The solid line is the return trip. As the voltage decreases, the PZT contracts, moving the dither mirror away from the combining splitter, adding path to beam 5.

corresponding voltage. An RT-Linux module written by the author speaks to the dither driver, giving it position commands. The RT dither module sends the dither mirror to a target position for each CCD exposure, so that each frame of data from the camera corresponds to a certain OPD position. The module ensures that the dither mirror moves smoothly from one target position to the next, which prevents over-stressing the PZT crystal.

The PZT has significant hysteresis that was measured by Neda Safzideh in 2002

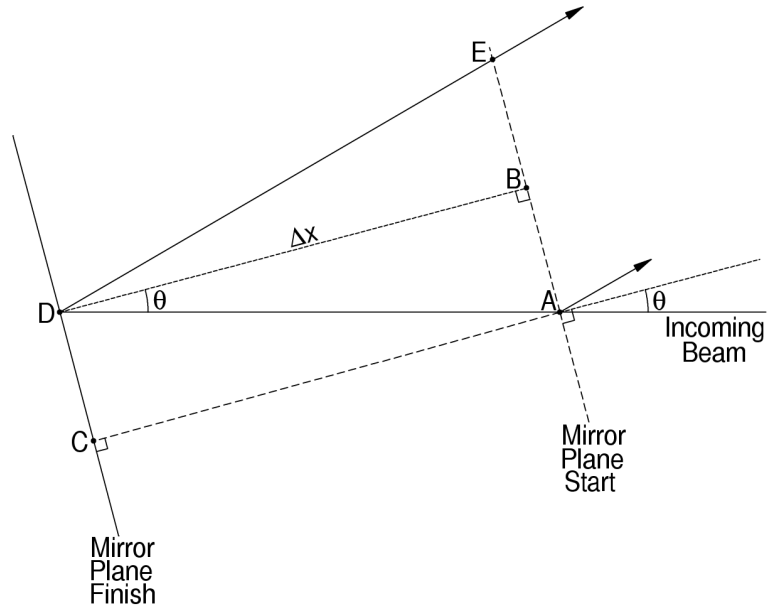


Figure 3.4: The geometry of the OPD introduced by moving the dither mirror. The incoming beam, at the lower right, reflects off of the dither mirror at point **A**. The incoming beam strikes it at angle θ from the surface normal (short-dashed line), and is reflected at angle θ on the other side of the surface normal as it heads toward the beam combining splitter. If the dither mirror moves Δx away from the combiner, the starlight beam reflects off of the mirror surface at **D** and travels back to point **E**, which is the same distance from the combining splitter as **A**. Line segments \overline{AD} and \overline{DE} each equal $\frac{\Delta x}{\cos \theta}$, so the starlight beam travels an extra $\frac{2\Delta x}{\cos \theta}$ before it reaches the beam combining splitter.

for calibration of the CHARA Classic system's dither mirror, which uses the same model PZT as the VIS system. The PZT hysteresis curve is shown in Figure 3.3. The hysteresis curve was used to generate a lookup table, which the RT dither module uses to generate a smooth, linear dither sweep.

The PZT has a hysteresis-calibrated movement range of $88.18 \mu\text{m}$. To find how this translates to OPD one must consider the geometry of the beam combiner, represented in Figure 3.4.

The central ray of the incoming starlight beam enters the figure at the lower right corner. The dither mirror is represented by the line labeled “Mirror Plane Start” in the figure, so the beam reflects off of the surface of the mirror at point **A**. The beam is at angle θ from the surface normal (a line perpendicular to the surface plane) of the mirror, so it reflects at the same angle on the other side of the surface normal. The reflected ray leaves the figure on its way to the beam combining splitter, whose surface is parallel to that of the dither mirror.

Consider the case where the dither mirror moves away from the beam combining splitter by the distance Δx . The starlight ray now travels to point **D** before it reaches the surface of the dither mirror, represented by the line labeled “Mirror Plane Finish.” Note that the ray strikes a different part of the mirror surface, as point **A** on the mirror surface moves to point **C**. The starlight beam moves across the surface of the dither mirror as it moves backward, so any perturbations made to the wavefront by variations in the mirror surface will translate sideways across the beam as the dither mirror moves.

The ray reflects off of the dither mirror at the same angles as it did previously, and when the ray reaches point **E**, it is the same distance away from the beam combining splitter as point **A**. The total path from the star to the beam combining splitter is increased by the length of line segments \overline{AD} and \overline{DE} .

Let point **B** be at the midpoint of line segment \overline{AE} . The right triangles $\triangle ADB$ and $\triangle DEB$ may be used to find the lengths of all desired line segments in terms of Δx . Using the definitions of cosine and tangent:

$$\overline{AD} = \overline{DE} = \frac{\Delta x}{\cos \theta}$$

and

$$\overline{\mathbf{AB}} = \overline{\mathbf{BE}} = \Delta x \tan \theta. \quad (3.1)$$

The extra optical path ΔOPD introduced by the dither mirror moving back Δx is

$$\Delta\text{OPD} = \frac{2\Delta x}{\cos \theta}. \quad (3.2)$$

In addition, the beam is offset by the length of $\overline{\mathbf{AE}}$. This offsetting of the beam without changing its angle is referred to as beam shear. The shear s , from (3.1) is

$$s = 2\Delta x \tan \theta. \quad (3.3)$$

The CHARA VIS combiner is set up such that the angle between the incoming and reflected beams is 30° , so θ equals 15° . At this angle, $\frac{1}{\cos \theta}$ is approximately 1.035, so ΔOPD is only 3.5% larger than $2\Delta x$, the change in OPD induced if the incoming ray were hitting the dither mirror straight on ($\theta = 0$).

$\tan \theta$ is approximately 0.2679, so the shear is about 53.6% of Δx . Because the dither range is $88.18 \mu\text{m}$, the beam shear between the extreme ends of the dither sweep is only $47.26 \mu\text{m}$. Compared to the size of the beam, the maximum shear is so small as to be negligible. We don't need to worry about visibility losses due to changes in the overlap of the two input beams at the combining splitter, because the induced shear is *much* smaller than beam shear due to alignment errors. The beam alignment scheme relies on centering the beam on a target, with errors on the order of 0.5 mm .

Equation 3.2 means that the VIS dither mirror has an OPD range of $182.58 \mu\text{m}$ on

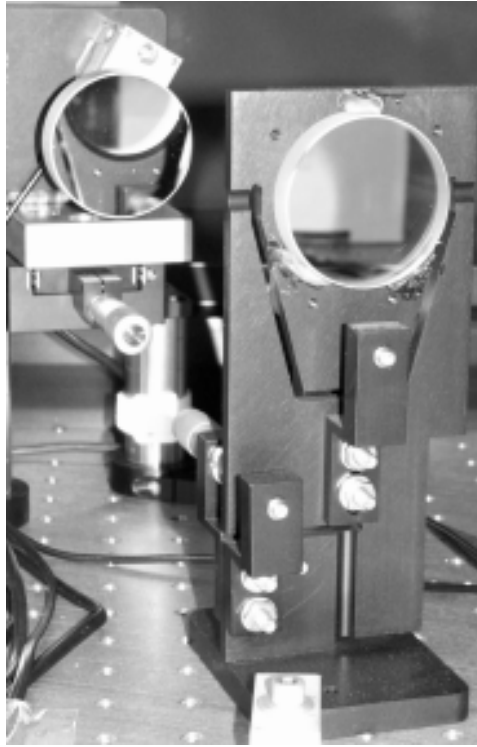


Figure 3.5: The VIS beam combining splitter. The VIS dither mirror is visible behind and to the left.

the sky. If the VIS system is used to measure fringes from the lab white light source, there are an additional two passes through Δx as the beam enters the system from the movable linear stage steering mirror. As a result, the VIS dither mirror has an OPD range for internal fringes that is $365.16 \mu\text{m}$. It is an easy number to remember, as it is almost exactly the number of days in a year!

3.3 Beam Combining Splitter

After beam 5 reflects off of the dither mirror, it meets beam 6 at the north surface of the beam combining splitter. The splitter is a flat with a $\frac{\lambda}{20}$ surface quality. The

combining surface is a multilayer coating that gives a 50/50 split across the entire VIS system wavelength range.

The splitter sits in a mount with tip and tilt axes driven by computer-controlled picomotors. The beam 5 alignment beam reflects off of the splitter as it enters the system, so the splitter angle must be adjusted as part of the nightly alignment routine.

3.4 Fiber Injectors

The beam from the BRTs is broken into sub-apertures by an array of lenses, each of which focuses onto a multi-mode optical fiber. The lens array and fibers are all mounted into a $\frac{15}{16}$ inch diameter by $1\frac{5}{8}$ inch long cylindrical chuck, the whole of which is referred to as the ‘fiber injector.’ The fiber injectors were custom ordered from Fiberguide Industries, whose engineers took the author’s simple design requirements and developed them into the fiber injector design as it stands.

The lenses are f/5.6 Edmund Industrial Optics achromats, custom made with a 6.25 mm diameter and a 35 mm focal length. They are held in the lens array with their edges touching, so that the diameter of the array is approximately 19 mm, about the same size as the incoming starlight beam under good seeing conditions.

The optical fiber is Fiberguide Industries’ Anhydroguide G fiber, with a numerical aperture (abbreviated NA) of 0.12. The numerical aperture is given by the equation (Fowles, 1989, Section 2.7):

$$\text{NA} = \sin \theta = \sqrt{n_{\text{core}}^2 - n_{\text{clad}}^2} \quad (3.4)$$

where θ is the maximum angle a ray may have with the fiber axis before entering the

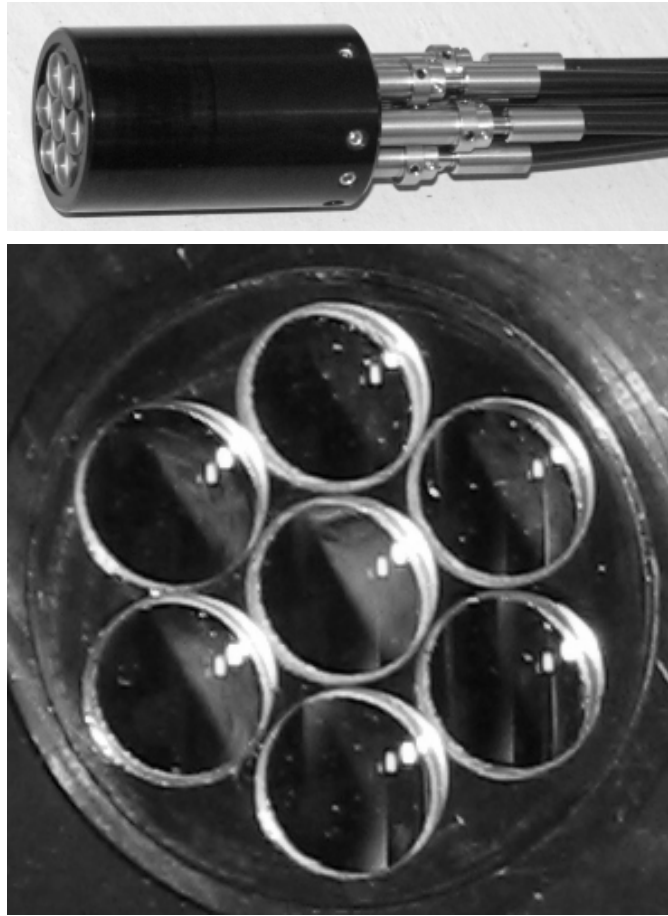


Figure 3.6: The fiber injector, side (above) and face-on (below) views. The lenses are 35 mm focal length Edmund Industrial Optics custom Achromats, designed for the 600-1000 nm wavelength range. Each lens is 6.25 mm in diameter, with the lens array about 19 mm in diameter, the same size as the (ideal) incoming starlight beam. Each lens acts as a separate sub-aperture of the beam, injecting its light into a unique multi-mode optical fiber. The optical fibers were co-aligned by Fiberguide Industries via a series of small discs into which each fiber is mounted off-center. The fibers are 1 m in length with FC connectors at the ends.

core. Any ray with an angle larger than θ will not have total internal reflection when reaching the boundary between the core and cladding inside the fiber. The core's index of refraction is $n_{\text{core}} = 1.457$ and the cladding index is $n_{\text{clad}} = 1.452$. An NA of

0.12 means that the fiber will accept light striking the core within 6.89° of the fiber's optical axis, or $f/4.17$.

The fiber is designed for transmitting in the visible to near-infrared wavelength regime, with a transmission of better than 99.83% per meter over the entire 600-1000 nm wave band, and better than 99.99% for wavelengths longer than 700 nm. The fiber core is 50 μm in diameter, so they are multi-mode fibers, meaning that there are multiple solutions for the angle at which light may propagate in the fiber. They are not usable for fiber-based beam combination, but rather are intended for use as a conduit for the post-combination beams to get to the detector in a more manageable way than bulk optics can provide. The fiber type was selected to match those in the fiber slit, described in Section 3.5.1.

The fibers are 1 m in length, just enough for the fibers to run upward from each injector, out through a hole in the dust cover of the VIS table, to the fiber switchboard that rests on top of the dust cover. The fiber ends are fitted with FC type connectors, which provide the most precise fiber tip positioning among the various types of fiber connectors available.

The fiber injector design has proven to be a good one overall. Its Achilles heel is the method of co-aligning all of the fibers. The fibers are each mounted off-center in a series of small concentric discs. By rotating the discs, one may position the fiber anywhere within in a small circle of the lens' focal plane. Fiberguide Industries handled the co-alignment of all of the fiber-lens pairs. They reported that the nested disk mechanism was somewhat troublesome to use, because the fiber tips moved slightly as the disk set screws were tightened. As a result of the alignment difficulty, the lens-fiber pairs are not perfectly co-aligned, but the author has found that there

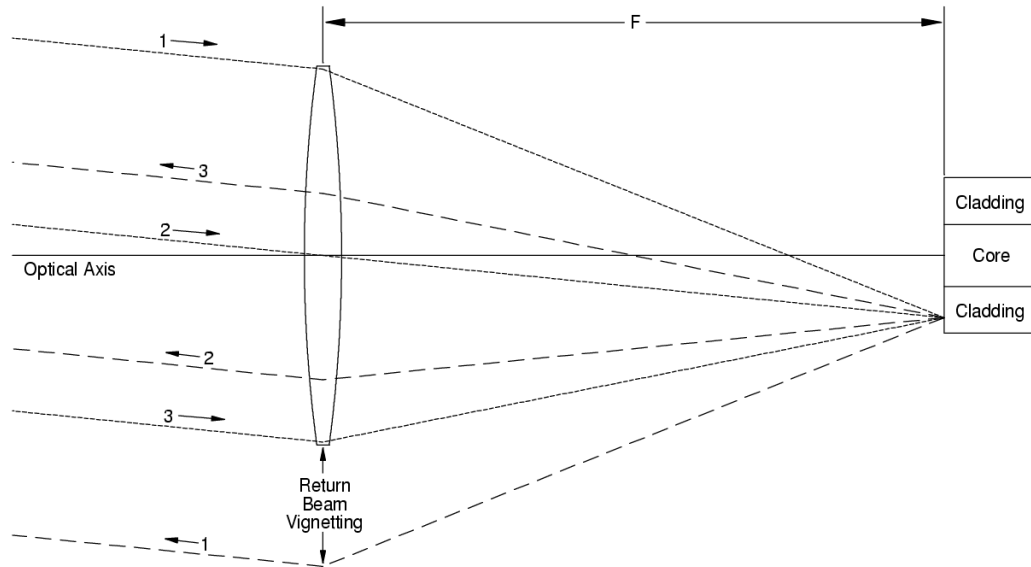


Figure 3.7: Fiber injector alignment. If the collimated incoming beam (short-dashed lines) is not parallel to the optical axis, it will focus to a spot off-center on the fiber tip. If the spot is not on the fiber core, most of the light will reflect back to the lens (long-dashed lines), where it will be re-collimated. The reflected beam leaves the lens parallel to the incoming beam, where it may be imaged onto a screen along with a beam projected out of the fiber. The projected beam defines the optical axis of the system, so overlapping the images of the reflected and projected beams aligns the injector to the reflected beam.

is a small angle inside of which all of the fibers do inject simultaneously.

3.4.1 Fiber Injector Alignment

The fiber injectors must be aligned to the output beams from the combiner. The beams themselves may not move, as their position is determined by the location of the beam splitters and the dither mirrors. The simplest way to align the injectors to the beams is to simply move the injectors themselves to the correct place. The injectors are clamped into a Newport U100-G gimbal mount, whose center of rotation

is at the center lens of the injector. This arrangement makes it possible to change the angle of the injector without changing its position with respect to the beam. The gimbal mount rests on a Newport 460A-XZ stage, so that the entire assembly may be translated in a plane perpendicular to the beam. This allows for the injector to be positioned such that the beam is centered on the lens array.

Consider a single lens-fiber pair in the fiber injector. If a collimated beam passes through the lens, it focuses to a spot on the fiber tip. If that spot is on the fiber core, the beam will inject into the fiber and can be seen coming out the other end. If the spot is not on the fiber core, it will not be injected, and most of the light will reflect off of the fiber tip. Because the fiber tip is at the focal length of the lens, the reflected beam will be re-collimated as it passes through the lens.

The reflected beam will return at the same angle it entered, but offset. The amount of the offset may be found by examining ray number 2 in Figure 3.7. Any ray that reflects off of a surface does so such that the incoming and reflected rays are symmetric about the surface normal (a line perpendicular to the surface at the point of ray reflection). Assuming that the fiber tip surface is perpendicular to the optical axis, any ray reflecting off of it will have the same slope as the incoming ray, but heading in the opposite direction. The distance a ray deviates from the optical axis when it hits the fiber tip is doubled when it returns to the lens. If ray number 2 hits the fiber tip at a distance Δy from the optical axis, it returns to the lens $2\Delta y$ from where it started. Upon examination of rays 1 and 3, which represent the peripheral rays of the incoming beam, one may see that the return beam is inverted, and the beam is vignetted by $2\Delta y$ at the ray 1 edge.

If the spot leaves the outer edge of the fiber cladding, the beam no longer reflects,

so the maximum offset of the return beam is twice the cladding outer radius. The fiber cladding is $125\ \mu\text{m}$ in diameter, also the largest offset in the reflected beam before it is lost entirely. $125\ \mu\text{m}$ is such a small amount compared to the diameter of the 6 mm lens that beam vignetting may be ignored.

The injector lenses are designed to have a spot under $30\ \mu\text{m}$ in diameter across the 600-1000 nm wave band. Because the fiber core is $50\ \mu\text{m}$ in diameter, that leaves about $20\ \mu\text{m}$ worth of room for alignment error while still injecting the entire spot into the core. With a lens focal length of 35 mm, the beam will still inject into the core as long as it is at an angle less than 1.07 arc minutes from the optical axis. This means that the fiber injector has a total acceptance cone of 2.14 arc minutes for injection.

If one assumes that the full spot must be on the cladding for the reflected beam to be seen adequately, the beam is required to be at an angle within 4.7 arc minutes of the optical axis. The fiber injector therefore has an acceptance cone of 9.3 arc minutes for beam reflection from the cladding. On the sky, tip-tilt errors must be under 4.7 arc minutes as well for the starlight to be injected into the fibers.

The alignment of the fiber injector is accomplished by using two laser beams simultaneously. The standard alignment beam shines into the lens array, reflecting back out if it is not parallel to the injector's optical axis. Before the alignment beam enters the VIS combiner, it passes through a cube splitter that splits off a secondary beam. The secondary beam is injected into a multimode fiber, identical to those used in the fiber injector. The fiber is patched to the center fiber of the injector, so the laser is projected out of the fiber and collimated by the center lens. If one can assume that the center fiber has been adequately aligned to its lens, the projected

beam comes out centered on and parallel to the injector's optical axis.

If the fiber injector is not aligned, the reflected beam will exit the injector at a different angle than the projected beam. A cube splitter is placed in front of the fiber injector; the reflected and projected beams each send part of their light out of the side of the splitter, where the beams may be viewed on a screen without obstructing the alignment beam. The reflected beam covers all 7 lenses in the injector, so it forms 7 spots in the same pattern as the lens array. It is easily distinguishable from the projected beam, which only comes out the center lens, forming 1 spot.

By changing the angle of the injector, the reflected and projected beams may be moved until they are superimposed. As the beams overlap, the reflected beams drop dramatically in intensity. This is because the spot from each lens has reached the fiber core, and the beams are being successfully injected into the fiber.

Once the input beam is injecting into the fibers, the spectrograph may be used to fine-tune the injector alignment. The relative brightness of each sub-aperture's spectrum shows how well the beam is injecting. As one changes the injector angle, the spectral intensities change accordingly.

In a perfect world, all of the spectra would reach their maximum brightness at exactly the same injector angle. However, there are small errors in the alignment of each lens-fiber pair with respect to the others in the injector array. The best that one can do is to try to maximize the brightness of as many spectra as possible. In practice, the injector can be aligned to the internal light sources such that all of the sub-apertures have maximized spectral intensities. On the sky it has proven more difficult to align the injectors, and to date 4 spectra out of the 6 outer sub-apertures are the most which have been aligned at once. It may be necessary to open up one

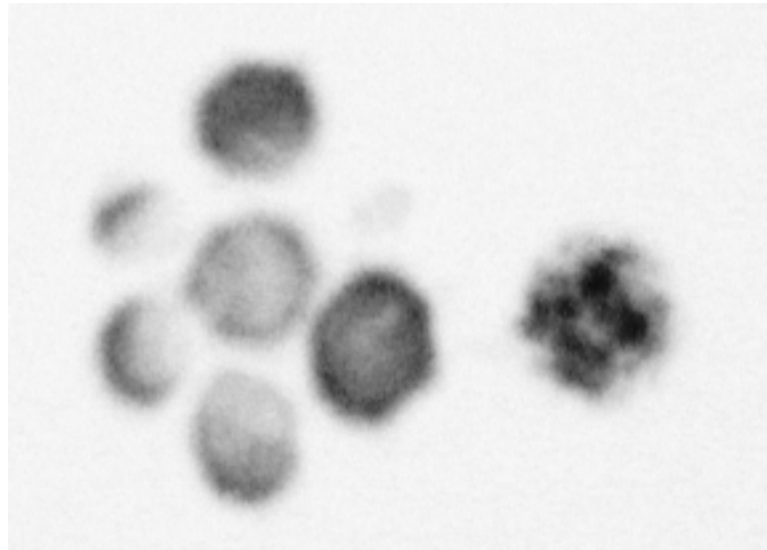


Figure 3.8: The reflected and projected beams from the fiber injector. This is a photograph of the two beams as seen on a screen at the output of the injector alignment cube splitter. The image values have been reversed for clarity in printing. The reflected beam (left) shows spots for nearly all of the lenses in the injector array. The projected beam (right) from the center lens of the array defines the injector's optical axis. By adjusting the injector angle until the two beams are superimposed, the fiber injector is brought into rough alignment. The missing image in the reflected beam could be due to the light beginning to inject into the fiber, or the focused beam could be at the edge of the fiber tip. If the latter case is true, that fiber-lens pair would be grossly out of alignment with the rest of the injector array.

of the fiber injectors and attempt to re-align it to improve the co-alignment of the fiber-lens pairs. Individual tip-tilt control for each sub-aperture would also fix this problem.

3.4.2 Fiber Switchboard

The fiber switchboard is where fibers from each injector may be coupled to fibers in the spectrograph slit. The switchboard is simply a long metal plate with 7 FC-to-FC

female connectors in a row, enough for all of the fibers in a given injector. FC-type connectors were chosen as they have the most rigid positioning of the fiber tip among the available connector types. The switchboard rests on top of the VIS table dust cover, directly above the fiber injector.

In retrospect, it would have been simpler to have all of the fiber injectors made with 5 m cables. This would have allowed enough room to make one master switchboard for all of the fiber injectors, which could have been mounted somewhere above the VIS table dust cover or below the VIS table itself.

3.5 Spectrograph

The CHARA visible to near-infrared spectrograph is a simple slit, collimator lens, prism, camera lens, and detector. The slit is composed of several optical fibers in a vertical row. Each fiber corresponds to a unique sub-aperture from one of the fiber injectors, so that imaging the slit is a way to process the light from each injector sub-aperture separately, but in parallel!

The prism provides a way to separate the light from each sub-aperture into various wavelength regimes. Once the light exits the prism, it is brought to a focus on the detector by a camera lens. Each fiber in the slit forms a short (about 4-5 pixel) spectrum on the detector. The vertical stack of fibers in the slit becomes a stack of spectra on the detector.

Each component of the spectrograph is described in detail in its following subsection.

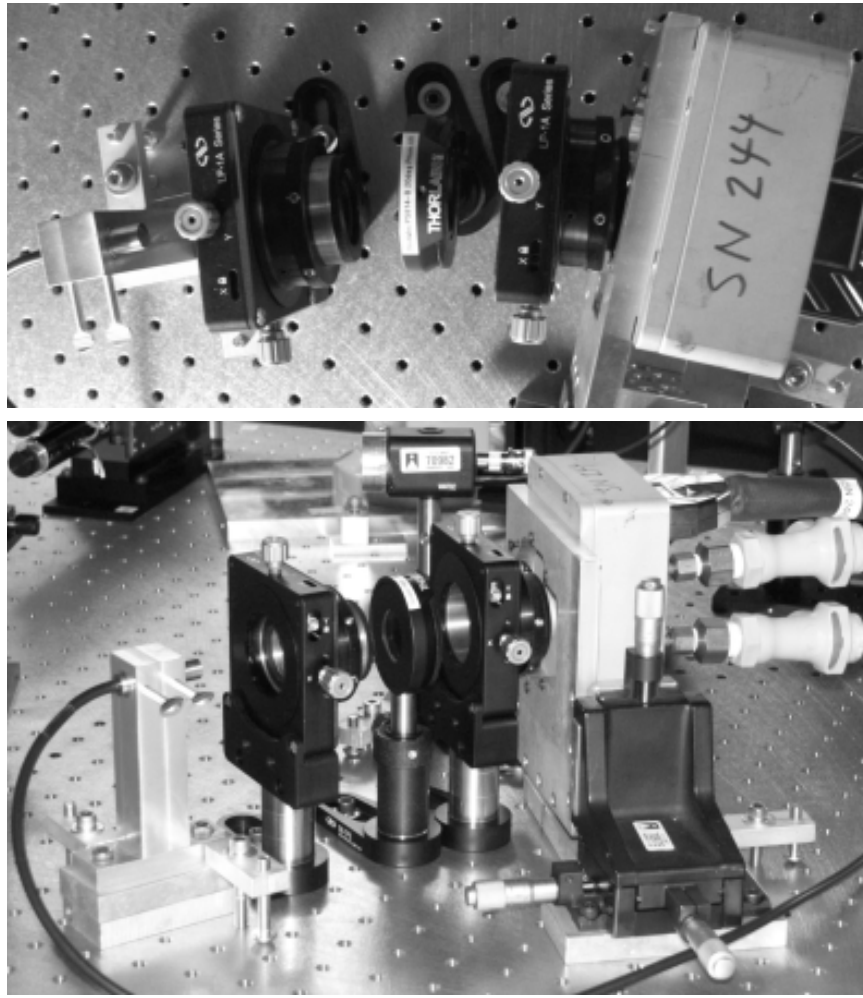


Figure 3.9: The spectrograph, top and side view. Light exits the fiber slit (silver mount, far left) in an $f/4.14$ cone. The collimator lens (black lens mount, left), bends the light into a 18 mm beam. The beam passes through the prism (circular mount, center). The prism pictured here has an 18° angle. The camera lens (black lens mount, right) focuses the spectrally dispersed beam onto the CCD detector (silver box, far right).

3.5.1 Fiber Slit

The fiber slit is a convenient way to put the light from many inputs into a small space that can be easily imaged onto a detector for parallel detection and processing. Each

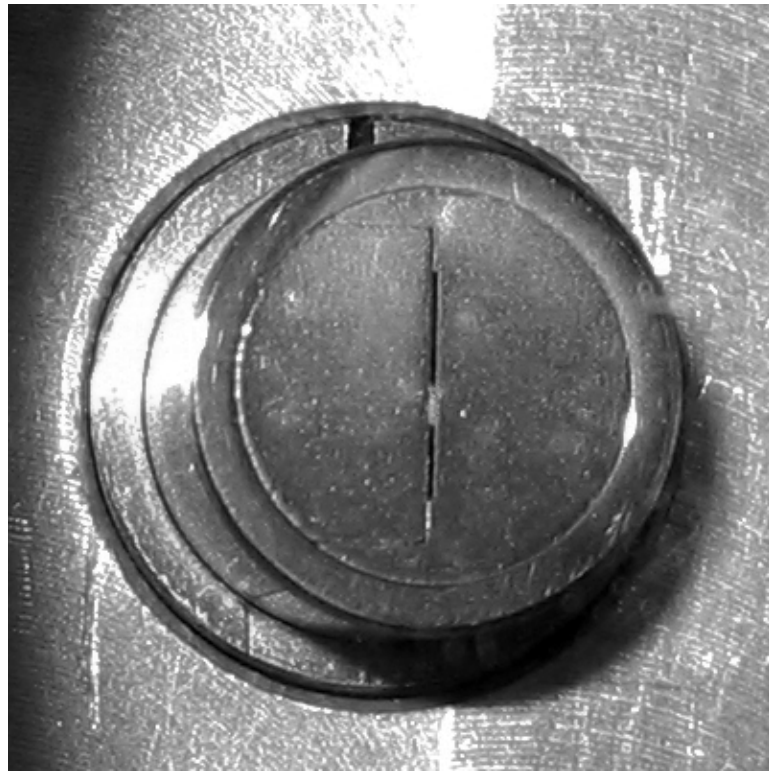


Figure 3.10: The fiber slit viewed face-on. 36 multimode fibers in a linear array form the slit. Their cores are $50\ \mu\text{m}$ and their cladding is $125\ \mu\text{m}$ in diameter. The fibers are stripped down to their cladding and placed side by side in V-shaped grooves etched into a silicon chip. The fibers are held with a spacing of $125\ \mu\text{m}$ center-to-center. Six fibers may be seen emitting light at the center of the slit.

fiber is coupled to a different fiber injector sub-aperture and pipes that sub-aperture's light independently to the slit.

The slit itself is a linear array of optical fibers, each laid in a V-shaped groove etched into a silicon chip. The grooves are $125\ \mu\text{m}$ from the bottom of one V to the next, and the fiber cladding is also $125\ \mu\text{m}$ in diameter, so adjacent fibers barely touch. There are 36 fibers, so the total length of the slit is 4.5 mm from topmost to bottom-most fiber cladding edge. The fibers are held in their respective grooves by a

flat plate, held against the silicon chip.

The fibers are Fiberguide Industries' Anhydroguide G fiber, optimized for transmission in the visible to near infrared. The fibers are multimode, 0.12 numerical aperture, with 50 μm diameter cores, suitable as a wave guide for starlight beams only after they have been combined. The fibers are 10 m long, with FC connectors at the ends.

The reason for such long fibers is so that the VIS spectrograph could be placed on any table in the beam combining lab. At the time that the VIS spectrograph was designed, it was not known if there would be enough room on the VIS table for the spectrograph, and it was thought that it may need to go on the table with the alignment laser and white light source. Because the fiber transmission is greater than 99.83% per meter across the entire 600-1000 nm band, 98% of the light is transmitted to the fiber tips at the slit. For wavelengths longer than 700 nm, greater than 99.7% of the light is transmitted to the slit.

The major light loss through the fiber portion of the system comes from the fiber surfaces. Each time the light crosses the boundary from air to fiber or vice-versa, about 3.45% of the light is reflected. This is based on the assumption that the light strikes the fiber at an angle small enough to be considered normal to the fiber tip surface. At normal incidence, the fraction of the energy that is reflected, known as the *reflectance* R , is given by (Born and Wolf, 1998, Section 1.5.3, equation 37):

$$R = \left(\frac{n_{\text{rel}} - 1}{n_{\text{rel}} + 1} \right)^2 \quad (3.5)$$

where n_{rel} is the relative index of refraction, the ratio between the refractive indices

of the fiber core and air. The index of refraction of air at 750 nm, 1 atmosphere of pressure, and 15° C is 1.000275. The fiber core's index of refraction is 1.457, so n_{rel} is about 1.4566. When the light exits the fiber, R turns out to be exactly the same as when the light enters.

The fraction of the energy that is transmitted at each boundary, the *transmittance* T , is simply $1-R$. In order to figure out what fraction of the light makes it all the way from the injector out of the slit, consider that the light must be transmitted through 4 fiber-air interfaces. The transmittance of this light is $(1 - R)^4$, about 86.88%. In addition, some of the light that was reflected from one of the intermediate surfaces may have reflected a second time and been transmitted out of the slit. The fraction of light that encountered 2 additional reflections is $(1 - R)^4 R^2$, about 0.31%. There are more terms, accounting for 4 or more reflections, but their contribution is negligible. In total, about 87% of the light that strikes the fiber tip at the injector makes it out of the fiber tip at the slit. With the additional absorption inside the fiber at 600 nm, the transmitted energy drops to about 82%, but remains at 87% above 700 nm.

The fiber slit is mounted in a $\frac{3}{8}$ inch diameter by $1\frac{9}{16}$ inch long cylindrical chuck, which can be held by a simple clamp-type mount. The spectrograph is built with the slit held in a rigid position, defining the optical axis of the instrument. The collimator, prism, camera, and detector are all on adjustable mounts, and are each moved into position one at a time as the spectrograph is aligned.

Fiberguide Industries designed and built the fiber slit based upon basic design parameters given them by the author. The slit has proven easy to use and free of any major design flaws, although the fibers are housed in a thin jacketing material, and must be handled with care.

3.5.2 Collimator and Camera Lenses

The collimator is a Thorlabs near-infrared stock achromat, model number AC254-075-B, 25.4 mm in diameter, with a 75 mm focal length. The collimator is guaranteed to have a clear aperture greater than 90% of the diameter, at least 22.86 mm.

The fibers in the slit have a 0.12 numerical aperture, so they project their light in a cone 6.89° from each fiber's axis. A cone of that angle is about $\frac{f}{4.14}$, or its length is about 4.14 times its base diameter. In order to contain the light cone from a given fiber, the collimator must be $\frac{f}{4}$ or shorter. This proves to be an important limitation when coupled with the magnification required, a discussion of which follows.

The collimator is $\frac{f}{3.28}$, which allows for the fact that the cones from the top and bottom fibers of the slit are offset by $4.375 \mu\text{m}$. The diameter of each fiber's light cone is 18.13 mm when it reaches the collimator, so with the offset between the top and bottom light cones added in, the vertical aperture size required is 22.5 mm, only 0.36 mm less than the guaranteed clear aperture diameter of the collimator.

The collimator is housed in a 3-axis Newport LP-1A lens mount. It is aligned to the fiber slit by injecting a beam from the white light source or alignment laser into the fiber injector, and patching the fibers from various sub-apertures of the injector to fibers at the top, bottom, and center of the slit. Laser light may be used to roughly position the collimator. The laser beam may be easily seen on a white piece of paper, and used to ensure that the beams from the fiber tips are not vignetted at the collimator aperture.

Once the collimator is aligned with the slit's optical axis, the beam collimation is checked by putting a small telescope behind the collimator. The telescope is already

set to focus at infinity, so a collimated beam passing into it will appear as a tightly focused spot in the telescope. The collimator is moved by small amounts along the optical axis until the spot in the telescope reaches its smallest focus. The white light source is best used for checking the beam collimation, as the laser's wavelength of 632 nm is on the extreme short end of the system's band width.

After the beams leave the collimator, they are not all parallel to the optical axis. Figure 3.11 shows the geometry of the situation. The slit length s is 4.375 mm from the center of the top fiber to the center of the bottom fiber. The bottom-most fiber is $\frac{s}{2}$ or 2.1875 mm below the optical axis. The light from the bottom-most fiber forms a beam at the collimator which is centered about a point that is also $\frac{s}{2}$ below the optical axis.

The angle of the beam as it leaves the collimator is simple to find: one need only remember that any ray passing through the center of the lens exits at the same angle that it entered (assuming the lens is very thin). By selecting a ray that starts at the bottom fiber tip and extends through the center of the collimator, we see that the ray forms an angle ϕ with the optical axis. All rays in the beam must be parallel to this ray if they are collimated. By inspection one can see that

$$\tan \phi = \frac{\left(\frac{s}{2}\right)}{f_{\text{coll}}} , \quad (3.6)$$

which gives a value for ϕ of 1.67° .

The beam starts out at the collimator centered on a point $\frac{s}{2}$ below the optical axis, but angles upward with a slope of $\frac{s}{2f_{\text{coll}}}$ (from Equation 3.6). If L is the separation between the collimator and camera, then the offset Δy of the beam when it reaches

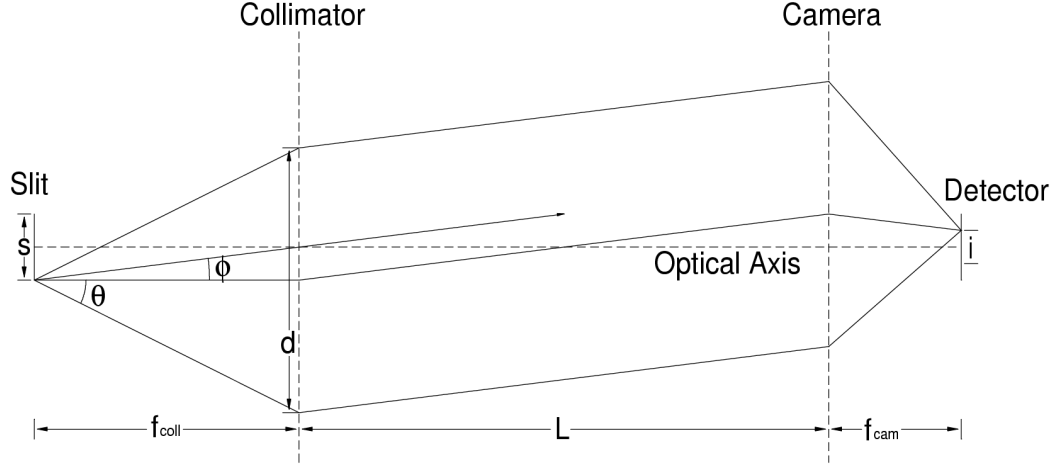


Figure 3.11: Vertical spreading of collimated beams due to slit length, side view. The slit fibers emit light in a cone of angle $\theta=6.89^\circ$ from the fiber axis. When each cone reaches the collimator lens with $f_{\text{coll}}=75$ mm, it has a diameter of $d=18.13$ mm. Each collimated beam maintains that diameter until it reaches the camera lens. The light cone from the bottom-most fiber in the slit is offset below the optical axis by $\frac{s}{2} = 2.1875$ mm. The ray from the bottom-most fiber tip through the center of the lens shows the angle $\phi=1.67^\circ$, which the collimated beam has with respect to the optical axis. The beam starts out $\frac{s}{2}$ below the optical axis at the collimator, but is angled upward. The offset of the beam from the optical axis when it reaches the camera is $\frac{s}{2}(L/f_{\text{coll}} - 1)$, where L is the separation between the camera and the collimator. If L is less than $2f_{\text{coll}}$, the beam will be less than $\frac{s}{2}$ above the optical axis when it reaches the camera.

the camera is

$$\Delta y = \frac{s}{2} \left(\frac{L}{f_{\text{coll}}} - 1 \right) . \quad (3.7)$$

The beam from the topmost fiber in the slit is symmetric with the bottom beam about the optical axis, so the total height of all beams together is $d + 2\Delta y$. If L is less than $2f_{\text{coll}}$, then Δy at the camera is less than $\frac{s}{2}$, and the camera has no worse vignetting than the collimator. The actual distance between the collimator and camera is about 150 mm, so the total height of all beams is the same as at the

collimator.

The camera lens is a Thorlabs near-infrared stock achromat, model number AC254-030-B, 25.4 mm in diameter, with a 30 mm focal length. The camera has the same guaranteed clear aperture as the collimator, 22.86 mm. The camera lens is aligned to the collimated laser beam, which is checked with a white card to ensure that there is no vignetting at the lens aperture. The reflected beam from the first surface of the camera lens may be centered on the incoming beam to ensure that the lens is perpendicular to the beam.

The camera lens is required to have a very short focal length because of the magnification required at the detector. The ARC CCD pixels are $24\text{ }\mu\text{m}$ square, so in order to image the slit fiber tips on every other row, the magnification must be the width of two pixels divided by the slit fiber spacing: $\frac{48\mu\text{m}}{125\mu\text{m}}$, or 0.384. The collimator focal length cannot be any longer than 75 mm because it must be able to catch all of the light from the fibers, which comes out in an $\frac{f}{4}$ cone. The magnification of the spectrograph is $\frac{f_{\text{cam}}}{f_{\text{coll}}}$, and because the collimator focal length is locked in, the camera focal length is forced to be about 28.8 mm. The actual 30 mm focal length of the camera gives a magnification of 0.4. The spectrograph has only been used to image six simultaneous spectra to date, but they did fall neatly onto alternating pixel rows, despite the slight deviation from the ideal magnification.

The fiber tips are $50\text{ }\mu\text{m}$ in diameter, and the light that exits them may be approximated by a Gaussian distribution with its full width at half the maximum value (abbreviated FWHM) at the fiber core edge. The image of the fiber tip is then a Gaussian with a FWHM of $20\text{ }\mu\text{m}$. On the detector image, one may see some light leakage from the spectra into the buffer rows of pixels in between them, due to the

tails of the Gaussian distribution, as well as some aberrations from the camera lens.

There are ways to reconfigure the spectrograph in the event that more than 6 spectra do not image on the detector with the right spacing.

- Custom optics could be ordered whose focal ratios are exactly 0.384.
- If aberrations due to the short camera focal length prove problematic, a custom camera lens with a 57.6 mm focal length could be ordered. This would create an image twice as large as the current design, spreading each spectrum across two rows of pixels, and skipping two rows before the next spectrum. If the detector were used in a mode that bins pixels 2x2, there would be no additional read noise, and the only loss would be that a maximum of 20 spectra could be imaged simultaneously.
- A positive field lens could be placed just after the fiber slit to make the fiber light cones subtend a narrower angle. A longer focal length collimator could then be used, which in turn would allow for a longer focal length camera lens, as long as the ratio of $\frac{f_{\text{cam}}}{f_{\text{coll}}}$ remains 0.384.

None of these options were tried because the system worked well enough for six spectra in the current configuration, corresponding to the six outer apertures.

3.5.3 Prism

The prism used in the spectrograph is a Thorlabs PS814-B beam steering prism, made of BK7 glass and coated with Thorlabs' B-type 600-1000 nm anti-reflective coating. The prism is a wedged circular window, 25.4 mm in diameter, with a wedge angle

of 18.15° . It disperses the 600-1000 nm wave band across an angle of 0.18° . With a camera focal length of 30 mm, the spectrum spreads across 5 pixels on the detector. A prism was used because it provides the required spectral resolution for low cost. A grating could just as easily be used, provided it can match the throughput of the prism.

The prism is held in a Thorlabs RSP1 rotation mount, which enables rotation of the prism about its optical axis. Prism rotation is used to change the tilt of the spectra so they line up parallel to the pixel rows on the detector. The mount is placed midway between the collimator and camera lenses. The dispersion induced by the lens is small enough that the collimated beams leaving the prism do not spread enough to induce any additional vignetting problems at the camera lens aperture.

The prism is necessary because if all of the light were to fall on a single pixel, its bandwidth would be so broad that the width of the fringe packet would be unusably small. From the definition of $\kappa \equiv \frac{1}{\lambda}$:

$$\Delta\kappa = \frac{1}{\lambda_{\text{short}}} - \frac{1}{\lambda_{\text{long}}},$$

so if λ_{short} is $0.600 \mu\text{m}$ and λ_{long} is $1.00 \mu\text{m}$, then $\Delta\kappa$ is $0.667 \frac{1}{\mu\text{m}}$. The fringe packet width is twice the coherence length, which in turn is $\frac{1}{\Delta\kappa}$, so width of the fringe packet is only $3.00 \mu\text{m}$.

If the whole band pass is assumed to be at uniform intensity (which it definitely is not, but it's close enough for the sake of this approximation), the fringes have a wavenumber κ_o of $1.333 \frac{1}{\mu\text{m}}$, or a wavelength of 750 nm. The fringe packet would be barely wide enough to contain 4 fringes, which is too narrow to be easily discerned in

low light conditions. By spreading the starlight across 5 pixels referred to as ‘spectral channels’ the fringe packets in each spectral channel are about 5 times wider, which is enough to make them usable.

Another reason for spectrally dispersing the starlight is longitudinal dispersion. The OPLE system propagates the beams through air in order to equalize the OPD between any two telescopes. However, the path from the star to each telescope went through equal amounts of air, so some amount of delay in vacuum is compensated by the OPLE delay in air. Because the index of refraction changes with wavelength, the zero OPD position will also depend upon wavelength. If the dispersion is too great, the fringes from the red end of the wave band are offset enough from the fringes at the blue end of the wave band as to blur the fringes and reduce their visibility.

David Berger developed the Longitudinal Dispersion Compensators to mitigate this effect by introducing an variable thicknesses of glass into each beam. By using a glass whose index of refraction vs. wavelength curve has a similar shape to that of air, one may compensate a certain amount of air dispersion with a certain amount of glass. The compensation is not perfect, but the residual dispersion is much smaller than that with no LDC.

The VIS system was intended to be used with the LDCs, but Berger’s model predicted that for the S1-S2 baseline, the longitudinal dispersion was not enough to cause significant loss of visibility. As a result, the VIS system was used without the LDCs to find first fringes.

3.5.4 ARC CCD

Astronomical Research Cameras (ARC) built the detector hardware. Two camera systems were purchased, with the second camera currently in use as the detector for the CHARA CCD tip-tilt system.

The ARC CCD camera system is composed of several parts:

- The camera head. A $5\frac{1}{4}$ inch tall, $3\frac{13}{16}$ inch wide, $1\frac{11}{16}$ inch deep housing contains the detector and readout electronics. The CCD chip is cooled by an electric Peltier cooling system that keeps the chip at -20.5° C by exchanging heat with a water-based cooling system, described below. The camera housing is attached to the control electronics (also described below) via a 2 meter long data and power cable. The camera housing is also attached to the water cooling system via two insulated hoses that keep water flowing through the camera head. The camera head is mounted on a custom-made mounting plate which in turn is held by a 3-axis stage. The 3-axis stage facilitates movement of the camera to get the detector into the camera lens' focal plane.
- Control electronics. The readout amplifiers, timing, communications, and other electronics are contained in a large casing separate from the camera head. The control electronics generate a substantial amount of heat, and the casing they are mounted in contains several cooling fans. Such a generator of heat and wind is not desirable near an interferometric beam combiner, and so a system has been devised to isolate the control electronics from the rest of the Beam Combining Lab, described in Section 3.5.4. The control electronics communicate with, and supply power to the camera head via a data cable, and communicate with the

PCI camera card (described below) in a control computer via a pair of standard telecommunication optical fibers.

- Power supply. This provides power to the control electronics, and is connected to them via a short cable.
- Peltier cooling power supply. This provides power for the camera head's electric cooling system. It is connected to the control electronics box via yet another cable. The control electronics box relays power for the cooling system to the camera head via the head's data/power cable.
- Water cooling system. A ThermoElectric Cooling America Corporation (TECA) model TLC-700 Solid State Liquid Chiller refrigerates distilled water to 5° C and circulates it through insulated hoses to and from the camera head. The water cooler makes a lot of noise and puts out quite a bit of wind, so it is housed outside the beam combining lab. The insulated hoses run across the roof of the BCL where they pass through a cabinet that holds the power supplies and control electronics. The hoses terminate in quick-disconnect type connectors at the bottom of the cabinet, where connectorized lengths in insulated hose may be attached to complete the circuit to and from the camera head.
- PCI camera card. Installed in a control computer just outside the Beam Combining Lab, the PCI camera card communicates with the control electronics via a pair of standard telecommunication optical fibers. The PCI card sends commands to the camera electronics and receives data upon completion of each exposure.

The detector is a Marconi CCD39-01 back-illuminated chip, with an array of 80x94 pixels, each $24\ \mu\text{m}$ square. The central 80x80 pixel region of the chip is the actual imaging area. The chip is split into 4 equal quadrants, each of which is read out from a register in its outer corner. Each time the control electronics read out a pixel from the CCD, a read is taken from all 4 quadrants in parallel. If all 4 quadrants of the CCD are used, this readout scheme has the capacity to cut the camera readout time to $\frac{1}{4}$ what it would be with a single-readout CCD.

The camera readout speed is $2.6\ \mu\text{s}$ per pixel, so it takes 4.89 ms to read out the entire chip. The shortest exposure time that the camera electronics will allow is 1 ms. There is an additional 0.38 ms processing time required by the computer that records the data, and $43\ \mu\text{s}$ to transfer the frame data to the control computer's DMA buffer, so the full-chip time per frame is 6.31 ms, a frame rate of 158 Hz.

In order to adequately sample fringes in a turbulent atmosphere, one would like to sample them quickly enough that the fringes are effectively frozen during the sampling time. The characteristic time for the fringe phase to change by 1 radian, τ_o , is r_o divided by the wind speed of the turbulent layer. A typical r_o for good seeing at Mount Wilson is 10 cm at 550 nm (which translates to 11 cm at 600 nm, the blue end of the VIS system band pass), and a typical wind speed is about $15\ \frac{\text{m}}{\text{s}}$. These numbers give a τ_o of 7.3 ms, and a characteristic turbulent frequency of the atmosphere of 136 Hz. If we require 4 samples per fringe to obtain usable data, then the CCD must run at a frame rate of 545 Hz.

In practice, observations with the CHARA array in K' band have shown that the entire fringe packet should be sampled within a few τ_o . Even for the longest wavelength spectral channel of the VIS system under ideal seeing conditions, this is

an unrealistic requirement. When the entire fringe packet cannot be sampled within a few τ_o , the atmospheric piston will stretch and shrink the fringes as they are recorded, causing a broadening in the power spectrum peak of the fringes (see Section 4.2.3).

Clearly the full frame rate for the ARC CCD is too slow. Tom Schneider of Riptide Realtime developed a RT-Linux driver for controlling the ARC CCD. The ARC CCD driver utilizes a capacity built into the camera hardware for reading out only a sub-array of the chip. By reading out only a portion of the array, the frame rate can be dramatically improved. The driver in its present state is only able to handle the upper-left quadrant of the chip in the sub-array mode. This has not been a problem so far, as the highest number of simultaneous spectra used in the VIS system to date is 6. By reading out a sub-array that is 8x16 pixels, more than enough to hold 6 spectra, the camera takes 1.71 ms per frame, or 583 Hz. In the limit where so few pixels are read out that their contribution to the frame time is negligible, the minimum frame time is 1 ms exposure time plus 0.38 ms control computer lag plus 43 μ s frame transfer time, a total of about 1.43 ms per frame, a frame rate of 700 Hz.

When the light levels get so low that the fluctuations due to the fringes are comparable to the noise, the fringes become undetectable. The camera generates read noise when it counts the number of electrons in each pixel. One would like the read noise to be low enough that it is not the limiting noise source.

Laszlo Sturmann measured the photon transfer curve of the ARC CCD in the summer of 2005 as part of the CCD tip-tilt system development. The photon transfer curve is a plot of the *noise* in the number of electrons detected as a function of the *total* number of electrons detected. The read noise is the constant to which the photon transfer curve approaches as the number of electrons detected goes to zero. The ARC

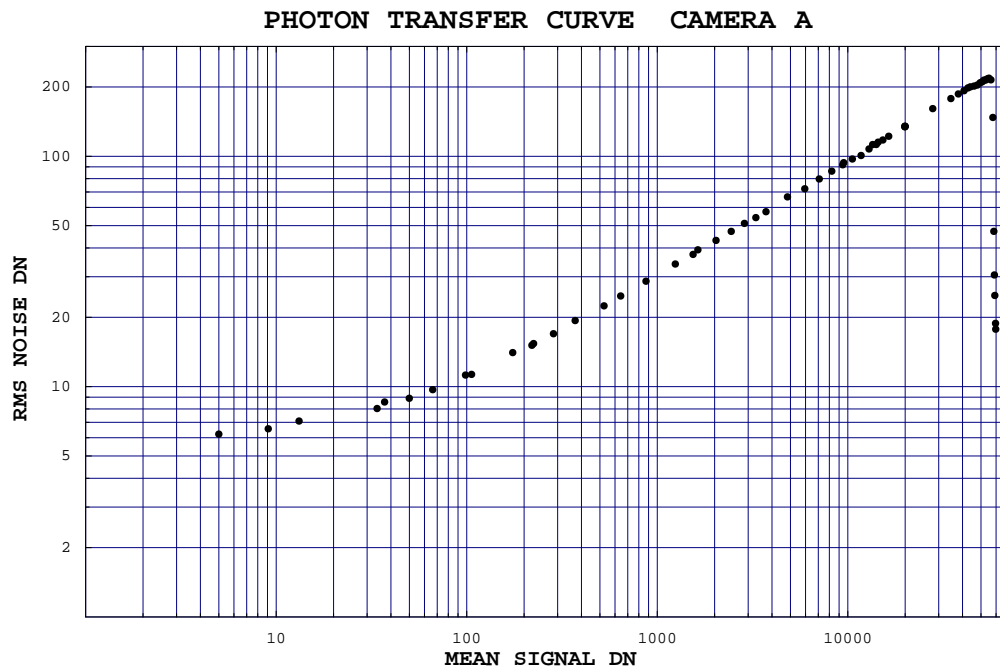


Figure 3.12: The ARC CCD photon transfer curve. Courtesy of Laszlo Sturmann, personal communication.

CCD photon transfer curve is shown in Figure 3.12. The read noise was measured at 6.6 electrons, and the linear part of the photon transfer curve showed a slope of 1.1 electrons per digital count.

The ARC CCD Cabinet and Water Cooling System

The heat and air currents produced by the ARC CCD control electronics are isolated from the BCL by mounting them in a cabinet on the ceiling of the BCL. The ARC CCD data cable is only 2 m long, and so the ceiling was the only place close enough to keep the electronics where they could be vented to the space between the inner and outer enclosures of the BCL. The cabinet is large enough to hold all of the



Figure 3.13: The ARC CCD electronics cabinet. The cabinet is large enough to hold two full sets of ARC CCD camera electronics, each on an independent computer-activated power strip. The cabinet is ventilated by a 4-inch electric fan directly into the BCL attic space. This avoids the generation of turbulent eddies in the BCL caused by heat dissipated from the camera electronics. Insulated hoses filled with water at 5°C run from a cooling system outside the BCL to connectors at the bottom of the cabinet, where connectorized hoses circulate the water to and from the ARC CCD camera. The ARC CCD data cable is only 2 m long, making a ceiling-mounted cabinet the only viable way to ventilate the camera electronics separately from the BCL.

electronics for both ARC CCD camera systems, and has independently wired power strips for each system. The power strips are plugged into a computer-controlled power distributor, so each system may be remotely turned on and off. Doors on both ends

of the cabinet provide access to the camera electronics.

The hoses that transport chilled water from the TECA cooler to the cameras are run across the top of the inner BCL enclosure and down through holes into the cabinet. The hoses terminate in quick-disconnect terminals mounted into the bottom of the cabinet, so that the plumbing from the cabinet to the cameras may be easily reconfigured. The water must go through both cameras in series before returning to the cabinet.

3.6 Spectrograph Calibration

The spectrograph was calibrated by measuring internal fringes with a white light source. If the dither position is well-calibrated, then the fringe frequency in each spectral channel gives the effective wavelength of that channel, provided one may assume that the spectral response over the channel is uniform. The assumption of uniform spectral response across each channel becomes worse with lower spectral dispersion, so this calibration cannot be completely accurate.

The fringe frequency is easiest to find by taking a power spectrum (the Fourier Transform of the data multiplied by its complex conjugate) of the fringe sweep data across some calibrated dither range. Because the data is a set of discrete values for each dither position, we use a Fast Fourier Transform algorithm to find the power spectrum. The Fast Fourier Transform (FFT) is a commonly used implementation of the Discrete Fourier Transform, used for data sets rather than continuous functions.

The power spectrum peak shows up at a bin number equal to the number of fringes

Table 3.1: Effective wave number κ and wavelength λ for VIS spectral channels.

Spectral Channel	$\kappa \left(\frac{1}{\mu\text{m}}\right)$	$\text{err}_\kappa \left(\frac{1}{\mu\text{m}}\right)$	$\lambda \text{ (nm)}$	$\text{err}_\lambda \text{ (nm)}$	$\Delta\kappa \left(\frac{1}{\mu\text{m}}\right)$
2	1.56	± 0.01	641	± 3	0.119
3	1.42	± 0.01	702	± 6	0.144
4	1.27	± 0.01	786	± 6	0.157
5	1.10	± 0.04	901	± 35	0.164

that would fit into the whole data set. The bin number n_{bin} is given by

$$n_{\text{bin}} = \frac{\Delta z_{\text{opd}}}{\lambda_{\text{eff}}} , \quad (3.8)$$

where z_{OPD} is the total OPD distance induced by dither sweep, and λ_{eff} is both the fringe wavelength and the effective wavelength of the spectral channel. This measure of n_{bin} is independent of the data sampling (as long as the fringes are adequately sampled), as a change in the step size would show up equally in both the numerator and denominator of Equation 3.8.

To find λ_{eff} one rearranges (3.8) to obtain

$$\lambda_{\text{eff}} = \frac{\Delta z_{\text{opd}}}{n_{\text{bin}}} . \quad (3.9)$$

Table 3.1 gives the latest calibration of the VIS spectral channels, done on May 12, 2005 during the VIS system first fringe search.

The spectrograph calibration will change any time that the spectra change position on the detector. Any time the spectrograph is adjusted, the spectra must be re-

calibrated. This is an easy process, one simply records a data set of internal white light fringes. The set of IDL tools written by David Berger and myself (see Appendix) for examining VIS system data is able to produce a mean power spectrum for each spectral channel, built by averaging the power spectra from all dither scans in the data set. One may find the bin number n_{bin} of each channel's power spectrum peak by examining the power spectrum plots, then divide Δz_{opd} ($356.16 \mu\text{m}$) by n_{bin} to obtain λ_{eff} .

In the event that the spectrograph requires frequent recalibration in the future, a program could easily be written that would automate the calibration process. In practice, only a few recalibrations have been necessary, and while troubleshooting the system, the author has restrained himself many times from adjusting the spectrograph until he was sure it was the source of the problem. One may save oneself hours of re-alignment and calibration work by keeping in mind the colloquialism: *If it ain't broke, don't fix it.*

3.7 Software

The camera software, is in three parts, of which the last two are included in full in Appendix B.

The first is a driver written by Tom Schneider (www.riptiderealttime.com) for the ARC CCD camera. The driver handles all direct communications with the camera, and takes requests from other software. It defines the size of sub-arrays of the CCD when it initializes the camera. Whenever the camera electronics behave erratically, they may usually be sorted out by running a script that makes calls directly to the

driver, running the camera through its basic initialization procedure. This script is usually run before running the regular camera control code, to ensure that the camera communications are behaving properly. Most problems with erratic camera behavior were recently traced to a faulty PCI card that communicates with the cameras electronics via optical fibers. Replacement of the PCI card should improve the camera behavior.

The second is an RT-Linux module, which is appended to the system kernel. As part of the operating system, it is able to handle real-time data handling tasks, with the caveat that if it is given too much to do, the operating system will never get around to processing any of the normal user program functions. The real time module registers a callback function with the ARC CCD driver, and whenever the CCD finished an exposure and puts the data in the DMA buffer, the callback function is given control. The callback function copies the data from requested pixels into dedicated memory spaces, and does whatever real-time analysis of the data is required.

The VIS system was originally designed as a group delay fringe tracker, which would monitor the OPD by means of the spatial frequency of fringes in the sub-aperture spectra, called channel fringes. The spatial frequency of the channel fringes is proportional to the OPD. The spectra were Fourier transformed, and the frequency of their power spectrum peak found. The VIS system proved too insensitive to group delay track on stars, but the real-time Fourier transform functionality remains in the code, in the event that channel spectra are desired in the VIS system while coupled to a separate fringe tracker.

The real-time module handles requests from the user side for data from each exposure. The module places data in a First-In-First-Out (FIFO) memory buffer,

which the user side programs may read from. As long as the user side systems can empty the FIFO fast enough to keep it from overflowing, the data pipeline works smoothly.

The real-time module also sends a flag to the module that controls the dither mirror position. The flag tells the dither module that the exposure is complete and it may move to the dither position for the next exposure. The dither module is serviced at a very fast clock rate, so it moves the dither mirror to its next position in small increments to avoid any excess ringing.

The third software component, the normal user side software, handles the major organizational tasks such as defining sub-arrays and requesting them from the driver, selecting the spectral rows and channels on the chip for which data will be recorded, setting up dither scans, taking bias data, and recording fringe data. A full list of available commands is contained in the file `std_arc_ccd_functs.c`, in Appendix B.

Research is what I'm doing when I don't know what I'm doing.

— Wernher von Braun

Chapter 4

Experimental Results

During the 2005 CHARA Array observing season, time was allotted for the VIS system first fringe search. A few small, bright stars were chosen as first fringe targets, described in Section 4.1. First fringes were successfully found on HR 5191, on May 14th Universal Time (UT.) Several nights were spent working kinks out of the system, culminating in a night of excellent seeing on June 26th (UT), during which the data in this chapter were taken on the S1-S2 baseline, with POPs 0 and 1 respectively.

All of the data analysis was done in IDL, starting with a set of stopgap data analysis and plotting tools written by David Berger. These tools were modified and expanded by the author as need arose. The tools eventually gained permanent status, as is the fate of most ‘temporary’ solutions.

4.1 The Stars

An extensive list of objects and calibrator stars was prepared for the first fringe search, but was quickly culled down to the smallest and brightest objects that would be high in the sky during the months of April to June. Listed here are only the objects from which interference fringes were detected. Previous measurements and estimates of object angular diameters were found using the CHARM2 (Richichi et al., 2004) and

CADARS (Pasinetti Fracassini et al., 2001) catalogs to ensure that the visibilities for the S1-S2 baseline would be high enough. All stars were checked for companions in the Washington Double Star Catalog (Mason et al., 2001), and all companion data listed in this section were obtained there.

HR 5191, Alcaid

Alcaid, known also as HR 5191, HD 120315, and η Ursa Majoris, is a spectral type B3V star with an R-band (670 nm) magnitude of 1.98, and an I-band (870 nm) magnitude of 2.16.

To date, no one has published an interferometric measurement of Alcaid's angular diameter, except to limit it to less than 2 mas (Bonneau et al., 1981). Eight estimates of Alcaid's angular diameter have been published based upon several methods using flux, spectral type, parallax, etc. Of these publications, one was thrown out because its errors were much larger than the others, and the remaining seven were averaged to give an angular diameter of 0.78 mas for Alcaid (Cayrel de Strobel, 1960; Wesselink et al., 1972; Underhill et al., 1979; Blackwell et al., 1980; Ochsenbein and Halbwachs, 1982; Morossi and Malagnini, 1985; Leggett et al., 1986).

Alcaid's angular diameter is so small that for S1-S2's 34 m baseline, the fringe visibility stays above 95% (see Figure 4.3 for a visibility plot). With such a high visibility for its brightness, Alcaid was chosen as the primary first fringe target. It is the end star in the Handle of the Big Dipper, and so is 6.7° from HR 5054 (Mizar) and 10.5° from HR 4905 (Alioth), the next two stars in the Handle.

The author had planned to record fringes from the bright stars in the Handle and Cup of the Big Dipper, using Alcaid and HR 4660 (Megrez) as calibrators for the

other stars whose visibilities were predicted to fall between 0.5 and 0.95. Alcaid has been used previously at NPOI as a calibrator star for Mizar (Hummel et al., 1998).

Sadly, this scheme is better suited for the spring months, and by the time the VIS system had found fringes, the Big Dipper was only high enough to observe at the beginning of the night. As a result, no usable fringes were found on the other stars in Ursa Major. An effort to measure all the Big Dipper stars would best be rescheduled for a late winter or early springtime run when Alcaid crosses the meridian around midnight.

HR 7001, Vega

Vega, known also as HR 7001, HD 172167, and α Lyrae, is a spectral type A0V star with an R-band (670 nm) magnitude of 0.07, and an I-band (870 nm) magnitude of 0.10. Vega has several optical companions listed in the Washington Double Star Catalog, but none of them are brighter than magnitude 9.5 and so shouldn't affect the detected visibility.

Many angular diameters have been published for Vega, obtained with various interferometers at wavelengths from 438.5 nm to 2.2 μm . Of these, a uniform disk diameter of 3.149 mas was measured at 800 nm using the Mark 3 interferometer (Mozurkewich et al., 2003). For purposes of estimating the VIS system visibility, this diameter will be used, because those made at other wavelengths may introduce small errors from a wavelength dependence in the angular diameter. At the S1-S2 baseline, Vega's visibility ranges from 0.4 to 0.7 in the VIS system wave band (see Figure 4.4 for a visibility plot.)

The author had planned to use HR 7178 (Sulafat) as a calibrator star, a spectral

type B9III star with R- and I- magnitudes of 3.27 and 3.28, respectively. Sulafat has an estimated diameter of 0.74 mas, which would give it a visibility above 0.95 with S1-S2 and the VIS system. It is part of a multiple system, but its companions are fainter than 10th magnitude, and may will not affect the detected visibility. Unfortunately, Sulafat's magnitude proved to be too dim for the VIS system to record usable fringes, so the Vega data cannot be calibrated.

HR 7924, Deneb

Deneb, known also as HR 7924, HD 197345, and α Cygni, is a spectral type A2Iae star with an R-band (670 nm) magnitude of 1.14, and an I-band (870 nm) magnitude of 1.04. It has an optical companion with a V magnitude of 11.7, which should not interfere with the visibility measurements.

Among the many interferometric angular diameters published for Deneb, three were recorded at wavelengths similar to the VIS system. The first was made at I2T in 1981, and had errors an order of magnitude larger than the others, and so was not considered. The second was made with the Mark III interferometer at 800 nm (Mozurkewich et al., 2003), and the third at NPOI across the 650-850 nm range (Aufdenberg et al., 2002). The average of the two measurements gives a uniform disk of 2.37 mas for Deneb. Deneb's visibility is in the 0.6-0.8 range for the S1-S2 baseline and the VIS system operating wavelengths (for a visibility plot, see Figure 4.5).

The calibrator for Deneb was HR 7528 (δ Cygni), a spectral type B9.5IV+ star with R- and I- band magnitudes of 2.88 and 2.90 respectively. It has a companion with a V magnitude of 6.7; the companion is separated from δ Cygni by an angle of about 2 arc seconds, and so should not affect the measured visibility. δ Cygni's

estimated angular diameter is 0.84 mas, giving a visibility above 0.93 with the S1-S2 baseline and the VIS system wavelengths.

As with Vega's calibrator, δ Cygni was too faint for the VIS system to record usable fringes, and so the author was unable to calibrate the Deneb visibility data.

4.2 The Fringe Data

The fringe data were obtained using the CHARA Classic system as a low-bandwidth fringe packet tracker. At the beginning of each night, the VIS system was carefully co-phased with the CHARA Classic system to within a few μm .

CHARA Classic sees fringes easier than the VIS system for several reasons. CHARA Classic uses the K' band filter ($2.13\mu\text{m}$), where all of the objects and calibrators have visibilities near unity for the S1-S2 baseline. Seeing effects that degrade the visibility are less pronounced in K' band as well, and the NIRO camera that CHARA Classic uses is more sensitive than the VIS system's ARC CCD. Flaws and dust at each optical surface also affect the visible wavelengths much more than the Infrared.

After acquisition of the star, the CHARA Classic system was used to scan for fringes. Once CHARA Classic found fringes it was set to scan back and forth through the fringe packet, keeping it centered in the dither sweep. Because the two systems had been co-phased previously, one could simply scan the VIS dither mirror and find VIS fringes as well.

The LDCs were not in use during the first fringe run, so differential longitudinal dispersion between the K' band and the VIS system band caused the fringes to appear at slightly different OPD positions. Depending upon the amount of delay that the

OPLEs had to compensate for, the relative offset between the IR and VIS fringes was observed to be up to about $90\ \mu\text{m}$ in OPD. This was enough to require that CHARA Classic be scanned with its fringe packet near one end of the dither scan in order to center the VIS fringes in their dither scan.

With longer baselines, the relative offset between IR and VIS fringes would change enough during the night that the two system's dither scans could not hold their respective fringe packets while still overlapping in OPD. In future use of the VIS system, the LDCs should be used to remove as much differential dispersion as possible, and should enable the use of CHARA Classic as a fringe scanner for the VIS system at all baselines.

The fringe data were recorded for the 6 outer sub-apertures of the fiber injector at the north output beam of the beam 5-6 combiner. Each spectrum had 4 consecutive channels (pixels) whose effective wavelengths were calibrated to be 640, 702, 785, and 901 nm. The ARC CCD was run with the minimum 1ms exposure times (587 Hz) in an effort to get as many fringe samples as possible per τ_o .

A set of bias data for the ARC CCD was recorded with 1 ms exposure times, and was automatically subtracted from each spectral channel. The bias data also function as dark data because the exposure time was selected to match the on-sky exposure time. The de-biased values were the quantity recorded for each spectral channel, and the bias data were saved at the end of each data set so that the unbiased values could be reconstructed, were they ever needed.

The dither sampling rate was set to 4 samples per fringe at 600 nm. Because the data at all spectral channels are recorded from a given CCD exposure, it is not possible to adjust the sampling of the spectral channels independently. One must

Table 4.1: Throughput calculated from Vega flux

λ (nm)	Flux ($\frac{\text{erg}}{\text{cm}^2 \text{ s nm}}$)	Energy (erg)	Photons	Photons Detected	Throughput
641	2.18×10^{-8}	1.01×10^{-12}	3.27×10^4	1402	4.3%
702	1.56×10^{-8}	9.55×10^{-11}	3.38×10^4	1916	5.7%
786	1.15×10^{-8}	9.62×10^{-11}	3.80×10^4	1314	3.5%
901	9.0×10^{-9}	1.1^{-12}	4.9×10^4	835	1.8%

choose the minimum acceptable sampling for the shortest wavelength channel, and the longer wavelength channels will have higher sampling by a factor of the ratio of the channel wavelengths. With 4 samples per fringe at 600 nm, the sampling in the 4 spectral channels was 4.27 (640 nm), 4.68 (702 nm), 5.23 (785 nm), and 6.01 (901 nm) samples per fringe.

4.2.1 Throughput

The VIS system throughput was calculated by comparing Vega's incident flux to the measured photon counts at the VIS system detector. Table 4.1 shows the various quantities leading to a throughput estimate for each spectral channel.

The incident flux was measured by Hubble Space Telescope absolute spectrophotometry (Bohlin and Gilliland, 2004), given in units of $\frac{\text{erg}}{\text{cm}^2 \text{ s nm}}$. The flux was averaged across each spectral channel's bandwidth to obtain the effective flux for that channel.

To obtain the incident energy per CCD exposure, the flux was multiplied by 1 ms exposure time, the spectral channel bandwidth in nm, and an area of a circle with a 33 cm radius. That circle is the size of a sub-aperture at the fiber injector projected back onto the primary, and is a good estimate of the effective light gathering power of that sub-aperture.

The incident energy was divided by the energy per photon ($\frac{hc}{\lambda}$) at each channel wavelength to obtain the number of photons that were incident on at the telescope for that sub-aperture and spectral channel during the exposure time. The system throughput was then found by dividing the incident photons by the number of photons detected. It is important to note that the incident photons were not corrected for camera sensitivity or atmospheric losses, so the throughput given in Table 4.1 is from the top of the atmosphere to detection.

The author found that when stars generated less than 100 photon counts per channel, fringes were generally inseparable from the noise. Using the relation between flux and apparent magnitude:

$$m_2 - m_1 = -2.5 \log \frac{f_2}{f_1} , \quad (4.1)$$

where f is the flux and m is the magnitude of each star, we can find a limiting magnitude when we get 100 photon counts in each channel.

Let m_1 be Vega's magnitude interpolated between R- and I-band for each spectral channel wavelength. Vega's R-band (670 nm) magnitude is 0.07, and the I-band (870 nm) magnitude is 0.10. Because flux is proportional to photon counts, we can replace $\frac{f_2}{f_1}$ with $\frac{N_2}{N_1}$, where N is the number of photon counts. N_1 is therefore the number of photon counts from Vega in a given spectral channel, and N_2 is 100.

Using the recorded photon counts from Table 4.1 for N_1 yields the following spectral channel limiting magnitudes: 2.9 (641 nm), 3.3 (702 nm), 2.9 (786 nm), and 2.4 (901 nm.)

After September of 2005, the CHARA system switched over to CCD-based tip-

tilt control. The CCD-based system uses a 50/50 gray split across the entire visible to near-IR band. The older PMT-based tip-tilt used a dichroic split and took all photons with wavelengths shorter than 600 nm. As a result, after conversion to the CCD tip-tilt system, the VIS system suffered an additional 50% loss of light across the entire operational wave band, but gained the capacity to detect light short of 600 nm, where the ARC CCD is much more sensitive. This dissertation was in its final writing stages at the end of September 2005, so the author was unable to include a calibration of the system throughput under CCD tip-tilt control.

4.2.2 The Standard Data Reduction Method

The standard CHARA data reduction method is laid out in the CHARA instrument paper (ten Brummelaar et al., 2005), and is based upon the paper by Benson et al. (1995).

As the dither mirror scans through the fringe packet, the detected signal on a given sub-aperture spectral channel is given by

$$D_i(t) = \frac{I_{i1} + I_{i2}}{2} + (-1)^i \sqrt{I_{i1} I_{i2}} \nu \operatorname{sinc}(\pi \Delta \kappa v_g t) \cos(2\pi \kappa_o v_g t + \phi), \quad (4.2)$$

where I_{ij} is the intensity of the light from telescope j which reaches beam combiner output i . The two beam combiner outputs are 180° out of phase, which can be seen by comparing $D_0(t)$ with $D_1(t)$. ν is the correlation between the two combiner input beams, and is equal to the product of the object and system visibilities, $V_{\text{sys}}V$. κ is the wavenumber $\frac{1}{\lambda}$, also commonly denoted by σ , as in the CHARA instrument paper. The bandwidth of the spectral channel is $\Delta\kappa$, and its effective wavenumber

is κ_o . The speed of the dither sweep determines the group velocity v_g , the speed at which the fringe packet moves through the detector's OPD position. The phase ϕ is a sum of fringe phase due to the object, phase introduced via atmospheric turbulence, and any phase introduced by the system optics. The function $\text{sinc}(x)$ is defined as $\frac{\sin x}{x}$.

$D_i(t)$ may be normalized by multiplying it by $\frac{2}{I_{i1}+I_{i2}}$. This results in the form

$$N_i(t) = 1 + (-1)^i \frac{2\sqrt{I_{i1} I_{i2}}}{I_{i1} + I_{i2}} \nu \text{sinc}(\pi \Delta \kappa v_g t) \cos(2\pi \kappa_o v_g t + \phi). \quad (4.3)$$

The normalized signal has values between 0 and 2. Both the sinc and cos functions reach values of ± 1 at the center of the fringe packet, so the fringe packet amplitude is $T_i \nu$, where

$$T_i = \frac{2\sqrt{I_{i1} I_{i2}}}{I_{i1} + I_{i2}} \quad (4.4)$$

is the transfer function due to any mismatch in the telescope beam intensities.

By normalizing the data, then filtering it in the Fourier plane to only allow frequencies near the actual band pass of the spectral channel, one obtains a clean signal of $N_i(t) - 1$. The signal may be fit to a model of a fringe packet, giving the amplitude $T_i \nu$ at the fringe packet center. Figure 4.1 shows a filtered signal of $N(t) - 1$ recorded on May 26, 2005 from Deneb.

While finding the fringe visibility from individual fringe envelopes works, a preferable method is to use the power spectrum of the fringe data. The power spectrum is the data's Fourier transform multiplied by its complex conjugate. The power spec-

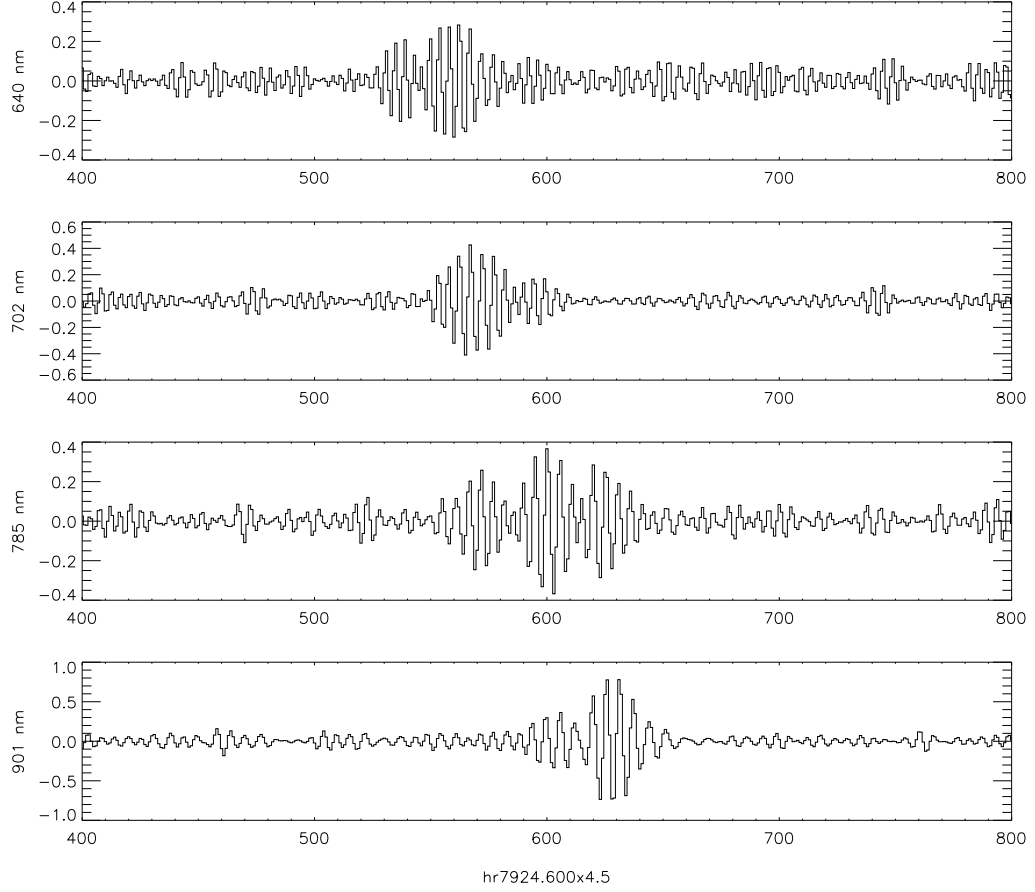


Figure 4.1: Normalized, filtered fringes from HR 7924, Deneb. The horizontal axis is exposure number, with exposures spaced every 150 nm in the dither sweep. The fringe packet amplitude is equal to $T_i \nu$, which in turn equals $2\sqrt{I_{i1} I_{i2}} / (I_{i1} + I_{i2}) V_{\text{sys}} V$.

Longitudinal dispersion is caused by not having the OPLE carts in vacuum, and results in a wavelength-dependent zero OPD position. Zero OPD for each spectral channel is at the center of each channel's fringe packet. Use of the LDC system would reduce the effects of longitudinal dispersion to a negligible amount.

trum of the normalized fringe signal $N_i(t) - 1$ is

$$\text{PS}[N_i(t) - 1] = T_i^2 \left(\frac{\nu}{\Delta \kappa v_g} \right)^2 \left[\Pi \left(\frac{\nu - \kappa_o v_g}{\Delta \kappa v_g} \right) + \Pi \left(\frac{\nu + \kappa_o v_g}{\Delta \kappa v_g} \right) \right], \quad (4.5)$$

where $\Pi(x)$ is the “boxcar function” defined as

$$\Pi(x) = \begin{cases} 1, & |x| < \frac{1}{2}, \\ \frac{1}{2}, & |x| = \frac{1}{2}, \\ 0, & |x| > \frac{1}{2}. \end{cases} \quad (4.6)$$

A basic property of Fourier transforms is that if a function is all real-valued, then its transform is anti-symmetric about 0. The power spectrum is then an anti-symmetric function squared, which is symmetric. Because the data must be real, its power spectrum is symmetric about 0 and the negative frequencies may be ignored, as they contain the same information as the positive ones.

Now we may write the power spectrum as

$$\text{PS}[N_i(t) - 1] = T_i^2 \left(\frac{\nu}{\Delta\kappa v_g} \right)^2 \Pi \left(\frac{\nu - \kappa_o v_g}{\Delta\kappa v_g} \right). \quad (4.7)$$

The power spectrum looks like a rectangle centered at $\kappa_o v_g$, with its edges at $\frac{1}{2}\Delta\kappa v_g$ above and below the center, giving it a width of $\Delta\kappa v_g$. The $\Pi(x)$ function is equal to 1 inside the rectangle, so the height of the rectangle is $T_i^2 \left(\frac{\nu}{\Delta\kappa v_g} \right)^2$.

The integral of the power spectrum S_i is simply the area of the rectangle, so

$$S_i = \int d\nu \text{PS}[N_i(t) - 1] = T_i^2 \frac{\nu^2}{\Delta\kappa v_g}. \quad (4.8)$$

Thus, the correlation can be extracted from the power spectrum of the fringe visibility. In practice, the power spectrum peak has the shape of the spectral channel band pass, which was assumed to be a boxcar function in the derivation of (4.2). The

shape of the power spectrum peak may change, but not its *area*. Regardless of the peak shape, Equation 4.8 still holds.

There are several sources of noise that must be removed in frequency space. By taking a set of data where the dither sweep does *not* pass through the fringe packet, one may obtain a power spectrum that shares the same noise properties as the fringe data. Fringe-free data may also be pieced together from the portions of the dither scan that do not contain the fringe packet. The noise power spectrum is subtracted from the fringe power spectrum prior to the integration in Equation 4.8.

4.2.3 Power Spectrum Peak Broadening Due To τ_o

Upon inspection of Figure 4.1, one may observe that the shape of the fringe packets in each channel are erratic, and do not conform closely to the sinc function envelope shape.

The reason for this is τ_o . An estimate of typical τ_o for Mount Wilson is about 6.7 ms, corresponding to an r_o of 10 cm and a wind speed at the optical turbulence generating layer of $15 \frac{m}{s}$ (see Table 1.1). For the fringe packet envelope to have a smooth, symmetric shape, the entire fringe packet must be sampled within a few τ_o . If the sampling is slower than that, the atmosphere shifts the fringe packet around. As a result of the fringe packet moving while it is being sampled, the detected fringe envelope becomes more erratic. More importantly, however, the fringes themselves become distorted. The stretching and squashing of fringes within the packet means that they will show up across a broader frequency range in the power spectrum. In effect, atmospheric turbulence broadens the power spectrum peak. Because the peak's area must remain unchanged, the broadened peak has a lower amplitude, reducing

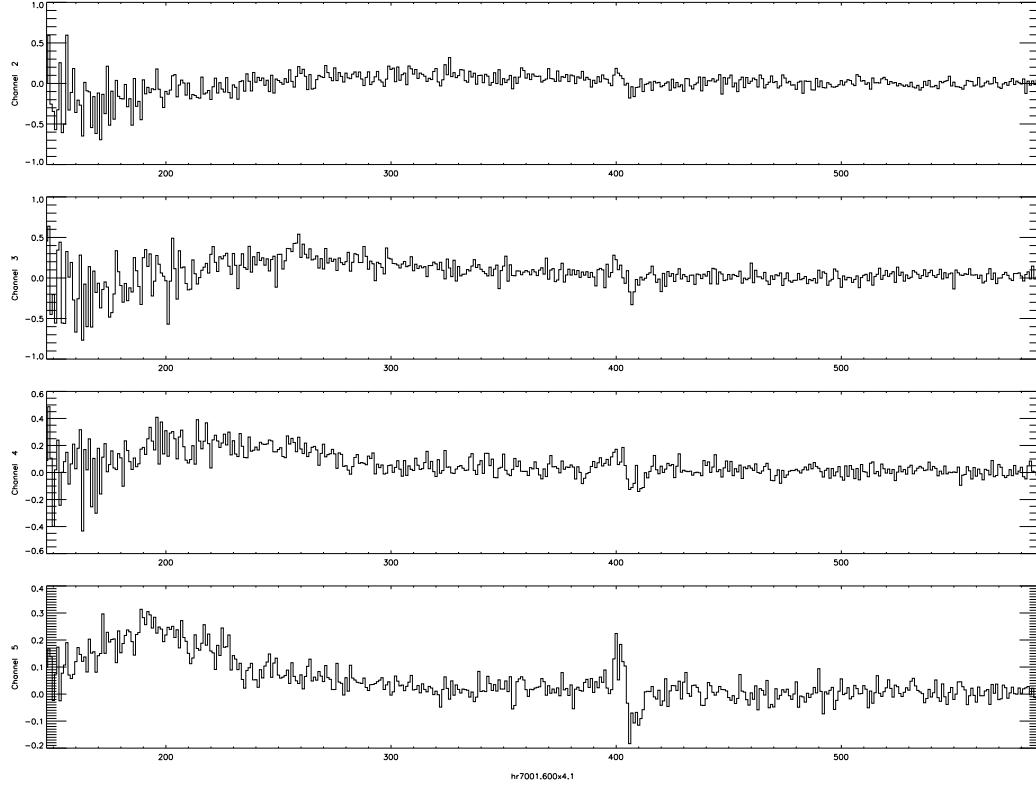


Figure 4.2: Power spectrum of raw fringe data from HR 7001, Vega. The power spectra for 44 dither scans and all 6 outer sub-apertures of the fiber injector have been averaged, and the noise power spectrum has been subtracted. The horizontal scale is the frequency bin number. The frequency bin where the fringe peak is centered is the dither sweep divided by the channel wavelength. Those values for the four channels shown are 285, 260, 233, and 203 from top to bottom. The fringe power spectrum peak is only readily identifiable in the bottom channel, centered near bin 203 as expected. τ_o scales as $\lambda^{6/5}$, meaning the shorter wavelengths have shorter τ_o , so the optical turbulence moves faster and broadens the fringe power peak more thoroughly. The feature at bin 400 is an artifact due to the dither mirror motion. It was slightly different in the on- vs. off- fringe data, and was not perfectly subtracted out as a result.

its ability to stand out from the noise.

Figure 4.2 shows a sample power spectrum from the June 26 fringe data. The power spectrum has already had as much noise removed as possible, and still only

the bottom channel (901 nm) has a recognizable peak. The peak only has a signal-to-noise ratio (SNR) of about 3, and this is the best data set of the entire night. In data sets where there was no recognizable power spectrum peak, fringe packets could still clearly be seen, though they were distorted. The power spectrum method was abandoned for these data, and another method attempted based upon the amplitude of each fringe packet.

4.2.4 Uncalibrated Visibilities From Fringe Packet Averages

Upon inspection of several nights' data, it was found that provided a spectral channel had about 100 counts on average, fringe packets could be discerned visually from among the filtered, normalized data.

The fringe data were normalized by dividing by the mean photon counts in each channel and multiplying by 2. The normalized data were then Fourier transformed and multiplied by a Gaussian filter function whose FWHM was twice the spectral channel bandwidth. After filtering, the data were inverse Fourier transformed to obtain a filtered, normalized fringe signal in the form of $N_o(t) - 1$. An example of such a fringe signal has been presented previously in Figure 4.1.

The fringe envelope amplitude, $T_o\nu$, was found as the data point with the maximum absolute deviation from the mean in each dither scan. This is a rather simplistic way to find the fringe envelope peak, and relies entirely upon the assumption that the fringe envelope has a stronger signal than the noise in each scan.

To establish an average fringe envelope amplitude over all scans in a data set, the amplitudes for each scan were combined in a weighted average. The weight each scan received was proportional to its amplitude, so that the scans that were more likely to

give erroneous amplitudes from noise were given less weight.

Up to this point, each sub-aperture and spectral channel had been processed independently. Because each sub-aperture had a different transfer function, the fringe amplitude $T_o\nu$ had to be divided by its corresponding transfer function before ν could be averaged over all sub-apertures for each spectral channel. The transfer function was calculated using two data sets, each with the light from only one telescope, which were taken immediately after the fringe data.

There was some concern that the mean counts of the individual telescope data sets did not add up exactly to the mean counts in the fringe data set. As long as one can assume that the *ratio* between the two telescope beams remained unchanged, it does not make a difference. If we express I_{i1} and I_{i2} as a ratio and a sum:

$$\begin{aligned} R &= \frac{I_{i1}}{I_{i2}} \\ S &= I_{i1} + I_{i2} \end{aligned} \tag{4.9}$$

then

$$\begin{aligned} I_{i1} &= \frac{RS}{R+1} \\ I_{i2} &= \frac{S}{R+1} \end{aligned} \tag{4.10}$$

can be plugged into the definition of the transfer function (4.4) to obtain

$$T_i = \frac{2\sqrt{\frac{RS^2}{(R+1)^2}}}{S} = \frac{2\sqrt{R}}{R+1}, \tag{4.11}$$

which depends only on the input beam ratio.

Because some spectral channels had more light than others, and some had a more even distribution of light between the two input beams, a weighted average was used to

find ν for each channel. Each sub-aperture ν was weighted by its mean photon counts and its transfer function. The transfer function drops below 0.9 only after the beam ratio drops below 0.4 or exceeds 2.5. By multiplying by the transfer function, sub-apertures with grossly mismatched input beams have their weight reduced. Weighting by photon counts as well gives more importance to channels with a higher signal-to-noise.

During the data processing, it was found that the fringes recorded while the dither mirror was moving “up” (increasing voltage, or toward the combining splitter, reducing the beam 5 path), consistently had a slightly higher visibility than the “down” scanned fringes. For this reason, the up and down data were processed independently, in parallel.

The following subsections show the uncalibrated visibility, that is to say the correlation $\nu = V_{\text{sys}}V$ of Alcaid, Deneb, and Vega.

The ‘ \diamond ’ symbols each correspond to a visibility averaged across all sub-apertures for a given spectral channel. This value was consistently lower than the visibility from only the brightest sub-aperture for each spectral channel, plotted with a ‘+’ symbol.

Occasionally the atmosphere would calm down just long enough for a lucky high-visibility fringe packet to be recorded. The single highest fringe packet visibility from the brightest sub-aperture for each spectral channel is plotted with a ‘ \times ’ symbol. These give a hint at what the system visibility might be with no atmospheric turbulence effects.

The plot points tend to cluster in pairs. Each pair tends to be the up- and down-dither visibilities for a particular spectral channel data set.

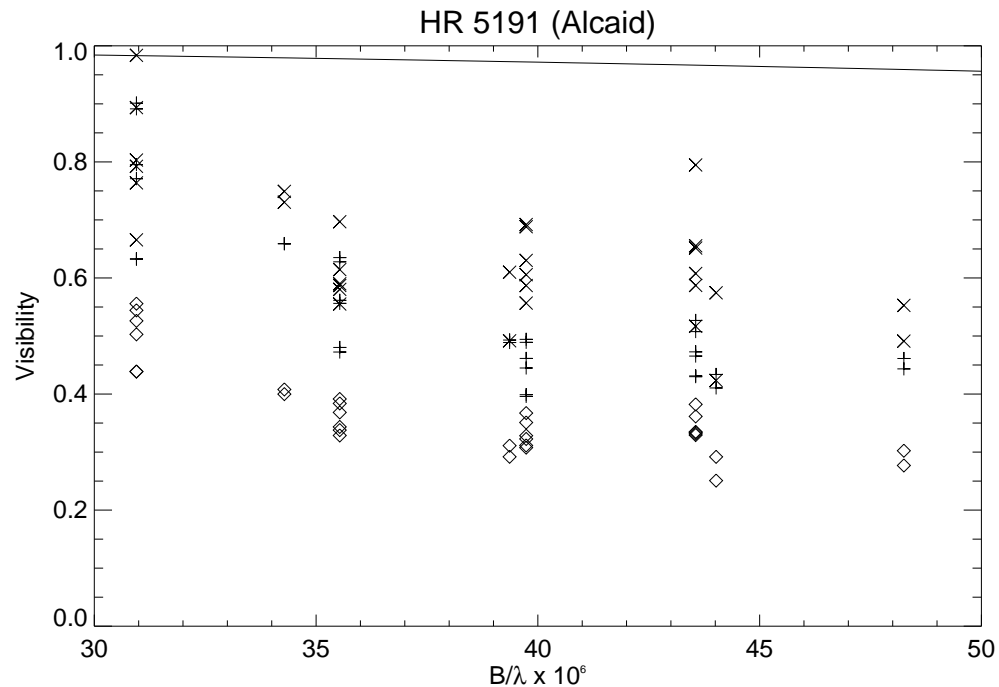


Figure 4.3: Uncalibrated Visibility vs. B/λ for HR 5191, Alcaid. The solid line is the expected visibility of the star based upon the angular diameter 0.78 mas, an average of several published estimates. The symbols are:
 ◇: V weighted average over all sub-apertures.
 +: V for only the brightest sub-aperture.
 ×: Highest single scan V from the brightest sub-aperture.

HR 5191, Alcaid

Figure 4.3 shows the uncalibrated visibilities measured for Alcaid, the result of four data sets. The projected baseline changed over the course of the observations, causing the spread in B/λ coverage for each spectral channel.

There is a considerable spread between the visibilities averaged over all sub-apertures (◇) and those from only the brightest sub-aperture (+). The fainter sub-apertures were below 100 counts, even in the brightest spectral channel, and were at

about 25% the intensity of each channel's brightest sub-aperture. With this in mind, perhaps the bright sub-aperture visibilities are more accurate.

Alcaid moved lower in the sky as the data sets were taken, and this shows in the spectral channel mean photon counts. The brightest sub-aperture starts at: 206 (641 nm), 272 (702 nm), 184 (785 nm), 114 (901 nm) and drops to 148 (641 nm), 165 (702 nm), 101 (785 nm), 57 (901 nm) by the 4th data set.

Because of the generally low numbers of photon counts per channel, these data are prone to a lot of misidentified noise passing as fringe peaks. There is a general downward trend in the data as B/λ increases, but this is probably more due to the system visibility dropping with shorter wavelengths than it is a reflection of Alcaid's visibility curve. For an estimate of the system visibility, see Section 4.2.5.

HR 7001, Vega

Figure 4.4 plots the uncalibrated visibilities for Vega, the result of two data sets taken one after the other during the best seeing of the night. The tip-tilt system reported r_o sizes of 12-15 cm during this brief period, although that number has tip-tilt system errors and internal seeing wrapped up in it as well.

Vega has only a 40-70% visibility, but is so bright that these data are the most consistent of the three objects. Typical mean photon counts from the brightest sub-aperture of each spectral channel were: 1472 (641 nm), 2019 (702 nm), 1388 (785 nm), and 883 (901 nm). Even the faintest sub-apertures in the 901 nm spectral channel stayed near or above 100 photon counts. With this in mind, the visibility averaged over all sub-apertures is probably the most reliable estimate, though it typically is less than 0.1 lower than the brightest sub-aperture visibility.

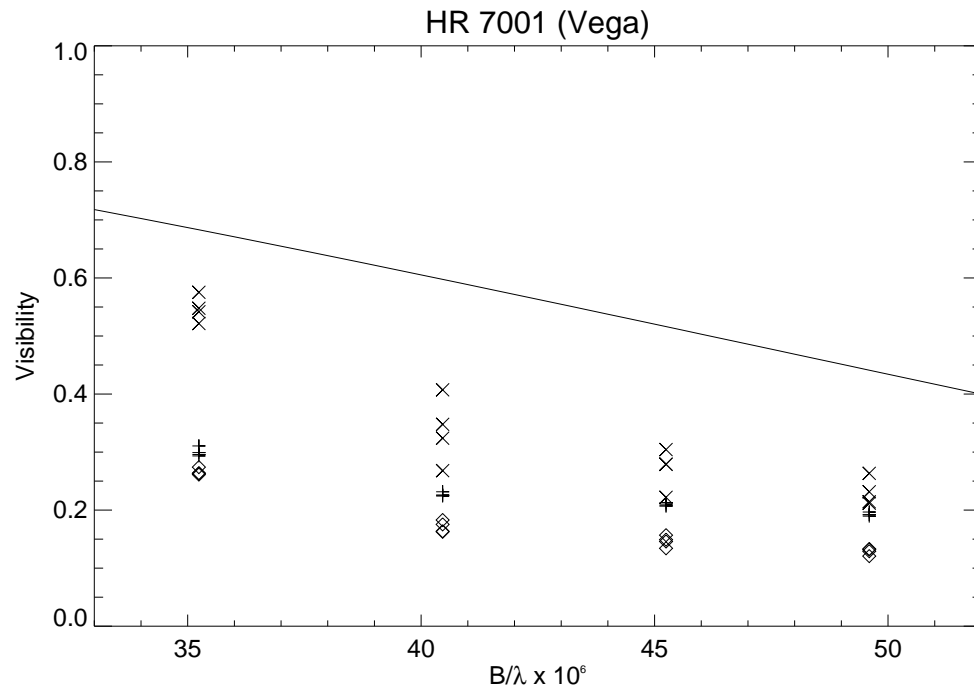


Figure 4.4: Uncalibrated Visibility vs. B/λ for HR 7001, Vega. The solid line is the expected visibility of the star based upon uniform disk angular diameter 3.15 mas, measured with the Mark III interferometer. The symbols are:

- ◊: V weighted average over all sub-apertures.
- +: V for only the brightest sub-aperture.
- ×: Highest single scan V from the brightest sub-aperture.

The single highest fringe packet visibility for the brightest sub-apertures is unlikely to be noise with such high photon counts, and it follows Vega's visibility curve quite well. These data would suggest that perhaps the system visibility does not drop precipitously as one nears the shorter wavelengths of the VIS system.

HR 7924, Deneb

Figure 4.5 shows the uncalibrated visibilities for Deneb over three data sets.

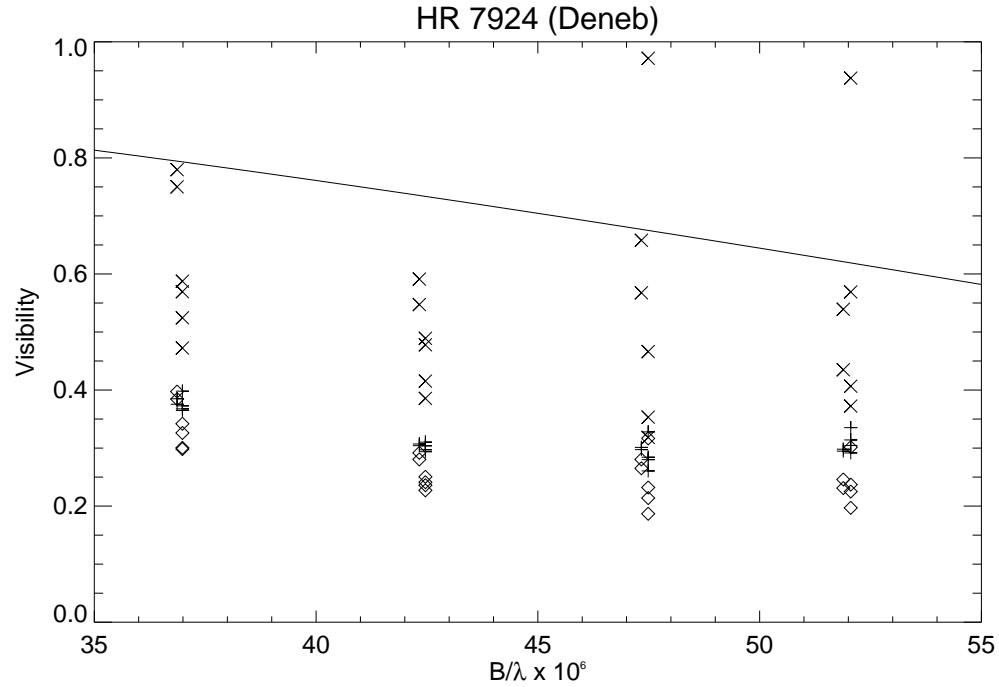


Figure 4.5: Uncalibrated Visibility vs. B/λ for HR 7924, Deneb. The solid line is the expected visibility of the star based upon uniform disk angular diameter 2.37 mas, measured with the Mark III and NPOI interferometers.

The symbols are:

- ◇: V weighted average over all sub-apertures.
- +: V for only the brightest sub-aperture.
- ×: Highest single scan V from the brightest sub-aperture.

Deneb has a higher visibility than Vega, 60-80% at the S1-S2 baseline for the VIS system, but not as many photons. Typical brightest sub-aperture mean photon counts were: 385 (641 nm), 529 (702 nm), 388 (785 nm), and 254 (901 nm). The faintest sub-apertures of the 702 nm channel stayed near 100 counts, but those of the 901 nm channel were around 60, indicating that perhaps the fainter sub-apertures are too noisy; perhaps the brightest sub-aperture visibilities are more trustworthy. The brightest sub-aperture visibilities (+) are clustered together quite tightly, especially

for the longer wavelength channels, which supports the idea that they are more accurate. The averaged sub-aperture visibilities tend to fall about 0.1 lower than the brightest sub-aperture visibilities, which is consistent with the behavior of the Vega data.

The two extremely high visibility points are the maximum single fringe packet visibilities for two spectral channels in the same dither scan, shown in Figure 4.6. Because Deneb is only supposed to have a 60% visibility in the 641 nm channel and a 65% visibility in the 702 nm channel, these fringes are a bit perplexing. The fringes are well-sampled enough that there is no doubt they are real.

4.2.5 System Visibility Estimates

The VIS system visibility may be estimated by using the known diameters of Alcaid, Vega, and Deneb. Because the uncalibrated visibility $\nu = V_{\text{sys}}V$, and V can be computed from each star's angular diameter, the system visibility is found by dividing ν/V .

Figure 4.7 shows the system visibility as a function of spectral channel wavelength, using the data from Alcaid, Vega, and Deneb. For the sake of clarity, only the visibilities averaged over all sub-apertures (\diamond) and those from the brightest sub-aperture (+) were plotted. The visibilities from the single highest visibility scan from the brightest sub-aperture filled in the range from the plotted visibilities up to 1.

The scatter in the brightest sub-aperture data probably looks worse than it is. There are 6 points in each spectral channel that are consistently much higher than the rest of the data at the two longer wavelengths, and these are from the Alcaid data, which is the noisiest. If we place less value on those points, the system visibility

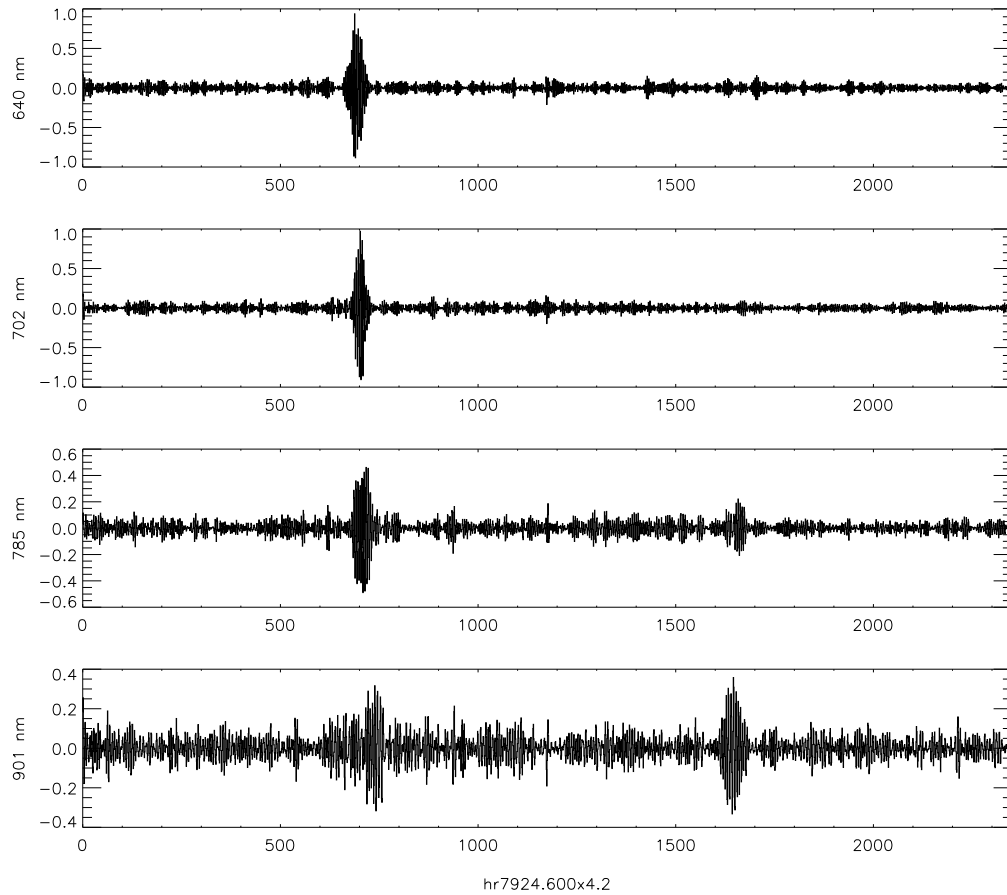


Figure 4.6: An unusually high visibility fringe scan from HR 7924, Deneb. Deneb’s visibility is only supposed to be 60% and 65% for the top two channels. The entire dither scan is shown, up and down. It is not uncommon in the data to see fringes in one direction, while the fringe packets are nearly impossible to make out in the other direction. This seems to be due to the fact that the sampling frequency is not sufficiently faster than the characteristic turbulence frequency, $1/\tau_0$.

seems nailed down to a relatively small range, regardless of which type of sub-aperture visibility data is preferred.

The VIS system visibility may be estimated to be 0.5 at 901 nm, down to 0.4 at 641 nm. These are reasonable numbers, not much smaller than the routinely

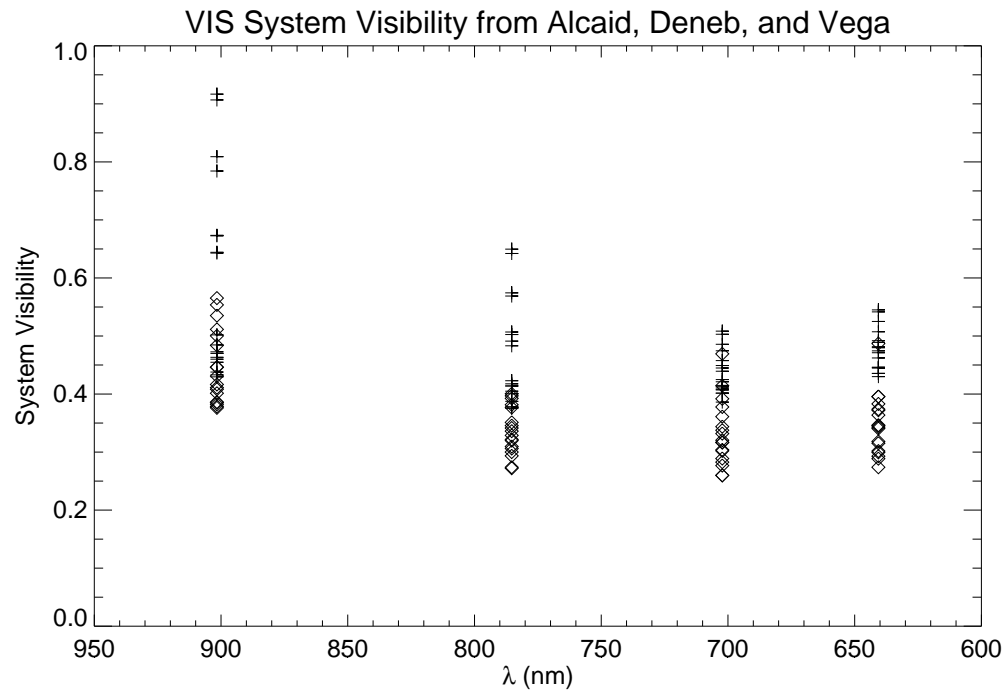


Figure 4.7: The VIS system visibility vs. λ from observations of Alcaid, Deneb, and Vega. The symbols are:

\diamond : V weighted average over all sub-apertures.

$+$: V for only the brightest sub-aperture.

measured CHARA IR system visibilities of 0.4 to 0.5.

4.3 Further Improvements

This work was only a demonstration of the VIS system's throughput efficiency and system visibility.

It has been shown that the VIS system suffers from two major drawbacks: it is too slow and too insensitive for general fringe packet scanning data collection.

There are three major ways to deal with this problem:

1. Get a faster camera. The ARC CCD camera's software is written so that it only accepts exposure times in units of whole milliseconds. While this would not generally be a problem, it limits us to running the camera at a theoretical 700 Hz while reading out only a few pixels (say, only the brightest sub-aperture spectrum.)

With bright stars like Vega (and there are not many stars that are both bright and small enough to have a visibility over 50%) the camera could be run with sub-millisecond exposures and still have enough light per exposure. With faster sampling, the entire fringe packet could be sampled in a few τ_o , which would cause less spreading of the power spectrum peak, enabling the power spectrum method of visibility estimation. It may be worthwhile to investigate how much programming work would be required to switch the ARC CCD software to handle the exposure time in units of microseconds instead. It may be as simple as adding a few multiples of 1000 here and there.

There are CCD cameras currently out on the market that make use of a multiplication register in their readout electronics, enabling them to operate as single photon counting devices, with effectively zero read noise. These cameras have approximately the same read speed per pixel as the ARC CCD, so they would have to be used on a small number of pixels, but they could be a simple way to increase the sampling rate *and* sensitivity.

2. Improve the measurement of fringe visibility by utilizing a fringe envelope tracker. When the fringe data is Fourier filtered by a windowing function in frequency space, the fringe envelope may be extracted by setting all of the neg-

ative frequencies to zero before inverse transforming the data back to the time domain. Such a treatment is known as a Hilbert transform (Bracewell, 1965). If the power spectrum is taken in the time domain of the Hilbert transformed data, it gives the fringe power at each point in the time, that is to say, the square of the fringe envelope.

The fringe envelope is immune to errors in the phase of the fringes, and is only susceptible to shifts in position due to atmospheric OPD changes. The peak of the fringe envelope may be found in each scan, and the data shifted and added to produce an averaged fringe envelope with a higher signal-to-noise. This improved fringe packet data may be used to determine the squared visibility of the fringes. It may also be averaged over only the most recent scans to produce a low-bandwidth envelope tracking signal that has better sensitivity than using the envelope from single scans. The SUSI interferometer has already demonstrated that this approach works (Tuthill et al., 2004), and with a much lower SNR!

This method should be combined with shorter dither scans to enable a more frequent sampling of the fringe envelope, and an increase in the envelope tracking bandwidth. The dither mirrors in the VIS and CHARA Classic systems have only been calibrated for hysteresis for the full $88.18\text{ }\mu\text{m}$ sweep length. It would be fairly simple to calibrate the dither mirrors for several sweep lengths, as well as for the central position of each sweep, because a sweep of a given length will probably have a different hysteresis curve when it is centered in the PZT range than when it is offset to one side. The fit coefficients for each hysteresis curve

as a function of sweep size and center position could easily be stored in a lookup table that the dither module could refer to in real-time to adaptively change the shape of the driving voltage curve to maintain linear dither sweeps.

The Hilbert transform was used by the author in troubleshooting the VIS on-sky data. The Hilbert transformed scans were added together in data where the fringe signal was too weak to discern in the individual filtered data scans. Because the envelopes add coherently, they build up a mean envelope with atmospheric piston offsets built in. As long as the mean atmospheric piston is on the order of the coherence length, the fringe envelopes overlap and the mean envelope SNR improves with the number of scans added. It proved to be a reliable method of determining if there were fringes present, and gave the offset between the VIS and K band systems due to uncompensated longitudinal dispersion.

3. Slave the VIS system to a separate fringe tracker. If the SNR of the VIS system proves too low to effectively utilize the fringe envelope tracking technique, the tracking could be done in CHARA Classic, while the VIS system scans through a dither sweep only 5 or so coherence lengths wide. By offloading the tracking offsets from CHARA Classic to the VIS dither mirror, the VIS system stays centered on the fringe packet even if it is too faint to detect in a single scan. Because CHARA Classic and the VIS system use the same model of PZT stage, the same dither sweep hysteresis curves and offsets should apply to both.

With a functioning fringe tracker running in the IR, τ_o at visible wavelengths would be greatly improved, even if the fringe tracker does not phase lock. With

a fringe tracker that truly locks on and follows a single fringe, the VIS system could step through the fringe packet as slowly as it needs to in order to collect enough photons to detect the fringes.

Piggybacking the VIS system on a fringe tracker would enable the study of objects from visible wavelengths all the way up to K' band, all done simultaneously. In addition, the VIS system enables CHARA to look at objects with a B/λ 3 to 4 times larger than with the K' band systems alone, which translates directly to a factor 3 to 4 increase in angular resolution.

The CHARA Array is poised on the brink of using 3, 4, and more telescopes simultaneously as part of standard observing. In September of 2005, the MIRC group demonstrated fringes with the CHARA Array on 4 telescopes. They are currently developing an IR fringe tracker for use with MIRC that will keep the 4 telescopes phase locked. While phase locking never works perfectly, it does greatly increase the effective τ_o .

With a phase locking fringe tracker, the VIS system could expose on each position in its dither sweep for as long as a few times the effective τ_{circ} . This would increase the VIS system's sensitivity by a factor of $\frac{\tau_o(\text{tracking})}{\tau_o}$, as this is the ratio of the number of photons detected in the tracking vs. non-tracking case.

It is the author's feeling that the next logical step in the development of the VIS system is slaving it to CHARA Classic, with CHARA Classic functioning as a fringe envelope tracker. CHARA Classic already has an envelope tracker algorithm in use, and it would be simple to improve it to use a leaky memory buffer to try to improve its fringe envelope detection. It will probably not even need to use more than one

scan at a time, because the visibility on the K band will always be higher than that of the VIS system, due to $r_o \propto \lambda^{\frac{6}{5}}$, not to mention that most sources will have higher visibility at K band (binary stars being the possible exception).

The identical dither mirrors to both systems greatly simplifies the problem of dither calibration for slaving the tracking offsets. The dither sweeps should be calibrated for shorter sweeps anyway, as improved envelope tracking in CHARA Classic already makes shorter sweeps an easy way to increase in the IR data throughput.

The next step would be to try fringe tracking with CHARA Classic and slaving the VIS dither sweep offsets to that. Theo ten Brummelaar has already been working on a phase locking algorithm, and this would be a dramatic showcase of its merits. Depending upon the bandwidth of the phase lock, it could even be used to phase up the VIS band light for use with the high resolution fiber-fed spectrograph that was built by Todd Hillwig, and sits in a state of readiness on the light source table, awaiting the day of phased visible light at CHARA.

With either fringe tracking scheme, first fringes on the S1-E1 baseline is an appealing experiment. It would triple or quadruple the world record for $\frac{B}{\lambda}$, depending upon the spectral channels in which the fringes may be found. A source that would have a high enough visibility in the VIS wave band to be detected would be unresolved in K', and hence very dim. Fringe tracking of some form will be essential for the experiment to be successful.

4.4 Conclusion

The VIS system has been shown to be very sensitive to r_o , with SNR approximately $\propto r_o^{\frac{3}{2}}$. The system visibility is acceptable, in the neighborhood of 0.4 at 650 nm to 0.5 at 900 nm. Its magnitude limit of approximately 3 at 700 nm may be greatly improved by the co-adding fringe envelopes, fringe packet scanning, slaving the VIS system to a fringe tracker, or using a photon counting CCD with zero effective read noise.

Many options are open to making CHARA's VIS system a scientifically useful subsystem. The author feels some regret about moving on from CHARA and missing out on all of the fun.

Appendix A

Diffraction Model IDL Code (on CD-ROM)

A.1 The Master Files

A.1.1 Visibility and SNR Measurement With Variable Aperture Size and Propagation Distance, Averaged Over Many Model Iterations: chara_diff_vis.pro

```
; chara_diff_vis.pro
; models diffraction using angular spectrum tools
; 2 independent instances
; Reduces beam at scope, propogates to brt, reduces beam at brt,
; propagates to beam combiner. Interferes them.
; Makes plots of pupil and frequency plane at each position.

PRO chara_diff_vis, outfile, lam, r0_550

; d1 is the distance from the telescopes to the BRTs
; d1 worst case E1-W2, vega rising.
; E1 Pop End, 90m delay: 478m
; W2 Pop 1, 0 delay: 80m
; E1-S1 similar case, but d2 is where the diff comes from.
; E1 as above,
; S1 Pop 1, 0 delay: 234m
; 250m "typical" of S1-S2

d1a = DOUBLE(478)
d1b = DOUBLE(80)

; total length of the grid in each x,y dimension

xtot = DOUBLE(8)
ytot = xtot

; number of samples in each dimension
; note that the smpling should be closer than 1.08r0 (Buscher 1988)
; and the total grid size should be > 6 Dtel (from Noll 1973)
; if tip/tilt is compensated.
; with xtot = 8, the long-wavelength criterion is satisfied.
; At this setting, here are the dx sizes for the number of samples
; in one dimension:
; 128: 6.25 cm
; 256: 3.125 cm
; 512: 1.5625 cm
; 1024: 7.8125 mm
; 2048: 3.90625 mm

nx = DOUBLE(256)
ny = nx

; number of independent cases to run

ncases = 50

; magnification factors at telescope and brt. 1/ because they're reducing.
mtel = 1/DOUBLE(8.0)
```

```

mbrt = 1/DOUBLE(6.6)

; telescope primary and secondary mirror radii in m

rprimary = DOUBLE(0.5)
rsecondary = DOUBLE(0.120)

; note when using subaperture dimensions after a beam reducer,
; multiply by that beam reducer's magnification.
; ex. subap_x at beam combiner = mtel * mbrt * subap_x.
; before brt: mbrt * subap_x.
; on primary mirror: subap_x.

; subaperture center on primary mirror in m
; the outside subapertures are 1/3 rprim in radius, centers 2/3 rprim out.

subap_y = rprimary*(2.0/3.0)
subap_x = DOUBLE(0.0)

; subaperture minimum and maximum radius on primary in m
; the max is made big enough so that the biggest size can hold the whole beam.
; rfixed is the radius of the lenses in the subaperture array.

subap_rfixed = rprimary/(3.0) ; check this. It's actually a little smaller.
subap_rmax = 2 * rprimary

; the radii array holds subaperture radii at which to calculate visibilities.
pts_per_r = 18 ; was 6
n_r_pts = pts_per_r * LONG(subap_rmax/subap_rfixed)
subap_r = DINDGEN(n_r_pts)
subap_r = subap_r + 1
subap_r = subap_r / n_r_pts
subap_r = subap_r * subap_rmax

; the station array holds distances past the brt at which to stop and calculate
; visibilities.
; the first station is for the telescope input aperture,
; the second is for the brt output
; beam combiner positions for beam 5
; scope nearest table furthest table
; S1 7.94 m 13.67 m
; S2 8.52 m 14.25 m
; W1 9.77 m 15.49 m
; W2 10.36 m 16.08 m
; E2 11.60 m 17.32 m
; E1 12.18 m 17.90 m

stationa = DOUBLE([0.0,0.0,12.18,17.90])
stationb = DOUBLE([0.0,0.0,10.36,16.08])
nstations = N_ELEMENTS(stationa)

; size in cm of ps output files
psx = 12
psy = 8

dx = xtot / nx
dy = ytot / ny

r0 = calculate_r0(r0_550,lam)
xya = 1
fxfya = 1
phase_xya = 1
phase_fxfya = 1
xyb = 1
fxfyb = 1
phase_xyb = 1
phase_fxfyb = 1

plotIntensity = 0
plotPhase = 0
kbhit = 0
make_jpg = 0
turb = 1

; make the file name additions

nstr = '.' + STRTRIM( STRING(ncases),2 ) + 'cases'
lam_str = '.' + STRTRIM( STRING( LONG(lam) ),2 ) + 'nm'
r0_str = '.ro' + STRTRIM( STRING( LONG(r0) ),2 ) + 'cm'
grid_size_str = '.m' + STRTRIM( STRING( LONG(xtot) ),2 )

```

```

grid_n_str = '.n' + STRTRIM( STRING( LONG(nx) ),2 )
grid_dx_str = '.dx' + STRTRIM( STRING( dx ),2 )
distr = '.brtA' + STRTRIM( STRING( LONG(dia) ),2 ) + $
'.brtB' + STRTRIM( STRING( LONG(dib) ),2 )
IF TURB GT 0 THEN $
turb_str = r0_str $
ELSE $
turb_str = '.plane'

outfile_name = outfile_name + lam_str + turb_str + nstr + $
grid_size_str + grid_n_str + grid_dx_str + distr

; set up the array to hold x, flux and n for each subap size, each case
; dimensions subscripts:
; 1: case. each independent generation of turbulence + diffraction propagation
; 2: station. point along the diffraction propagation where x,flux etc. were
; calculated. Diffraction distances stored in separate station[] array.
; 3: r. specific subaperture radius. subap_r[] array holds values.
; 4: |X|, |flux|, n for each point.

varr = DBLARR(ncases,nstations,n_r_pts,3)

; begin the big loop
k=0
FOR k=0, ncases-1 DO BEGIN

print, k, ' Setting up xy arrays'

; just use setup_xy, nx, ny, dx, dy if you want a plane wave
setup_xy, xya, nx, ny, dx, dy
setup_xy, xyb, nx, ny, dx, dy

IF turb EQ 1 THEN BEGIN

; instance A

; make the turbulent phase array
print, k, ' Generating turbulent phase array a'

turbulent_phase, phase_xya, phase_fxhya, r0, nx, ny, dx, dy

print, k, ' Finished generating turbulent phase array a'

; remove the tip/tilt component of the phase
; the second argument is the effective radius of the aperture to
; use for tip/tilt removal

remove_tip_tilt, phase_xya, 1.0, 1.0

phase_fxhya = update_fxhy(phase_xya)

print, k, ' Finished removing tip/tilt from phase array a'

; now apply the phase offset to the wavefront array

apply_phase, phase_xya, xya

; instance B
; make the turbulent phase array
print, k, ' Generating turbulent phase array b'

turbulent_phase, phase_xyb, phase_fxhyb, r0, nx, ny, dx, dy

print, k, ' Finished generating turbulent phase array b'

; remove the tip/tilt component of the phase
; the second argument is the effective radius of the aperture to
; use for tip/tilt removal

remove_tip_tilt, phase_xyb, 1.0, 1.0

phase_fxhyb = update_fxhy(phase_xyb)

print, k, ' Finished removing tip/tilt from phase array b'

; now apply the phase offset to the wavefront array

```



```

apply_phase, phase_xyb, xyb

ENDIF

;establish the xy offset from the beam center
; it's the coordinates of the beam center.
xyoffset = [ -xya[0,0].x, -xya[0,0].y]

;establish the frequency spectrum
print, k," Updating fxfy arrays to turbulent xy wavefronts"
fxfya = update_fxfy(xya)
fxfyb = update_fxfy(xyb)

print, k," Applying telescope aperture"
; apply the telescope aperture
xya = round_aperture(xya, rprimary, rsecondary, 0,0)
xyb = round_aperture(xyb, rprimary, rsecondary, 0,0)

; normalize the vectors so they'll integrate to 1 over the aperture.

normalize_power, xya
normalize_power, xyb

fxfya = update_fxfy(xya)
fxfyb = update_fxfy(xyb)

print, k," finished applying the telescope aperture stop:"

; calculate X and flux

x = calculate_x( xya, xyb )
flux = calculate_flux( xya, xyb )

; calculate visibilities at the input aperture. no magnification yet.
j=0

FOR j=0, n_r_pts-1 DO BEGIN

temp = vis_aperture(x,flux,subap_x, subap_y, subap_r[j])

varr[k,0,j,*] = temp

ENDFOR

; Rescale the pupil at the scopes. Beam reduction 8x.
print, k," Rescaling the pupil at telescopes"
rescale_pupil, xya, fxfya, mtel, mtel
rescale_pupil, xyb, fxfyb, mtel, mtel

; recalculate the xy offset

xyoffset = [ -xya[0,0].x, -xya[0,0].y]

print, k," rescaled the pupil at telescope:"

; calculate X and flux

x = calculate_x( xya, xyb )
flux = calculate_flux( xya, xyb )

; propagate to the brts
print, k, " Propagating to BRTs"
fxfya = propagator(fxfya,d1a,lam)
fxfyb = propagator(fxfyb,d1b,lam)

; update the pupil plane

xya = update_xy(fxfya, xyoffset)
xyb = update_xy(fxfyb, xyoffset)

print, k," Propagated to the BRTs"

; calculate X and flux

x = calculate_x( xya, xyb )
flux = calculate_flux( xya, xyb )

```

```

; rescale the pupil at the brts. Beam reduction 6.6x
print, k, " Rescaling the pupil at BRTs"
rescale_pupil, xya, fxhya, mbrr, mbrr
rescale_pupil, xyb, fxhyb, mbrr, mbrr

; recalculate the xy offset
xyoffset = [ -xya[0,0].x, -xya[0,0].y]

print, k, " Rescaled pupil at BRTs"

; calculate X and flux
x = calculate_x( xya, xyb )
flux = calculate_flux( xya, xyb )

; this is for station 1, before propagating away from the beam combiner
j=0
FOR j=0, n_r_pts-1 DO BEGIN

temp = vis_aperture(x,flux,subap_x * mtel * mbrr, $
subap_y * mtel * mbrr, subap_r[j] * mtel * mbrr)

varr[k,1,j,*] = temp

ENDFOR

; now propagate to each remaining station and calculate the visibility
l=0
FOR l=2, nstations-1 DO BEGIN

; propagate to the next station
print, k, " Propagating to station", l
fxhya = propagator(fxhya, stationa[l]-stationa[l-1], lam)
fxhyb = propagator(fxhyb, stationb[l]-stationb[l-1], lam)

; update the pupil plane
xya = update_xy(fxhya, xyoffset)
xyb = update_xy(fxhyb, xyoffset)

print,k, " Propagated to station", l

; calculate X and flux
x = calculate_x( xya, xyb )
flux = calculate_flux( xya, xyb )

j=0
FOR j=0, n_r_pts-1 DO BEGIN

temp = vis_aperture(x,flux,subap_x * mtel * mbrr, $
subap_y * mtel * mbrr, subap_r[j] * mtel * mbrr)

varr[k,l,j,*] = temp

ENDFOR

ENDFOR ; this ends the station propagation loop

ENDFOR ; this ends the big loop

; now process the data for all instances
x2avg = DBLARR(nstations, n_r_pts, 4)
fluxavg = DBLARR(nstations, n_r_pts, 4)

FOR l=0, nstations-1 DO BEGIN
FOR j=0, n_r_pts-1 DO BEGIN

; find the stats of X^2: <|X|^2>

temp = MOMENT( varr[* ,l,j,0]^2 )
x2avg[l,j,*] = temp

;find the stats of flux: <flux>

```

```

temp = MOMENT( varr[*,1,j,1] )
fluxavg[1,j,*] = temp

ENDFOR
ENDFOR

; Vrms = 2 sqrt( <|X|^2>/<F>^2)
; subscript1: station
; subscript2: r

vrms = 2.0 * SQRT( x2avg[*,*,0] / fluxavg[*,*,0]^2 )

; SNR ~ NV^2
; subscript1: station
; subscript2: r
snr = x2avg[*,*,0] / fluxavg[*,*,0]

; make plots

; flux plots
plot_vis_stats, fluxavg[*,*,0], subap_r, 'nowrite', 'Average Flux', $
'Subaperture Radius', kbhit, 0, psx, psy

; postscript plot too
plot_vis_stats, fluxavg[*,*,0], subap_r, outfilename + '.flux.ps', $
'Average Flux', 'Subaperture Radius', 0, 1, psx, psy

; vrms plots
plot_vis_stats, vrms, subap_r, 'nowrite', 'rms Visibility', $
'Subaperture Radius', kbhit, 0, psx, psy

; do the postscript plots as well
plot_vis_stats, vrms, subap_r, outfilename + '.Vrms.ps', 'rms Visibility', $
'Subaperture Radius', 0, 1, psx, psy

; SNR plots
plot_vis_stats, snr/snr[0,n_r_pts-1], subap_r, 'nowrite', 'Signal to Noise', $
'Subaperture Radius', kbhit, 0, psx, psy

; do the postscript plots as well
plot_vis_stats, snr/snr[0,n_r_pts-1], subap_r, outfilename + '.snr.ps', $
'Signal to Noise', 'Subaperture Radius', 0, 1, psx, psy

; Now the log plots

; flux plots
plot_vis_stats, ALOG10(fluxavg[*,*,0]), subap_r, 'nowrite', 'Log Average Flux', $
'Subaperture Radius', kbhit, 0, psx, psy

; postscript plot too
plot_vis_stats, ALOG10(fluxavg[*,*,0]), subap_r, outfilename + '.logflux.ps', $
'Log Average Flux', 'Subaperture Radius', 0, 1, psx, psy

; vrms plots
plot_vis_stats, ALOG10(vrms), subap_r, 'nowrite', 'Log rms Visibility', $
'Subaperture Radius', kbhit, 0, psx, psy

; do the postscript plots as well
plot_vis_stats, ALOG10(vrms), subap_r, outfilename + '.logVrms.ps', $
'Log rms Visibility', 'Subaperture Radius', 0, 1, psx, psy

; SNR plots
plot_vis_stats, ALOG10(snr/snr[0,n_r_pts-1]), subap_r, 'nowrite', $
'Log Signal to Noise', 'Subaperture Radius', kbhit, 0, psx, psy

; do the postscript plots as well
plot_vis_stats, ALOG10(snr/snr[0,n_r_pts-1]), subap_r, $
outfilename + '.logsnr.ps', 'Log Signal to Noise', $
'Subaperture Radius', 0, 1, psx, psy

; record data

; write the binary file

```

```

OPENW,1,outfilename+'.vis.bin.dat'

WRITEU, 1, LONG(N_ELEMENTS(stationa))
WRITEU, 1, stationa

WRITEU, 1, LONG(N_ELEMENTS(stationb))
WRITEU, 1, stationb

WRITEU, 1, LONG(N_ELEMENTS(subap_r))
WRITEU, 1, subap_r

WRITEU, 1, x2avg
WRITEU, 1, fluxavg
WRITEU, 1, vrms
WRITEU, 1, snr

CLOSE, 1

;write the ascii file

OPENW,1,outfilename+'.vis.ascii.dat'

PRINTF, 1, 'Elements in station arraya:', nstations
PRINTF, 1, 'Station distatnces from BRTa (m):'
PRINTF, 1, stationa
PRINTF, 1, stationb

PRINTF, 1, 'Elements in subaperture radius array:', n_r_pts
PRINTF, 1, 'Subaperture radii on primary (m) :'
PRINTF, 1, subap_r

i=0
FOR i=0, nstations-1 DO BEGIN
  PRINTF, 1, 'Station', i
  PRINTF, 1, '<|X|^2>(r_subap):'
  PRINTF, 1, x2avg[i,*,0]
  PRINTF, 1, '<Flux>(r_subap):'
  PRINTF, 1, fluxavg[i,*,0]
  PRINTF, 1, 'Vrms(r_subap):'
  PRINTF, 1, vrms[i,*,0]
  PRINTF, 1, 'SNR(r_subap):'
  PRINTF, 1, snr[i,*,0]
ENDFOR

CLOSE, 1

END

```

A.1.2 A Single Iteration of Two Beam Diffraction and Interference Through The Sytem Which Generates Jpeg Images of All Array Values at Several Propagation Distances: chara_diff2.pro

```

; chara_diff2.pro
; models diffraction using angular spectrum tools
; 2 independent instances
; Reduces beam at scope, propogates to brt, reduces beam at brt,
; propagates to beam combiner. Interferes them.
; Makes plots of pupil and frequency plane at each position.

PR0 chara_diff2, outfilename, lam, r0_550

; d1 is the distance from the telescopes to the BRTs
; d1 worst case E1-W2, vega rising.
; E1 Pop End, 90m delay: 478m
; W2 Pop 1, 0 delay: 80m
; E1-S1 similar case, but d2 is where the diff comes from.
; E1 as above,
; S1 Pop 1, 0 delay: 234m

```

```

; 250m "typical" of S1-S2

dia = DOUBLE(478)
dib = DOUBLE(80)

; d2 is the distance from the brts to the beam combiner
; beam combiner positions for beam 5
; scope      nearest table    furthest table
; S1         7.94 m           13.67 m
; S2         8.52 m           14.25 m
; W1         9.77 m           15.49 m
; W2         10.36 m          16.08 m
; E2         11.60 m          17.32 m
; E1         12.18 m          17.90 m

d2a = DOUBLE([12.18,17.90])
d2b = DOUBLE([10.36,16.08])

; total length of the grid in each x,y dimension
xtot = DOUBLE(8)
ytot = xtot

; number of samples in each dimension
; note that the smpling should be closer than 1.08r0 (Buscher 1988)
; and the total grid size should be > 6 Dtel (from Noll 1973)
; if tip/tilt is compensated.
; with xtot = 8, the long-wavelength criterion is satisfied.
; At this setting, here are the dx sizes for the number of samples
; in one dimension:
; 128: 6.25 cm
; 256: 3.125 cm
; 512: 1.5625 cm
; 1024: 7.8125 mm
; 2048: 3.90625 mm

nx = DOUBLE(512)
ny = nx

; magnification factors at telescope and brt. 1/ because they're reducing.

mtel = 1/DOUBLE(8.0)
mbrt = 1/DOUBLE(6.6)

; telescope primary and secondary mirror radii in m

rprimary = DOUBLE(0.5)
rsecondary = DOUBLE(0.120)

; length of central grid for smaller displays

inside_l = 4 * rprimary
n_inside = (inside_l / xtot) * nx

; these are the start and ending indices for subarrays which are twice the
; telescope aperture size before beam reduction. Good for displaying
; results without all of the extra real estate

inside_start = LONG(nx/2 - n_inside/2)
inside_stop = LONG(nx/2 + n_inside/2 - 1)

dx = xtot / nx
dy = ytot / ny

r0 = calculate_r0(r0_550,lam)
xya = 1
fxfyb = 1
phase_xya = 1
phase_fxgya = 1
xyb = 1
fxfyb = 1
phase_xyb = 1
phase_fxgyb = 1

plotIntensity = 1
plotPhase = 1

; wait for key hit after each plot?
kbhit = 0

```

```

; make jpegs of plots?
make_jpg = 1

; use modeled atmospheric turbulence in the input wavefronts?
turb = 0

; display the full array?
disp_full = 1

; display the smaller inside array?
disp_small = 1

; make the file name additions

lam_str = '.' + STRTRIM( STRING( LONG(lam) ),2 ) + 'nm'
r0_str = '.ro' + STRTRIM( STRING( LONG(r0) ),2 ) + 'cm'
grid_size_str = '.m' + STRTRIM( STRING( LONG(xtot) ),2 )
grid_n_str = '.n' + STRTRIM( STRING( LONG(nx) ),2 )
grid_dx_str = '.dx' + STRTRIM( STRING( dx ),2 ) + '$
distr = '.brtA' + STRTRIM( STRING( LONG(dia) ),2 ) + $
'.brtB' + STRTRIM( STRING( LONG(dib) ),2 )
IF TURB GT 0 THEN $
turb_str = r0_str $
ELSE $
turb_str = '.plane'

outfile_name = outfile_name + lam_str + turb_str + grid_size_str + grid_n_str + $
grid_dx_str + distr

instr = '.in' + STRTRIM( STRING( LONG(inside_1) ),2 ) + 'm'

; just use setup_xy, nx, ny, dx, dy if you want a plane wave
setup_xy, xya, nx, ny, dx, dy
setup_xy, xyb, nx, ny, dx, dy

IF turb EQ 1 THEN BEGIN

; instance A

; make the turbulent phase array

turbulent_phase, phase_xya, phase_fxhya, r0, nx, ny, dx, dy

print, 'Finished generating turbulent phase array'

; plot the real component of the phase array
; make plots
; plot_phase, xy
print, 'plotting phase_xya real'
IF disp_full GT 0 THEN $
plot_real, phase_xya, outfile_name + '.rawturb.phasexya.real' $
, kbhit, make_jpg, 0
IF disp_small GT 0 THEN $
plot_real, phase_xya[inside_start:inside_stop, $
inside_start:inside_stop], $
outfile_name + '.rawturb.phasexya.real'+instr $
, kbhit, make_jpg, 0
print, 'plotting phase_xya log real'
IF disp_full GT 0 THEN $
plot_log_real, phase_xya, outfile_name + $
'.rawturb.phasexya.logreal' , kbhit, make_jpg
IF disp_small GT 0 THEN $
plot_log_real, phase_xya[inside_start:inside_stop, $
inside_start:inside_stop], $
outfile_name + '.rawturb.phasexya.logreal'+instr $
, kbhit, make_jpg

freq_disp_prep, phase_fxhya
print, 'plotting phase_fxhya real'
IF disp_full GT 0 THEN $
plot_real, phase_fxhya, outfile_name + '.rawturb.phasefxhya.real' $
, kbhit, make_jpg, 0
IF disp_small GT 0 THEN $
plot_real, phase_fxhya[inside_start:inside_stop, $
inside_start:inside_stop], $
outfile_name + '.rawturb.phasefxhya.real'+instr $
, kbhit, make_jpg, 0

```

```

print,'plotting phase_fxfsa log real'
IF disp_full GT 0 THEN $
    plot_log_real, phase_fxfsa, outfilename + $
    'rawturb.phasefxfsa.logreal', kbhit, make_jpg
IF disp_small GT 0 THEN $
    plot_log_real, phase_fxfsa[inside_start:inside_stop, $
    inside_start:inside_stop], $
    outfilename + 'rawturb.phasefxfsa.logreal'+instr $
    , kbhit, make_jpg
freq_calc_prep, phase_fxfsa

; remove the tip/tilt component of the phase
; the second argument is the effective radius of the aperture to
; use for tip/tilt removal

remove_tip_tilt, phase_xya, 1.0, 1.0

phase_fxfsa = update_fxfsa(phase_xya)

print, 'Finished removing tip/tilt from phase array'

; plot the real component of the phase array
;make plots
;plot_phase, xy
print,'plotting phase_xya real'
IF disp_full GT 0 THEN $
    plot_real, phase_xya, outfilename + '.tt_turb.phasexya.real' $
    , kbhit, make_jpg, 0
IF disp_small GT 0 THEN $
    plot_real, phase_xya[inside_start:inside_stop, $
    inside_start:inside_stop], $
    outfilename + '.tt_turb.phasexya.real'+instr $
    , kbhit, make_jpg, 0
print,'plotting phase_xya log real'
IF disp_full GT 0 THEN $
    plot_log_real, phase_xya, outfilename + $
    '.tt_turb.phasexya.logreal', kbhit, make_jpg
IF disp_small GT 0 THEN $
    plot_log_real, phase_xya[inside_start:inside_stop, $
    inside_start:inside_stop], $
    outfilename + '.tt_turb.phasexya.logreal'+instr $
    , kbhit, make_jpg

freq_disp_prep, phase_fxfsa
print,'plotting phase_fxfsa real'
IF disp_full GT 0 THEN $
    plot_real, phase_fxfsa, outfilename + '.tt_turb.phasefxfsa.real' $
    , kbhit, make_jpg, 0
IF disp_small GT 0 THEN $
    plot_real, phase_fxfsa[inside_start:inside_stop, $
    inside_start:inside_stop], $
    outfilename + '.tt_turb.phasefxfsa.real'+instr $
    , kbhit, make_jpg, 0
print,'plotting phase_fxfsa log real'
IF disp_full GT 0 THEN $
    plot_log_real, phase_fxfsa, outfilename + $
    '.tt_turb.phasefxfsa.logreal', kbhit, make_jpg
IF disp_small GT 0 THEN $
    plot_log_real, phase_fxfsa[inside_start:inside_stop, $
    inside_start:inside_stop], $
    outfilename + '.tt_turb.phasefxfsa.logreal'+instr $
    , kbhit, make_jpg
freq_calc_prep, phase_fxfsa

; now apply the phase offset to the wavefront array

apply_phase, phase_xya, xya

; instance B
; make the turbulent phase array

turbulent_phase, phase_xyb, phase_fxfsb, r0, nx, ny, dx, dy

print,'Finished generating turbulent phase array'

; plot the real component of the phase array
;make plots
;plot_phase, xy

print,'plotting phase_xyb real'

```

```

IF disp_full GT 0 THEN $
    plot_real, phase_xyb, outfilename + '.rawturb.phasexyb.real' $
    , kbhit, make_jpg, 0
IF disp_small GT 0 THEN $
    plot_real, phase_xyb[inside_start:inside_stop, $
        inside_start:inside_stop], $
        outfilename + '.rawturb.phasexyb.real'+instr $
        , kbhit, make_jpg, 0
print, 'plotting phase_xyb log real'
IF disp_full GT 0 THEN $
    plot_log_real, phase_xyb, outfilename + $
    '.rawturb.phasexyb.logreal' , kbhit, make_jpg
IF disp_small GT 0 THEN $
    plot_log_real, phase_xyb[inside_start:inside_stop, $
        inside_start:inside_stop], $
        outfilename + '.rawturb.phasexyb.logreal'+instr $
        , kbhit, make_jpg

freq_disp_prep, phase_fxzyb
print, 'plotting phase_fxzyb real'
IF disp_full GT 0 THEN $
    plot_real, phase_fxzyb, outfilename + '.rawturb.phasefxzyb.real' $
    , kbhit, make_jpg, 0
IF disp_small GT 0 THEN $
    plot_real, phase_fxzyb[inside_start:inside_stop, $
        inside_start:inside_stop], $
        outfilename + '.rawturb.phasefxzyb.real'+instr $
        , kbhit, make_jpg, 0
print, 'plotting phase_fxzyb log real'
IF disp_full GT 0 THEN $
    plot_log_real, phase_fxzyb, outfilename + $
    '.rawturb.phasefxzyb.logreal' , kbhit, make_jpg
IF disp_small GT 0 THEN $
    plot_log_real, phase_fxzyb[inside_start:inside_stop, $
        inside_start:inside_stop], $
        outfilename + '.rawturb.phasefxzyb.logreal'+instr $
        , kbhit, make_jpg
freq_calc_prep, phase_fxzyb

; remove the tip/tilt component of the phase
; the second argument is the effective radius of the aperture to
; use for tip/tilt removal

remove_tip_tilt, phase_xyb, 1.0, 1.0

phase_fxzyb = update_fxzyb(phase_xyb)

print, 'Finished removing tip/tilt from phase array'

; plot the real component of the phase array
; make plots
; plot phase, xy

print, 'plotting phase_xyb real'
IF disp_full GT 0 THEN $
    plot_real, phase_xyb, outfilename + '.tt_turb.phasexyb.real' $
    , kbhit, make_jpg, 0
IF disp_small GT 0 THEN $
    plot_real, phase_xyb[inside_start:inside_stop, $
        inside_start:inside_stop], $
        outfilename + '.tt_turb.phasexyb.real'+instr $
        , kbhit, make_jpg, 0
print, 'plotting phase_xyb log real'
IF disp_full GT 0 THEN $
    plot_log_real, phase_xyb, outfilename + $
    '.tt_turb.phasexyb.logreal' , kbhit, make_jpg
IF disp_small GT 0 THEN $
    plot_log_real, phase_xyb[inside_start:inside_stop, $
        inside_start:inside_stop], $
        outfilename + '.tt_turb.phasexyb.logreal'+instr $
        , kbhit, make_jpg

freq_disp_prep, phase_fxzyb
print, 'plotting phase_fxzyb real'
IF disp_full GT 0 THEN $
    plot_real, phase_fxzyb, outfilename + '.tt_turb.phasefxzyb.real' $
    , kbhit, make_jpg, 0
IF disp_small GT 0 THEN $
    plot_real, phase_fxzyb[inside_start:inside_stop, $
        inside_start:inside_stop], $

```



```

                                outfilename + '.tt_turb.phasefxfyb.real'+instr $
                                , kbhit, make_jpg, 0
print,'plotting phase_fxfyb log real'
IF disp_full GT 0 THEN $
    plot_log_real, phase_fxfyb, outfilename + $
    '.tt_turb.phasefxfyb.logreal' , kbhit, make_jpg
IF disp_small GT 0 THEN $
    plot_log_real, phase_fxfyb[inside_start:inside_stop, $
    inside_start:inside_stop], $
    outfilename + '.tt_turb.phasefxfyb.logreal'+instr $
    , kbhit, make_jpg
freq_calc_prep, phase_fxfyb

; now apply the phase offset to the wavefront array
apply_phase, phase_xyb, xyb
ENDIF

;establish the xy offset from the beam center
; it's the coordinates of the beam center.
xyoffset = [ -xya[0,0].x, -xya[0,0].y]

;establish the frequency spectrum
fxfy_a = update_fxfy(xya)
fxfy_b = update_fxfy(xyb)

print, "finished making arrays:"
;array_print, xy, 'xy_start'
;array_print, fxfy, 'fxfy_start'

;print, "testing freq_disp_prep"
;freq_disp_prep, fxfy
;array_print, fxfy, 'fxfy_disp'

;print, "testing freq_calc_prep"
;freq_calc_prep, fxfy
;array_print, fxfy, 'fxfy_calc'

;make plots
IF plotPhase EQ 1 THEN BEGIN
print,'plotting xya phase'
IF disp_full GT 0 THEN $
    plot_phase, xya, outfilename + '.pretel.xya.phase' $
    ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_phase, xya[inside_start:inside_stop, $
        inside_start:inside_stop], $
        outfilename + '.pretel.xya.phase' +instr $
        ,kbhit, make_jpg
print,'plotting xyb phase'
IF disp_full GT 0 THEN $
    plot_phase, xyb, outfilename + '.pretel.xyb.phase' $
    ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_phase, xyb[inside_start:inside_stop, $
        inside_start:inside_stop], $
        outfilename + '.pretel.xyb.phase' +instr $
        ,kbhit, make_jpg
ENDIF

IF plotIntensity EQ 1 THEN BEGIN
print,'plotting xya intensity'
IF disp_full GT 0 THEN $
    plot_intensity, xya,outfilename+'.pretel.xya.I',kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_intensity, xya[inside_start:inside_stop, $
        inside_start:inside_stop], $
        outfilename + '.pretel.xya.I' +instr, kbhit, make_jpg
print,'plotting xyb intensity'
IF disp_full GT 0 THEN $
    plot_intensity, xyb,outfilename+'.pretel.xyb.I',kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_intensity, xyb[inside_start:inside_stop, $
        inside_start:inside_stop], $
        outfilename + '.pretel.xyb.I' +instr, kbhit, make_jpg
print,'plotting xya log intensity'
IF disp_full GT 0 THEN $
    plot_log_intensity, xya, outfilename + '.pretel.xya.logI'$

```

```

, kbhit, make_jpg
IF disp_small GT 0 THEN $
    plot_log_intensity, xya[inside_start:inside_stop, $
        inside_start:inside_stop], $
        outfilename + '.pretel.xya.logI' +instr, kbhit, make_jpg
print, 'plotting xyb log intensity'
IF disp_full GT 0 THEN $
    plot_log_intensity, xyb, outfilename + '.pretel.xyb.logI'$
    , kbhit, make_jpg
IF disp_small GT 0 THEN $
    plot_log_intensity, xyb[inside_start:inside_stop, $
        inside_start:inside_stop], $
        outfilename + '.pretel.xyb.logI' +instr, kbhit, make_jpg
freq_disp_prep, fxhya
freq_disp_prep, fxhyb
print, 'plotting fxhya intensity'
IF disp_full GT 0 THEN $
    plot_intensity, fxhya, outfilename + '.pretel.fxhya.I' $
, kbhit, make_jpg
IF disp_small GT 0 THEN $
    plot_intensity, fxhya[inside_start:inside_stop, $
        inside_start:inside_stop], $
        outfilename + '.pretel.fxhya.I' +instr, kbhit, make_jpg
print, 'plotting fxhyb intensity'
IF disp_full GT 0 THEN $
    plot_intensity, fxhyb, outfilename + '.pretel.fxhyb.I' $
, kbhit, make_jpg
IF disp_small GT 0 THEN $
    plot_intensity, fxhyb[inside_start:inside_stop, $
        inside_start:inside_stop], $
        outfilename + '.pretel.fxhyb.I' +instr, kbhit, make_jpg
print, 'plotting fxhya log intensity'
IF disp_full GT 0 THEN $
    plot_log_intensity, fxhya, outfilename + '.pretel.fxhya.logI'$
    , kbhit, make_jpg
IF disp_small GT 0 THEN $
    plot_log_intensity, fxhya[inside_start:inside_stop, $
        inside_start:inside_stop], $
        outfilename + '.pretel.fxhya.logI' +instr, $
kbhit, make_jpg
print, 'plotting fxhyb log intensity'
IF disp_full GT 0 THEN $
    plot_log_intensity, fxhyb, outfilename + '.pretel.fxhyb.logI'$
    , kbhit, make_jpg
IF disp_small GT 0 THEN $
    plot_log_intensity, fxhyb[inside_start:inside_stop, $
        inside_start:inside_stop], $
        outfilename + '.pretel.fxhyb.logI' +instr, $
kbhit, make_jpg
freq_calc_prep, fxhya
freq_calc_prep, fxhyb
ENDIF

; apply the telescope aperture
xya = round_aperture(xya, rprimary, rsecondary, 0,0)
fxhya = update_fxhy(xya)
xyb = round_aperture(xyb, rprimary, rsecondary, 0,0)
fxhyb = update_fxhy(xyb)

print, "finished applying the telescope aperture stop:"
;array_print, xy, 'xy_tel_in'
;array_print, fxhy, 'fxhy_tel_in'

; calculate X and flux

x = calculate_x( xya, xyb )
flux = calculate_flux( xya, xyb )
v = flux
v.vec = 2*ABS(x.vec)/flux.vec
cleanup_nan, v.vec
PRINT, 'V max:', MAX(v.vec)

print, 'plotting flux'
plot_real, flux, outfilename + '.telin.flux', kbhit, make_jpg, 0
; plot just the inner part
plot_real, flux[inside_start:inside_stop, inside_start:inside_stop], $
outfilename + '.telin.flux'+instr, kbhit, make_jpg, 0

print, 'plotting V'

```

```

plot_real, v, outfilename + '.telin.V', kbhit, make_jpg, 1
; plot just the inner part
plot_real,v[inside_start:inside_stop,inside_start:inside_stop],$
outfilename + '.telin.V'+instr, kbhit, make_jpg, 1

freq_disp_prep, fxhya
freq_disp_prep, fxhyb

x_f = calculate_X( fxhya, fxhyb )
flux_f = calculate_flux( fxhya, fxhyb )
v_f = flux_f
v_f.vec = 2*ABS(x_f.vec)/flux_f.vec
cleanup_nan, v_f.vec
PRINT, 'V_f max:', MAX(v_f.vec)

print, 'plotting flux_f'
plot_real, flux_f, outfilename + '.telin.flux_f', kbhit, make_jpg, 0
; plot just the inner part
plot_real,flux_f[inside_start:inside_stop,inside_start:inside_stop],$
outfilename + '.telin.flux_f'+ instr, kbhit, make_jpg, 0

print, 'plotting V_f'
plot_real, v_f, outfilename + '.telin.V_f', kbhit, make_jpg, 1
; plot just the inner part
plot_real,v_f[inside_start:inside_stop,inside_start:inside_stop],$
outfilename + '.telin.V_f'+instr, kbhit, make_jpg, 1

;make plots
IF plotPhase EQ 1 THEN BEGIN
print,'plotting xya phase'
IF disp_full GT 0 THEN $
    plot_phase, xya, outfilename + '.telin.xya.phase' $
    ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_phase, xya[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.telin.xya.phase' +instr $
            ,kbhit, make_jpg
print,'plotting xyb phase'
IF disp_full GT 0 THEN $
    plot_phase, xyb, outfilename + '.telin.xyb.phase' $
    ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_phase, xyb[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.telin.xyb.phase' +instr $
            ,kbhit, make_jpg
print,'plotting x phase'
IF disp_full GT 0 THEN $
    plot_phase, x, outfilename + '.telin.x.phase' $
    ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_phase, x[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.telin.x.phase' +instr $
            ,kbhit, make_jpg
print,'plotting fxhya phase'
IF disp_full GT 0 THEN $
    plot_phase, fxhya, outfilename + '.telin.fxhya.phase' $
    ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_phase, fxhya[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.telin.fxhya.phase' +instr $
            ,kbhit, make_jpg
print,'plotting fxhyb phase'
IF disp_full GT 0 THEN $
    plot_phase, fxhyb, outfilename + '.telin.fxhyb.phase' $
    ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_phase, fxhyb[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.telin.fxhyb.phase' +instr $
            ,kbhit, make_jpg
print,'plotting x_f phase'
IF disp_full GT 0 THEN $
    plot_phase, x_f, outfilename + '.telin.x_f.phase' $
    ,kbhit, make_jpg
    IF disp_small GT 0 THEN $

```

```

        plot_phase, x_f[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.telin.x_f.phase' +instr $
            ,kbhit, make_jpg
ENDIF

IF plotIntensity EQ 1 THEN BEGIN
print,'plotting xya intensity'
    IF disp_full GT 0 THEN $
        plot_intensity, xya,outfilename+'.telin.xya.I',kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_intensity, xya[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.telin.xya.I' +instr, kbhit, make_jpg
print,'plotting xyb intensity'
    IF disp_full GT 0 THEN $
        plot_intensity, xyb,outfilename+'.telin.xyb.I',kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_intensity, xyb[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.telin.xyb.I' +instr, kbhit, make_jpg
print,'plotting xya log intensity'
    IF disp_full GT 0 THEN $
        plot_log_intensity, xya, outfilename + '.telin.xya.logI'$
            ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_log_intensity, xya[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.telin.xya.logI' +instr, kbhit, make_jpg
print,'plotting xyb log intensity'
    IF disp_full GT 0 THEN $
        plot_log_intensity, xyb, outfilename + '.telin.xyb.logI'$
            ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_log_intensity, xyb[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.telin.xyb.logI' +instr, kbhit, make_jpg
print,'plotting fxhya intensity'
    IF disp_full GT 0 THEN $
        plot_intensity, fxhya,outfilename+'.telin.fxhya.I' $
            ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_intensity, fxhya[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.telin.fxhya.I' +instr, kbhit, make_jpg
print,'plotting fxhyb intensity'
    IF disp_full GT 0 THEN $
        plot_intensity, fxhyb,outfilename+'.telin.fxhyb.I' $
            ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_intensity, fxhyb[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.telin.fxhyb.I' +instr, kbhit, make_jpg
print,'plotting fxhya log intensity'
    IF disp_full GT 0 THEN $
        plot_log_intensity, fxhya, outfilename + '.telin.fxhya.logI'$
            ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_log_intensity, fxhya[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.telin.fxhya.logI' +instr, $
            kbhit, make_jpg
print,'plotting fxhyb log intensity'
    IF disp_full GT 0 THEN $
        plot_log_intensity, fxhyb, outfilename + '.telin.fxhyb.logI'$
            ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_log_intensity, fxhyb[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.telin.fxhyb.logI' +instr, $
            kbhit, make_jpg
ENDIF
freq_calc_prep, fxhya
freq_calc_prep, fxhyb

; Rescale the pupil at the scopes. Beam reduction 8x.

rescale_pupil, xya, fxhya, mtel, mtel
rescale_pupil, xyb, fxhyb, mtel, mtel

```

```

; recalculate the xy offset
xyoffset = [ -xya[0,0].x, -xya[0,0].y]

print, "rescaled the pupil at telescope:"
;array_print, xy, 'xy_tel_out'
;array_print, fxfy, 'xfxy_tel_out'

; calculate X and flux
x = calculate_X( xya, xyb )
flux = calculate_flux( xya, xyb )
v = flux
v.vec = 2*ABS(x.vec)/flux.vec
cleanup_nan, v.vec

PRINT, 'V max:', MAX(v.vec)

print, 'plotting flux'
plot_real, flux, outfilename + '.telout.flux', kbhit, make_jpg, 0
; plot just the inner part
plot_real, flux[inside_start:inside_stop, inside_start:inside_stop], $
outfilename + '.telout.flux'+instr, kbhit, make_jpg, 0

print, 'plotting V'
plot_real, v, outfilename + '.telout.V', kbhit, make_jpg, 1
; plot just the inner part
plot_real, v[inside_start:inside_stop, inside_start:inside_stop], $
outfilename + '.telout.V'+instr, kbhit, make_jpg, 1

freq_disp_prep, fxfya
freq_disp_prep, fxfyb

x_f = calculate_X( fxfya, fxfyb )
flux_f = calculate_flux( fxfya, fxfyb )
v_f = flux_f
v_f.vec = 2*ABS(x_f.vec)/flux_f.vec
cleanup_nan, v_f.vec
PRINT, 'V_f max:', MAX(v_f.vec)

print, 'plotting flux_f'
plot_real, flux_f, outfilename + '.telout.flux_f', kbhit, make_jpg, 0
; plot just the inner part
plot_real, flux_f[inside_start:inside_stop, inside_start:inside_stop], $
outfilename + '.telout.flux_f'+instr, kbhit, make_jpg, 0

print, 'plotting V_f'
plot_real, v_f, outfilename + '.telout.V_f', kbhit, make_jpg, 1
; plot just the inner part
plot_real, v_f[inside_start:inside_stop, inside_start:inside_stop], $
outfilename + '.telout.V_f'+instr, kbhit, make_jpg, 1

;make plots
IF plotPhase EQ 1 THEN BEGIN
print, 'plotting xya phase'
IF disp_full GT 0 THEN $
    plot_phase, xya, outfilename + '.telout.xya.phase' $
    , kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_phase, xya[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.telout.xya.phase' +instr $
            , kbhit, make_jpg
print, 'plotting xyb phase'
IF disp_full GT 0 THEN $
    plot_phase, xyb, outfilename + '.telout.xyb.phase' $
    , kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_phase, xyb[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.telout.xyb.phase' +instr $
            , kbhit, make_jpg
print, 'plotting x phase'
IF disp_full GT 0 THEN $
    plot_phase, x, outfilename + '.telout.x.phase' $
    , kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_phase, x[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.telout.x.phase' +instr $

```

```

,kbhit, make_jpg
print,'plotting fxhya phase'
IF disp_full GT 0 THEN $
    plot_phase, fxhya, outfilename + '.telout.fxhya.phase' $
    ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_phase, fxhya[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.telout.fxhya.phase' +instr $
            ,kbhit, make_jpg
print,'plotting fxhyb phase'
IF disp_full GT 0 THEN $
    plot_phase, fxhyb, outfilename + '.telout.fxhyb.phase' $
    ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_phase, fxhyb[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.telout.fxhyb.phase' +instr $
            ,kbhit, make_jpg
print,'plotting x_f phase'
IF disp_full GT 0 THEN $
    plot_phase, x_f, outfilename + '.telout.x_f.phase' $
    ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_phase, x_f[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.telout.x_f.phase' +instr $
            ,kbhit, make_jpg
ENDIF

IF plotIntensity EQ 1 THEN BEGIN
print,'plotting xya intensity'
    IF disp_full GT 0 THEN $
        plot_intensity, xya,outfilename+'.telout.xya.I',kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_intensity, xya[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.telout.xya.I' +instr, kbhit, make_jpg
print,'plotting xyb intensity'
    IF disp_full GT 0 THEN $
        plot_intensity, xyb,outfilename+'.telout.xyb.I',kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_intensity, xyb[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.telout.xyb.I' +instr, kbhit, make_jpg
print,'plotting xya log intensity'
    IF disp_full GT 0 THEN $
        plot_log_intensity, xya, outfilename + '.telout.xya.logI'$
        ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_log_intensity, xya[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.telout.xya.logI' +instr, kbhit, make_jpg
print,'plotting xyb log intensity'
    IF disp_full GT 0 THEN $
        plot_log_intensity, xyb, outfilename + '.telout.xyb.logI'$
        ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_log_intensity, xyb[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.telout.xyb.logI' +instr, kbhit, make_jpg
print,'plotting fxhya intensity'
    IF disp_full GT 0 THEN $
        plot_intensity, fxhya,outfilename+'.telout.fxhya.I' $
        ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_intensity, fxhya[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.telout.fxhya.I' +instr, kbhit, make_jpg
print,'plotting fxhyb intensity'
    IF disp_full GT 0 THEN $
        plot_intensity, fxhyb,outfilename+'.telout.fxhyb.I' $
        ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_intensity, fxhyb[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.telout.fxhyb.I' +instr, kbhit, make_jpg
print,'plotting fxhya log intensity'
    IF disp_full GT 0 THEN $
        plot_log_intensity, fxhya, outfilename + '.telout.fxhya.logI'$

```

```

                                ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_log_intensity, fxfya[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.telout.fxfya.logI' +instr, $
kbhit, make_jpg
print, 'plotting fxfyb log intensity'
    IF disp_full GT 0 THEN $
        plot_log_intensity, fxfyb, outfilename + '.telout.fxfyb.logI' $
            ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_log_intensity, fxfyb[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.telout.fxfyb.logI' +instr, $

kbhit, make_jpg
ENDIF
freq_calc_prep, fxfya
freq_calc_prep, fxfyb

; propagate to the brts

fxfya = propagator(fxfya,dia,lam)
fxfyb = propagator(fxfyb,dib,lam)

; update the pupil plane

xya = update_xy(fxfya, xyoffset)
xyb = update_xy(fxfyb, xyoffset)

print, "propagated to the brts"
;array_print, xy, 'xy_brt_in'
;array_print, fxfy, 'fxfy_brt_in'

; calculate X and flux

x = calculate_X( xya, xyb )
flux = calculate_flux( xya, xyb )
v = flux
v.vec = 2*ABS(x.vec)/flux.vec
cleanup_nan, v.vec
PRINT, 'V max:', MAX(v.vec)

print, 'plotting flux'
plot_real, flux, outfilename + '.brtin.flux', kbhit, make_jpg, 0
; plot just the inner part
plot_real, flux[inside_start:inside_stop,inside_start:inside_stop], $
outfilename + '.brtin.flux'+ instr, kbhit, make_jpg, 0
print, 'plotting V'
plot_real, v, outfilename + '.brtin.V', kbhit, make_jpg, 1
; plot just the inner part
plot_real, v[inside_start:inside_stop,inside_start:inside_stop], $
outfilename + '.brtin.V'+instr, kbhit, make_jpg, 1

freq_disp_prep, fxfya
freq_disp_prep, fxfyb

x_f = calculate_X( fxfya, fxfyb )
flux_f = calculate_flux( fxfya, fxfyb )
v_f = flux_f
v_f.vec = 2*ABS(x_f.vec)/flux_f.vec
cleanup_nan, v_f.vec
PRINT, 'V_f max:', MAX(v_f.vec)

print, 'plotting flux_f'
plot_real, flux_f, outfilename + '.brtin.flux_f', kbhit, make_jpg, 0
; plot just the inner part
plot_real, flux_f[inside_start:inside_stop,inside_start:inside_stop], $
outfilename + '.brtin.flux_f'+ instr, kbhit, make_jpg, 0

print, 'plotting V_f'
plot_real, v_f, outfilename + '.brtin.V_f', kbhit, make_jpg, 1
; plot just the inner part
plot_real, v_f[inside_start:inside_stop,inside_start:inside_stop], $
outfilename + '.brtin.V_f'+instr, kbhit, make_jpg, 1

;make plots
IF plotPhase EQ 1 THEN BEGIN
print, 'plotting xya phase'
IF disp_full GT 0 THEN $
    plot_phase, xya, outfilename + '.brtin.xya.phase' $

```

```

, kbhit, make_jpg
IF disp_small GT 0 THEN $
  plot_phase, xya[inside_start:inside_stop, $
    inside_start:inside_stop], $
    outfilename + '.brtin.xya.phase' +instr $
    , kbhit, make_jpg
print, 'plotting xyb phase'
IF disp_full GT 0 THEN $
  plot_phase, xyb, outfilename + '.brtin.xyb.phase' $
    , kbhit, make_jpg
IF disp_small GT 0 THEN $
  plot_phase, xyb[inside_start:inside_stop, $
    inside_start:inside_stop], $
    outfilename + '.brtin.xyb.phase' +instr $
    , kbhit, make_jpg
print, 'plotting x phase'
IF disp_full GT 0 THEN $
  plot_phase, x, outfilename + '.brtin.x.phase' $
    , kbhit, make_jpg
IF disp_small GT 0 THEN $
  plot_phase, x[inside_start:inside_stop, $
    inside_start:inside_stop], $
    outfilename + '.brtin.x.phase' +instr $
    , kbhit, make_jpg
print, 'plotting fxfya phase'
IF disp_full GT 0 THEN $
  plot_phase, fxfya, outfilename + '.brtin.fxfya.phase' $
    , kbhit, make_jpg
IF disp_small GT 0 THEN $
  plot_phase, fxfya[inside_start:inside_stop, $
    inside_start:inside_stop], $
    outfilename + '.brtin.fxfya.phase' +instr $
    , kbhit, make_jpg
print, 'plotting fxfyb phase'
IF disp_full GT 0 THEN $
  plot_phase, fxfyb, outfilename + '.brtin.fxfyb.phase' $
    , kbhit, make_jpg
IF disp_small GT 0 THEN $
  plot_phase, fxfyb[inside_start:inside_stop, $
    inside_start:inside_stop], $
    outfilename + '.brtin.fxfyb.phase' +instr $
    , kbhit, make_jpg
print, 'plotting x_f phase'
IF disp_full GT 0 THEN $
  plot_phase, x_f, outfilename + '.brtin.x_f.phase' $
    , kbhit, make_jpg
IF disp_small GT 0 THEN $
  plot_phase, x_f[inside_start:inside_stop, $
    inside_start:inside_stop], $
    outfilename + '.brtin.x_f.phase' +instr $
    , kbhit, make_jpg
ENDIF

IF plotIntensity EQ 1 THEN BEGIN
print, 'plotting xya intensity'
  IF disp_full GT 0 THEN $
    plot_intensity, xya, outfilename + '.brtin.xya.I', kbhit, make_jpg
  IF disp_small GT 0 THEN $
    plot_intensity, xya[inside_start:inside_stop, $
      inside_start:inside_stop], $
      outfilename + '.brtin.xya.I' +instr, kbhit, make_jpg
print, 'plotting xyb intensity'
  IF disp_full GT 0 THEN $
    plot_intensity, xyb, outfilename + '.brtin.xyb.I', kbhit, make_jpg
  IF disp_small GT 0 THEN $
    plot_intensity, xyb[inside_start:inside_stop, $
      inside_start:inside_stop], $
      outfilename + '.brtin.xyb.I' +instr, kbhit, make_jpg
print, 'plotting xya log intensity'
  IF disp_full GT 0 THEN $
    plot_log_intensity, xya, outfilename + '.brtin.xya.logI'$
      , kbhit, make_jpg
  IF disp_small GT 0 THEN $
    plot_log_intensity, xya[inside_start:inside_stop, $
      inside_start:inside_stop], $
      outfilename + '.brtin.xya.logI' +instr, kbhit, make_jpg
print, 'plotting xyb log intensity'
  IF disp_full GT 0 THEN $
    plot_log_intensity, xyb, outfilename + '.brtin.xyb.logI'$
      , kbhit, make_jpg

```



```

        IF disp_small GT 0 THEN $
            plot_log_intensity, xyb[inside_start:inside_stop, $
                inside_start:inside_stop], $
                outfilename + '.brtin.xyb.logI' +instr, kbhit, make_jpg
print,'plotting fxfya intensity'
        IF disp_full GT 0 THEN $
            plot_intensity, fxfya,outfilename+'.brtin.fxfya.I' $
,kbhit, make_jpg
        IF disp_small GT 0 THEN $
            plot_intensity, fxfya[inside_start:inside_stop, $
                inside_start:inside_stop], $
                outfilename + '.brtin.fxfya.I' +instr, kbhit, make_jpg
print,'plotting fxfyb intensity'
        IF disp_full GT 0 THEN $
            plot_intensity, fxfyb,outfilename+'.brtin.fxfyb.I' $
,kbhit, make_jpg
        IF disp_small GT 0 THEN $
            plot_intensity, fxfyb[inside_start:inside_stop, $
                inside_start:inside_stop], $
                outfilename + '.brtin.fxfyb.I' +instr, kbhit, make_jpg
print,'plotting fxfya log intensity'
        IF disp_full GT 0 THEN $
            plot_log_intensity, fxfya, outfilename + '.brtin.fxfya.logI'$
,kbhit, make_jpg
        IF disp_small GT 0 THEN $
            plot_log_intensity, fxfya[inside_start:inside_stop, $
                inside_start:inside_stop], $
                outfilename + '.brtin.fxfya.logI' +instr, $
kbhit, make_jpg
print,'plotting fxfyb log intensity'
        IF disp_full GT 0 THEN $
            plot_log_intensity, fxfyb, outfilename + '.brtin.fxfyb.logI'$
,kbhit, make_jpg
        IF disp_small GT 0 THEN $
            plot_log_intensity, fxfyb[inside_start:inside_stop, $
                inside_start:inside_stop], $
                outfilename + '.brtin.fxfyb.logI' +instr, $
kbhit, make_jpg
ENDIF
freq_calc_prep, fxfya
freq_calc_prep, fxfyb

; rescale the pupil at the brts. Beam reduction 6.6x

rescale_pupil, xya, fxfya, mbrr, mbrr
rescale_pupil, xyb, fxfyb, mbrr, mbrr

; recalculate the xy offset

xyoffset = [ -xya[0,0].x, -xya[0,0].y]

print, "recaled beam at brts"
;array_print, xy,'xy_brt_out'
;array_print, fxfy,'fxfy_brt_out'

; calculate X and flux

x = calculate_X( xya, xyb )
flux = calculate_flux( xya, xyb )
v = flux
v.vec = 2*ABS(x.vec)/flux.vec
cleanup_nan, v.vec
PRINT, 'V max:', MAX(v.vec)

print, 'plotting flux'
plot_real, flux, outfilename + '.brtout.flux', kbhit, make_jpg, 0
; plot just thie inner part
plot_real,flux[inside_start:inside_stop,inside_start:inside_stop],$
outfilename + '.brtout.flux'+ instr, kbhit, make_jpg, 0

print, 'plotting V'
plot_real, v, outfilename + '.brtout.V', kbhit, make_jpg, 1
; plot just thie inner part
plot_real,v[inside_start:inside_stop,inside_start:inside_stop],$
outfilename + '.brtout.V'+instr, kbhit, make_jpg, 1

freq_disp_prep, fxfya
freq_disp_prep, fxfyb

```

```

x_f = calculate_X( fxfya, fxfyb )
flux_f = calculate_flux( fxfya, fxfyb )
v_f = flux_f
v_f.vec = 2*ABS(x_f.vec)/flux_f.vec
cleanup_nan, v_f.vec
PRINT, 'V_f max:', MAX(v_f.vec)

print, 'plotting flux_f'
plot_real, flux_f, outfilename + '.brtout.flux_f', kbhit, make_jpg, 0
; plot just the inner part
plot_real, flux_f[inside_start:inside_stop, inside_start:inside_stop], $
outfilename + '.brtout.flux_f' + instr, kbhit, make_jpg, 0

print, 'plotting V_f'
plot_real, v_f, outfilename + '.brtout.V_f', kbhit, make_jpg, 1
; plot just the inner part
plot_real, v_f[inside_start:inside_stop, inside_start:inside_stop], $
outfilename + '.brtout.V_f' + instr, kbhit, make_jpg, 1

;make plots
IF plotPhase EQ 1 THEN BEGIN
print, 'plotting xya phase'
IF disp_full GT 0 THEN $
    plot_phase, xya, outfilename + '.brtout.xya.phase' $
    , kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_phase, xya[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.brtout.xya.phase' + instr $
            , kbhit, make_jpg
print, 'plotting xyb phase'
IF disp_full GT 0 THEN $
    plot_phase, xyb, outfilename + '.brtout.xyb.phase' $
    , kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_phase, xyb[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.brtout.xyb.phase' + instr $
            , kbhit, make_jpg
print, 'plotting x phase'
IF disp_full GT 0 THEN $
    plot_phase, x, outfilename + '.brtout.x.phase' $
    , kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_phase, x[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.brtout.x.phase' + instr $
            , kbhit, make_jpg
print, 'plotting fxfya phase'
IF disp_full GT 0 THEN $
    plot_phase, fxfya, outfilename + '.brtout.fxfya.phase' $
    , kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_phase, fxfya[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.brtout.fxfya.phase' + instr $
            , kbhit, make_jpg
print, 'plotting fxfyb phase'
IF disp_full GT 0 THEN $
    plot_phase, fxfyb, outfilename + '.brtout.fxfyb.phase' $
    , kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_phase, fxfyb[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.brtout.fxfyb.phase' + instr $
            , kbhit, make_jpg
print, 'plotting x_f phase'
IF disp_full GT 0 THEN $
    plot_phase, x_f, outfilename + '.brtout.x_f.phase' $
    , kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_phase, x_f[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + '.brtout.x_f.phase' + instr $
            , kbhit, make_jpg
ENDIF

IF plotIntensity EQ 1 THEN BEGIN
print, 'plotting xya intensity'

```

```

        IF disp_full GT 0 THEN $
            plot_intensity, xya,outfilename+'.brtout.xya.I',kbhit, make_jpg
        IF disp_small GT 0 THEN $
            plot_intensity, xya[inside_start:inside_stop, $
                inside_start:inside_stop], $
                outfilename + '.brtout.xya.I' +instr, kbhit, make_jpg
print,'plotting xyb intensity'
        IF disp_full GT 0 THEN $
            plot_intensity, xyb,outfilename+'.brtout.xyb.I',kbhit, make_jpg
        IF disp_small GT 0 THEN $
            plot_intensity, xyb[inside_start:inside_stop, $
                inside_start:inside_stop], $
                outfilename + '.brtout.xyb.I' +instr, kbhit, make_jpg
print,'plotting xya log intensity'
        IF disp_full GT 0 THEN $
            plot_log_intensity, xya, outfilename + '.brtout.xya.logI'$
            ,kbhit, make_jpg
        IF disp_small GT 0 THEN $
            plot_log_intensity, xya[inside_start:inside_stop, $
                inside_start:inside_stop], $
                outfilename + '.brtout.xya.logI' +instr, kbhit, make_jpg
print,'plotting xyb log intensity'
        IF disp_full GT 0 THEN $
            plot_log_intensity, xyb, outfilename + '.brtout.xyb.logI'$
            ,kbhit, make_jpg
        IF disp_small GT 0 THEN $
            plot_log_intensity, xyb[inside_start:inside_stop, $
                inside_start:inside_stop], $
                outfilename + '.brtout.xyb.logI' +instr, kbhit, make_jpg
print,'plotting fxfya intensity'
        IF disp_full GT 0 THEN $
            plot_intensity, fxfya,outfilename+'.brtout.fxfya.I' $
            ,kbhit, make_jpg
        IF disp_small GT 0 THEN $
            plot_intensity, fxfya[inside_start:inside_stop, $
                inside_start:inside_stop], $
                outfilename + '.brtout.fxfya.I' +instr, kbhit, make_jpg
print,'plotting fxfyb intensity'
        IF disp_full GT 0 THEN $
            plot_intensity, fxfyb,outfilename+'.brtout.fxfyb.I' $
            ,kbhit, make_jpg
        IF disp_small GT 0 THEN $
            plot_intensity, fxfyb[inside_start:inside_stop, $
                inside_start:inside_stop], $
                outfilename + '.brtout.fxfyb.I' +instr, kbhit, make_jpg
print,'plotting fxfya log intensity'
        IF disp_full GT 0 THEN $
            plot_log_intensity, fxfya, outfilename + '.brtout.fxfya.logI'$
            ,kbhit, make_jpg
        IF disp_small GT 0 THEN $
            plot_log_intensity, fxfya[inside_start:inside_stop, $
                inside_start:inside_stop], $
                outfilename + '.brtout.fxfya.logI' +instr, $
            kbhit, make_jpg
print,'plotting fxfyb log intensity'
        IF disp_full GT 0 THEN $
            plot_log_intensity, fxfyb, outfilename + '.brtout.fxfyb.logI'$
            ,kbhit, make_jpg
        IF disp_small GT 0 THEN $
            plot_log_intensity, fxfyb[inside_start:inside_stop, $
                inside_start:inside_stop], $
                outfilename + '.brtout.fxfyb.logI' +instr, $
            kbhit, make_jpg
ENDIF
freq_calc_prep, fxfya
freq_calc_prep, fxfyb

; propagate to the beam combiner positions
FOR i=0, N_ELEMENTS(d2a)-1 DO BEGIN

d2str = '.bcA' + STRTRIM( STRING( LONG(d2a[i]) ),2 ) + $
'.bcB' + STRTRIM( STRING( LONG(d2b[i]) ),2 )

fxfya = propagator(fxfya,d2a[i],lam)
fxfyb = propagator(fxfyb,d2b[i],lam)

; update the pupil plane

xya = update_xy(fxfya, xyoffset)

```

```

xyb = update_xy(fxfyb, xyoffset)

print,"propagated to the beam combiner position :", d2a[i],d2b[i]
;array_print, xy,'xy_bc'
;array_print, fxfy,'xfyb'

; calculate X and flux

x = calculate_X( xya, xyb )
flux = calculate_flux( xya, xyb )
v = flux
v.vec = 2*ABS(x.vec)/flux.vec
cleanup_nan, v.vec
PRINT, 'V max:', MAX(v.vec)

freq_disp_prep, fxfya
freq_disp_prep, fxfyb

x_f = calculate_X( fxfya, fxfyb )
flux_f = calculate_flux( fxfya, fxfyb )
v_f = flux_f
v_f.vec = 2*ABS(x_f.vec)/flux_f.vec
cleanup_nan, v_f.vec
PRINT, 'V_f max:', MAX(v_f.vec)

print, 'plotting flux'
plot_real, flux, outfilename + d2str+ '.bc.flux', kbhit, make_jpg, 0
; plot just the inner part
plot_real,flux[inside_start:inside_stop,inside_start:inside_stop],$
outfilename + d2str+'.bc.flux'+ instr, kbhit, make_jpg, 0

print, 'plotting V'
plot_real, v, outfilename + d2str+'.bc.V', kbhit, make_jpg, 1
; plot just the inner part
plot_real,v[inside_start:inside_stop,inside_start:inside_stop],$
outfilename + d2str+ '.bc.V'+instr, kbhit, make_jpg, 1

print, 'plotting flux_f'
plot_real, flux_f, outfilename + d2str+ '.bc.flux_f', kbhit, make_jpg, 0
; plot just the inner part
plot_real,flux_f[inside_start:inside_stop,inside_start:inside_stop],$
outfilename + d2str+'.bc.flux_f'+ instr, kbhit, make_jpg, 0

print, 'plotting V_f'
plot_real, v_f, outfilename + d2str+'.bc.V_f', kbhit, make_jpg, 1
; plot just the inner part
plot_real,v_f[inside_start:inside_stop,inside_start:inside_stop],$
outfilename + d2str+ '.bc.V_f'+instr, kbhit, make_jpg, 1

; make plots
IF plotPhase EQ 1 THEN BEGIN
print,'plotting xya phase'
IF disp_full GT 0 THEN $
    plot_phase, xya, outfilename + d2str+ '.bc.xya.phase' $
    ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_phase, xya[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + d2str+'.bc.xya.phase' +instr $
            ,kbhit, make_jpg
print,'plotting xyb phase'
IF disp_full GT 0 THEN $
    plot_phase, xyb, outfilename + d2str+'.bc.xyb.phase' $
    ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_phase, xyb[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + d2str+'.bc.xyb.phase' +instr $
            ,kbhit, make_jpg
print,'plotting x phase'
IF disp_full GT 0 THEN $
    plot_phase, x, outfilename + d2str+'.bc.x.phase' $
    ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_phase, x[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + d2str+'.bc.x.phase' +instr $
            ,kbhit, make_jpg
print,'plotting fxfya phase'
IF disp_full GT 0 THEN $

```

```

        plot_phase, fxhya, outfilename + d2str+ '.bc.fxhya.phase' $
        ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_phase, fxhya[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + d2str+'.bc.fxhya.phase' +instr $
            ,kbhit, make_jpg
print,'plotting fxhyb phase'
IF disp_full GT 0 THEN $
    plot_phase, fxhyb, outfilename + d2str+'.bc.fxhyb.phase' $
    ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_phase, fxhyb[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + d2str+'.bc.fxhyb.phase' +instr $
            ,kbhit, make_jpg
print,'plotting x_f phase'
IF disp_full GT 0 THEN $
    plot_phase, x_f, outfilename + d2str+'.bc.x_f.phase' $
    ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_phase, x_f[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + d2str+'.bc.x_f.phase' +instr $
            ,kbhit, make_jpg
ENDIF

IF plotIntensity EQ 1 THEN BEGIN
print,'plotting xya intensity'
    IF disp_full GT 0 THEN $
        plot_intensity, xya, outfilename+d2str+'.bc.xya.I', $
        kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_intensity, xya[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + d2str+'.bc.xya.I' +instr, kbhit, make_jpg
print,'plotting xyb intensity'
    IF disp_full GT 0 THEN $
        plot_intensity, xyb, outfilename+d2str+'.bc.xyb.I', $
        kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_intensity, xyb[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + d2str+'.bc.xyb.I' +instr, kbhit, make_jpg
print,'plotting xya log intensity'
    IF disp_full GT 0 THEN $
        plot_log_intensity, xya, outfilename + d2str+'.bc.xya.logI'$
        ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_log_intensity, xya[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + d2str+'.bc.xya.logI' +instr, $
        kbhit, make_jpg
print,'plotting xyb log intensity'
    IF disp_full GT 0 THEN $
        plot_log_intensity, xyb, outfilename + d2str+'.bc.xyb.logI'$
        ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_log_intensity, xyb[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + d2str+'.bc.xyb.logI' +instr, $
        kbhit, make_jpg
print,'plotting fxhya intensity'
    IF disp_full GT 0 THEN $
        plot_intensity, fxhya, outfilename+d2str+'.bc.fxhya.I' $
        ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_intensity, fxhya[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + d2str+'.bc.fxhya.I' +instr, $
        kbhit, make_jpg
print,'plotting fxhyb intensity'
    IF disp_full GT 0 THEN $
        plot_intensity, fxhyb, outfilename+d2str+'.bc.fxhyb.I' $
        ,kbhit, make_jpg
    IF disp_small GT 0 THEN $
        plot_intensity, fxhyb[inside_start:inside_stop, $
            inside_start:inside_stop], $
            outfilename + d2str+'.bc.fxhyb.I' +instr, $
        kbhit, make_jpg

```

```

print,'plotting fxfya log intensity'
  IF disp_full GT 0 THEN $
    plot_log_intensity, fxfya, outfilename + d2str+'.bc.fxfya.logI'$
    ,kbhit, make_jpg
  IF disp_small GT 0 THEN $
    plot_log_intensity, fxfya[inside_start:inside_stop, $
      inside_start:inside_stop], $
      outfilename + d2str+'.bc.fxfya.logI' +instr, $
kbhit, make_jpg
print,'plotting fxfyb log intensity'
  IF disp_full GT 0 THEN $
    plot_log_intensity, fxfyb, outfilename + d2str+'.bc.fxfyb.logI'$
    ,kbhit, make_jpg
  IF disp_small GT 0 THEN $
    plot_log_intensity, fxfyb[inside_start:inside_stop, $
      inside_start:inside_stop], $
      outfilename + d2str+'.bc.fxfyb.logI' +instr, $
kbhit, make_jpg
ENDIF
freq_calc_prep, fxfya
freq_calc_prep, fxfyb

ENDFOR ; this ends the loop which steps through different d2 positions

END

```

A.2 Building the Turbulent Wavefront

A.2.1 Generate the Turbulent Phase Array: turbulent_phase.pro

```

;generate a simulated wavefront perturbed by atmospheric turbulence
; creates white noise in frequency space, filters it with the
; appropriate structure function, then ifft to xy space
; to obtain the perturbed wavefront

PRO turbulent_phase, xy, fxfy, r0_cm, nx, ny, dx, dy

r0_m = r0_cm/100.0

      ;setup the array. this is in xy space
      setup_xy, xy, nx, ny, dx, dy

; get to frequency space
fxfy = update_fxfy(xy)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This was when I was trying to generate the power spectrum
; in freq space first. Now I think it's better to generate the
; frequencies themselves, and make sure they have a power spectrum
; which fits  $\Phi(s) = 0.229 r_0^{(-5/3)} s^{(-11/3)}$ 

      ; generate random complex vectors with mean magnitude of 1
; add the exp part to give the vectors random phases.
; I don't think this is necessary, because we're building the
; angular power spectrum which is the IFT of
; the phase structure function.

      fxfy.vec = $
; DCOMPLEX( $
; 2.0 * RANDOMU(undef,nx,ny,/DOUBLE) $
; ,0)
; * $
      ;exp(DCOMPLEX( 0, 2*PI*RANDOMU(undef,nx,ny,/DOUBLE) ))

      ; filter these to fit  $\Phi(s)=0.0229 r_0^{(-5/3)} s^{(-11/3)}$ 

      fxfy.vec = fxfy.vec * 0.0229 * r0_m^{(-5.0/3.0)} * $
      (sqrt( fxfy.x^2 + fxfy.y^2 ))^{(-11.0/3.0)}

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This generates the frequencies, and their power spectrum
; should fit  $\Phi(s) = 0.229 r_0^{(-5/3)} s^{(-11/3)}$ 
; That means that if frequencies are generated with magnitude 1
; and random phase, their average value will come out to 1.

```

```

; Then multiply them by the square root of Phi(s).
; On top of that, we actually want their magnitudes to be random, but still
; come out with <magnitude^2> = 1.
; This works if their magnitudes are randomly distributed
; in the range 0..3^(1/3). Proof in the comments to follow.
; The average value of <f(s) f*(s)> should come out to 1 times Phi(s).

; generate random complex vectors with mean magnitude^2 of 1.
; add the exp part to give the vectors random phases.

; <x^2> = 1 is what we want.
; <x^2> = integral from 0 to x' of x'^2 dx
; = evaluation from 0 to x' of 1/3 x'^3 = 1/3 x'^3 - 0
; set = 1 and solve for x':
; x' = 3^(1/3) = 1.44224957...

; So if we give the vectors random amplitudes from 0 to 3^(1/3),
; the average value of their amplitudes squared will be 1.
; Give them random phases on top of that, and their
; <amplitude^2> remains unchanged.

fxfy.vec = 3.0^(1.0/3.0) * RANDOMU(undef,nx,ny,/DOUBLE) * $
exp(DCOMPLEX( 0, 2.0*!DPI*RANDOMU(undef,nx,ny,/DOUBLE) ))

; filter these to fit
; (Phi(s))^(1/2) = ( 0.0229 r0^(-5/3) s^(-11/3) )^(1/2)

; The factor of 10 and the 0.0257 are coarse empirical kludges I
; had to throw in to make the structure function of
; phi(r) fit the theoretical structure function
; of Dphi(r) = 6.88(r/r0)^(5/3)

fxfy.vec = fxfy.vec * 10 *$
sqrt( $
0.0257 $
; 0.0229 $
* r0_m^(-5.0/3.0) * $
(sqrt( fxfy.x^2 + fxfy.y^2 ))^( $
-12.0/3.0 $
; -11.0/3.0 $
) $
);this ends the square root around Phi(s)

; Buscher used 0.0190 r0^(-2) s^(-4)

; fxfy.vec = fxfy.vec * 10 *$
; sqrt( $
; 0.0190 * ( r0_m * $
; ( fxfy.x^2 + fxfy.y^2 ) $
; )^(-2) $
; );this ends the square root around Phi(s)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; set the 0 frequency to zero.
fxfy[0,0].vec = DOUBLE(0)

; inverse transform to get to xy space
xyoffset = [ -xy[0,0].x, -xy[0,0].y]

xy = update_xy(fxfy,xyoffset)

; at this point real(xy) is the phase offset due to turbulence,
; not the wavefront itself.

RETURN
END

```

A.2.2 Remove Tip/Tilt: remove_tip_tilt.pro

```

; remove_tip_tilt.pro
;
; Finds the best fit plane to the real component of phase array xy
; in a square centered on xy's origin
; where xy.x and xy.y = 0.
; the square is sqrt(pi) * r (that's ~ 1.77 r) to a side. That's so it has

```

```

; the same area as a circle of radius r.
; This is because the function calls SFIT(), which only takes rectangular
; arrays and doesn't know how to ignore any of the array data points.
; I figure that a square array of the wavefront with the same area as the
; aperture will have the statistically closest atmospheric behavior possible
; to the round aperture itself.
;
; Once the best fit plane is found, it is multiplied by the efficiency
; and subtracted from the wavefront. This is so you can leave in residual
; tip/tilt if you like. If you want all possible tip/tilt removed,
; simply set the efficiency to 1.

PRO remove_tip_tilt, phase_xy, radius, efficiency

PRINT, 'Beginning remove_tip_tilt procedure.'

phase_size = SIZE(phase_xy.vec,/DIMENSIONS)
phase_nx = phase_size[0]
phase_ny = phase_size[1]

PRINT, 'Phase array size: ', phase_size

phase_subscript = WHERE(phase_xy.x EQ 0 AND phase_xy.y EQ 0)

phase_origin = [ phase_subscript MOD phase_ny, $
LONG(phase_subscript)/LONG(phase_nx)]

PRINT, 'Phase array origin found at index: ', phase_origin

side = radius * sqrt(!DPI)

; we want a subarray centered at x,y = 0 and with sides
; a = sqrt(pi) r
; the array index should then be a/dx away from the index of the
; origin point.
; That is to say a/2 in either direction from the origin.
; We're assuming that the xy array has dx=dy and is square.

index_width = LONG( (sqrt(!DPI)*radius/2.0)/phase_xy[0,0].dx )

PRINT, 'Subarray index width:', index_width

subarray = phase_xy[ $
phase_origin[0]-index_width:phase_origin[0]+index_width, $
phase_origin[1]-index_width:phase_origin[1]+index_width $
].vec

PRINT, 'From', phase_origin[0] - index_width, ' to', $
phase_origin[0] + index_width

fitarr= SFIT(subarray,1,kx = fit_coeffs)

PRINT, 'Fit Coeffs: mx = ', fit_coeffs[2], ' my = ', fit_coeffs[1]

;fit coefficients are ordered a little strange to me.
; [const, y]
; [x, xy]
; These coefficients assume that the array origin is at index [0,0]
; and that dx,dy = 1. In this coordinate system,
; the data point's x,y value is it's array index number.
; The shift in origin isn't a problem, that just changes the
; constant offset, and we don't worry about that when
; removing tip/tilt. The x and y slopes will remain unchanged.

phase_xy.vec = phase_xy.vec - efficiency * $
( $
fit_coeffs[1] * phase_xy.y / phase_xy[0,0].dy + $
fit_coeffs[2] * phase_xy.x / phase_xy[0,0].dx $
)

; sanity check. Comment this out when the routine is proven to work.
; do a fit again, and see if the coefficients are basically zero.

subarray = phase_xy[ $
phase_origin[0]-index_width:phase_origin[0]+index_width, $
phase_origin[1]-index_width:phase_origin[1]+index_width $
].vec

fitarr = SFIT(subarray,1,kx = check_coeffs)

PRINT, 'Check Fit Coeffs: mx = ', check_coeffs[2], $

```



```
' my = ', check_coeffs[1]
```

```
RETURN
END
```

A.2.3 Apply the Phase Offset Array to the Wavefront Array: apply_phase.pro

```
; apply_phase.pro
; assumes that the real component of phase.vec array is the phase
; which must be applied to wavefront.vec array.
```

```
PRO apply_phase, phase, wavefront
```

```
; let wavefront.vec = A e^(i phi0)
; where A is the complex vector amplitude,
; and phi0 is the phase which that vector has to start with.
; We want to add phi1 to this vector's phase.
; A e^(i (phi0+phi1)) = A e^(i phi0) e^(i phi1)
; and e^(i phi1) = cos(phi1) + i sin(phi1)
```

```
wavefront.vec = wavefront.vec * DCOMPLEX( $
; real component of e^(i phase) to be added
; the DOUBLE call inside is so that we only take
; the real component of phase.vec.
; An imaginary component of phase will cause
; a decay in amplitude, ie absorption.
COS( DOUBLE( phase.vec ) ), $
; imaginary component of e^(i phase) to be added
SIN( DOUBLE( phase.vec ) ) $
) ; this ends the DCOMPLEX function call
```

```
RETURN
END
```

A.3 Propagating the Wavefront Through the System

A.3.1 Apply the Telescope Aperture to the Wavefront: round_aperture.pro

```
; The roundaperture function is a transfer function in xy space. It simply
; sets to zero anything outside of r_outer or inside r_inner centered on
; the point (xoffset,yoffset).
; The round aperture can also be applied as a spatial filter in the fx,fy plane.
```

```
FUNCTION round_aperture, xypoint, r_outer, r_inner, xoffset, yoffset
```

```
    tempxy = xypoint
```

```
; Set to zero if outside of aperture
; aka if (x-xoffset)^2 + (y-yoffset)^2 GT r^2
```

```
    test = (tempxy.x - xoffset)^2 + (tempxy.y - yoffset)^2
```

```
i=0
j=0
FOR i=0, N_ELEMENTS(tempxy[*,0].x)-1 DO BEGIN
FOR j=0, N_ELEMENTS(tempxy[0,*].y)-1 DO BEGIN
; PRINT, 'x:', tempxy[i,j].x, 'y:', tempxy[i,j].y $
; , 'test:', test[i,j], 'r^2:', r^2
; this sets vector values to zero outside r from xoff,yoff
IF (test[i,j] GT r_outer^2) OR (test[i,j] LT r_inner^2) THEN $
tempxy[i,j].vec = DCOMPLEX(0.0)
```

```
ENDFOR ; this ends the j loop
ENDFOR ; this ends the i loop
```

```
    RETURN, tempxy
```

```
END
```

A.3.2 Rescale the Arrays at a Beam Reducer: rescale_pupil.pro

```
; rescale_pupil is for expanding/reducing the beam in the xy plane.
; This causes
; an reduction/expansion of the beam's angular spectrum by the same factor.
; This is based on the property of fourier transforms:
; IF F(fx) = FT(f(x))
; f(x) <--> F(fx)
; f(ax) <--> 1/a F(fx/a)
; 2d beam expansion by factor of a in x, b in y:
; f(ax,by) <--> 1/(ab) F(fx/a,fy/b)
; 2d beam reduction by factor of a in x, b in y:
; f(x/a,y/b) <--> ab F(afx, bfy)
; funny, but this didn't work. I ended up using parseval's theorem
; to insure that the integrated power before and after were the same.
; that's where the sqrt(a*b) stuff below comes from.

PRO rescale_pupil, xy, fxfy, a, b

    xy.x = xy.x * a
    xy.y = xy.y * b

xy.vec = xy.vec /SQRT(a*b)

    fxfy.x = fxfy.x / a
    fxfy.y = fxfy.y /b

    fxfy.vec = fxfy.vec * SQRT(a*b)

; what about dx and dy!

xy.dx = xy.dx*a
xy.dy = xy.dy*b

fxfy.dx = fxfy.dx / a
fxfy.dy = fxfy.dy / b

END
```

A.3.3 Propagate the f_x, f_y Array a Given Distance: propagator.pro

```
; The propagator is a transfer function in angular spectrum space.
; Multiply an angular spectrum by the propagator to get the angular spectrum
; at a point z further along the optical axis.

FUNCTION propagator, fxfypoint, z, lam

; convert lam to units of meters
; it is assumed that lambda was given in nm.

lam_m = DOUBLE(lam)/1000000000

    tempxy = fxfypoint
; Phase added to each point is 2pi z/lam sqrt(1-(lam fx)^2 - (lam fy^2))
; and for all (lam fx)^2 + (lam fy)^2 >= 1 propagator = 0

    fxfyradius_sqrd = lam_m^2 * ((tempxy.x)^2 + (tempxy.y)^2)

    phase = 2.0 * !DPI * z / lam_m * SQRT(1 - fxfyradius_sqrd)

i=0
j=0
FOR i=0, N_ELEMENTS(tempxy[,0]).x-1 DO BEGIN
FOR j=0, N_ELEMENTS(tempxy[0,*]).y-1 DO BEGIN

    IF fxfyradius_sqrd[i,j] GE 1.0 THEN BEGIN
        tempxy[i,j].vec = 0.0
    ENDIF ELSE BEGIN
        tempxy[i,j].vec = tempxy[i,j].vec * $
DCOMPLEX(COS(phase[i,j]),SIN(phase[i,j]))
    ENDELSE
ENDFOR ; this ends the j loop
ENDFOR ; this ends the i loop
```

```

;          tempxy.total_phase = fxfypoint.total_phase + phase
RETURN, tempxy
END

```

A.4 Visibility and SNR Measurement

A.4.1 Calculate X and Flux in a Given Aperture Size and Position For Two Wavefront Arrays:

vis_aperture.pro

```

; vis_aperture.pro
;
; Finds the value of X and Flux integrated over the aperture.
; For 2 complex functions A and B, X = A B*, Flux = A A* + B B*
; Visibility = 2|X|/F
; Since you may need to figure out Visibility using averages of
; Flux and X, this function returns an array of three values:
; [X,Flux,npoints]
; as input, the function needs 2 arrays of field points:
; xarr: array whose .vec values are pre-calculated X values
; fluxarr: array whose .vec values are the flux.
; use calculae_flux and calculate_x procedures to generate
; these arrays beforehand.
; make sure they have the same dimensions, x, y, dx and dy values.
;
; integrates in an aperture of radius r, centered at x0, y0
; all lengths in meters.

FUNCTION vis_aperture, xarr, fluxarr, x0, y0, r

arrsz = SIZE(xarr,/DIMENSIONS)
nx = arrsz[0]
ny = arrsz[1]
i=0
j=0
npoints=0
xtot = DCOMPLEX(0.0)
fluxtot = DCOMPLEX(0.0)

test = ((xarr.x - x0)^2 + (xarr.y - y0)^2)

FOR i=0, nx-1 DO BEGIN
FOR j=0, ny-1 DO BEGIN

IF test[i,j] LE r^2 THEN BEGIN

xtot = xtot + xarr[i,j].vec
fluxtot = fluxtot + fluxarr[i,j].vec
npoints = npoints + 1
ENDIF

ENDFOR
ENDFOR

; the actual integral is int(A B* dx dy), similarly for flux

xint = xtot * xarr[0,0].dx * xarr[0,0].dy
fluxint = fluxtot * fluxarr[0,0].dx * fluxarr[0,0].dy

retarr = [ABS(xint), ABS(fluxint),DOUBLE(npoints)]

RETURN, retarr

END

```

A.5 General Utilities

A.5.1 Array Setup:

setup_xy.pro

```

; setup_xy sets up nx x ny array of field_point objects and
; stes the x,y positions such that they are centered about 0,0

PRO setup_xy, xy, nx, ny, dx, dy

FP = {field_point, x:DOUBLE(0.0), y:DOUBLE(0.0), vec:DCOMPLEX(1.0), $
      total_phase:DOUBLE(0.0) , dx:DOUBLE(dx), dy:DOUBLE(dy) }

      xy = REPLICATE(FP, nx, ny)

      temp = DINDGEN(nx)
;subtract half of nx
      temp = temp - LONG((nx-1)/2)
      temp = temp * dx

i=0
j=0

      FOR i=0, nx-1 DO $
          xy[i,*].x = temp[i]

      temp = DINDGEN(ny)
      temp = temp - LONG((ny-1)/2)
      temp = temp * dy

      FOR j=0, ny-1 DO $
          xy[* ,j].y = temp[j]

; debug messages
; PRINT, "setup_xy: Finished setting up xy. Size:",SIZE(xy)
; PRINT, "setup_xy: xy[0,0]=",xy[0,0]
RETURN
END

```

A.5.2 Update the f_x, f_y Plane After Changing the x, y Plane:

update_fxfy.pro

```

; Update_xy returns an array of field points which is the fourier transform
; of the field point array xypoints. Use this to update your angular frequency
; points after doing work in the xy plane. Has to apply shift and scaling
; to fxfy after the FFT since the FFT format assumes
; that the data starts at 0,0 and goes to m,n.

FUNCTION update_fxfy, xypoints

    fxfytemp = xypoints

; The FFT is going to assume that the data starts at 0,0

    fxfytemp.vec = FFT(xypoints.vec)

    fxfylen = SIZE(fxfytemp.vec,/DIMENSIONS)

; the scale in freq space (width of frequency bins) is
; 1/(N*dx) where dx is the sampling interval in x space. Same for y.

    fxfytemp.dx = 1 / (xypoints[0].dx * DOUBLE(fxfylen[0]))
    fxfytemp.dy = 1 / (xypoints[0].dy * DOUBLE(fxfylen[1]))

; Now set the fx,fy values for each field point. Scale point
; separations to reflect that the xy data had spacing of dx,dy
; rather than 1,1 as assumed in the fft.

i=0
j=0

    FOR i=0, fxfylen[0]-1 DO BEGIN
    FOR j=0, fxfylen[1]-1 DO BEGIN

```

```

        IF i LE fxfylen[0]/2 THEN BEGIN $
            fxfytemp[i,*].x = DOUBLE(i) * fxfytemp[0,0].dx
        ENDIF ELSE BEGIN $
            fxfytemp[i,*].x = DOUBLE(- fxfylen[0] + i) * $
            fxfytemp[0,0].dx
        ENDELSE

        IF j LE fxfylen[1]/2 THEN BEGIN $
            fxfytemp[* ,j].y = DOUBLE(j) * fxfytemp[0,0].dy
        ENDIF ELSE BEGIN $
            fxfytemp[* ,j].y = DOUBLE(- fxfylen[1] + j) * $
            fxfytemp[0,0].dy
        ENDELSE

    ENDFOR ; end j loop
    ENDFOR ; end i loop

    ; after setting the fx, fy values, the field value also must be scaled.
    ; remember f(ax) <--> 1/|a| F(fx/a)
    ; f(ax, by) <--> 1/|ab| F(fx/a, fy/b)
    ; Note that with noscaling, if dx = 1, dfx = 1/N. dx=a, dfx = 1/(N*a)
; in this case I want to divide by a & b

    ; fxfytemp.vec = fxfytemp.vec $
; / (xypoints[0].dx * xypoints[0].dy)

; try mult. by sqrt(nx/2 ny/2) from conservation of power...
; this is a kludge, but has been tested and gives the same result
; as scaling the power using Parseval's Theorem as below.

fxfytemp.vec = fxfytemp.vec * $
SQRT(fxfylen[0]*fxfylen[1])/2

; apply Parseval's Theorem. Equal Power in both xy and freq space.

print,'Scaling fxfy power to xy'

scale_power, fxfytemp, xypoints

RETURN, fxfytemp
END

```

A.5.3 Update the x, y Plane After Changing the f_x, f_y Plane: update_xy.pro

```

; Update_xy returns an array of field points which is the fourier transform
; of the field point array fxfypoints. Use this to update your pupil plane
; points after you have done some work in angular spectrum space.
; xyoffset is a 2 element array of the x,y poosition of where the center
; of the beam is.

FUNCTION update_xy, fxfypoints, xyoffset

    xytemp = fxfypoints

    ; apply phase shift. Necessary if the xy data doesn't start at 0,0
    ; A translation in the position plane introduces a linear phase shift
    ; in the frequency plane.
    ; f(x-a,y-b) <--> F(fx,fy) * exp(-2 pi i (fx*a + fy*b))
    ; This is probably not necessary, and would have to be removed before
    ; inverse transforming.

    ; The FFT is going to assume that the xy data starts at 0,0

    xytemp.vec = FFT(fxfypoints.vec,/INVERSE)

    ; Now shift the data. Comes out of FFT as 0,1,...,f,Nyq,-f,...-1
    ; Nyquist bin is # n/2 where n is # points in FFT data
    ; shift right so it goes -f,...,-1,0,1,...,f,nyq
    ; if test is n elements long, neven, shift n/2 - 1
    ; if test n is odd, shift n/2 (ignore remainder)

    xylen = SIZE(xytemp.vec,/DIMENSIONS)

```

```

; the scale in freq space (width of frequency bins) is
; 1/(N*dx) where dx is the sampling interval in x space. Same for y.
; 1/ ( 1/ (N*dx) * N) = 1/( N/(N*dx) ) = 1/(1/dx) = dx

xytemp.dx = 1 / (fxfypoints[0].dx * DOUBLE(xylen[0]))
xytemp.dy = 1 / (fxfypoints[0].dy * DOUBLE(xylen[1]))

; set the x,y values to 0..nx
nx = N_ELEMENTS(xytemp[*,0].x)
temp = DINDGEN(nx)
i=0
FOR i=0, nx-1 DO $
xytemp[i,*].x = temp[i]

ny = N_ELEMENTS(xytemp[0,*].y)
temp = DINDGEN(ny)
j=0
FOR j=0, ny-1 DO $
xytemp[*,j].y = temp[j]

; Now set the x,y values for each field point. Scale point
; separations to reflect that the xy data had spacing of dx,dy
; rather than 1,1 as assumed in the fft.

xytemp.x = xytemp.x * xytemp[0].dx - xyoffset[0]
xytemp.y = xytemp.y * xytemp[0].dy - xyoffset[1]

; after setting the fx, fy values, the field value also must be scaled.
; rmember f(ax) <--> 1/|a| F(fx/a)
; f(ax, by) <--> 1/|ab| F(fx/a, fy/b)
; Note that with noscaling, if dx = 1, dfx = 1/N. dx=a, dfx = 1/(N*a)
; the 1/N part is already covered by the inverse transform.

xytemp.vec = xytemp.vec $
; / SQRT(xytemp[0].dx * xytemp[0].dy)

; To make sure the amplitude comes out right, apply Parseval's Theorem,
; which states that the total power Integ[f(x)^2 dx]=Integ[F(s)^2 ds]

print, 'scaling xy power to fxfy'

scale_power, xytemp, fxfypoints

RETURN, xytemp
END

```

A.5.4 Normalize the Power in an Array: normalize_power.pro

```

; normalize_power.pro
; for array ar1, finds the volume ar1^2 dx dy, the volume under the surface
; formed by ar1^2. This is the total power in ar1.
; divides ar1 by its power, so now it has a power of 1.

PRO normalize_power, ar1

ar1_pow = TOTAL( DOUBLE( ar1.vec * CONJ(ar1.vec) ) ) $
* ar1[0,0].dx * ar1[0,0].dy
PRINT, 'Array 1 power:', ar1_pow

ar1.vec = ar1.vec / sqrt(ar1_pow)
ar1_check = TOTAL( DOUBLE( ar1.vec * CONJ(ar1.vec) ) ) $
* ar1[0,0].dx * ar1[0,0].dy
PRINT, 'Array 1 normalized power: ', ar1_check

RETURN
END

```

A.5.5 Scale the Power in an Array to That of Another Array: scale_power.pro

```
; scale_power.pro
; for array ar1, finds the volume ar1^2 dx dy, the volume under the surface
; formed by ar1^2. This is the total power in ar1.
; divides ar1 by its power, so now it has a power of 1.
; multiplies ar by the power of ar2. Now ar1 has the same power.

PRO scale_power, ar1, ar2

ar1_pow = TOTAL( DOUBLE( ar1.vec * CONJ(ar1.vec) ) ) $
* ar1[0,0].dx * ar1[0,0].dy
PRINT, 'Array 1 power:', ar1_pow

ar2_pow = TOTAL( DOUBLE( ar2.vec * CONJ(ar2.vec) ) ) $
* ar2[0,0].dx * ar2[0,0].dy
PRINT, 'Array 2 power:', ar2_pow

; ar1.vec = ar1.vec / sqrt(ar1_pow)
; ar1_check = TOTAL( DOUBLE( ar1.vec * CONJ(ar1.vec) ) ) $
; * ar1[0,0].dx * ar1[0,0].dy
; PRINT, 'Array 1 normalized power: ', ar1_check
; ar1.vec = ar1.vec * sqrt(ar2_pow)

ar1.vec = ar1.vec * sqrt(ar2_pow/ar1_pow)

ar1_check = TOTAL( DOUBLE( ar1.vec * CONJ(ar1.vec) ) ) $
* ar1[0,0].dx * ar1[0,0].dy
PRINT, 'Array 1 scaled power: ', ar1_check

RETURN
END
```

A.5.6 Generate a Plot of V, SNR, or Flux Vs. Sub-Aperture Radius: plot_vis_stats.pro

```
; plot_vis_stats.pro
;
; intended for plotting the results of chara_diff_vis.pro

PRO plot_vis_stats, yarr, xarr, outfilename, yname, xname, getkey, ps, psx, psy

; find the sizes of the arrays
xlen = N_ELEMENTS(xarr)

; the y array is 2d, will step through 1st index for each plot
ylen = SIZE(yarr,/DIMENSIONS)

IF ps GT 0 THEN BEGIN
SET_PLOT, 'PS'
DEVICE, FILENAME=outfilename, XSIZE = psx, YSIZE = psy
ENDIF

; plot yarr wrt xarr

; plot the 0 index

PLOT, xarr, yarr[0,*], YTITLE = yname, XTITLE = xname, $
; yrange = [0,1.8], ystyle = 1

; oplot the remaining indices
i=1
FOR i=1, ylen[0]-1 DO BEGIN
OPlot, xarr, yarr[i,*], linestyle = i
ENDFOR

IF ps GT 0 THEN BEGIN
DEVICE, /CLOSE
SET_PLOT, 'X'
ENDIF
```

```

IF getkey GT 0 THEN $
key = GET_KBRD(1)

RETURN
END

```

A.5.7 Generate an Intensity Jpeg Plot of an Array: plot_intensity.pro

```

; plot_intensity
PRO plot_intensity, xy, name, getkey, makejpg

WINDOW, 0, XSIZE = N_ELEMENTS(xy[*,0].x), $
YSIZE = N_ELEMENTS(xy[0,*].y) , TITLE = name

TVSCL, xy.vec * CONJ(xy.vec);, XSIZE = !d.x_size, YSIZE = !d.y_size

IF getkey GT 0 THEN $
key = GET_KBRD(1)

    IF makejpg GT 0 THEN $
        out_jpg, name

WDELETE, 0
END

```

A.5.8 Read a Saved File of Visibility and SNR Data: read_vis_stats.pro

```

; read_vis_stats.pro
;
; Procedure to read in results from chara_diff_vis.pro for later processing.
; reads in variables in the order they were written at the end of
; chara_diff_vis.pro

PRO read_vis_stats, infilename, stationa, stationb, subap_r, x2avg, fluxavg, vrms, snr

ns = LONG(0)
nr = LONG(0)

OPENR, 1, infilename

; this is the length of the station a array
READU, 1, ns

stationa = DBLARR(ns)

READU, 1, stationa

PRINT, 'Station A array:', stationa

; this is the length of the station a array
READU, 1, ns

stationb = DBLARR(ns)

READU, 1, stationb

PRINT, 'Station B array:', stationb

; this is the length of the subap_r array
READU, 1, nr

subap_r = DBLARR(nr)

READU, 1, subap_r

PRINT, 'Subaperture radii:', subap_r

```



```

; the x2avg array is [ns,nr,4]
; the last subscript is moment. 0 = mean... see MOMENT IDL function.
x2avg = DBLARR(ns,nr,4)
READU, 1,x2avg
; the fluxavg array is [ns,nr,4]
; the last subscript is moment. 0 = mean... see MOMENT IDL function.
fluxavg = DBLARR(ns,nr,4)
READU, 1,fluxavg
; the vrms array is [ns,nr]
vrms = DBLARR(ns,nr)
READU, 1,vrms
; the snr array is [ns,nr]
snr = DBLARR(ns,nr)
READU, 1,snr
CLOSE, 1
RETURN
END

```

Appendix B

ARC CCD Control Code (on CD-ROM)

B.1 The Header Files

B.1.1 arc_ccd.h

```

/*****
/* arc_ccd.h */
/*
/* Header file for the ARC_CCD Camera Driver. */
/*****
/*
/*      Center for High Angular Resolution Astronomy
/* Georgia State University, Atlanta GA 30303-3083, U.S.A.
/*
/*
/*
/* Telephone: 1-626-796-5405
/* Fax      : 1-626-796-6717
/* email    : theo@chara.gsu.edu
/* WWW      : http://www.chara.gsu.edu/~theo/theo.html
/*
/*
/* (C) This source code and its associated executable
/* program(s) are copyright.
/*
/*****
/*
/* Author : Theo ten Brummelaar based on code by Laszlo Sturmann
/* Date   : January 2000 */
/* Modified from Dalsa to Arc_ccd by Chad Ogden, Oct 2002 */
/*****

#ifndef __ARC_CCD__
#define __ARC_CCD__

/*****
/* The stuff for both MODULES and User Interfaces */
/*****

#include <chara.h>
#include <clock.h>
#include <unistd.h>
#include <linux/types.h>
#include <rtccd.h>
#include <rtccd_io.h>
#include "../vis_dither/vis_dither.h"

// this is where arcccd should put high bandwidth streaming data.
// not over the network!

#define ARC_CCD_DATA_DIRECTORY "/gdt/data/"

// uncomment this to enable code which sends ople commands.  Only for use as
// an extension of ople, not for arc_ccd as a stand-alone.

// #define TALK_TO_OPLE

#ifdef TALK_TO_OPLE

#include "../ople/ople.h"

#endif

```

```

//define space and enter keys
// this should maybe be incorporated into charaui.h

#define KEY_ENTER '\015'
#define KEY_SPACE '\040'

// Size of FFT used
//If FFTs will be 64 bits, that's 2^6.
//If FFTs will be 16 bits, that's 2^4
//If FFTs will be 4 bits, that's 2^2

#define FFT_POW_2      3
#define FFT_SIZE      (1<<FFT_POW_2)
#define SPEC_SIZE FFT_SIZE
#define PS_SIZE        ((1<<(FFT_POW_2-1))+1)

// the maximum number of spectra on the ccd which will be simultaneously
// processed
// 12 is enough for the outer apertures of 2 fiber injectors. Both sides
// of a 2 beam combiner.
// 6 for each beam combiner output x 2 beam combiner outputs
// for a telescope pair.
// 36 for 3 telescopes pairwise
// 12 x n for n telescopes in a pairwise ring

#define MAX_NUM_SIMULTANEOUS_SPECTRA 12

// the maximum number of spectrum stacks. A stack is a group of spectra whose
// power spectrum data can be averaged. Typically all of the spectra in a
// stack should be subapertures from the same beam combiner output,
// or both sides of a pairwise beam combiner output.
// Generally, you'll want the same number of stacks as
// pairwise telescope combinations.
// 1 stack for 2 telescopes pairwise
// 3 stacks for 3 telescopes pairwise
// n stacks for n telescopes in a pairwise ring

#define MAX_SPECTRUM_STACKS 1

// the default number of spectra

#define DEFAULT_NUM_OF_SPECTRA 6

// number of microseconds to wait for shutters to open/close

#define SHUTTER_WAIT_TIME 1000000

// this triggers code which uses the rtccdAPI to send commands to the
// rtccd module. Commenting this out will cause the use of older code which
// speaks directly to the rtccd module.

#define USE_RTCCD_API

// this is where the .lod files for the arc ccd camera are located

#define ARC_CCD_LOAD_FILES  "/ctrscrut/chara/etc/pci.interrupts.lod", "/ctrscrut/chara/etc/tim030403.lod"

//comment this out to omit the overscan areas from the default ccd bounding box

//#define USE_CCD_OVERSCAN

// ROI coordinates for useful area of ARC CCD
// These define the default bounding box if USE_CCD_OVERSCAN isn't defined

//upper left corner coords

#define USEFUL_ULX 7
#define USEFUL_ULY 79

//lower right corner coords

#define USEFUL_LRX 86
#define USEFUL_LRY 0

// in terms of x,y,dx,dy roi coords:
// (dx,dy is total number of pix in that dim.)

#define USEFUL_X USEFUL_ULX
#define USEFUL_Y USEFUL_LRY

```

```

#define USEFUL_DX (USEFUL_LRX - USEFUL_ULX + 1)
#define USEFUL_DY (USEFUL_ULY - USEFUL_LRY + 1)

// leftmost and rightmost pixel numbers for a spectrum,
// centered left-right.

#define SPECTRUM_WIDTH FFT_SIZE
#define SINGLE_ROW_BB_ULX ( (CCD_X_PIXELS/2) - (SPECTRUM_WIDTH/2) )
#define SINGLE_ROW_BB_LRX ( (CCD_X_PIXELS/2) + (SPECTRUM_WIDTH/2) - 1)

// default value for spectrum offset from edge of chip
#define DEFAULT_SPEC_ROW_OFFSET 0

// avg readout time per pixel, for purposes of timing data read from RT FIFO
// system waits for readout + exposure time before polling FIFO

#define MICROSEC_PER_PIXEL 4

// number of microsec to wait between FIFO polls
#define FIFO_POLLING_INTERVAL 100

/*
 * Some macros
 */

#define MIN_EXPOSURE 1
#define MAX_EXPOSURE 2000
#define DEFAULT_FRAMES_IN_BIAS 20
#define ALLOWABLE_MAKE_BIAS_DROPPED_FRAMES 100
#define MAX_POW_SPECTRA_IN_MEAN 100

// Well depth of the arc ccd camera
#define ARC_CCD_WELL_DEPTH 65535.0

// Let arccd see the pointers to the vis_dither dither position arrays
// although the pointers will be declared as constant, so vis_dither should
// be safe

#define USE_VIS_DITHER_SWEEP_POINTERS

// some constants for use when setting flags on rt side

#define FLAG_ID_BIAS_HAS_BEEN_SET 1
#define FLAG_ID_SPEC_FLAT_HAS_BEEN_SET 2
#define FLAG_ID_PS_FLAT_HAS_BEEN_SET 3
#define FLAG_ID_USE_BIAS 4
#define FLAG_ID_USE_SPEC_FLAT 5
#define FLAG_ID_USE_PS_FLAT 6
#define FLAG_ID_USE_FFT 7
#define FLAG_ID_USE_LPT_TOGGLE 8
#define FLAG_ID_PS_SERVO_ON 9
#define FLAG_ID_SQUAREWAVE_ON 10
#define FLAG_ID_USE_PS 11

// constants for use with ARC_CCD_RT_TWEAK_PID

#define TWEAK_PS_SERVO_P 0
#define TWEAK_PS_SERVO_I 1
#define TWEAK_PS_SERVO_D 2
#define TWEAK_PS_SERVO_GAIN 3

// default values of visible fringe tracking ps_servo parameters

#define PS_SERVO_P 1.0
#define PS_SERVO_I 0.0
#define PS_SERVO_D -0.1
#define PS_SERVO_GAIN 1.0
#define PS_SERVO_LEAKY_COEFF 0.9
#define PS_SERVO_TARGET_FREQ FFT_SIZE / 4.0

// The frequency range for the power spectrum is 0-32 fringes across the full
// 64 data points in the spectrum. This includes the 0 buffers around the
// 80 pixels on the chip or however many pixels wide the bounding box is.
// Trying to track on a target too close to one edge of the power spectrum
// will cause errors due to crossing the 0 or nyquist (32) frequencies.
// A special function will be required to implement jumps across these
// boundaries while tracking.

```

```

#define PS_SERVO_TARGET_MIN 3.0
#define PS_SERVO_TARGET_MAX PS_SIZE - 1.0

/* The functions */
// The vis_dither driver has already registered a bunch of commands, so
// the ones in this file make sure they follow the end of the list
// defined in vis_dither.h, which is included by arc_ccd.h.

#define ARC_CCD_RT_SET_ROW (FIRST_ARCCCD_COMMAND + 0)
#define ARC_CCD_RT_GET_ROW (FIRST_ARCCCD_COMMAND + 1)
#define ARC_CCD_RT_REQ_PS (FIRST_ARCCCD_COMMAND + 2)
#define ARC_CCD_RT_REQ_FRAME (FIRST_ARCCCD_COMMAND + 3)
#define ARC_CCD_RT_SYNC_BB (FIRST_ARCCCD_COMMAND + 4)
#define ARC_CCD_RT_CHECK_BB (FIRST_ARCCCD_COMMAND + 5)
#define ARC_CCD_RT_SET_BIAS (FIRST_ARCCCD_COMMAND + 6)
#define ARC_CCD_RT_SET_SPEC_FLAT (FIRST_ARCCCD_COMMAND + 7)
#define ARC_CCD_RT_CHECK_FLAG (FIRST_ARCCCD_COMMAND + 8)
#define ARC_CCD_RT_TOGGLE_FLAG (FIRST_ARCCCD_COMMAND + 9)
#define ARC_CCD_RT_SET_SUPPRESS_PIXELS (FIRST_ARCCCD_COMMAND + 10)
#define ARC_CCD_RT_GET_SUPPRESS_PIXELS (FIRST_ARCCCD_COMMAND + 11)
#define ARC_CCD_RT_REQ_SPECTRUM (FIRST_ARCCCD_COMMAND + 12)
#define ARC_CCD_RT_SET_PS_MEAN_NUM (FIRST_ARCCCD_COMMAND + 13)
#define ARC_CCD_RT_REQ_PURGE_PS_BUF (FIRST_ARCCCD_COMMAND + 14)
#define ARC_CCD_RT_INIT_FFT (FIRST_ARCCCD_COMMAND + 15)
#define ARC_CCD_RT_REQ_PS_STATS (FIRST_ARCCCD_COMMAND + 16)
#define ARC_CCD_RT_START_RECORD (FIRST_ARCCCD_COMMAND + 17)
#define ARC_CCD_RT_END_RECORD (FIRST_ARCCCD_COMMAND + 18)
#define ARC_CCD_RT_TWEAK_PID (FIRST_ARCCCD_COMMAND + 19)
#define ARC_CCD_RT_NEW_TARGET_FREQ (FIRST_ARCCCD_COMMAND + 20)
#define ARC_CCD_RT_SET_FLAG_TRUE (FIRST_ARCCCD_COMMAND + 21)
#define ARC_CCD_RT_SET_FLAG_FALSE (FIRST_ARCCCD_COMMAND + 22)
#define ARC_CCD_RT_SETUP_DITHER_SCAN (FIRST_ARCCCD_COMMAND + 23)
#define ARC_CCD_RT_ACTIVATE_DITHER_SCAN (FIRST_ARCCCD_COMMAND + 24)
#define ARC_CCD_RT_REQ_CHANNELS (FIRST_ARCCCD_COMMAND + 25)
#define ARC_CCD_RT_SET_CHANNELS (FIRST_ARCCCD_COMMAND + 26)
#define ARC_CCD_RT_SET_PS_FLAT (FIRST_ARCCCD_COMMAND + 27)
#define ARC_CCD_RT_REQ_VD_STATS (FIRST_ARCCCD_COMMAND + 28)

/* Definition of data types used in both modules and user interfaces */

struct arcccd_power_spectrum_stats_struct {

    CHARA_TIME time;
    float bin;
    float height;
    float power;
    float width;
    float snr;
    float skew;
    float sigma;
    float phase;
    float errsignal;
    float tracksignal;
    float dith_dac;
    float dith_um;
    float frame_rate;
};

typedef struct arcccd_power_spectrum_stats_struct PS_STATS;

// this is for dither scans run by the arc_ccd_rt module

struct arcccd_dither_scan_parameters {

    int lower_limit;
    int upper_limit;
    int dx;
    int dx_index;
    int scan_index;
    int total_scans;
    bool on;
    bool going_up;
};

typedef struct arcccd_dither_scan_parameters ARCCCD_DITH_SCAN;

// This is used in determining the display color for the channels.

```

```

typedef struct {
double r,g,b;
} COLOR;

/*****
/* This stuff for modules only
*****/

#ifdef MODULE

#include <rt_fft.h>
#include "f_arctanlookup.c"

// make sure booleans are defined

//ifndef bool
//typedef unsigned char bool;
//endif
//ifndef TRUE
//define TRUE 1
//endif
//ifndef FALSE
//define FALSE 0
//endif

// The number of sigma above the average that a ps bin must have
// to be counted as part of the peak. Maybe this would be better served by
// a variable and a little smarts.

#define MINIMUM_SIGNIFICANT_PEAK 3.0

//include "rtccd_cb.h"
// for prototyping the camera driver's register callback function
// taken from rtccd_main.c in gdt:/home/tgs/chara/dev/rtccd/driver/

int rt_astropci_reg_callback(uint32_t (*CB)(uint16_t *, CHARA_TIME, uint32_t));

/*
 * Globals
 */

/*
 * Some local prototypes.
 */

int init_module(void);
int arc_ccd_rt_set_row(unsigned char arg);
int arc_ccd_rt_get_row(unsigned char arg);
int arc_ccd_rt_req_ps(unsigned char arg);
int arc_ccd_rt_get_ps(void);
int arc_ccd_rt_req_spectrum(unsigned char arg);
int arc_ccd_rt_get_spectrum(void);
int arc_ccd_rt_req_frame(unsigned char arg);
int arc_ccd_rt_get_frame(void);
int arc_ccd_rt_req_ps_stats(unsigned char arg);
int arc_ccd_rt_get_ps_stats(void);
int arc_ccd_rt_req_channels(unsigned char arg);
int arc_ccd_rt_get_channels(void);
int arc_ccd_rt_req_vd_stats(unsigned char arg);
int arc_ccd_rt_get_vd_stats(void);
int arc_ccd_rt_sync_bb(unsigned char arg);
int arc_ccd_rt_check_bb(unsigned char arg);
int arc_ccd_rt_set_bias(unsigned char arg);
int arc_ccd_rt_set_spec_flat(unsigned char arg);
int arc_ccd_rt_set_ps_flat(unsigned char arg);
uint32_t arc_ccd_rt_callback(uint16_t *data, CHARA_TIME fr_time, uint32_t fr_number);
int arc_ccd_rt_calculate_ps(void);
int arc_ccd_rt_ps_mean(void);
int arc_ccd_rt_calculate_ps_stats(void);
int arc_ccd_rt_apply_bias(unsigned char arg);
int arc_ccd_rt_apply_spec_flat(void);
int arc_ccd_rt_apply_ps_flat(void);
int arc_ccd_rt_toggle_flag(unsigned char arg);
int arc_ccd_rt_check_flag(unsigned char arg);
int arc_ccd_rt_set_suppress_pixels(unsigned char arg);
int arc_ccd_rt_get_suppress_pixels(unsigned char arg);
int arc_ccd_rt_toggle_lpt(unsigned char arg);

```

```

int arc_ccd_rt_set_ps_mean_num(unsigned char arg);
int arc_ccd_rt_purge_ps_buf(void);
int arc_ccd_rt_req_purge_ps_buf(unsigned char arg);
int arc_ccd_rt_init_fft(unsigned char arg);
int arc_ccd_rt_start_record(unsigned char arg);
int arc_ccd_rt_end_record(unsigned char arg);
void arc_ccd_rt_ps_servo(void);
int arc_ccd_rt_tweak_pid(unsigned char arg);
void arc_ccd_rt_squarewave(void);
int arc_ccd_rt_new_target_freq(unsigned char arg);
int arc_ccd_rt_set_flag_true(unsigned char arg);
int arc_ccd_rt_set_flag_false(unsigned char arg);
void arc_ccd_rt_dither_scan(void);
int arc_ccd_rt_setup_dither_scan(unsigned char arg);
int arc_ccd_rt_activate_dither_scan(unsigned char arg);
void arc_ccd_rt_sort_full_frame(void);
void arc_ccd_rt_pixel_doctor(uint16_t *dat);
int arc_ccd_rt_set_channels(unsigned char arg);
void arc_ccd_rt_calculate_frame_rate(void);
void cleanup_module(void);

#else

/*****
/* This stuff only for user interfaces */
*****/

#include <stdint.h>
#include <charaui.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <nsimpleX.h>
#include <uishut.h>
#include <nrc.h>
#include <filter.h>

/* Globals */

extern char *default_display_machine;
extern char *bias_types[3];

// in arc_ccd_comms:

extern CHARA_TIME frame_time;
extern uint32_t frame_number;
extern int exposure;
extern SROI bbdata;
extern int bbquadrant;
extern PS_STATS ps_peak;
extern bool freeze_stats_query;
extern VIS_DITH vd_par;

#ifdef USE_VIS_DITHER_SWEEP_POINTERS
extern const float *vis_dith_sweep_um;
extern const short int *vis_dith_sweep_dac;
#endif

// these are the user side copies of arc_ccd_rt flags

// in arc_ccd_control:

extern bool use_fft;
extern bool use_lpt_toggle;
extern bool bb_movie_running;
extern int spec_shut;
extern bool recording;
extern float ps_servo_target;
extern bool ps_servo_on;
extern bool ps_servo_squarewave_on;
extern float ps_servo_p;
extern float ps_servo_i;
extern float ps_servo_d;
extern float ps_servo_gain;
extern bool single_quad_mode;

// in arc_ccd_channels:

extern bool channel_mask[FFT_SIZE];

```

```

extern int channel_mask_total;

// in arc_ccd_bias:

extern bool use_bias;
extern bool bias_has_been_set;
extern long int *bias;

// in arc_ccd_spect:

extern bool spec_flat_has_been_set;
extern bool ps_flat_has_been_set;
extern bool use_spec_flat;
extern bool use_ps_flat;
extern int ps_mean_num;
extern bool dps_running;
extern float spec_flat[MAX_NUM_SIMULTANEOUS_SPECTRA][SPEC_SIZE];
extern float ps_flat[PS_SIZE];
extern int spec_row[MAX_NUM_SIMULTANEOUS_SPECTRA + 1];
extern int spec_row_offset;
extern int spec_stack[2][MAX_SPECTRUM_STACKS + 1];

/* end globals */

/* function prototypes */

/* arc_ccd_comms.c */

void dbi(void);
int init_arc_ccd(void);
int open_driver(void);
int close_driver(void);
int reset_arc_ccd(void);
#ifdef USE_RTCCD_API
int send_ascii_command(unsigned int *data);
#endif
int set_exposure(unsigned int exposure);
int set_bb(int x, int y, int dx, int dy);
uint16_t *get_bb(void);
int set_rt_flag(unsigned char flag_id, bool value);
int set_rt_flag_blind(unsigned char flag_id, bool value);
int get_ps_stats(void);
int get_vd_stats(void);
int req_vis_dither_goto(int target);
int req_vis_dither_move_rel(int change);
int set_ps_servo_target(float target);
int tweak_ps_servo_pid(int param_id, float value);
int toggle_ps_servo(bool value);
int toggle_ps_servo_squarewave(bool value);
int setup_dither_scan(ARCCCD_DITH_SCAN scan);
int activate_dither_scan(unsigned char arg);

/* arc_ccd_control.c */

int call_set_exposure(int argc, char **argv);
int clear_window(int argc, char **argv);
void arc_ccd_close(void);
void arc_ccd_open(void);
int call_set_bb(int argc, char **argv);
int set_single_quad_bb(int argc, char **argv);
int call_set_bb_row_64(int argc, char **argv);
int call_set_bb_row_16(int argc, char **argv);
int clear_bb(int argc, char **argv);
int single_bb(int argc, char **argv);
int display_bb(char *display, uint16_t *buffer);
int display_bb_overlay(char *display, uint16_t *buffer, int ulx, int uly,
int lrx, int lry);
int bb_movie(int argc, char **argv);
int bb_background_movie(int argc, char **argv);
int bb_movie_background_job(void);
int save_bb(int argc, char **argv);
int call_make_bias(int argc, char **argv);
int call_toggle_bias(int argc, char **argv);
int send_ints_on(int argc, char **argv);
int send_ints_off(int argc, char **argv);
int toggle_use_fft(int argc, char **argv);
int toggle_use_lpt(int argc, char **argv);
int init_fft(int argc, char **argv);

```



```

int save_spectrum(int argc, char **argv);
bool read_arc_data(char *data, int n);
int load_arc_ccd(int argc, char **argv);
int call_req_vis_dither_goto(int argc, char **argv);
int call_req_vis_dither_move_rel(int argc, char **argv);
int call_set_ps_servo_target(int argc, char **argv);
int call_toggle_ps_servo(int argc, char **argv);
int call_toggle_ps_servo_squarewave(int argc, char **argv);
int call_tweak_ps_servo_pid(int argc, char **argv);
int call_setup_dither_scan(int argc, char **argv);
int call_activate_dither_scan(int argc, char **argv);
int set_picture_scale(int argc, char **argv);
int set_picture_display_type(int argc, char **argv);

/* arc_ccd_channels.c */
float *get_channels(void);
int set_channel_list(void);
int call_set_channel_list(int argc, char **argv);
int get_disp_color(double v, double vmin, double vmax, COLOR *c, double scale);
int save_channels(int argc, char **argv);

/* arc_ccd_spect.c */

float *get_power_spec(void);
float *get_spectrum(void);
float *display_power_spectrum(Window win, int width, int height, bool no_fft);
int call_display_power_spec(int argc, char **argv);
int power_spec_bias(int argc, char **argv);
int toggle_ps_bias(int argc, char **argv);
int set_num_pix_zero(int argc, char **argv);
int call_set_spec_row(int argc, char **argv);
int set_spec_row(int *row, int row_offset);
int make_spec_flat(int argc, char **argv);
int toggle_spec_flat(int argc, char **argv);
int make_ps_flat(int argc, char **argv);
int toggle_ps_flat(int argc, char **argv);
int toggle_auto_range(int argc, char **argv);
int set_spect_cutoff(int argc, char **argv);
int set_ps_mean_num(int argc, char **argv);
int request_purge_ps(int argc, char **argv);
int change_spec_display_range(float min, float max);
int call_change_spec_display_range(int argc, char **argv);

// no longer in use:

float *bb_power_spec(bool no_fft);
int calc_data_dark(int argc, char **argv);
int toggle_data_dark(int argc, char **argv);

/* arc_ccd_bias.c */

int free_bias(void);
int toggle_bias(void);
int make_bias(int frames);

// no longer in use:

int apply_bias(uint16_t *data);
#endif

/* arc_ccd_background.c */

void setup_arc_ccd_background_jobs(void);
void unsetup_arc_ccd_background_jobs(void);
int arc_ccd_linux_time_status(void);
int arc_ccd_rt_time_status(void);
int arc_ccd_status(void);

#endif

```

B.1.2 std_arc_ccd_funcs.h

```

/*****
/* std_arc_ccd_funcs.h */
*/
/* Standard functions in the Arc_ccd lib. */
/*****
*/

```

```

/*          CHARA ARRAY USER INTERFACE */
/*          Based on the SUSI User Interface */
/* In turn based on the CHIP User interface */
/*
/*          Center for High Angular Resolution Astronomy
/*          Mount Wilson Observatory, CA 91001, USA */
/*
/* Telephone: 1-626-796-5405
/* Fax       : 1-626-796-6717
/* email     : theo@chara.gsu.edu
/* WWW      : http://www.chara.gsu.edu
/*
/* (C) This source code and its associated executable
/* program(s) are copyright.
/*
/******
/*
/* Author : Theo ten Brummelaar
/* Date   : Jan 2000 */
/* Modified from Dalsa to Arc_ccd by Chad Ogden, Oct 2002 */
/******

#ifndef __STD_ARC_CCD_FUNCTS__
#define __STD_ARC_CCD_FUNCTS__
#endif
{"exp",call_set_exposure},
{"x",call_set_exposure},
{"frame",single_bb},
{"fr",single_bb},
{"clear",clear_window},
{"clr",clear_window},
{"movie",bb_movie},
{"mov",bb_movie},
{"bgmovie",bb_background_movie},
{"bgm",bb_background_movie},
{"sbb",call_set_bb},
// {"sbb_64",call_set_bb_row_64},
{"sbb_16",call_set_bb_row_16},
{"sbb_2d",call_set_bb},
{"cbb",clear_bb},
{"bb",single_bb},
{"svf",save_bb},
{"svb",save_bb},
{"dps",call_display_power_spec},
{"cpsb",power_spec_bias},
{"tpsb",toggle_ps_bias},
{"spsz",set_num_pix_zero},
{"row",call_set_spec_row},
{"flat",make_ps_flat},
{"psflat",make_ps_flat},
{"mkflat",make_ps_flat},
{"specflat",make_spec_flat},
{"tflat",toggle_ps_flat},
{"tpsflat",toggle_ps_flat},
{"tspecflat",toggle_spec_flat},
//no longer used{"cdark",calc_data_dark},
//no longer used{"tdark",toggle_data_dark},
{"tauto",toggle_auto_range},
{"ssc",set_spect_cutoff},
{"tb",call_toggle_bias},
{"tbias",call_toggle_bias},
{"mbias",call_make_bias},
{"mkbias",call_make_bias},
{"inton",send_ints_on},
{"into",send_ints_off},
{"tfft",toggle_use_fft},
{"tlpt",toggle_use_lpt},
{"psmean",set_ps_mean_num},
{"purgeps",request_purge_ps},
{"purge",request_purge_ps},
{"initfft",init_fft},
{"ifft",init_fft},
{"svps",save_spectrum},
// {"load_arc_ccd",load_arc_ccd},
// {"load",load_arc_ccd},
{"goto",call_req_vis_dither_goto},
{"load_vis_dither_offsets",call_load_dither_offsets},
{"mvrel",call_req_vis_dither_move_rel},
{"frgtarg",call_set_ps_servo_target},
{"vistarget",call_set_ps_servo_target},

```

```

{"vistgt", call_set_ps_servo_target},
{"servo", call_toggle_ps_servo},
{"squarewave", call_toggle_ps_servo_squarewave},
{"tweak", call_tweak_ps_servo_pid},
{"enable_vis_dither_offsets", call_enable_dither_offsets},
// {"setup_arcccd_dither_scan", call_setup_dither_scan},
// {"scds", call_setup_dither_scan},
// {"activate_arcccd_dither_scan", call_activate_dither_scan},
// {"acds", call_activate_dither_scan},
{"spec_disp", call_change_spec_display_range},
{"spec_scale", call_change_spec_display_range},
{"setup_vis_dither_scan", new_dither},
{"svds", new_dither},
{"scale", set_picture_scale},
{"frame_scale", set_picture_scale},
{"scale_type", set_picture_display_type},
{"channels", call_set_channel_list},
{"chan", call_set_channel_list},
{"svch", save_channels},
{"save_channels", save_channels},
{"vd", call_enable_vis_dither_sweep},
{"vis_dither", call_enable_vis_dither_sweep},
{"vdauto", call_enable_vis_dither_sweep_automatic},
{"vis_dither_automaitc",
call_enable_vis_dither_sweep_automatic},
{"vdreset", call_sweep_reset},
{"vddamp", call_change_damping},

```

B.1.3 arc_ccd_channel_cal.h

```

/*****
/* arc_ccd_channel_cal.h */
/*
/* Calibration center wavenumbers and channel widths. */
/*****
/*
/* CHARA ARRAY USER INTERFACE */
/* Based on the SUSI User Interface */
/* In turn based on the CHIP User interface */
/*
/* Center for High Angular Resolution Astronomy
/* Mount Wilson Observatory, CA 91001, USA */
/*
/* Telephone: 1-626-796-5405
/* Fax : 1-626-796-6717
/* email : theo@chara.gsu.edu
/* WWW : http://www.chara.gsu.edu
/*
/* (C) This source code and its associated executable
/* program(s) are copyright.
/*
/*****
/*
/* Author : Chad Ogden
/* Date : May 2005 */
/*****

const float channel_wn[] = {
// 0 lam: 0.5336
1.8740 ,
// 1 lam: 0.5773
1.7323 ,
// 2 lam: 0.6287
1.5905 ,
// 3 lam: 0.6916
1.4459 ,
// 4 lam: 0.7668
1.3042 ,
// 5 lam: 0.8603
1.1624 ,
// 6 lam: 0.9798
1.0206 ,
// 7 lam: 1.1378
0.8789 };

const float channel_delta_wn = 0.1418;

```

```

/****
* This is the wavenumber info for the spectrograph with the 60 degree prism
*
// channel lamda should be moved to a config file
// spec size currently 16
// these data determined from line fitting. These start at channel 0 on the
// quadrant, so take into account the offset of the spectrum row.

// I have shifted everything by 2 pixels, because I think the ccd has been
// bumped and I recalibrated using dither size and location of ps peaks.

const float channel_wn[] = {
// 0 lam: 0.5521      1.8113 ,
// 1 lam: 0.5647      1.7710 ,
// 2 lam: 0.5778      1.7307 ,
// 3 lam: 0.5916      1.6904 ,
// 4 lam: 0.6060      1.6502 ,
// 5 lam: 0.6211      1.6100 ,
// 6 lam: 0.6370      1.5698 ,
// 7 lam: 0.6537      1.5297 ,
// 8 lam: 0.6713      1.4896 ,
// 9 lam: 0.6899      1.4496 ,
// 10 lam: 0.7095     1.4095 ,
// 11 lam: 0.7302     1.3695 ,
// 12 lam: 0.7521     1.3296 ,
// 13 lam: 0.7754     1.2896 ,
// 14 lam: 0.8002     1.2498 ,
// 15 lam: 0.8265     1.2099 ,
// 16 lam: 0.8547     1.1701 ,
// 17 lam: 0.8848     1.1303 ,
// 18 lam: 0.9170     1.0905 ,
// 19 lam: 0.9517     1.0508 ,
// 20 lam: 0.9891     1.0111 ,
// 21 lam: 1.0294     0.9714 ,
// 22 lam: 1.0732     0.9318 ,
// 23 lam: 1.1209     0.8922 ,
// 24 lam: 1.1729     0.8526 ,
// 25 lam: 1.2299     0.8131 ,
// 26 lam: 1.2927     0.7736 ,
// 27 lam: 1.3622     0.7341
};

// This is the wavenumber width of each spectral channel, assuming a linear
// fit of channel number vs. wavenumber

const float channel_delta_wn = 0.0400;

****/

```

B.2 The Real-Time Linux Module, arc_ccd_rt.c

```

#define MODULE
/*****
/* arc_ccd_rt.c */
*/
/* Functions used for handling the data sent back by the arc ccd */
/* camera driver, but separate from it. Must be used in context of */
/* a callback function registered with the camera driver. */
*****/
/*
/* Center for High Angular Resolution Astronomy
/* Georgia State University, Atlanta GA 30303-3083, U.S.A.
*/
/* Telephone: 1-626-796-5405
/* Fax : 1-626-796-6717
/* email : theo@chara.gsu.edu
/* WWW : http://www.chara.gsu.edu/~theo/theo.html
*/
/* (C) This source code and its associated executable
/* program(s) are copyright.
*/
*****/
/*
/* Author : Chad Ogden */
/* Date : Nov 2002 */
*****/

/* Include the header file */
#include "arc_ccd.h"

#define DEFAULT_SUPPRESS_PIXELS 0

// comment this out to turn off varied to/from FIFO debug messages.
// #define DEBUG_MESSAGES

// comment this out to turn off spectrum to FIFO debug messages.
// #define SPECTRUM_IO_DEBUG_MESSAGES

// comment this out to turn off ccd frame to FIFO debug messages.
// #define FRAME_IO_DEBUG_MESSAGES

// comment this out to turn off channel to FIFO debug messages.
// #define CHANNEL_IO_DEBUG_MESSAGES

// comment this out to turn off vis dither data to FIFO debug messages.
// #define DITHER_IO_DEBUG_MESSAGES

// uncomment this to normalise the spectrum when applying the flat
// #define NORMAILZE_WHEN_APPLYING_SPEC_FLAT

// comment this out to turn off lpt toggle around the fft
// #define FFT_LPT_TOGGLE

// comment this out to turn off lpt toggle around callback function
#define CALLBACK_LPT_TOGGLE

// comment out to turn off toggle in ps -> FIFO routine
// #define GET_PS_LPT_TOGGLE

// comment out to turn off toggle in req_ps -> FIFO routine
// #define REQ_PS_LPT_TOGGLE

#define TOGGLE_UP 1
#define TOGGLE_DOWN 0
#define TOGGLE_OPPOSITE 2

/* These are defined in vis_dither.h, here is a copy for reference purposes
*

```

```

* #define AMPLIFIER_GAIN 15.0
* #define DAC_GAIN      (10.0/32767)
* #define TOTAL_GAIN    (AMPLIFIER_GAIN*DAC_GAIN)
*
* #define pos_to_volts_linear(x) (1.46534*x)
* #define volts_to_pos_linear(x) (x/1.46534)
*
*
* // old version v to dac
*
* // #define volts_to_dac(x)          ((short int)(x / TOTAL_GAIN + 0.5))
*
* // new version of v to dac leaves out short int conversion. do later.
*
* #define volts_to_dac(x)          (x / TOTAL_GAIN + 0.5)
* #define dac_to_volts(x)         (x * TOTAL_GAIN)
*/

#define fringe_freq_to_pos(x) (x/1.6)
#define pos_to_fringe_freq(x) (1.6 * x)

#define freq_to_dac pos_to_volts_linear(fringe_freq_to_pos(1.0))/TOTAL_GAIN
#define freq_to_dac_offset 0.5

// fringe freq amplitude of injected squarewave for ps servo tuning
#define SQUAREWAVE_AMPLITUDE 20.0

// number of loops before squarewave toggles
#define SQUAREWAVE_PERIOD 500

// signal to noise below which the servo stops tracking
#define PS_SERVO_SNR_CUTOFF 10.0

// how many frames to wait before calculating frame rate
#define NUMBER_OF_FRAMES_PER_FRAME_RATE_CALCULATION 100

/* Globals*/

static bool arc_ccd_bias_has_been_set;
static bool arc_ccd_spec_flat_has_been_set;
static bool arc_ccd_ps_flat_has_been_set;
static bool arc_ccd_use_lpt_toggle;
static bool arc_ccd_use_bias;
static bool arc_ccd_use_spec_flat;
static bool arc_ccd_use_ps_flat;
static bool arc_ccd_use_fft;
static bool arc_ccd_lpt_high;
static bool arc_ccd_ps_servo_on;
static bool arc_ccd_usable_peak;
static bool arc_ccd_use_ps;
static bool arc_ccd_recording;

// flags to request data for non-rt side
static bool arc_ccd_req_ps;
static bool arc_ccd_req_frame;
static bool arc_ccd_req_channels;
static bool arc_ccd_req_spectrum;
static bool arc_ccd_req_ps_stats;
static bool arc_ccd_req_vd_stats;

static bool arc_ccd_purge_ps_flag;
static bool arc_ccd_init_fft_flag;

static int arc_ccd_ps_int_bin;

// arc_ccd_ps_buf[0][] is used to store the mean. arc_ccd_ps_buf[1]
// is where the most recent power spectrum is put.

static float arc_ccd_ps_buf[MAX_POW_SPECTRA_IN_MEAN+1][PS_SIZE];

// This index is used to indicate which slot in arc_ccd_ps_buf[] the
// most recent power spectrum will be inserted into.

static int arc_ccd_ps_index;

static float arc_ccd_re_fft_buf[MAX_NUM_SIMULTANEOUS_SPECTRA][FFT_SIZE];

```

```

static float arc_ccd_im_fft_buf[MAX_NUM_SIMULTANEOUS_SPECTRA][FFT_SIZE];
static float arc_ccd_spec_buf[MAX_NUM_SIMULTANEOUS_SPECTRA][FFT_SIZE];
static float arc_ccd_spec_flat[MAX_NUM_SIMULTANEOUS_SPECTRA][FFT_SIZE];
static float arc_ccd_ps_flat[PS_SIZE];
static long int arc_ccd_bias[CCD_X_PIXELS*CCD_Y_PIXELS];
static float arc_ccd_channel_buf[MAX_NUM_SIMULTANEOUS_SPECTRA][FFT_SIZE];
static int arc_ccd_channel_list[SPEC_SIZE+1];

// row[0] gives the total number of rows
static int arc_ccd_row[MAX_NUM_SIMULTANEOUS_SPECTRA+1];
static int arc_ccd_row_offset;
static int arc_ccd_suppress_pixels;
static SROI arc_ccd_bbdata;
static int arc_ccd_bbquadrant;
static int arc_ccd_num_quadrants;
static bool arc_ccd_single_quad_mode;
static CHARA_TIME arc_ccd_frame_time;
static uint32_t arc_ccd_frame_number;
static int arc_ccd_pow_spectra_in_mean;
static PS_STATS arc_ccd_ps_peak;
static float arc_ccd_frame_rate;

//static int arc_ccd_vis_dither_counter;
static const float *arc_ccd_vis_dith_sweep_um;
static const short int *arc_ccd_vis_dith_sweep_dac;

static float arc_ccd_ps_servo_target_freq;
static float arc_ccd_ps_servo_p;
static float arc_ccd_ps_servo_i;
static float arc_ccd_ps_servo_d;
static float arc_ccd_ps_servo_gain;
static float arc_ccd_ps_servo_last_errsignal;
static float arc_ccd_ps_servo_last_tracksignal;
static float arc_ccd_ps_servo_leaky_errsignal;

static int arc_ccd_squarewave_loop;
static bool arc_ccd_squarewave_on;

// this is for dither run by the arc_ccd_rt module

static ARCCCD_DITH_SCAN arcccd_dith_scan;

// this is a copy of the vis_dither stats

static VIS_DITH arc_ccd_vd_par;

#ifdef USE_CCD_OVERSCAN
uint16_t sorted_frame[CCD_X_PIXELS*CCD_Y_PIXELS];
#else
uint16_t sorted_frame[USEFUL_DX*USEFUL_DY];
#endif

// this pointer only to be used by functions called by the callback function
uint16_t *frame;

#ifdef SPECTRUM_IO_DEBUG_MESSAGES
static float test_ps_avg;
#endif

/*****
/* init_module() */
/* */
/* This code initializes the module and the printer port itself. */
*****/

int init_module(void)
{
    int i,j;

    // arc_ccd_row < 0 means row not set.

    for(i=0;i<MAX_NUM_SIMULTANEOUS_SPECTRA; i++)
        arc_ccd_row[i] = -1;

    arc_ccd_row_offset = DEFAULT_SPEC_ROW_OFFSET;

    // initially suppress first SUPPRESS_PIXELS pixels in power spectrum
    arc_ccd_suppress_pixels = DEFAULT_SUPPRESS_PIXELS;

```

```

arc_ccd_frame_rate = 0;
arc_ccd_frame_time = 0;
arc_ccd_ps_peak.time = 0;
arc_ccd_frame_number = 0;

arc_ccd_ps_peak.height = 0.0;
arc_ccd_ps_int_bin = 0;
arc_ccd_ps_peak.bin = 0.0;
arc_ccd_ps_peak.power = 0.0;
arc_ccd_ps_peak.width = 0.0;
arc_ccd_ps_peak.snr = 0.0;
arc_ccd_ps_peak.skew = 0.0;
arc_ccd_ps_peak.errsignal = 0.0;
arc_ccd_ps_peak.tracksignal = 0.0;
arc_ccd_ps_peak.phase = 0.0;

arc_ccd_ps_servo_target_freq = PS_SERVO_TARGET_FREQ;
arc_ccd_ps_servo_p = PS_SERVO_P;
arc_ccd_ps_servo_i = PS_SERVO_I;
arc_ccd_ps_servo_d = PS_SERVO_D;
arc_ccd_ps_servo_gain = PS_SERVO_GAIN;

//arc_ccd_ps_buf initialize to zeros

arc_ccd_rt_purge_ps_buf();

//set the arc_ccd_ps_index to 1
arc_ccd_ps_index = 1;

//set the default number of pow spectra in mean to 1.

arc_ccd_pow_spectra_in_mean = 1;

// default bounding box values

#ifdef USE_CCD_OVERSCAN
arc_ccd_bbdata.x = 0;
arc_ccd_bbdata.y = 0;
arc_ccd_bbdata.dx = CCD_X_PIXELS ;
arc_ccd_bbdata.dy = CCD_Y_PIXELS ;
#else
arc_ccd_bbdata.x = USEFUL_X;
arc_ccd_bbdata.y = USEFUL_Y;
arc_ccd_bbdata.dx = USEFUL_DX/2;
arc_ccd_bbdata.dy = USEFUL_DY/2;
arc_ccd_bbquadrant = 2;
arc_ccd_num_quadrants = 1;
arc_ccd_single_quad_mode = TRUE;
#endif

// default flags

arc_ccd_use_spec_flat = arc_ccd_use_bias = FALSE;
arc_ccd_use_ps_flat = FALSE;
arc_ccd_bias_has_been_set = arc_ccd_spec_flat_has_been_set = FALSE;
arc_ccd_ps_flat_has_been_set = FALSE;
arc_ccd_use_fft = TRUE;
arc_ccd_use_ps = TRUE;

arc_ccd_use_lpt_toggle = FALSE;
arc_ccd_lpt_high = FALSE;

arc_ccd_req_ps = FALSE;
arc_ccd_req_frame = FALSE;
arc_ccd_req_channels = FALSE;
arc_ccd_req_spectrum = FALSE;
arc_ccd_req_ps_stats = FALSE;
arc_ccd_req_vd_stats = FALSE;
arc_ccd_purge_ps_flag = FALSE;
arc_ccd_init_fft_flag = FALSE;
arc_ccd_recording = FALSE;
arc_ccd_ps_servo_on = FALSE;
arc_ccd_squarewave_on = FALSE;
arc_ccd_squarewave_loop = 0;
arc_ccd_usable_peak = FALSE;

arcccd_dith_scan.lower_limit = RTDAC_LOWER_LIMIT;
arcccd_dith_scan.upper_limit = RTDAC_UPPER_LIMIT;
arcccd_dith_scan.dx = MAX_DIFF;
arcccd_dith_scan.on = FALSE;

```



```

arcccd_dith_scan.going_up = TRUE;
arcccd_dith_scan.dx_index = 0;
arcccd_dith_scan.scan_index = 0;
arcccd_dith_scan.total_scans = 0;

// arc_ccd_vis_dither_on = FALSE;
// arc_ccd_vis_dither_counter = 0;
arc_ccd_vis_dith_sweep_um = vis_dither_rt_sweep_um_ptr();
arc_ccd_vis_dith_sweep_dac = vis_dither_rt_sweep_dac_ptr();

// set the vis dither parameters to the same as the vis dither module
arc_ccd_vd_par = vis_dither_rt_query_stats();

// initialize dummy array. used as imaginary input for fft functs.
for (j=0; j<MAX_NUM_SIMULTANEOUS_SPECTRA; j++)
{
for (i=0; i<FFT_SIZE; i++)
arc_ccd_im_fft_buf[j][i] = 0.0;
}

// initialize channel mask. Which pixels in spectrum will be sent vis
// send_channels.
arc_ccd_channel_list[0] = SPEC_SIZE;

for (i=0; i<FFT_SIZE; i++) arc_ccd_channel_list[i] = i;

#ifdef SPECTRUM_IO_DEBUG_MESSAGES
test_ps_avg = -1.0;
#endif

/* Add our local commands */

    if (chara_add_command(ARC_CCD_RT_SET_ROW,arc_ccd_rt_set_row) < 0)
    {
        printk("example: Could not get command %d.\n",
ARC_CCD_RT_SET_ROW);
        return -EINVAL;
    }

    if (chara_add_command(ARC_CCD_RT_GET_ROW,arc_ccd_rt_get_row) < 0)
    {
chara_remove_command(ARC_CCD_RT_SET_ROW);
        printk("example: Could not get command %d.\n",
ARC_CCD_RT_GET_ROW);
        return -EINVAL;
    }

    if (chara_add_command(ARC_CCD_RT_REQ_PS,arc_ccd_rt_req_ps) < 0)
    {
chara_remove_command(ARC_CCD_RT_GET_ROW);
chara_remove_command(ARC_CCD_RT_SET_ROW);
        printk("example: Could not get command %d.\n",
ARC_CCD_RT_REQ_PS);
        return -EINVAL;
    }

    if (chara_add_command(ARC_CCD_RT_REQ_FRAME,arc_ccd_rt_req_frame) < 0)
    {
chara_remove_command(ARC_CCD_RT_REQ_PS);
chara_remove_command(ARC_CCD_RT_GET_ROW);
chara_remove_command(ARC_CCD_RT_SET_ROW);
        printk("example: Could not get command %d.\n",
ARC_CCD_RT_REQ_FRAME);
        return -EINVAL;
    }

    if (chara_add_command(ARC_CCD_RT_SYNC_BB,arc_ccd_rt_sync_bb) < 0)
    {
chara_remove_command(ARC_CCD_RT_REQ_FRAME);
chara_remove_command(ARC_CCD_RT_REQ_PS);
chara_remove_command(ARC_CCD_RT_GET_ROW);
chara_remove_command(ARC_CCD_RT_SET_ROW);
        printk("example: Could not get command %d.\n",
ARC_CCD_RT_SYNC_BB);
        return -EINVAL;
    }

```

```

        if (chara_add_command(ARC_CCD_RT_CHECK_BB,arc_ccd_rt_check_bb) < 0)
        {
            chara_remove_command(ARC_CCD_RT_SYNC_BB);
            chara_remove_command(ARC_CCD_RT_REQ_FRAME);
            chara_remove_command(ARC_CCD_RT_REQ_PS);
            chara_remove_command(ARC_CCD_RT_GET_ROW);
            chara_remove_command(ARC_CCD_RT_SET_ROW);
            printk("example: Could not get command %d.\n",
ARC_CCD_RT_CHECK_BB);
            return -EINVAL;
        }

        if (chara_add_command(ARC_CCD_RT_SET_BIAS,arc_ccd_rt_set_bias)
< 0)
        {
            chara_remove_command(ARC_CCD_RT_CHECK_BB);
            chara_remove_command(ARC_CCD_RT_SYNC_BB);
            chara_remove_command(ARC_CCD_RT_REQ_FRAME);
            chara_remove_command(ARC_CCD_RT_REQ_PS);
            chara_remove_command(ARC_CCD_RT_GET_ROW);
            chara_remove_command(ARC_CCD_RT_SET_ROW);
            printk("example: Could not get command %d.\n",
ARC_CCD_RT_SET_BIAS);
            return -EINVAL;
        }

        if (chara_add_command(ARC_CCD_RT_SET_SPEC_FLAT,arc_ccd_rt_set_spec_flat) < 0)
        {
            chara_remove_command(ARC_CCD_RT_SET_BIAS);
            chara_remove_command(ARC_CCD_RT_CHECK_BB);
            chara_remove_command(ARC_CCD_RT_SYNC_BB);
            chara_remove_command(ARC_CCD_RT_REQ_FRAME);
            chara_remove_command(ARC_CCD_RT_REQ_PS);
            chara_remove_command(ARC_CCD_RT_GET_ROW);
            chara_remove_command(ARC_CCD_RT_SET_ROW);
            printk("example: Could not get command %d.\n",
ARC_CCD_RT_SET_SPEC_FLAT);
            return -EINVAL;
        }

        if (chara_add_command(ARC_CCD_RT_CHECK_FLAG,arc_ccd_rt_check_flag) < 0)
        {
            chara_remove_command(ARC_CCD_RT_SET_SPEC_FLAT);
            chara_remove_command(ARC_CCD_RT_SET_BIAS);
            chara_remove_command(ARC_CCD_RT_CHECK_BB);
            chara_remove_command(ARC_CCD_RT_SYNC_BB);
            chara_remove_command(ARC_CCD_RT_REQ_FRAME);
            chara_remove_command(ARC_CCD_RT_REQ_PS);
            chara_remove_command(ARC_CCD_RT_GET_ROW);
            chara_remove_command(ARC_CCD_RT_SET_ROW);
            printk("example: Could not get command %d.\n",
ARC_CCD_RT_CHECK_FLAG);
            return -EINVAL;
        }

        if (chara_add_command(ARC_CCD_RT_TOGGLE_FLAG,arc_ccd_rt_toggle_flag)
< 0)
        {
            chara_remove_command(ARC_CCD_RT_CHECK_FLAG);
            chara_remove_command(ARC_CCD_RT_SET_SPEC_FLAT);
            chara_remove_command(ARC_CCD_RT_SET_BIAS);
            chara_remove_command(ARC_CCD_RT_CHECK_BB);
            chara_remove_command(ARC_CCD_RT_SYNC_BB);
            chara_remove_command(ARC_CCD_RT_REQ_FRAME);
            chara_remove_command(ARC_CCD_RT_REQ_PS);
            chara_remove_command(ARC_CCD_RT_GET_ROW);
            chara_remove_command(ARC_CCD_RT_SET_ROW);
            printk("example: Could not get command %d.\n",
ARC_CCD_RT_TOGGLE_FLAG);
            return -EINVAL;
        }

        if (chara_add_command(ARC_CCD_RT_SET_SUPPRESS_PIXELS,
arc_ccd_rt_set_suppress_pixels) < 0)
        {
            chara_remove_command(ARC_CCD_RT_TOGGLE_FLAG);
            chara_remove_command(ARC_CCD_RT_CHECK_FLAG);
            chara_remove_command(ARC_CCD_RT_SET_SPEC_FLAT);
            chara_remove_command(ARC_CCD_RT_SET_BIAS);
            chara_remove_command(ARC_CCD_RT_CHECK_BB);

```

```

chara_remove_command(ARC_CCD_RT_SYNC_BB);
chara_remove_command(ARC_CCD_RT_REQ_FRAME);
chara_remove_command(ARC_CCD_RT_REQ_PS);
chara_remove_command(ARC_CCD_RT_GET_ROW);
chara_remove_command(ARC_CCD_RT_SET_ROW);
    printk("example: Could not get command %d.\n",
ARC_CCD_RT_SET_SUPPRESS_PIXELS);
    return -EINVAL;
}

    if (chara_add_command(ARC_CCD_RT_GET_SUPPRESS_PIXELS,
arc_ccd_rt_get_suppress_pixels) < 0)
    {
chara_remove_command(ARC_CCD_RT_SET_SUPPRESS_PIXELS);
chara_remove_command(ARC_CCD_RT_TOGGLE_FLAG);
chara_remove_command(ARC_CCD_RT_CHECK_FLAG);
chara_remove_command(ARC_CCD_RT_SET_SPEC_FLAT);
chara_remove_command(ARC_CCD_RT_SET_BIAS);
chara_remove_command(ARC_CCD_RT_CHECK_BB);
chara_remove_command(ARC_CCD_RT_SYNC_BB);
chara_remove_command(ARC_CCD_RT_REQ_FRAME);
chara_remove_command(ARC_CCD_RT_REQ_PS);
chara_remove_command(ARC_CCD_RT_GET_ROW);
chara_remove_command(ARC_CCD_RT_SET_ROW);
        printk("example: Could not get command %d.\n",
ARC_CCD_RT_GET_SUPPRESS_PIXELS);
        return -EINVAL;
    }

    if (chara_add_command(ARC_CCD_RT_REQ_SPECTRUM,
arc_ccd_rt_req_spectrum) < 0)
    {
chara_remove_command(ARC_CCD_RT_GET_SUPPRESS_PIXELS);
chara_remove_command(ARC_CCD_RT_SET_SUPPRESS_PIXELS);
chara_remove_command(ARC_CCD_RT_TOGGLE_FLAG);
chara_remove_command(ARC_CCD_RT_CHECK_FLAG);
chara_remove_command(ARC_CCD_RT_SET_SPEC_FLAT);
chara_remove_command(ARC_CCD_RT_SET_BIAS);
chara_remove_command(ARC_CCD_RT_CHECK_BB);
chara_remove_command(ARC_CCD_RT_SYNC_BB);
chara_remove_command(ARC_CCD_RT_REQ_FRAME);
chara_remove_command(ARC_CCD_RT_REQ_PS);
chara_remove_command(ARC_CCD_RT_GET_ROW);
chara_remove_command(ARC_CCD_RT_SET_ROW);
        printk("example: Could not get command %d.\n",
ARC_CCD_RT_REQ_SPECTRUM);
        return -EINVAL;
    }

    if (chara_add_command(ARC_CCD_RT_SET_PS_MEAN_NUM,
arc_ccd_rt_set_ps_mean_num) < 0)
    {
chara_remove_command(ARC_CCD_RT_REQ_SPECTRUM);
chara_remove_command(ARC_CCD_RT_GET_SUPPRESS_PIXELS);
chara_remove_command(ARC_CCD_RT_SET_SUPPRESS_PIXELS);
chara_remove_command(ARC_CCD_RT_TOGGLE_FLAG);
chara_remove_command(ARC_CCD_RT_CHECK_FLAG);
chara_remove_command(ARC_CCD_RT_SET_SPEC_FLAT);
chara_remove_command(ARC_CCD_RT_SET_BIAS);
chara_remove_command(ARC_CCD_RT_CHECK_BB);
chara_remove_command(ARC_CCD_RT_SYNC_BB);
chara_remove_command(ARC_CCD_RT_REQ_FRAME);
chara_remove_command(ARC_CCD_RT_REQ_PS);
chara_remove_command(ARC_CCD_RT_GET_ROW);
chara_remove_command(ARC_CCD_RT_SET_ROW);
        printk("example: Could not get command %d.\n",
ARC_CCD_RT_SET_PS_MEAN_NUM);
        return -EINVAL;
    }

    if (chara_add_command(ARC_CCD_RT_REQ_PURGE_PS_BUF,
arc_ccd_rt_req_purge_ps_buf) < 0)
    {
chara_remove_command(ARC_CCD_RT_SET_PS_MEAN_NUM);
chara_remove_command(ARC_CCD_RT_REQ_SPECTRUM);
chara_remove_command(ARC_CCD_RT_GET_SUPPRESS_PIXELS);
chara_remove_command(ARC_CCD_RT_SET_SUPPRESS_PIXELS);
chara_remove_command(ARC_CCD_RT_TOGGLE_FLAG);
chara_remove_command(ARC_CCD_RT_CHECK_FLAG);
chara_remove_command(ARC_CCD_RT_SET_SPEC_FLAT);

```

```

chara_remove_command(ARC_CCD_RT_SET_BIAS);
chara_remove_command(ARC_CCD_RT_CHECK_BB);
chara_remove_command(ARC_CCD_RT_SYNC_BB);
chara_remove_command(ARC_CCD_RT_REQ_FRAME);
chara_remove_command(ARC_CCD_RT_REQ_PS);
chara_remove_command(ARC_CCD_RT_GET_ROW);
chara_remove_command(ARC_CCD_RT_SET_ROW);
    printk("example: Could not get command %d.\n",
ARC_CCD_RT_REQ_PURGE_PS_BUF);
    return -EINVAL;
}

    if (chara_add_command(ARC_CCD_RT_INIT_FFT,
arc_ccd_rt_init_fft) < 0)
    {
chara_remove_command(ARC_CCD_RT_REQ_PURGE_PS_BUF);
chara_remove_command(ARC_CCD_RT_SET_PS_MEAN_NUM);
chara_remove_command(ARC_CCD_RT_REQ_SPECTRUM);
chara_remove_command(ARC_CCD_RT_GET_SUPPRESS_PIXELS);
chara_remove_command(ARC_CCD_RT_SET_SUPPRESS_PIXELS);
chara_remove_command(ARC_CCD_RT_TOGGLE_FLAG);
chara_remove_command(ARC_CCD_RT_CHECK_FLAG);
chara_remove_command(ARC_CCD_RT_SET_SPEC_FLAT);
chara_remove_command(ARC_CCD_RT_SET_BIAS);
chara_remove_command(ARC_CCD_RT_CHECK_BB);
chara_remove_command(ARC_CCD_RT_SYNC_BB);
chara_remove_command(ARC_CCD_RT_REQ_FRAME);
chara_remove_command(ARC_CCD_RT_REQ_PS);
chara_remove_command(ARC_CCD_RT_GET_ROW);
chara_remove_command(ARC_CCD_RT_SET_ROW);
    printk("example: Could not get command %d.\n",
ARC_CCD_RT_INIT_FFT);
    return -EINVAL;
}

    if (chara_add_command(ARC_CCD_RT_REQ_PS_STATS,
arc_ccd_rt_req_ps_stats) < 0)
    {
chara_remove_command(ARC_CCD_RT_INIT_FFT);
chara_remove_command(ARC_CCD_RT_REQ_PURGE_PS_BUF);
chara_remove_command(ARC_CCD_RT_SET_PS_MEAN_NUM);
chara_remove_command(ARC_CCD_RT_REQ_SPECTRUM);
chara_remove_command(ARC_CCD_RT_GET_SUPPRESS_PIXELS);
chara_remove_command(ARC_CCD_RT_SET_SUPPRESS_PIXELS);
chara_remove_command(ARC_CCD_RT_TOGGLE_FLAG);
chara_remove_command(ARC_CCD_RT_CHECK_FLAG);
chara_remove_command(ARC_CCD_RT_SET_SPEC_FLAT);
chara_remove_command(ARC_CCD_RT_SET_BIAS);
chara_remove_command(ARC_CCD_RT_CHECK_BB);
chara_remove_command(ARC_CCD_RT_SYNC_BB);
chara_remove_command(ARC_CCD_RT_REQ_FRAME);
chara_remove_command(ARC_CCD_RT_REQ_PS);
chara_remove_command(ARC_CCD_RT_GET_ROW);
chara_remove_command(ARC_CCD_RT_SET_ROW);
    printk("example: Could not get command %d.\n",
ARC_CCD_RT_REQ_PS_STATS);
    return -EINVAL;
}

    if (chara_add_command(ARC_CCD_RT_START_RECORD,
arc_ccd_rt_start_record) < 0)
    {
chara_remove_command(ARC_CCD_RT_REQ_PS_STATS);
chara_remove_command(ARC_CCD_RT_INIT_FFT);
chara_remove_command(ARC_CCD_RT_REQ_PURGE_PS_BUF);
chara_remove_command(ARC_CCD_RT_SET_PS_MEAN_NUM);
chara_remove_command(ARC_CCD_RT_REQ_SPECTRUM);
chara_remove_command(ARC_CCD_RT_GET_SUPPRESS_PIXELS);
chara_remove_command(ARC_CCD_RT_SET_SUPPRESS_PIXELS);
chara_remove_command(ARC_CCD_RT_TOGGLE_FLAG);
chara_remove_command(ARC_CCD_RT_CHECK_FLAG);
chara_remove_command(ARC_CCD_RT_SET_SPEC_FLAT);
chara_remove_command(ARC_CCD_RT_SET_BIAS);
chara_remove_command(ARC_CCD_RT_CHECK_BB);
chara_remove_command(ARC_CCD_RT_SYNC_BB);
chara_remove_command(ARC_CCD_RT_REQ_FRAME);
chara_remove_command(ARC_CCD_RT_REQ_PS);
chara_remove_command(ARC_CCD_RT_GET_ROW);
chara_remove_command(ARC_CCD_RT_SET_ROW);
    printk("example: Could not get command %d.\n",

```

```

ARC_CCD_RT_START_RECORD);
    return -EINVAL;
}

    if (chara_add_command(ARC_CCD_RT_END_RECORD,
arc_ccd_rt_end_record) < 0)
    {
chara_remove_command(ARC_CCD_RT_START_RECORD);
chara_remove_command(ARC_CCD_RT_REQ_PS_STATS);
chara_remove_command(ARC_CCD_RT_INIT_FFT);
chara_remove_command(ARC_CCD_RT_REQ_PURGE_PS_BUF);
chara_remove_command(ARC_CCD_RT_SET_PS_MEAN_NUM);
chara_remove_command(ARC_CCD_RT_REQ_SPECTRUM);
chara_remove_command(ARC_CCD_RT_GET_SUPPRESS_PIXELS);
chara_remove_command(ARC_CCD_RT_SET_SUPPRESS_PIXELS);
chara_remove_command(ARC_CCD_RT_TOGGLE_FLAG);
chara_remove_command(ARC_CCD_RT_CHECK_FLAG);
chara_remove_command(ARC_CCD_RT_SET_SPEC_FLAT);
chara_remove_command(ARC_CCD_RT_SET_BIAS);
chara_remove_command(ARC_CCD_RT_CHECK_BB);
chara_remove_command(ARC_CCD_RT_SYNC_BB);
chara_remove_command(ARC_CCD_RT_REQ_FRAME);
chara_remove_command(ARC_CCD_RT_REQ_PS);
chara_remove_command(ARC_CCD_RT_GET_ROW);
chara_remove_command(ARC_CCD_RT_SET_ROW);
        printk("example: Could not get command %d.\n",
ARC_CCD_RT_END_RECORD);
        return -EINVAL;
    }

    if (chara_add_command(ARC_CCD_RT_TWEAK_PID,
arc_ccd_rt_tweak_pid) < 0)
    {
chara_remove_command(ARC_CCD_RT_END_RECORD);
chara_remove_command(ARC_CCD_RT_START_RECORD);
chara_remove_command(ARC_CCD_RT_REQ_PS_STATS);
chara_remove_command(ARC_CCD_RT_INIT_FFT);
chara_remove_command(ARC_CCD_RT_REQ_PURGE_PS_BUF);
chara_remove_command(ARC_CCD_RT_SET_PS_MEAN_NUM);
chara_remove_command(ARC_CCD_RT_REQ_SPECTRUM);
chara_remove_command(ARC_CCD_RT_GET_SUPPRESS_PIXELS);
chara_remove_command(ARC_CCD_RT_SET_SUPPRESS_PIXELS);
chara_remove_command(ARC_CCD_RT_TOGGLE_FLAG);
chara_remove_command(ARC_CCD_RT_CHECK_FLAG);
chara_remove_command(ARC_CCD_RT_SET_SPEC_FLAT);
chara_remove_command(ARC_CCD_RT_SET_BIAS);
chara_remove_command(ARC_CCD_RT_CHECK_BB);
chara_remove_command(ARC_CCD_RT_SYNC_BB);
chara_remove_command(ARC_CCD_RT_REQ_FRAME);
chara_remove_command(ARC_CCD_RT_REQ_PS);
chara_remove_command(ARC_CCD_RT_GET_ROW);
chara_remove_command(ARC_CCD_RT_SET_ROW);
        printk("example: Could not get command %d.\n",
ARC_CCD_RT_TWEAK_PID);
        return -EINVAL;
    }

    if (chara_add_command(ARC_CCD_RT_NEW_TARGET_FREQ,
arc_ccd_rt_new_target_freq) < 0)
    {
chara_remove_command(ARC_CCD_RT_TWEAK_PID);
chara_remove_command(ARC_CCD_RT_END_RECORD);
chara_remove_command(ARC_CCD_RT_START_RECORD);
chara_remove_command(ARC_CCD_RT_REQ_PS_STATS);
chara_remove_command(ARC_CCD_RT_INIT_FFT);
chara_remove_command(ARC_CCD_RT_REQ_PURGE_PS_BUF);
chara_remove_command(ARC_CCD_RT_SET_PS_MEAN_NUM);
chara_remove_command(ARC_CCD_RT_REQ_SPECTRUM);
chara_remove_command(ARC_CCD_RT_GET_SUPPRESS_PIXELS);
chara_remove_command(ARC_CCD_RT_SET_SUPPRESS_PIXELS);
chara_remove_command(ARC_CCD_RT_TOGGLE_FLAG);
chara_remove_command(ARC_CCD_RT_CHECK_FLAG);
chara_remove_command(ARC_CCD_RT_SET_SPEC_FLAT);
chara_remove_command(ARC_CCD_RT_SET_BIAS);
chara_remove_command(ARC_CCD_RT_CHECK_BB);
chara_remove_command(ARC_CCD_RT_SYNC_BB);
chara_remove_command(ARC_CCD_RT_REQ_FRAME);
chara_remove_command(ARC_CCD_RT_REQ_PS);
chara_remove_command(ARC_CCD_RT_GET_ROW);
chara_remove_command(ARC_CCD_RT_SET_ROW);
    }

```

```

        printk("example: Could not get command %d.\n",
ARC_CCD_RT_NEW_TARGET_FREQ);
        return -EINVAL;
    }

    if (chara_add_command(ARC_CCD_RT_SET_FLAG_TRUE,
arc_ccd_rt_set_flag_true) < 0)
    {
        chara_remove_command(ARC_CCD_RT_NEW_TARGET_FREQ);
        chara_remove_command(ARC_CCD_RT_TWEAK_PID);
        chara_remove_command(ARC_CCD_RT_END_RECORD);
        chara_remove_command(ARC_CCD_RT_START_RECORD);
        chara_remove_command(ARC_CCD_RT_REQ_PS_STATS);
        chara_remove_command(ARC_CCD_RT_INIT_FFT);
        chara_remove_command(ARC_CCD_RT_REQ_PURGE_PS_BUF);
        chara_remove_command(ARC_CCD_RT_SET_PS_MEAN_NUM);
        chara_remove_command(ARC_CCD_RT_REQ_SPECTRUM);
        chara_remove_command(ARC_CCD_RT_GET_SUPPRESS_PIXELS);
        chara_remove_command(ARC_CCD_RT_SET_SUPPRESS_PIXELS);
        chara_remove_command(ARC_CCD_RT_TOGGLE_FLAG);
        chara_remove_command(ARC_CCD_RT_CHECK_FLAG);
        chara_remove_command(ARC_CCD_RT_SET_SPEC_FLAT);
        chara_remove_command(ARC_CCD_RT_SET_BIAS);
        chara_remove_command(ARC_CCD_RT_CHECK_BB);
        chara_remove_command(ARC_CCD_RT_SYNC_BB);
        chara_remove_command(ARC_CCD_RT_REQ_FRAME);
        chara_remove_command(ARC_CCD_RT_REQ_PS);
        chara_remove_command(ARC_CCD_RT_GET_ROW);
        chara_remove_command(ARC_CCD_RT_SET_ROW);
        printk("example: Could not get command %d.\n",
ARC_CCD_RT_SET_FLAG_TRUE);
        return -EINVAL;
    }

    if (chara_add_command(ARC_CCD_RT_SET_FLAG_FALSE,
arc_ccd_rt_set_flag_false) < 0)
    {
        chara_remove_command(ARC_CCD_RT_SET_FLAG_TRUE);
        chara_remove_command(ARC_CCD_RT_NEW_TARGET_FREQ);
        chara_remove_command(ARC_CCD_RT_TWEAK_PID);
        chara_remove_command(ARC_CCD_RT_END_RECORD);
        chara_remove_command(ARC_CCD_RT_START_RECORD);
        chara_remove_command(ARC_CCD_RT_REQ_PS_STATS);
        chara_remove_command(ARC_CCD_RT_INIT_FFT);
        chara_remove_command(ARC_CCD_RT_REQ_PURGE_PS_BUF);
        chara_remove_command(ARC_CCD_RT_SET_PS_MEAN_NUM);
        chara_remove_command(ARC_CCD_RT_REQ_SPECTRUM);
        chara_remove_command(ARC_CCD_RT_GET_SUPPRESS_PIXELS);
        chara_remove_command(ARC_CCD_RT_SET_SUPPRESS_PIXELS);
        chara_remove_command(ARC_CCD_RT_TOGGLE_FLAG);
        chara_remove_command(ARC_CCD_RT_CHECK_FLAG);
        chara_remove_command(ARC_CCD_RT_SET_SPEC_FLAT);
        chara_remove_command(ARC_CCD_RT_SET_BIAS);
        chara_remove_command(ARC_CCD_RT_CHECK_BB);
        chara_remove_command(ARC_CCD_RT_SYNC_BB);
        chara_remove_command(ARC_CCD_RT_REQ_FRAME);
        chara_remove_command(ARC_CCD_RT_REQ_PS);
        chara_remove_command(ARC_CCD_RT_GET_ROW);
        chara_remove_command(ARC_CCD_RT_SET_ROW);
        printk("example: Could not get command %d.\n",
ARC_CCD_RT_SET_FLAG_FALSE);
        return -EINVAL;
    }

    if (chara_add_command(ARC_CCD_RT_SETUP_DITHER_SCAN,
arc_ccd_rt_setup_dither_scan) < 0)
    {
        chara_remove_command(ARC_CCD_RT_SET_FLAG_FALSE);
        chara_remove_command(ARC_CCD_RT_SET_FLAG_TRUE);
        chara_remove_command(ARC_CCD_RT_NEW_TARGET_FREQ);
        chara_remove_command(ARC_CCD_RT_TWEAK_PID);
        chara_remove_command(ARC_CCD_RT_END_RECORD);
        chara_remove_command(ARC_CCD_RT_START_RECORD);
        chara_remove_command(ARC_CCD_RT_REQ_PS_STATS);
        chara_remove_command(ARC_CCD_RT_INIT_FFT);
        chara_remove_command(ARC_CCD_RT_REQ_PURGE_PS_BUF);
        chara_remove_command(ARC_CCD_RT_SET_PS_MEAN_NUM);
        chara_remove_command(ARC_CCD_RT_REQ_SPECTRUM);
        chara_remove_command(ARC_CCD_RT_GET_SUPPRESS_PIXELS);
        chara_remove_command(ARC_CCD_RT_SET_SUPPRESS_PIXELS);
    }

```

```

chara_remove_command(ARC_CCD_RT_TOGGLE_FLAG);
chara_remove_command(ARC_CCD_RT_CHECK_FLAG);
chara_remove_command(ARC_CCD_RT_SET_SPEC_FLAT);
chara_remove_command(ARC_CCD_RT_SET_BIAS);
chara_remove_command(ARC_CCD_RT_CHECK_BB);
chara_remove_command(ARC_CCD_RT_SYNC_BB);
chara_remove_command(ARC_CCD_RT_REQ_FRAME);
chara_remove_command(ARC_CCD_RT_REQ_PS);
chara_remove_command(ARC_CCD_RT_GET_ROW);
chara_remove_command(ARC_CCD_RT_SET_ROW);
        printk("example: Could not get command %d.\n",
ARC_CCD_RT_SETUP_DITHER_SCAN);
        return -EINVAL;
    }

    if (chara_add_command(ARC_CCD_RT_ACTIVATE_DITHER_SCAN,
arc_ccd_rt_activate_dither_scan) < 0)
    {
chara_remove_command(ARC_CCD_RT_SETUP_DITHER_SCAN);
chara_remove_command(ARC_CCD_RT_SET_FLAG_FALSE);
chara_remove_command(ARC_CCD_RT_SET_FLAG_TRUE);
chara_remove_command(ARC_CCD_RT_NEW_TARGET_FREQ);
chara_remove_command(ARC_CCD_RT_TWEAK_PID);
chara_remove_command(ARC_CCD_RT_END_RECORD);
chara_remove_command(ARC_CCD_RT_START_RECORD);
chara_remove_command(ARC_CCD_RT_REQ_PS_STATS);
chara_remove_command(ARC_CCD_RT_INIT_FFT);
chara_remove_command(ARC_CCD_RT_REQ_PURGE_PS_BUF);
chara_remove_command(ARC_CCD_RT_SET_PS_MEAN_NUM);
chara_remove_command(ARC_CCD_RT_REQ_SPECTRUM);
chara_remove_command(ARC_CCD_RT_GET_SUPPRESS_PIXELS);
chara_remove_command(ARC_CCD_RT_SET_SUPPRESS_PIXELS);
chara_remove_command(ARC_CCD_RT_TOGGLE_FLAG);
chara_remove_command(ARC_CCD_RT_CHECK_FLAG);
chara_remove_command(ARC_CCD_RT_SET_SPEC_FLAT);
chara_remove_command(ARC_CCD_RT_SET_BIAS);
chara_remove_command(ARC_CCD_RT_CHECK_BB);
chara_remove_command(ARC_CCD_RT_SYNC_BB);
chara_remove_command(ARC_CCD_RT_REQ_FRAME);
chara_remove_command(ARC_CCD_RT_REQ_PS);
chara_remove_command(ARC_CCD_RT_GET_ROW);
chara_remove_command(ARC_CCD_RT_SET_ROW);
        printk("example: Could not get command %d.\n",
ARC_CCD_RT_ACTIVATE_DITHER_SCAN);
        return -EINVAL;
    }

    if (chara_add_command(ARC_CCD_RT_REQ_CHANNELS,
arc_ccd_rt_req_channels) < 0)
    {
chara_remove_command(ARC_CCD_RT_ACTIVATE_DITHER_SCAN);
chara_remove_command(ARC_CCD_RT_SETUP_DITHER_SCAN);
chara_remove_command(ARC_CCD_RT_SET_FLAG_FALSE);
chara_remove_command(ARC_CCD_RT_SET_FLAG_TRUE);
chara_remove_command(ARC_CCD_RT_NEW_TARGET_FREQ);
chara_remove_command(ARC_CCD_RT_TWEAK_PID);
chara_remove_command(ARC_CCD_RT_END_RECORD);
chara_remove_command(ARC_CCD_RT_START_RECORD);
chara_remove_command(ARC_CCD_RT_REQ_PS_STATS);
chara_remove_command(ARC_CCD_RT_INIT_FFT);
chara_remove_command(ARC_CCD_RT_REQ_PURGE_PS_BUF);
chara_remove_command(ARC_CCD_RT_SET_PS_MEAN_NUM);
chara_remove_command(ARC_CCD_RT_REQ_SPECTRUM);
chara_remove_command(ARC_CCD_RT_GET_SUPPRESS_PIXELS);
chara_remove_command(ARC_CCD_RT_SET_SUPPRESS_PIXELS);
chara_remove_command(ARC_CCD_RT_TOGGLE_FLAG);
chara_remove_command(ARC_CCD_RT_CHECK_FLAG);
chara_remove_command(ARC_CCD_RT_SET_SPEC_FLAT);
chara_remove_command(ARC_CCD_RT_SET_BIAS);
chara_remove_command(ARC_CCD_RT_CHECK_BB);
chara_remove_command(ARC_CCD_RT_SYNC_BB);
chara_remove_command(ARC_CCD_RT_REQ_FRAME);
chara_remove_command(ARC_CCD_RT_REQ_PS);
chara_remove_command(ARC_CCD_RT_GET_ROW);
chara_remove_command(ARC_CCD_RT_SET_ROW);
        printk("example: Could not get command %d.\n",
ARC_CCD_RT_REQ_CHANNELS);
        return -EINVAL;
    }
}

```

```

        if (chara_add_command(ARC_CCD_RT_SET_CHANNELS,
arc_ccd_rt_set_channels) < 0)
        {
            chara_remove_command(ARC_CCD_RT_REQ_CHANNELS);
            chara_remove_command(ARC_CCD_RT_ACTIVATE_DITHER_SCAN);
            chara_remove_command(ARC_CCD_RT_SETUP_DITHER_SCAN);
            chara_remove_command(ARC_CCD_RT_SET_FLAG_FALSE);
            chara_remove_command(ARC_CCD_RT_SET_FLAG_TRUE);
            chara_remove_command(ARC_CCD_RT_NEW_TARGET_FREQ);
            chara_remove_command(ARC_CCD_RT_TWEAK_PID);
            chara_remove_command(ARC_CCD_RT_END_RECORD);
            chara_remove_command(ARC_CCD_RT_START_RECORD);
            chara_remove_command(ARC_CCD_RT_REQ_PS_STATS);
            chara_remove_command(ARC_CCD_RT_INIT_FFT);
            chara_remove_command(ARC_CCD_RT_REQ_PURGE_PS_BUF);
            chara_remove_command(ARC_CCD_RT_SET_PS_MEAN_NUM);
            chara_remove_command(ARC_CCD_RT_REQ_SPECTRUM);
            chara_remove_command(ARC_CCD_RT_GET_SUPPRESS_PIXELS);
            chara_remove_command(ARC_CCD_RT_SET_SUPPRESS_PIXELS);
            chara_remove_command(ARC_CCD_RT_TOGGLE_FLAG);
            chara_remove_command(ARC_CCD_RT_CHECK_FLAG);
            chara_remove_command(ARC_CCD_RT_SET_SPEC_FLAT);
            chara_remove_command(ARC_CCD_RT_SET_BIAS);
            chara_remove_command(ARC_CCD_RT_CHECK_BB);
            chara_remove_command(ARC_CCD_RT_SYNC_BB);
            chara_remove_command(ARC_CCD_RT_REQ_FRAME);
            chara_remove_command(ARC_CCD_RT_REQ_PS);
            chara_remove_command(ARC_CCD_RT_GET_ROW);
            chara_remove_command(ARC_CCD_RT_SET_ROW);
            printk("example: Could not get command %d.\n",
ARC_CCD_RT_SET_CHANNELS);
            return -EINVAL;
        }

        if (chara_add_command(ARC_CCD_RT_SET_PS_FLAT,
arc_ccd_rt_set_ps_flat) < 0)
        {
            chara_remove_command(ARC_CCD_RT_SET_CHANNELS);
            chara_remove_command(ARC_CCD_RT_REQ_CHANNELS);
            chara_remove_command(ARC_CCD_RT_ACTIVATE_DITHER_SCAN);
            chara_remove_command(ARC_CCD_RT_SETUP_DITHER_SCAN);
            chara_remove_command(ARC_CCD_RT_SET_FLAG_FALSE);
            chara_remove_command(ARC_CCD_RT_SET_FLAG_TRUE);
            chara_remove_command(ARC_CCD_RT_NEW_TARGET_FREQ);
            chara_remove_command(ARC_CCD_RT_TWEAK_PID);
            chara_remove_command(ARC_CCD_RT_END_RECORD);
            chara_remove_command(ARC_CCD_RT_START_RECORD);
            chara_remove_command(ARC_CCD_RT_REQ_PS_STATS);
            chara_remove_command(ARC_CCD_RT_INIT_FFT);
            chara_remove_command(ARC_CCD_RT_REQ_PURGE_PS_BUF);
            chara_remove_command(ARC_CCD_RT_SET_PS_MEAN_NUM);
            chara_remove_command(ARC_CCD_RT_REQ_SPECTRUM);
            chara_remove_command(ARC_CCD_RT_GET_SUPPRESS_PIXELS);
            chara_remove_command(ARC_CCD_RT_SET_SUPPRESS_PIXELS);
            chara_remove_command(ARC_CCD_RT_TOGGLE_FLAG);
            chara_remove_command(ARC_CCD_RT_CHECK_FLAG);
            chara_remove_command(ARC_CCD_RT_SET_SPEC_FLAT);
            chara_remove_command(ARC_CCD_RT_SET_BIAS);
            chara_remove_command(ARC_CCD_RT_CHECK_BB);
            chara_remove_command(ARC_CCD_RT_SYNC_BB);
            chara_remove_command(ARC_CCD_RT_REQ_FRAME);
            chara_remove_command(ARC_CCD_RT_REQ_PS);
            chara_remove_command(ARC_CCD_RT_GET_ROW);
            chara_remove_command(ARC_CCD_RT_SET_ROW);
            printk("example: Could not get command %d.\n",
ARC_CCD_RT_SET_PS_FLAT);
            return -EINVAL;
        }

        if (chara_add_command(ARC_CCD_RT_REQ_VD_STATS,
arc_ccd_rt_req_vd_stats) < 0)
        {
            chara_remove_command(ARC_CCD_RT_SET_PS_FLAT);
            chara_remove_command(ARC_CCD_RT_SET_CHANNELS);
            chara_remove_command(ARC_CCD_RT_REQ_CHANNELS);
            chara_remove_command(ARC_CCD_RT_ACTIVATE_DITHER_SCAN);
            chara_remove_command(ARC_CCD_RT_SETUP_DITHER_SCAN);
            chara_remove_command(ARC_CCD_RT_SET_FLAG_FALSE);
            chara_remove_command(ARC_CCD_RT_SET_FLAG_TRUE);

```



```

chara_remove_command(ARC_CCD_RT_NEW_TARGET_FREQ);
chara_remove_command(ARC_CCD_RT_TWEAK_PID);
chara_remove_command(ARC_CCD_RT_END_RECORD);
chara_remove_command(ARC_CCD_RT_START_RECORD);
chara_remove_command(ARC_CCD_RT_REQ_PS_STATS);
chara_remove_command(ARC_CCD_RT_INIT_FFT);
chara_remove_command(ARC_CCD_RT_REQ_PURGE_PS_BUF);
chara_remove_command(ARC_CCD_RT_SET_PS_MEAN_NUM);
chara_remove_command(ARC_CCD_RT_REQ_SPECTRUM);
chara_remove_command(ARC_CCD_RT_GET_SUPPRESS_PIXELS);
chara_remove_command(ARC_CCD_RT_SET_SUPPRESS_PIXELS);
chara_remove_command(ARC_CCD_RT_TOGGLE_FLAG);
chara_remove_command(ARC_CCD_RT_CHECK_FLAG);
chara_remove_command(ARC_CCD_RT_SET_SPEC_FLAT);
chara_remove_command(ARC_CCD_RT_SET_BIAS);
chara_remove_command(ARC_CCD_RT_CHECK_BB);
chara_remove_command(ARC_CCD_RT_SYNC_BB);
chara_remove_command(ARC_CCD_RT_REQ_FRAME);
chara_remove_command(ARC_CCD_RT_REQ_PS);
chara_remove_command(ARC_CCD_RT_GET_ROW);
chara_remove_command(ARC_CCD_RT_SET_ROW);
        printk("example: Could not get command %d.\n",
ARC_CCD_RT_REQ_VD_STATS);
        return -EINVAL;
    }

// setup fft
fft_setup_fft(FFT_POW_2);

// register callback function with camera driver
rt_astropci_reg_callback(arc_ccd_rt_callback);

/* That is all */
return 0;
} /* init_module() */

/*****
/* arc_ccd_rt_set_row() */
/* */
/* This function sets which row from the bb will be used for the */
/* spectrum. Row number is in raster order (from top of window) */
*****/

int arc_ccd_rt_set_row(unsigned char arg)
{
    int i;
    int err = 0;

    if ( (err=rtf_get(DATA_IN_FIFO, arc_ccd_row, sizeof(arc_ccd_row)*(MAX_NUM_SIMULTANEOUS_SPECTRA+1))
    < 0)

    return err;

    if ( (err=rtf_get(DATA_IN_FIFO, &arc_ccd_row_offset,
    sizeof(arc_ccd_row_offset)))
    < 0)

    return err;

    // if row value doesn't lie in bounding box
    for (i=1; i<arc_ccd_row[0];i++)

    if ( arc_ccd_row[i] > (arc_ccd_bbdata.dy - 1) ||
    arc_ccd_row[i] < 0 )
    {
        arc_ccd_row[i] = -1;
        arc_ccd_row[0] = 0;
#ifdef DEBUG_MESSAGES

        chara_printerr(
        "Row %d is outside of bounding box. Row not set."
        ,arc_ccd_row);

#endif
    }
    return err;
}

```

```

}

// if row offset puts spectrum outside of quadrant
if ( arc_ccd_row_offset + SPECTRUM_WIDTH > arc_ccd_bbdata.dx ||
    arc_ccd_row_offset < 0)
{
    arc_ccd_row_offset = DEFAULT_SPEC_ROW_OFFSET;

    chara_printerr(
        "Row offset places spectrum outside of chip quadrant. Offset reset to default value."
        ,arc_ccd_row_offset);
}

#ifdef DEBUG_MESSAGES
chara_printerr("I got the number of rows: %d.",arc_ccd_row[0]);
if(arc_ccd_row[0] > 0)
{
    for(i=1;i<arc_ccd_row[0];i++)
        chara_printerr("I got the row value: %d.",arc_ccd_row[i]);
}
chara_printerr("I got the row offset value: %d.",arc_ccd_row_offset);
#endif

arc_ccd_purge_ps_flag = TRUE;

return 0;
} // arc_ccd_rt_set_row()

/*****
/* arc_ccd_rt_get_row() */
/* */
/* This function is used to verify which row(s) out of the ccd is being */
/* used for the spectrum. */
*****/

int arc_ccd_rt_get_row(unsigned char arg)
{
    int err = 0;

    if ((err=rtdf_put(DATA_OUT_FIFO, arc_ccd_row, sizeof(arc_ccd_row)*(MAX_NUM_SIMULTANEOUS_SPECTRA+1) ))<0)
        return err;

#ifdef DEBUG_MESSAGES
chara_printerr("I sent the number of rows: %d.",arc_ccd_row[0]);
if(arc_ccd_row[0] > 0)
{
    for(i=1;i<=arc_ccd_row[0];i++)
        chara_printerr("I sent the row value: %d.",arc_ccd_row[i]);
}
#endif
return 0;
} /* arc_ccd_rt_get_row() */

/*****
/* arc_ccd_rt_req_ps() */
/* */
/* Sets arc_ccd_req_ps flag to true so callback function will send data */
/* to FIFO using arc_ccd_get_ps() */
*****/

int arc_ccd_rt_req_ps(unsigned char arg)
{
#ifdef REQ_PS_LPT_TOGGLE
    if(arc_ccd_use_lpt_toggle)
    {
        chara_set_bit_lpt(0);
    }
#endif
    arc_ccd_req_ps = TRUE;

#ifdef REQ_PS_LPT_TOGGLE
    if(arc_ccd_use_lpt_toggle)
    {

```

```

chara_unset_bit_lpt(0);
    }
#endif
return 0;
} // arc_ccd_rt_req_ps()

/*****
/* arc_ccd_rt_get_ps() */
/* */
/* Used by user side to request the latest power spectrum. */
/* Puts arc_ccd_ps_buf[] into DATA_OUT_FIFO. */
*****/

int arc_ccd_rt_get_ps(void)
{
#ifdef SPECTRUM_IO_DEBUG_MESSAGES
int i;
#endif
int err = 0;

// toggle lpt port, for use when timing the get_ps function

#ifdef GET_PS_LPT_TOGGLE

if(arc_ccd_use_lpt_toggle)
{
chara_set_bit_lpt(0);
}
#endif

if ( (err=rtf_put( DATA_OUT_FIFO, &(arc_ccd_ps_buf[0][0]),
PS_SIZE * sizeof(float) ))<0 )
{
chara_printerr("Failed to send data to UI (err = %d).",
err);
return err;
}

#ifdef SPECTRUM_IO_DEBUG_MESSAGES

chara_printerr("I sent %d bytes of arc_ccd_ps_buf[0][].\nShould be %d.",
err, PS_SIZE * sizeof(float));

// check to see that all of the ps is there. for use when
// using a fake stair-step ps

test_ps_avg = 0.0;
for (i=0; i<PS_SIZE; i++)
test_ps_avg += arc_ccd_ps_buf[0][i];
test_ps_avg /= PS_SIZE;
chara_printerr("PS mean being sent = %d", (int)test_ps_avg);
#endif

// toggle lpt port, for use when timing the get_ps function

#ifdef GET_PS_LPT_TOGGLE

if(arc_ccd_use_lpt_toggle)
{
chara_unset_bit_lpt(0);
}
#endif

return 0;
} // arc_ccd_rt_get_ps()

/*****
/* arc_ccd_rt_req_spectrum() */
/* */
/* Sets arc_ccd_req_spectrum flag to true so callback function will */
/* send data to FIFO using arc_ccd_get_ps() */
/* */
*****/

int arc_ccd_rt_req_spectrum(unsigned char arg)
{
arc_ccd_req_spectrum = TRUE;

return 0;
} // arc_ccd_rt_req_spectrum()

```

```

/*****
/* arc_ccd_rt_get_spectrum() */
/* */
/* Used by user side to request the latest spectrum. */
/* Puts arc_ccd_spect_buf[] into DATA_OUT_FIFO. */
*****/

int arc_ccd_rt_get_spectrum(void)
{
    int i;
    int err = 0;

    for(i=0; i<arc_ccd_row[0]; i++)
    {
        if ( (err=rtf_put( DATA_OUT_FIFO, &(arc_ccd_spect_buf[i][0]),
            SPEC_SIZE * sizeof(float) )<0 )
        {
            chara_printerr("Failed to send data to UI (err = %d).",
                           err);
            return err;
        }

#ifdef SPECTRUM_IO_DEBUG_MESSAGES

        chara_printerr("I sent %d bytes of arc_ccd_spect_buf[].",err);

#endif
    }

    return 0;
} // arc_ccd_rt_get_spectrum()

/*****
/* arc_ccd_rt_req_frame() */
/* */
/* Sets arc_ccd_req_frame flag to true so callback function will send */
/* data to FIFO using arc_ccd_get_frame() */
/* */
*****/
int arc_ccd_rt_req_frame(unsigned char arg)
{
    arc_ccd_req_frame = TRUE;

    return 0;
} // arc_ccd_rt_req_frame()

/*****
/* arc_ccd_rt_get_frame() */
/* */
/* User side request for current frame. Puts arc_ccd_frame_buf[] into */
/* DATA_OUT_FIFO. */
*****/

int arc_ccd_rt_get_frame(void)
{
    int err = 0;

    if ((err=rtf_put(DATA_OUT_FIFO, &arc_ccd_frame_time,
        sizeof(arc_ccd_frame_time))<0)
    {
        chara_printerr("Failed to send data to UI (err = %d).",
                       err);
        return err;
    }

    if ((err=rtf_put(DATA_OUT_FIFO, &arc_ccd_frame_number,
        sizeof(arc_ccd_frame_number))<0)
    {
        chara_printerr("Failed to send data to UI (err = %d).",
                       err);
        return err;
    }

    if ( (err=rtf_put( DATA_OUT_FIFO, frame,
        (arc_ccd_num_quadrants * arc_ccd_bbdata.dx
        * arc_ccd_bbdata.dy) * sizeof(*frame) ) ) <0 )
    {

```

```

chara_printerr("Failed to send data to UI (err = %d).",
               err);
return err;
}

#ifdef FRAME_IO_DEBUG_MESSAGES

chara_printerr("I sent frame time: %ld.",arc_ccd_frame_time);
chara_printerr("I sent frame number: %lu.",arc_ccd_frame_number);
chara_printerr("I sent %d bytes of data.",err);

#endif
return 0;
} // arc_ccd_rt_get_frame()

/*****
/* arc_ccd_rt_req_ps_stats() */
/* */
/* Sets arc_ccd_req_ps_stats flag to true so callback function will */
/* send data to FIFO using arc_ccd_get_ps() */
/* */
*****/

int arc_ccd_rt_req_ps_stats(unsigned char arg)
{
arc_ccd_req_ps_stats = TRUE;

return 0;
} // arc_ccd_rt_req_ps_stats()

/*****
/* arc_ccd_rt_get_ps_stats() */
/* */
/* Used by user side to request the latest power spectrum stats. */
/* Puts the arc_ccd_ps_peak data structure into DATA_OUT_FIFO. */
*****/

int arc_ccd_rt_get_ps_stats(void)
{
int err = 0;

if ((err=rtdf_put(DATA_OUT_FIFO, &arc_ccd_ps_peak,
sizeof(arc_ccd_ps_peak)))<0)
{
chara_printerr("Failed to send ps stats to UI (err = %d).",
               err);
return err;
}

return 0;
} // arc_ccd_rt_get_ps_stats()

/*****
/* arc_ccd_rt_req_channels() */
/* */
/* Sets arc_ccd_req_channels flag to true so callback function will */
/* send data to FIFO using arc_ccd_get_channels(). */
/* */
*****/

int arc_ccd_rt_req_channels(unsigned char arg)
{
arc_ccd_req_channels = TRUE;

return 0;
} // arc_ccd_rt_req_channels()

/*****
/* arc_ccd_rt_get_channels() */
/* */
/* User side request for channels from current spectrum. */
/* Puts channel values from spect_buf which are on the channel_list */
/* into DATA_OUT_FIFO. */
*****/

int arc_ccd_rt_get_channels(void)
{
int i;
int err = 0;

```

```

        if ((err=rtf_put(DATA_OUT_FIFO, &arc_ccd_frame_time,
sizeof(arc_ccd_frame_time)))<0)
        {
        chara_printerr("Failed to send frame time to UI (err = %d).",
                        err);
        return err;
        }

        if ((err=rtf_put(DATA_OUT_FIFO, &arc_ccd_frame_number,
sizeof(arc_ccd_frame_number)))<0)
        {
        chara_printerr("Failed to send frame number to UI (err = %d).",
                        err);
        return err;
        }

        for(i=0; i<arc_ccd_row[0]; i++)
        {
        if ( (err=rtf_put( DATA_OUT_FIFO,
&(arc_ccd_channel_buf[i][0]),
arc_ccd_channel_list[0] *
sizeof(arc_ccd_channel_buf[0][0]) )
) < 0
        )
        {
        chara_printerr(
        "Failed to send channel data to UI (err = %d).",
        err);
        return err;
        }
        }

#ifdef CHANNEL_IO_DEBUG_MESSAGES

        chara_printerr("I sent frame time: %ld.",arc_ccd_frame_time);
        chara_printerr("I sent frame number: %lu.",arc_ccd_frame_number);
        chara_printerr("I sent %d bytes of data.",err);

#endif
        return 0;
    } // arc_ccd_rt_get_channels()

    /*****
    /* arc_ccd_rt_req_vd_stats() */
    /* */
    /* Sets arc_ccd_req_vd_stats flag to true so callback function will */
    /* send data to FIFO using arc_ccd_get_vd() */
    /* */
    *****/

    int arc_ccd_rt_req_vd_stats(unsigned char arg)
    {
    arc_ccd_req_vd_stats = TRUE;

    return 0;
    } // arc_ccd_rt_req_vd_stats()

    /*****
    /* arc_ccd_rt_get_vd_stats() */
    /* */
    /* Used by user side to request the latest vis_dither data. */
    /* Puts the arc_ccd_vd_par data structure into DATA_OUT_FIFO. */
    *****/

    int arc_ccd_rt_get_vd_stats(void)
    {
    int err = 0;

        if ((err=rtf_put(DATA_OUT_FIFO, &arc_ccd_vd_par,
sizeof(arc_ccd_vd_par)))<0)
        {
        chara_printerr("Failed to send vis dither data to UI (err = %d).",
                        err);
        return err;
        }

#ifdef DITHER_IO_DEBUG_MESSAGES

        chara_printerr("I sent %d bytes of dither data.",err);

```

```

#endif
return 0;

} // arc_ccd_rt_get_vd_stats()

/*****
/* arc_ccd_rt_sync_bb() */
/* */
/* This function should be called whenever RTCCD_SET_ROI is sent to the */
/* camera. Keeps this copy of arc_ccd_bbdata current with the camera. */
*****/

int arc_ccd_rt_sync_bb(unsigned char arg)
{
    int err = 0;

    if ( (err=rtf_get(DATA_IN_FIFO, &arc_ccd_bbdata, sizeof(SROI))) < 0 ||
        rtf_get(DATA_IN_FIFO, &arc_ccd_bbquadrant,
        sizeof(arc_ccd_bbquadrant)) < 0)
    {

        // default bounding box values

#ifdef USE_CCD_OVERSCAN
        arc_ccd_bbdata.x = 0;
        arc_ccd_bbdata.y = 0;
        arc_ccd_bbdata.dx = CCD_X_PIXELS/2 ;
        arc_ccd_bbdata.dy = CCD_Y_PIXELS/2 ;
#else
        arc_ccd_bbdata.x = USEFUL_X;
        arc_ccd_bbdata.y = USEFUL_Y;
        arc_ccd_bbdata.dx = USEFUL_DX/2;
        arc_ccd_bbdata.dy = USEFUL_DY/2;
        arc_ccd_bbquadrant = 0;
        arc_ccd_num_quadrants = 4;
#endif
        return err;
    }

    //check to see if bb outside of chip

    if ( arc_ccd_bbdata.x < 0 ||
        arc_ccd_bbdata.x >= USEFUL_X + USEFUL_DX/2 - 1 ||
        arc_ccd_bbdata.y < 0 ||
        arc_ccd_bbdata.y >= USEFUL_Y + USEFUL_DY/2 - 1 ||
        (arc_ccd_bbdata.x+arc_ccd_bbdata.dx) >
        (USEFUL_X + USEFUL_DX/2) ||
        (arc_ccd_bbdata.y+arc_ccd_bbdata.dy) >
        (USEFUL_Y + USEFUL_DY/2) ||
        arc_ccd_bbdata.dx == 0 || arc_ccd_bbdata.dy ==0 )
    {
#ifdef DEBUG_MESSAGES

        chara_pinterr("Bounding box coords not possible. Box reset.");

#endif

        // default bounding box values
#ifdef USE_CCD_OVERSCAN
        arc_ccd_bbdata.x = 0;
        arc_ccd_bbdata.y = 0;
        arc_ccd_bbdata.dx = CCD_X_PIXELS/2 ;
        arc_ccd_bbdata.dy = CCD_Y_PIXELS/2 ;
#else
        arc_ccd_bbdata.x = USEFUL_X;
        arc_ccd_bbdata.y = USEFUL_Y;
        arc_ccd_bbdata.dx = USEFUL_DX/2;
        arc_ccd_bbdata.dy = USEFUL_DY/2;
        arc_ccd_bbquadrant = 0;
        arc_ccd_num_quadrants = 4;
#endif
        return err;
    }

    // set the number of quadrants, a pixel multiplier
    // used in figuring out buffer sizes

    if(arc_ccd_bbquadrant<1 || arc_ccd_bbquadrant >4)

```

```

arc_ccd_num_quadrants = 4;

else

arc_ccd_num_quadrants = 1;

#ifdef DEBUG_MESSAGES

chara_printerr("I got the bb: x:%d y:%d dx:%d dy:%d.",
arc_ccd_bbdata.x,arc_ccd_bbdata.y,arc_ccd_bbdata.dx,
arc_ccd_bbdata.dy);

#endif
return 0;

} // arc_ccd_rt_sync_bb()

/*****
* arc_ccd_rt_check_bb() */
* */
/* This function is used to check the bounding box values stored by */
/* this module, and make sure they follow those given to the camera. */
*****/

int arc_ccd_rt_check_bb(unsigned char arg)
{
    int err = 0;

    if ((err=rtf_put(DATA_OUT_FIFO, &arc_ccd_bbdata, sizeof(SROI)))<0 ||
rtf_put(DATA_OUT_FIFO, &arc_ccd_bbquadrant,
sizeof(arc_ccd_bbquadrant)) < 0 )

return err;

#ifdef DEBUG_MESSAGES

chara_printerr("I sent the bounding box: x: %d y: %d dx: %d dy: %d",
arc_ccd_bbdata.x,arc_ccd_bbdata.y,arc_ccd_bbdata.dx,
arc_ccd_bbdata.dy);

#endif
return 0;

} // arc_ccd_rt_check_bb()

/*****
* arc_ccd_rt_set_bias() */
* */
/* Reads a arc_ccd_bias frame from the DATA_IN_FIFO. Sets */
/* arc_ccd_bias_has_been_set and arc_ccd_use_bias flags to TRUE. */
*****/

int arc_ccd_rt_set_bias(unsigned char arg)
{
    int err = 0;

    if ( (err=rtf_get(DATA_IN_FIFO, arc_ccd_bias,
arc_ccd_num_quadrants * arc_ccd_bbdata.dx
* arc_ccd_bbdata.dy * sizeof(long int)) ) < 0)

return err;

arc_ccd_bias_has_been_set = TRUE;
arc_ccd_use_bias = TRUE;

// purge ps_buf

arc_ccd_purge_ps_flag = TRUE;

return err;
} // arc_ccd_rt_set_bias()

/*****
* arc_ccd_rt_set_spec_flat() */
* */
/* Reads a spectrum flat from DATA_IN_FIFO. Sets */
/* arc_ccd_spec_flat_has_been_set and arc_ccd_use_spec_flat flags */
/* to TRUE. */

```



```

/*****

int arc_ccd_rt_set_spec_flat(unsigned char arg)
{
int i;
int err=0;

for(i=0; i<arc_ccd_row[0]; i++)
{
    if ( (err=rtf_get(DATA_IN_FIFO, &(arc_ccd_spec_flat[i][0]),
FFT_SIZE*sizeof(float))) < 0)

return err;
}
arc_ccd_spec_flat_has_been_set = TRUE;
arc_ccd_use_spec_flat = TRUE;

// purge ps_buf

arc_ccd_purge_ps_flag = TRUE;

return err;
} // arc_ccd_rt_set_spec_flat()

/*****
/* arc_ccd_rt_set_ps_flat() */
/* */
/* Reads a power spectrum flat from DATA_IN_FIFO. Sets */
/* arc_ccd_ps_flat_has_been_set and arc_ccd_use_ps_flat flags */
/* to TRUE. */
*****/

int arc_ccd_rt_set_ps_flat(unsigned char arg)
{
int err=0;

    if ( (err=rtf_get(DATA_IN_FIFO, arc_ccd_ps_flat,
PS_SIZE*sizeof(float))) < 0)

return err;

arc_ccd_ps_flat_has_been_set = TRUE;
arc_ccd_use_ps_flat = TRUE;

// purge ps_buf

arc_ccd_purge_ps_flag = TRUE;

return err;
} // arc_ccd_rt_set_ps_flat()

/*****
/* arc_ccd_rt_callback() */
/* */
/* This function is registered with the ccd driver when this module is */
/* installed. This function is called by the ccd driver every time it */
/* services a data_ready interrupt from the camera. */
*****/

uint32_t arc_ccd_rt_callback(uint16_t *data, CHARA_TIME fr_time,
uint32_t fr_number)
{
int i,j,k,leading_zeros;
int err = 0;

// toggle lpt port, for use when timing the callback function

#ifdef CALLBACK_LPT_TOGGLE

if(arc_ccd_use_lpt_toggle)
{
chara_set_bit_lpt(1);
}
#endif

// update vis_dither information
// the increment will only happen if vd_par.sweep_on is TRUE

arc_ccd_vd_par = vis_dither_sweep_increment();

```

```

arc_ccd_ps_peak.dith_dac = arc_ccd_vd_par.dac_curr;

// trigger any dithers that are going
// dither dictated by this module

if(arc_ccd_dith_scan.on && !arc_ccd_ps_servo_on)
{
arc_ccd_rt_dither_scan();
}

// dither dictated by vis_dither

else if(arc_ccd_vd_par.sweep_on && !arc_ccd_ps_servo_on)
{
// arc_ccd_vd_par=vis_dither_sweep_increment();

// note that the um measurement is only valid for vis_dither
// controlled sweeps, not
// for other dither functions

arc_ccd_ps_peak.dith_um = arc_ccd_vd_par.sweep_um;

/* This way uses the pointers to vis_dither's sweep data arrays
* if(arc_ccd_vis_dith_sweep_dac != NULL)
* {
* arc_ccd_ps_peak.dith_dac = *(
* arc_ccd_vis_dith_sweep_dac +
* arc_ccd_vis_dither_counter);
* }
*
* if(arc_ccd_vis_dith_sweep_um != NULL)
* {
* arc_ccd_ps_peak.dith_um = *(
* arc_ccd_vis_dith_sweep_um +
* arc_ccd_vis_dither_counter);
* }
*/
}

// fix the first 3 readout pixels of desired quadrant(s)

arc_ccd_rt_pixel_doctor(data);

if(!arc_ccd_single_quad_mode)
{

// if using only one quadrant of the detector

if(arc_ccd_num_quadrants == 1)
{
//make frame pointer point to the appropriate quadrant data
// the (quadrant + 2) % 4 bit is because the data comes from
// the camera in the quadrant order 2,3,4,1. The value of
// this quantity is 3,0,1,2 for quadrants 1,2,3,4 respectively.
frame = data + ((arc_ccd_bbquadrant + 2) % 4) *
arc_ccd_bbdata.dx * arc_ccd_bbdata.dy;

}

else //full frame
{
frame = data;
arc_ccd_rt_sort_full_frame();
frame = sorted_frame;
}

}

else // single quad mode
{
frame = data;
}

arc_ccd_ps_peak.time = arc_ccd_frame_time = fr_time;
arc_ccd_frame_number = fr_number;

arc_ccd_rt_calculate_frame_rate();
arc_ccd_ps_peak.frame_rate = arc_ccd_frame_rate;

```

```

if( arc_ccd_init_fft_flag )
{
fft_setup_fft(FFT_POW_2);
arc_ccd_init_fft_flag = FALSE;
chara_printerr("FFT re-initialized");
}

// if a row is set, do spectrum stuff
if (arc_ccd_row[0] >= 0)
{
// copy row data to spectrum

// this is the number of zeros padded in front of the data
leading_zeros = ((FFT_SIZE-arc_ccd_bbdata.dx)/2);
if(leading_zeros < 0)
leading_zeros = 0;

//the outer loop steps through the spectral rows
for(j=0; j<arc_ccd_row[0]; j++)
{
//the inner loop copies data for a given row
for (i=0 ; i<FFT_SIZE; i++)
{

// zero pad at beginning
if( i < leading_zeros )

arc_ccd_spect_buf[j][i] = 0.0;

// data in the middle
// the -leading_zeros is because
// i = leading_zeros
// at datum number 0.

else if ( i < (leading_zeros
+ arc_ccd_bbdata.dx) )
{
// the bit with num_quadrants
//comes out to 2*dx
// when all 4 quadrants are being used

arc_ccd_spect_buf[j][i] = (float)
*(frame + (i - leading_zeros) +
( (1 + arc_ccd_num_quadrants / 4) *
arc_ccd_bbdata.dx )
* arc_ccd_row[j+1] + arc_ccd_row_offset);
}
// zero pad the end

else

arc_ccd_spect_buf[j][i] = 0.0;

} // done copying data for a given row

} //done stepping through all of the requested rows

// bias the spectrum?

if (arc_ccd_use_bias)
arc_ccd_rt_apply_bias((char)leading_zeros);

// flat the spectrum?

if (arc_ccd_use_spec_flat)
arc_ccd_rt_apply_spec_flat();

// copy unmasked channels to channel buffer

//outer loop steps through requested spectral rows
for(k=0; k<arc_ccd_row[0]; k++)
{
//i represents the index in the channel_list
// fills in channel_buf with channel_list[0] entries
// from spect_buf using numbers in channel list
// as indices of spect_buf

```

```

for(i=1; i<=arc_ccd_channel_list[0]; i++)
{
    arc_ccd_channel_buf[k][i-1]
    = arc_ccd_spect_buf[k][arc_ccd_channel_list[i]];

} // done copying channels from row

// old way using boolean channel mask
/*
* //inner loop steps through individual pixels in a row
* //and copies to channel buffer if channel mask
* //is TRUE for that pixel
*
* for (i=0,j=0; i<FFT_SIZE; i++)
* {
*     if(arc_ccd_channel_mask[i])
*     {
*         *( &(arc_ccd_channel_buf[k][0]) + j )
*         = arc_ccd_spect_buf[k][i];
*         j++;
*     }
* }
* } // done copying channels from row
*/

} //done copying channel data for all rows

//calculate ps and incorporate into power spectrum mean
arc_ccd_rt_ps_mean();

// calculate power spectrum statistics
arc_ccd_rt_calculate_ps_stats();

// servo
arc_ccd_rt_ps_servo();

} // done doing spectrum stuff

//send data to user side if request flags have been set
if(arc_ccd_req_ps_stats == TRUE)
{
    if(arc_ccd_rt_get_ps_stats() != 0) arc_ccd_recording = FALSE;

    //if not in record mode, this will turn off the request flag
    if(!arc_ccd_recording) arc_ccd_req_ps_stats = FALSE;
}

if(arc_ccd_req_vd_stats == TRUE)
{
    if(arc_ccd_rt_get_vd_stats() != 0) arc_ccd_recording = FALSE;

    //if not in record mode, this will turn off the request flag
    if(!arc_ccd_recording) arc_ccd_req_vd_stats = FALSE;
}

if(arc_ccd_req_ps == TRUE)
{
    if(arc_ccd_rt_get_ps() != 0) arc_ccd_recording = FALSE;

    //if not in record mode, this will turn off the request flag
    if(!arc_ccd_recording) arc_ccd_req_ps = FALSE;
}

if(arc_ccd_req_spectrum == TRUE)
{
    if(arc_ccd_rt_get_spectrum() != 0) arc_ccd_recording = FALSE;

    //if not in record mode, this will turn off the request flag
    if(!arc_ccd_recording) arc_ccd_req_spectrum = FALSE;
}

if(arc_ccd_req_frame == TRUE)
{
    arc_ccd_rt_get_frame();
    arc_ccd_req_frame = FALSE;
}

```

```

}

if(arc_ccd_req_channels == TRUE)
{
if(arc_ccd_rt_get_channels() != 0) arc_ccd_recording = FALSE;

//if not in record mode, this will turn off the request flag
if(!arc_ccd_recording) arc_ccd_req_channels = FALSE;
}

// toggle lpt port, for use when timing the callback function
#ifdef CALLBACK_LPT_TOGGLE

if(arc_ccd_use_lpt_toggle)
{
chara_unset_bit_lpt(1);
}
#endif

return err;
} // arc_ccd_rt_callback()

/*****
/* arc_ccd_rt_calculate_ps() */
/* */
/* Called by ...callback() if the row is >= 0 and if the arc_ccd_use_fft*/
/* flag is TRUE. Calculates power spectrum , locates the peak. */
/* If arc_ccd_use_lpt_toggle is TRUE, will toggle printer port up at */
/* beginning, down at end of function for timing purposes. */
*****/

int arc_ccd_rt_calculate_ps(void)
{
int i,j;

//this outer loop steps through the index for each spectral row
for(j=0;j<arc_ccd_row[0];j++)
{

// toggle printer port up, for use when timing the fft routine

#ifdef FFT_LPT_TOGGLE

if(arc_ccd_use_lpt_toggle)
{
chara_set_bit_lpt(0);
}
#endif

fft_bitrev_float32(FFT_SIZE, &(arc_ccd_spect_buf[j][0]),
&(arc_ccd_re_fft_buf[j][0]));

for (i=0; i<FFT_SIZE; i++) arc_ccd_im_fft_buf[j][i] = 0.0;

fft_cfft_dit_float32(FFT_POW_2, &(arc_ccd_re_fft_buf[j][0]),
&(arc_ccd_im_fft_buf[j][0]));

#ifdef FFT_LPT_TOGGLE
if(arc_ccd_use_lpt_toggle)
{
chara_unset_bit_lpt(0);
}
#endif

// calculate power spec. Not bothering with redundant points.
for (i=0 ; i <PS_SIZE ; i++)
{
//set the value from the first spectrum
if(j==0)
{
arc_ccd_ps_buf[arc_ccd_ps_index][i] =

```

```

    arc_ccd_re_fft_buf[0][i] *
    arc_ccd_re_fft_buf[0][i]
    + arc_ccd_im_fft_buf[0][i] *
    arc_ccd_im_fft_buf[0][i];
}

//add to it with subsequent spectra
else
{
    arc_ccd_ps_buf[arc_ccd_ps_index][i] +=
    arc_ccd_re_fft_buf[j][i] *
    arc_ccd_re_fft_buf[j][i]
    + arc_ccd_im_fft_buf[j][i] *
    arc_ccd_im_fft_buf[j][i];
}
}

} // done stepping through all spectra

// now divide by number of spectra to obtain average power spectrum
if(arc_ccd_row[0] > 0)
{
    for(i=0; i<PS_SIZE; i++)
    {
        arc_ccd_ps_buf[arc_ccd_ps_index][i] /=
        (float)arc_ccd_row[0];
    }
}

// if row[0] is not set, this function never should have been
// called. Just in case, this zeros out the ps rather than report
// a bogus one.
else
{
    for(i=0; i<PS_SIZE; i++)
    {
        arc_ccd_ps_buf[arc_ccd_ps_index][i] = 0;
    }
}

    return 0;
} // arc_ccd_rt_calculate_ps()

/*****
/* arc_ccd_rt_ps_mean() */
/* */
/* Calculates the mean of arc_ccd_pow_spectra_in_mean spectra, which */
/* has been set by the user. Stores the mean power spectrum in */
/* arc_ccd_ps_buf[0][]. */
/* Figures mean by subtracting oldest power spectrum from mean and */
/* adding newest. */
*****/

int arc_ccd_rt_ps_mean(void)
{
    int i,j;

    ///////////calculate mean power spectrum

    //zero out ps_buf if any request has been made

    if( arc_ccd_purge_ps_flag )
    {
        arc_ccd_rt_purge_ps_buf();
        arc_ccd_purge_ps_flag = FALSE;
    }

    // shift the ps_index to next slot up in the ps_buf ring.
    //if index is at 1, wrap around to bottom again.

    if (arc_ccd_ps_index <= 1 ||
    arc_ccd_ps_index > arc_ccd_pow_spectra_in_mean)

    arc_ccd_ps_index = arc_ccd_pow_spectra_in_mean;
    else
    arc_ccd_ps_index--;

    //now subtract the old PS in the slot from the mean

```

```

for(i=0; i<PS_SIZE; i++)
{
arc_ccd_ps_buf[0][i] -= arc_ccd_ps_buf[arc_ccd_ps_index][i];
}

//put current ps in arc_ccd_ps_buf[index][]
if (arc_ccd_use_fft)
{
arc_ccd_rt_calculate_ps();

//flat the power spectrum if requested
if(arc_ccd_use_ps_flat)
arc_ccd_rt_apply_ps_flat();
}

// if no fft, copy arc_ccd_spect_buf to arc_ccd_ps_buf[index][]
else
{
for(i=0; i<PS_SIZE; i++)

arc_ccd_ps_buf[arc_ccd_ps_index][i] =
arc_ccd_spect_buf[0][i];
}

// add current ps into mean ps
// divide by number of spectra in mean to get a good average.
// it's done this way since one ps is added
// to the running mean each frame.
for(j=0; j<PS_SIZE; j++)

arc_ccd_ps_buf[0][j] += arc_ccd_ps_buf[arc_ccd_ps_index][j]
/(float)arc_ccd_pow_spectra_in_mean;

// suppress pixels
for (j=0; j<arc_ccd_suppress_pixels; j++)
{
arc_ccd_ps_buf[0][j] = 0.0;
}

return 0;
} // arc_ccd_rt_ps_mean()

/*****
/* arc_ccd_rt_calculate_ps_stats() */
/* */
/* Finds ps peak bin, peak value, effective peak bin using weighted */
/* average, peak power peak width, peak snr, and peak skew. */
*****/

int arc_ccd_rt_calculate_ps_stats(void)
{
int i;
float min_sig_value;
float peak_left_edge = 0.0;
bool left_edge_set = FALSE;
float peak_right_edge = 0.0;
float weighted_sum = 0.0;
float ps_average = 0.0;
float std_deviation = 0.0;
float deviations[PS_SIZE];

// find peak

arc_ccd_ps_int_bin = 0;
arc_ccd_ps_peak.height = 0.0;
arc_ccd_ps_peak.power = 0.0;

for (i=0; i < PS_SIZE ; i++)
{

if(arc_ccd_ps_buf[0][i] > arc_ccd_ps_peak.height)
{
arc_ccd_ps_peak.height = arc_ccd_ps_buf[0][i];

```

```

arc_ccd_ps_int_bin = i;
}

// start calculating a spectrum average by summing the terms.
ps_average += arc_ccd_ps_buf[0][i];
}

// now complete the average by dividing by the number of terms
ps_average /= PS_SIZE;

//calculate the fringe phase from the complex fft at the peak bin
//f_arctanlookup is a function which uses a lookup table and
// takes 2 args, (y,x) ,and gives phase withing +/- pi rad.

arc_ccd_ps_peak.phase = f_arctanlookup(
arc_ccd_im_fft_buf[0][arc_ccd_ps_int_bin] ,
arc_ccd_re_fft_buf[0][arc_ccd_ps_int_bin] );

//convert to degrees
arc_ccd_ps_peak.phase *= 180.0 / M_PI;

// calculate the standard deviation
// standard deviation is ~1.25 x average deviation for large data sets.
// we'll assume that statistical fluctuation errors are small
// since avg deviation is quicker to calculate.
// average deviation = sum(|dx|)/k.
// dx is individual deviations, k is number of samples.

for( i=0; i<PS_SIZE; i++)
{
deviations[i] = arc_ccd_ps_buf[0][i] - ps_average;

// we're interested in the absolute value

if( deviations[i] < 0.0 )

std_deviation -= deviations[i];

else

std_deviation += deviations[i];
}

std_deviation *= (1.25 / PS_SIZE);

// calculate how many sigma the peak is.
arc_ccd_ps_peak.sigma = deviations[arc_ccd_ps_int_bin] / std_deviation;

// define the minimum significant value as
// MINIMUM_SIGNIFICANT_PEAK * sigma

min_sig_value = MINIMUM_SIGNIFICANT_PEAK * std_deviation;

for(i=0; i < PS_SIZE; i++)
{
// Do this if pixel value is significant
// and has positive deviation.

if ( deviations[i] >= min_sig_value)
{
// find effective peak with weighted average
// note bin number = i+1 so ps_buf[0][0] is bin 1

weighted_sum += arc_ccd_ps_buf[0][i] * (i + 1);
arc_ccd_ps_peak.power += arc_ccd_ps_buf[0][i];

//the first time through this will set the left edge

if(!left_edge_set)
{
peak_left_edge = (float)i;
left_edge_set = TRUE;
}
}
}

```



```

//the last significant bin will mark the right edge
peak_right_edge = (float)i;
}

}

// calculate the effective ps peak location
if(arc_ccd_ps_peak.power <= 0.0)
{
// this happens if no bins were more than
// MINIMUM_SIGNIFICANT_PEAK * sigma bigger than the ps average

arc_ccd_ps_peak.bin = (float)arc_ccd_ps_int_bin;
arc_ccd_usable_peak = FALSE;
}

// the -1.0 in the following line is so bins are numbered starting
// with bin 0, not bin 1.

else
{
arc_ccd_ps_peak.bin = (weighted_sum / arc_ccd_ps_peak.power)
- 1.0;

arc_ccd_usable_peak = TRUE;
}

// protect against divide by zero while calculating ps peak snr
if(ps_average == 0.0)

arc_ccd_ps_peak.snr = 0.0;

else
{
// snr = peak val / avg noise val.
// avg noise val = noise_integral / noise_bins

arc_ccd_ps_peak.snr = arc_ccd_ps_peak.height / ps_average;
}

// calc peak width

arc_ccd_ps_peak.width = peak_right_edge - peak_left_edge + 1;

// calc peak skew = (eff_pk_bin - left_edge) - (right_edge - eff_pk_bin)

arc_ccd_ps_peak.skew = 2.0 * arc_ccd_ps_peak.bin - peak_left_edge
- peak_right_edge;

return 0;

} // arc_ccd_rt_calculate_ps_stats()

/*****
/* arc_ccd_rt_apply_bias() */
/* */
/* Subtracts arc_ccd_bias[] from arc_ccd_spect_buf[] . arc_ccd_bbdata */
/* had better be current! */
/* arg = the number of leading zeros padding the spectrum */
*****/
int arc_ccd_rt_apply_bias(unsigned char arg)
{
int i,j;
int err=0;

if (!arc_ccd_bias_has_been_set)
{
#ifdef DEBUG_MESSAGES

chara_pinterr("Bias has not been set.");

#endif
err = -1;
return err;
}

```

```

//this loop steps through the spectra
for(j=0; j<arc_ccd_row[0]; j++)
{
    //this loop steps through the pixels in a spectrum
    for (i=0; i< arc_ccd_bbdata.dx; i++)
    {
        // the bit about num_quadrants comes out to 2*dx
        // when all 4 quadrants are in use

        arc_ccd_spect_buf[j][((int)arg) + i] -=
        arc_ccd_bias[i +
        ( (1 + arc_ccd_num_quadrants/4) *
        arc_ccd_bbdata.dx * arc_ccd_row[j+1])
        + arc_ccd_row_offset];

    }//done stepping through the pixels in a spectrum

    }//done stepping through the spectra

    return 0;

} // arc_ccd_rt_apply_bias()

/*****
/* arc_ccd_rt_apply_spec_flat() */
/* */
/* Divides arc_ccd_spect_buf[] by arc_ccd_spec_flat[], ckecking for */
/* divisions by zero. */
*****/

int arc_ccd_rt_apply_spec_flat(void)
{
    int i,j;
    int err=0;
    float total=0.0;

    if (!arc_ccd_spec_flat_has_been_set)
    {
#ifdef DEBUG_MESSAGES

        chara_pinterr("Spectrum flat has not been set.");

#endif
        err = -1;
        return err;
    }

    // this loop steps through the spectra
    for(j=0; j<arc_ccd_row[0]; j++)
    {

        // calculate pixel total for normalizing

        for (i=0; i<FFT_SIZE; i++)

            total += arc_ccd_spect_buf[j][i];

        // apply spectrum flat

        for (i=0; i<FFT_SIZE ; i++)
        {
            if (arc_ccd_spec_flat[j][i] == 0.0)

                arc_ccd_spect_buf[j][i] = 0;

            else

#ifdef NORMAILZE_WHEN_APPLYING_SPEC_FLAT
                arc_ccd_spect_buf[j][i] = (
                arc_ccd_spect_buf[j][i]
                / (total*arc_ccd_spec_flat[j][i])
                ) - 1.0;
            #else
                arc_ccd_spect_buf[j][i] = (
                arc_ccd_spect_buf[j][i]
                / arc_ccd_spec_flat[j][i]);
            #endif
        }
    }

```

```

} // done stepping through all spectra

return err;

} // arc_ccd_rt_apply_spec_flat()

/*****
 * arc_ccd_rt_apply_ps_flat() */
*/
/* Subtracts arc_ccd_ps_flat[] from arc_ccd_ps_buf[arc_ccd_ps_index][]. */
*****/

int arc_ccd_rt_apply_ps_flat(void)
{
    int i;
    int err=0;

    if(!arc_ccd_ps_flat_has_been_set)
    {
#ifdef DEBUG_MESSAGES
        chara_pnterr("Power spectrum flat has not been set.");
#endif
        err = -1;
        return err;
    }

    for(i=0; i<PS_SIZE; i++)
    {
        arc_ccd_ps_buf[arc_ccd_ps_index][i] -=
        arc_ccd_ps_flat[i];
    }

    return 0;
} // arc_ccd_rt_apply_ps_flat()

/*****
 * arc_ccd_rt_toggle_flag() */
*/
/* Users can call this function to toggle flags on the rt side. */
/* Value of arg tells which flag to toggle. Value of the flag is return.*/
/* Defs of flag identifier values in arc_ccd.h. */
*****/

int arc_ccd_rt_toggle_flag(unsigned char arg)
{
    bool return_value;
    int err = 0;

    // old way
    // if ( (err=rtf_get(DATA_IN_FIFO, &flag_id, sizeof(flag_id))) < 0)
    // return err;

    // new way, uses arg to pass flag id

    switch(arg)
    {
        case FLAG_ID_BIAS_HAS_BEEN_SET:
            arc_ccd_bias_has_been_set =
            !arc_ccd_bias_has_been_set;
            return_value = arc_ccd_bias_has_been_set;
            break;
        case FLAG_ID_SPEC_FLAT_HAS_BEEN_SET:
            arc_ccd_spec_flat_has_been_set =
            !arc_ccd_spec_flat_has_been_set;
            return_value = arc_ccd_spec_flat_has_been_set;
            break;
        case FLAG_ID_PS_FLAT_HAS_BEEN_SET:
            arc_ccd_ps_flat_has_been_set =
            !arc_ccd_ps_flat_has_been_set;
            return_value = arc_ccd_ps_flat_has_been_set;
            break;
        case FLAG_ID_USE_BIAS:
            arc_ccd_use_bias = !arc_ccd_use_bias;
            return_value = arc_ccd_use_bias;
            arc_ccd_purge_ps_flag = TRUE;
            break;
    }

```

```

case FLAG_ID_USE_SPEC_FLAT:
arc_ccd_use_spec_flat = !arc_ccd_use_spec_flat;
return_value = arc_ccd_use_spec_flat;
arc_ccd_purge_ps_flag = TRUE;
break;
case FLAG_ID_USE_PS_FLAT:
arc_ccd_use_ps_flat = !arc_ccd_use_ps_flat;
return_value = arc_ccd_use_ps_flat;
arc_ccd_purge_ps_flag = TRUE;
break;
case FLAG_ID_USE_FFT:
arc_ccd_use_fft = !arc_ccd_use_fft;
return_value = arc_ccd_use_fft;
arc_ccd_purge_ps_flag = TRUE;
break;
case FLAG_ID_USE_LPT_TOGGLE:
arc_ccd_use_lpt_toggle = !arc_ccd_use_lpt_toggle;
return_value = arc_ccd_use_lpt_toggle;
break;
case FLAG_ID_PS_SERVO_ON:
arc_ccd_ps_servo_on = !arc_ccd_ps_servo_on;
return_value = arc_ccd_ps_servo_on;
break;
case FLAG_ID_SQUAREWAVE_ON:
arc_ccd_squarewave_on = !arc_ccd_squarewave_on;
return_value = arc_ccd_squarewave_on;
break;
case FLAG_ID_USE_PS:
arc_ccd_use_ps = !arc_ccd_use_ps;
return_value = arc_ccd_use_ps;
break;
default:
err = -1;
return err;
}

    if ( (err=rtf_put(DATA_OUT_FIFO, &return_value, sizeof(return_value)) )
) < 0)
return err;

return 0;

} // arc_ccd_rt_toggle_flag()

/*****
/* arc_ccd_rt_check_flag() */
/* */
/* Users can call this function to check flag values on the rt side. */
/* Value of arg tells which flag to check. Value of the flag is return. */
/* Defs of flag identifier values in arc_ccd.h. */
*****/

int arc_ccd_rt_check_flag(unsigned char arg)
{
bool return_value;
int err = 0;

// old way
// if ( (err=rtf_get(DATA_IN_FIFO, &flag_id, sizeof(flag_id))) < 0)
// return err;

switch(arg)
{
case FLAG_ID_BIAS_HAS_BEEN_SET:
return_value = arc_ccd_bias_has_been_set;
break;
case FLAG_ID_SPEC_FLAT_HAS_BEEN_SET:
return_value = arc_ccd_spec_flat_has_been_set;
break;
case FLAG_ID_PS_FLAT_HAS_BEEN_SET:
return_value = arc_ccd_ps_flat_has_been_set;
break;
case FLAG_ID_USE_BIAS:
return_value = arc_ccd_use_bias;
break;
case FLAG_ID_USE_SPEC_FLAT:
return_value = arc_ccd_use_spec_flat;
break;

```

```

case FLAG_ID_USE_PS_FLAT:
return_value = arc_ccd_use_ps_flat;
break;
case FLAG_ID_USE_FFT:
return_value = arc_ccd_use_fft;
break;
case FLAG_ID_USE_LPT_TOGGLE:
return_value = arc_ccd_use_lpt_toggle;
break;
case FLAG_ID_PS_SERVO_ON:
return_value = arc_ccd_ps_servo_on;
break;
case FLAG_ID_SQUAREWAVE_ON:
return_value = arc_ccd_squarewave_on;
break;
case FLAG_ID_USE_PS:
return_value = arc_ccd_use_ps;
break;
default:
err = -1;
return err;
}

    if ( (err=rtf_put(DATA_OUT_FIFO, &return_value, sizeof(return_value))
) < 0)

return err;

return 0;
} // arc_ccd_rt_check_flag()

/*****
/* arc_ccd_rt_set_suppress_pixels() */
/* */
/* This function sets the number of pixels at the beginning of the */
/* power spectrum which will be forced to zero. */
*****/

int arc_ccd_rt_set_suppress_pixels(unsigned char arg)
{
int err = 0;

// request re-zero ps_buf

arc_ccd_purge_ps_flag = TRUE;

    if ( (err=rtf_get(DATA_IN_FIFO, &arc_ccd_suppress_pixels,
sizeof(arc_ccd_suppress_pixels))) < 0)

return err;

// if row value doesn't lie in bounding box

if ( arc_ccd_suppress_pixels < 0 ||
arc_ccd_suppress_pixels > (FFT_SIZE/2 +1) )
{
arc_ccd_suppress_pixels = 0;

#ifdef DEBUG_MESSAGES

chara_printerr(
"%d is not a valid number of pixels.  None suppressed."
,arc_ccd_suppress_pixels);

#endif
return err;
}

#ifdef DEBUG_MESSAGES

chara_printerr("I got the arc_ccd_suppress_pixels value: %d."
,arc_ccd_suppress_pixels);

#endif
return 0;
} // arc_ccd_rt_set_suppress_pixels()

```

```

/*****
/* arc_ccd_rt_get_suppress_pixels() */
/* */
/* This function is used to verify how many pixels at the beginning of */
/* the power spectrum are being forced to zero. */
*****/

int arc_ccd_rt_get_suppress_pixels(unsigned char arg)
{
    int err = 0;

    if ((err=rtf_put(DATA_OUT_FIFO, &arc_ccd_suppress_pixels,
sizeof(arc_ccd_suppress_pixels) ) ) <0 )

return err;

#ifdef DEBUG_MESSAGES

chara_printerr("I sent the arc_ccd_suppress_pixels number: %d.",
arc_ccd_suppress_pixels);

#endif
return 0;

} /* arc_ccd_rt_get_suppress_pixels() */

/*****
/* arc_ccd_rt_toggle_lpt() */
/* Toggles printer port using value in global bool arc_ccd_lpt_high. */
/* if arc_ccd_lpt_high == TRUE, will set it FALSE and set lpt pins low. */
/* if arc_ccd_lpt_high == FALSE, will set it TRUE and set lpt pins high.*/
*****/
int arc_ccd_rt_toggle_lpt(unsigned char arg)
{
    // toggle up

    if(arg == 1)
    {
        // all pins high no matter what

        chara_set_lpt(0xff);
        arc_ccd_lpt_high = TRUE;
    }

    // toggle down

    else if (arg == 0)
    {
        // all pins low no matter what

        chara_set_lpt(0x00);
        arc_ccd_lpt_high = FALSE;
    }

    // plain old toggle whatever

    else
    {
        if(arc_ccd_lpt_high)
        {
            // all 9 pins set low.

            chara_set_lpt(0x00);
            arc_ccd_lpt_high = FALSE;
        }

        //lpt is low, toggle it high

        else
        {
            // 0xFF is 255 in hex. all 8 pins will be set high.

            chara_set_lpt(0xFF);
            arc_ccd_lpt_high = TRUE;
        }
    }

    return 0;
} /* arc_ccd_rt_toggle_lpt()*/

```

```

/*****
/* arc_ccd_rt_set_ps_mean_num() */
/* */
/* This function sets how many power spectra will be used in the mean. */
*****/

int arc_ccd_rt_set_ps_mean_num(unsigned char arg)
{
    int err = 0;

    // zero out ps_buf

    arc_ccd_purge_ps_flag = TRUE;

    // get user defined value of number of power spectra in mean

        if ( (err=rtf_get(DATA_IN_FIFO, &arc_ccd_pow_spectra_in_mean,
sizeof(arc_ccd_pow_spectra_in_mean) ) ) < 0)
        {
            arc_ccd_pow_spectra_in_mean = 1;
            return err;
        }

    if(arc_ccd_pow_spectra_in_mean > MAX_POW_SPECTRA_IN_MEAN
|| arc_ccd_pow_spectra_in_mean < 1)
    {
        arc_ccd_pow_spectra_in_mean = 1;
    }

#ifdef DEBUG_MESSAGES

    chara_printerr(
"%d is outside of allowable range. Mean will be 1."
,arc_ccd_pow_spectra_in_mean);

#endif
    return err;
}

#ifdef DEBUG_MESSAGES

    chara_printerr("I got the arc_ccd_pow_spectra_in_mean value: %d."
,arc_ccd_pow_spectra_in_mean);

#endif
    return 0;
} // arc_ccd_rt_set_ps_mean_num()

/*****
/* arc_ccd_rt_purge_ps_buf() */
/* */
/* This zeros out the ps_buf. */
*****/

int arc_ccd_rt_purge_ps_buf(void)
{
    int i,j;

    for(j=0; j<MAX_POW_SPECTRA_IN_MEAN+1; j++)
    {
        for(i=0; i< PS_SIZE; i++)

            arc_ccd_ps_buf[j][i] = 0.0;
    }

    return 0;
} // arc_ccd_rt_purge_ps_buf()

/*****
/* arc_ccd_rt_req_purge_ps_buf() */
/* */
/* Callable from user side to request ps_buf purge. */
*****/

int arc_ccd_rt_req_purge_ps_buf(unsigned char arg)
{
    arc_ccd_purge_ps_flag = TRUE;

    return 0;
} // arc_ccd_rt_req_purge_ps_buf()

```

```

/*****
/* arc_ccd_rt_init_fft() */
/* */
/* Callable from user side to request fft init. */
*****/

int arc_ccd_rt_init_fft(unsigned char arg)
{
    arc_ccd_init_fft_flag = TRUE;

    return 0;
} // arc_ccd_rt_init_fft()

/*****
/* arc_ccd_rt_start_record() */
/* */
/* Sets arc_ccd_recording flag to true so callback function will */
/* send data to FIFO each interrupt. Request flags for ps_stats, */
/* ps_buf[0][] and spectrum_buf[] can be kept in the on position for */
/* the duration of the recording. The flags to be set are designated */
/* by the char "arg" passed from the user side. In binary form: */
/* arg bit 0: ps_stats */
/* arg bit 1: power spectrum */
/* arg bit 2: spectrum */
/* arg bit 3: individual spectrum channels */
/* arg bit 4: vis_dither data */
*****/

int arc_ccd_rt_start_record(unsigned char arg)
{
    arc_ccd_recording = TRUE;

    // if bit 0 is on, the request ps_stats
    if( (arg & 1) == 1 ) arc_ccd_req_ps_stats = TRUE;

    // if bit 1 is on, request power spectrum
    if( ( (arg >> 1) & 1 ) == 1 ) arc_ccd_req_ps = TRUE;

    // if bit 2 is on, request spectrum
    if( ( (arg >> 2) & 1 ) == 1 ) arc_ccd_req_spectrum = TRUE;

    // if bit 3 is on, request channels
    if( ( (arg >> 3) & 1 ) == 1 ) arc_ccd_req_channels = TRUE;

    // if bit 4 is on, request vis_dither_data
    if( ( (arg >> 4) & 1 ) == 1 ) arc_ccd_req_vd_stats = TRUE;

    return 0;
} // arc_ccd_rt_start_record()

/*****
/* arc_ccd_rt_end_record() */
/* */
/* Sets arc_ccd_recording flag to false so callback function will */
/* stop sending data to FIFO each interrupt. Request flags for */
/* ps_stats, ps_buf[0][], and spectrum_buf[] set to false. */
*****/

int arc_ccd_rt_end_record(unsigned char arg)
{
    arc_ccd_recording = FALSE;

    arc_ccd_req_ps_stats = FALSE;

    arc_ccd_req_ps = FALSE;

    arc_ccd_req_spectrum = FALSE;

    arc_ccd_req_channels = FALSE;

    arc_ccd_req_vd_stats = FALSE;

```



```

return 0;
} // arc_ccd_rt_end_record()

/*****
/* arc_ccd_rt_ps_servo() */
/* */
/* Calculates the next tracking signal to be sent to the vis_dither */
/* mirror. Sends new dac request via vis_dither_rt_goto, part of */
/* the vis_dither_rt module. */
*****/
void arc_ccd_rt_ps_servo(void)
{
// move whatever is in the current errsignal spot into
// the last errsignal, same for tracksignal

arc_ccd_ps_servo_last_errsignal = arc_ccd_ps_peak.errsignal;

arc_ccd_ps_servo_last_tracksignal = arc_ccd_ps_peak.tracksignal;

// the errsignal is the target freq - the current freq.
// bin refers to the frequency bin of the ps peak.

arc_ccd_ps_peak.errsignal = arc_ccd_ps_servo_target_freq
- arc_ccd_ps_peak.bin;

//inject squarewave for servo tuning?

if(arc_ccd_squarewave_on)
arc_ccd_rt_squarewave();

// the leaky errsignal is like an integral

arc_ccd_ps_servo_leaky_errsignal = arc_ccd_ps_peak.errsignal +
PS_SERVO_LEAKY_COEFF * arc_ccd_ps_servo_leaky_errsignal;

//track = G( P(err) + I(leakyerr) + D(err-lasterr))
// this assumes a constant dt between loops. Maybe dt should
// be implemented, but if the loop happens many times per clock tick,
// it could be worse

arc_ccd_ps_peak.tracksignal = arc_ccd_ps_servo_gain *
( arc_ccd_ps_servo_p * arc_ccd_ps_peak.errsignal)
+ (arc_ccd_ps_servo_i * arc_ccd_ps_servo_leaky_errsignal)
+ (arc_ccd_ps_servo_d *
(arc_ccd_ps_peak.errsignal - arc_ccd_ps_servo_last_errsignal)
);
// now tracksignal is a change in units of freq
// now change the trackingsignal into dac units

arc_ccd_ps_peak.tracksignal *= freq_to_dac ;

arc_ccd_ps_peak.tracksignal += freq_to_dac_offset;

//divide by 2 because delay changes twice as much as mirror move.
// perhaps this should just be absorbed into the gain, and
// require less processing time.

arc_ccd_ps_peak.tracksignal /= 2.0;

// now tracksignal is a position change

// now send the new position change to the vis_dither
// this assumes that adding delay will increase fringe freq. on
// opposite side of 0 freq, the behavior is reversed.

if(arc_ccd_ps_servo_on && arc_ccd_usable_peak)
{
if( vis_dither_rt_move_relative(
(int)arc_ccd_ps_peak.tracksignal ) != 0 )
{
chara_printerr(
"Failed to set vis_dither target in servo.");
}
}
}

} // arc_ccd_rt_ps_servo()

/*****/

```

```

/* arc_ccd_rt_tweak_pid() */
/* */
/* Function callable by the user side to tweak PID servo parameters. */
/* The value of arg tells which parameter to tweak, and the value must */
/* be passed down through the RT FIFO. Returns 0 if all goes well. */
/*****
int arc_ccd_rt_tweak_pid(unsigned char arg)
{
    float new_value;
    int err = 0;

    if ( (err=rtf_get(DATA_IN_FIFO, &new_value, sizeof(new_value)))
    < 0)

    return err;

#ifdef DEBUG_MESSAGES

    chara_printerr("I got the parameter value: %f.",new_value);

#endif

    // The value of arg says which parameter to tweak:
    // TWEAK_PS_SERVO_P = 0
    // TWEAK_PS_SERVO_I = 1
    // TWEAK_PS_SERVO_D = 2
    // TWEAK_PS_SERVO_GAIN = 3
    // defined in arcccd.h

    switch((int)arg)
    {
    case TWEAK_PS_SERVO_P:
        arc_ccd_ps_servo_p = new_value;
        break;

    case TWEAK_PS_SERVO_I:
        arc_ccd_ps_servo_i = new_value;
        break;

    case TWEAK_PS_SERVO_D:
        arc_ccd_ps_servo_d = new_value;
        break;

    case TWEAK_PS_SERVO_GAIN:
        arc_ccd_ps_servo_gain = new_value;
        break;

    default:
        chara_printerr(
        "No servo parameter mapped to %d.",(int)arg);
        err = (int)arg;
        break;
    }

    return err;
} // arc_ccd_rt_tweak_pid

/*****
/* arc_ccd_rt_squarewave() */
/* */
/* When arc_ccd_squarewave_on is TRUE, this function adds a square */
/* wave to the errsignal in arc_ccd_rt_ps_servo(). For use in tuning */
/* the servo parameters. */
/*****
void arc_ccd_rt_squarewave(void)
{
    arc_ccd_squarewave_loop++;

    // add SQUAREWAVE_AMPLITUDE to errsignal in first half of period
    if(arc_ccd_squarewave_loop < (SQUAREWAVE_PERIOD >> 1) )
        arc_ccd_ps_peak.errsignal += SQUAREWAVE_AMPLITUDE;

    // if loop counter goes over SQUAREWAVE_PERIOD, reset to zero.
    else if(arc_ccd_squarewave_loop >= SQUAREWAVE_PERIOD)

```

```

arc_ccd_squarewave_loop = 0;

} // arc_ccd_rt_squarewave()

/*****
/* arc_ccd_rt_new_target_freq() */
** */
/* Function for user side to request a new fringe frequency target for */
/* the fringe tracking servo. Not usable for crossing zero or nyquist */
/* frequency boundaries. Targets will be forced to the range between */
/* PS_SERVO_TARGET_MIN and PS_SERVO_TARGET_MAX. */
*****/
int arc_ccd_rt_new_target_freq(unsigned char arg)
{
    float new_value;
    int err = 0;

    if ( (err=rtf_get(DATA_IN_FIFO, &new_value, sizeof(new_value)))
    < 0)

    return err;

#ifdef DEBUG_MESSAGES

    chara_printrerr("I got the fringe frequency target value: %f.",
    new_value);

#endif

    // check that the new target is within acceptable limits.
    // change the target if necessary. Note that this means you
    // can't cross zero fringe or nyquist freq. using this function.

    if ( new_value < PS_SERVO_TARGET_MIN)

    new_value = PS_SERVO_TARGET_MIN;

    else if ( new_value > PS_SERVO_TARGET_MAX)

    new_value = PS_SERVO_TARGET_MAX;

    // set new target frequency value

    arc_ccd_ps_servo_target_freq = new_value;

    return err;

} // arc_ccd_rt_new_target_freq()

/*****
/* arc_ccd_rt_set_flag_true() */
** */
/* Users can call this function to set flag values on the rt side. */
/* Value of the arg tells which flag to check. */
/* Defs of flag identifier values in arc_ccd.h. */
*****/
int arc_ccd_rt_set_flag_true(unsigned char arg)
{
    switch(arg)
    {
        case FLAG_ID_BIAS_HAS_BEEN_SET:
            arc_ccd_bias_has_been_set = TRUE;
            break;
        case FLAG_ID_SPEC_FLAT_HAS_BEEN_SET:
            arc_ccd_spec_flat_has_been_set = TRUE;
            break;
        case FLAG_ID_PS_FLAT_HAS_BEEN_SET:
            arc_ccd_ps_flat_has_been_set = TRUE;
            break;
        case FLAG_ID_USE_BIAS:
            arc_ccd_use_bias = TRUE;
            break;
        case FLAG_ID_USE_SPEC_FLAT:
            arc_ccd_use_spec_flat = TRUE;
            break;
        case FLAG_ID_USE_PS_FLAT:
            arc_ccd_use_ps_flat = TRUE;
            break;
    }
}

```

```

case FLAG_ID_USE_FFT:
arc_ccd_use_fft = TRUE;
break;
case FLAG_ID_USE_LPT_TOGGLE:
arc_ccd_use_lpt_toggle = TRUE;
break;
case FLAG_ID_PS_SERVO_ON:
arc_ccd_ps_servo_on = TRUE;
break;
case FLAG_ID_SQUAREWAVE_ON:
arc_ccd_squarewave_on = TRUE;
break;
case FLAG_ID_USE_PS:
arc_ccd_use_ps = TRUE;
break;
default:
return -1;
}

return 0;

} // arc_ccd_set_flag_true()

/*****
/* arc_ccd_rt_set_flag_false() */
/* */
/* Users can call this function to set flag values on the rt side. */
/* Value of the arg tell which flag to check. */
/* Defs of flag identifier values in arc_ccd.h. */
*****/

int arc_ccd_rt_set_flag_false(unsigned char arg)
{

switch(arg)
{
case FLAG_ID_BIAS_HAS_BEEN_SET:
arc_ccd_bias_has_been_set = FALSE;
break;
case FLAG_ID_SPEC_FLAT_HAS_BEEN_SET:
arc_ccd_spec_flat_has_been_set = FALSE;
break;
case FLAG_ID_PS_FLAT_HAS_BEEN_SET:
arc_ccd_ps_flat_has_been_set = FALSE;
break;
case FLAG_ID_USE_BIAS:
arc_ccd_use_bias = FALSE;
break;
case FLAG_ID_USE_SPEC_FLAT:
arc_ccd_use_spec_flat = FALSE;
break;
case FLAG_ID_USE_PS_FLAT:
arc_ccd_use_ps_flat = FALSE;
break;
case FLAG_ID_USE_FFT:
arc_ccd_use_fft = FALSE;
break;
case FLAG_ID_USE_LPT_TOGGLE:
arc_ccd_use_lpt_toggle = FALSE;
break;
case FLAG_ID_PS_SERVO_ON:
arc_ccd_ps_servo_on = FALSE;
break;
case FLAG_ID_SQUAREWAVE_ON:
arc_ccd_squarewave_on = FALSE;
break;
case FLAG_ID_USE_PS:
arc_ccd_use_ps = FALSE;
break;
default:
return -1;
}

return 0;

} // arc_ccd_set_flag_false()

/*****
/* arc_ccd_rt_dither_scan() */
/* */

```

```

/* Handler called by the callback function which manages a dither */
/* mirror scan. Scans the dither mirror between */
/* arcccd_dith_scan.lower_limit and arcccd_dith_scan.upper_limit with */
/* steps of arcccd_dith_scan.dx DAC steps. Executes */
/* arcccd_dith_scan.scan_index scans and */
/* then turns arcccd_dith_scan.on to FALSE. */
/* Relies on arc_ccd_rt_setup_dither_scan() to set */
/* upper and lower limits, size of dx, scans left, etc. */
/*****/

void arc_ccd_rt_dither_scan(void)
{
    int target;

    // target the position (begin + dx * index)

    target = arcccd_dith_scan.lower_limit + arcccd_dith_scan.dx
    * arcccd_dith_scan.dx_index;

    if (arcccd_dith_scan.going_up)
    {
        // if the full number of scans has been completed,
        // turn off scanning

        if ( arcccd_dith_scan.scan_index >= arcccd_dith_scan.total_scans && arcccd_dith_scan.total_scans > 0 )
        {
            arcccd_dith_scan.on = FALSE;

            return;
        }

        // is that beyond the limits of the dither?

        if ( target > arcccd_dith_scan.upper_limit)
        {
            //switch scan direction

            arcccd_dith_scan.going_up = FALSE;

            // take two steps down since you're
            // over the upper limit, and just one
            // step down would result in two samples at the
            // same spot.

            arcccd_dith_scan.dx_index--;
            arcccd_dith_scan.dx_index--;

            // make a nested call to dither scan.
            // This should access the code block going down.

            arc_ccd_rt_dither_scan();

            return;
        }

        // increment the dx index

        arcccd_dith_scan.dx_index++;

    } // end if going up

    // going down

    else
    {
        // is that beyond the limits of the dither?

        if ( target < arcccd_dith_scan.lower_limit)
        {
            //switch scan direction

            arcccd_dith_scan.going_up = TRUE;

            // take 2 step up since you're below the
            // lower limit, and only one step up would result in
            // two samples at the same spot.

            arcccd_dith_scan.dx_index++;
            arcccd_dith_scan.dx_index++;
        }
    }
}

```

```

//increment the number of scans done
if(arcccd_dith_scan.total_scans > 0)

arcccd_dith_scan.scan_index++;

// make a nested call to dither scan.
// This should access the code block going down.

arc_ccd_rt_dither_scan();

return;
}

// decrement the dx index
arcccd_dith_scan.dx_index--;

} // end if going down

//send the command to the dither mirror
vis_dither_rt_goto(target);

return;

}
/* arc_ccd_rt_dither_scan() */

/*****
/* arc_ccd_rt_setup_dither_scan() */
/* */
/* Sets all of the variables and starts recording for */
/* arc_ccd_dither_scan(). */
/* Callable from the user side. */
*****/

int arc_ccd_rt_setup_dither_scan(unsigned char arg)
{
int err;
int temp;

// get parameters for dither scan from user side

if ( (err=rtf_get(DATA_IN_FIFO, &arcccd_dith_scan, sizeof(arcccd_dith_scan)))
< 0)
{
// reset the default values of the scan parameters

arcccd_dith_scan.lower_limit = RTDAC_LOWER_LIMIT;
arcccd_dith_scan.upper_limit = RTDAC_UPPER_LIMIT;
arcccd_dith_scan.dx = MAX_DIFF;
arcccd_dith_scan.on = FALSE;
arcccd_dith_scan.going_up = TRUE;
arcccd_dith_scan.dx_index = 0;
arcccd_dith_scan.scan_index = 0;
arcccd_dith_scan.total_scans = 0;

chara_printerr(
"Unable to get dither scan parameters from DATA_IN_FIFO.");

return err;
}

// check limits and step size

if(arcccd_dith_scan.lower_limit == arcccd_dith_scan.upper_limit)
{
// what were they thinking?
// reset the default values of the scan parameters

arcccd_dith_scan.lower_limit = RTDAC_LOWER_LIMIT;
arcccd_dith_scan.upper_limit = RTDAC_UPPER_LIMIT;
arcccd_dith_scan.dx = MAX_DIFF;
arcccd_dith_scan.on = FALSE;
arcccd_dith_scan.going_up = TRUE;
arcccd_dith_scan.dx_index = 0;
arcccd_dith_scan.scan_index = 0;

```

```

arcccd_dith_scan.total_scans = 0;

chara_printerr(
"Invalid scan limits.  Scan aborted.");

return -1;
}

if(arcccd_dith_scan.upper_limit < arcccd_dith_scan.lower_limit)
{
// swtich the two values

temp = arcccd_dith_scan.upper_limit;
arcccd_dith_scan.upper_limit = arcccd_dith_scan.lower_limit;
arcccd_dith_scan.lower_limit = temp;
}

if(arcccd_dith_scan.lower_limit < RTDAC_LOWER_LIMIT)
{
// reset limit

arcccd_dith_scan.lower_limit = RTDAC_LOWER_LIMIT;

chara_printerr(
"Requested scan lower limit out of bounds.  Reset.");
}

if(arcccd_dith_scan.upper_limit > RTDAC_UPPER_LIMIT)
{
// reset limit

arcccd_dith_scan.upper_limit = RTDAC_UPPER_LIMIT;

chara_printerr(
"Requested scan upper limit out of bounds.  Reset.");
}

if(arcccd_dith_scan.dx < 1 || arcccd_dith_scan.dx >
(arcccd_dith_scan.upper_limit - arcccd_dith_scan.lower_limit))
{
// reset dx
if(MAX_DIFF > (arcccd_dith_scan.upper_limit -
arcccd_dith_scan.lower_limit))

arcccd_dith_scan.dx = 1;

else

arcccd_dith_scan.dx = MAX_DIFF;

chara_printerr(
"Requested dither scan step size out of bounds.  Reset.");
}

return 0;
} /* arc_ccd_rt_setup_dither_scan() */

/*****
/* arc_ccd_rt_activate_dither_scan() */
/* */
/* User callable function which turns the arcccd_dith_scan.on boolean */
/* to TRUE or FALSE depending on the value of arg. */
/* arg = 1, turn scan on. 2 = restart scan. else turn scan off. */
*****/
int arc_ccd_rt_activate_dither_scan(unsigned char arg)
{
if( arg == 1)

arcccd_dith_scan.on = TRUE;

else if( arg == 2)
{
// start the scan sequence at the beginning
// with current limit and step size settings.

arcccd_dith_scan.on = TRUE;
arcccd_dith_scan.scan_index = 0;
arcccd_dith_scan.dx_index = 0;
arcccd_dith_scan.going_up = TRUE;

```

```

}
else

arccd_dith_scan.on = FALSE;

return 0;
} /* arc_ccd_rt_activate_dither_scan() */

/*****
/* arc_ccd_rt_sort_full_frame() */
*/
/* Takes bb data from the camera buffer pointed to by *frame, and re-*/
/* sorts it into raster order rather than in discrete quadrants in */
/* raster order. Assumes that the arccd_API is ROI mode, with the data*/
/* coming from the camera quadrant order, each in raster order. */
/* Sorted data in sorted_frame[] */
*****/

void arc_ccd_rt_sort_full_frame(void)
{
int i,j;

// Quadrant numbers are 1 at lower left, ascending clockwise.

//>>> Data sections numbered 1 at UPPER left, clockwise.
//>>> Data comes on order: quadrant 2,3,4,1

// raster from the upper left.

// origin is at upper left corner, +y is down.

for(j=0; j < arc_ccd_bbdata.dy*2; j++)
{
for(i=0; i < arc_ccd_bbdata.dx*2; i++)
{
// else in upper half of image

if(j < arc_ccd_bbdata.dy)
{
// if in left half of image
// quadrant 2

if(i < arc_ccd_bbdata.dx)
{
// data from first section

sorted_frame[j*arc_ccd_bbdata.dx*2 + i]
= *(frame
+ j * arc_ccd_bbdata.dx
+ i );
}

// else in right half of image
// quadrant 3

else
{
//data from second section

sorted_frame[j*arc_ccd_bbdata.dx*2 + i]
= *(frame
+ (arc_ccd_bbdata.dx
* arc_ccd_bbdata.dy)
+ j * arc_ccd_bbdata.dx
+ (i - arc_ccd_bbdata.dx) );
}
}

// if in lower half of image

else
{
// if in left half of image
// quadrant 1

if(i < arc_ccd_bbdata.dx)
{
// data from fourth section

sorted_frame[j*arc_ccd_bbdata.dx*2 + i]

```



```

= *(frame
+ 3 * (arc_ccd_bbdata.dx
* arc_ccd_bbdata.dy)
+ (j - arc_ccd_bbdata.dy)
* arc_ccd_bbdata.dx
+ i );
}

// else in right half of image
// quadrant 4

else
{
// data from third section

sorted_frame[j*arc_ccd_bbdata.dx*2 + i]
= *(frame
+ 2 * (arc_ccd_bbdata.dx
* arc_ccd_bbdata.dy)
+ (j - arc_ccd_bbdata.dy)
* arc_ccd_bbdata.dx
+ (i - arc_ccd_bbdata.dx) );
}
}
}
}

return;

} /* arc_ccd_rt_sort_full_frame() */

/*****
/* arc_ccd_rt_pixel_doctor() */
/* */
/* Sets the first n readout pixels in each quadrant to pixel 4's value,*/
/* rather than to 0. Determines which quadrants to fix by */
/* arc_ccd_bbquadrant. n defaults to 3. */
*****/

void arc_ccd_rt_pixel_doctor(uint16_t *dat)
{
uint16_t *p;
int i,n;
bool ll,ul,ur,lr;

ll = ul = ur = lr = FALSE;

// n is the number of pixels to fix
n=3;

// fix the first n zeroed readout pixels, at the outside
// corner of the quadrant. Set them equal to the n+1 pixel,
// and the user should know not to rely on the values in
// these pixels.

// remember that the data comes from the camera in quadrant order:
// 2,3,4,1. Quad 1 is lower left.

switch(arc_ccd_bbquadrant)
{
case 1: ll = TRUE;
break;

case 2: ul = TRUE;
break;

case 3: ur = TRUE;
break;

case 4: lr = TRUE;
break;

default: ll = ul = ur = lr = TRUE;
break;
}

if(ll)
{
// move the pointer to the beginning of the
// data for this quadrant.

```

```

// quad 1 is the 4th chunk of data

p = dat + 3 *
(arc_ccd_bbdata.dx * arc_ccd_bbdata.dy);

// the n zeroed pixels are in the lower left
// corner of the roi.

p += (arc_ccd_bbdata.dx *
(arc_ccd_bbdata.dy - 1));

for (i=0; i<n; i++)
{
// set equal to n+1th readout pixel

*(p+i) = *(p+n);
}

if(ul)
{
// move the pointer to the beginning of the
// data for this quadrant.

// quad 2 is the 1st chunk of data

p = dat;

// if it's in single quad mode (ul only)
// then fix 8 pixels

if(arc_ccd_bbquadrant == 2)
n = 8;

// kludge to fix pixels in lower right corner
if(arc_ccd_single_quad_mode)
{
// the n zeroed pixels are in the lower right
// corner of the roi.

p += (arc_ccd_bbdata.dx * arc_ccd_bbdata.dy)
- 1;

for (i=0; i<n; i++)
{
// set equal to n+1th readout pixel
// note the readout order on the right
// side of the chip is right to left

*(p-i) = *(p-n);
}
}

else
{
// the n zeroed pixels are in the upper left
// corner of the roi.

for (i=0; i<n; i++)
{
// set equal to n+1th readout pixel

*(p+i) = *(p+n);
}
}

if(ur)
{
// move the pointer to the beginning of the
// data for this quadrant.

// quad 3 is the 2nd chunk of data

p = dat +
(arc_ccd_bbdata.dx * arc_ccd_bbdata.dy);

// the n zeroed pixels are in the upper right
// corner of the roi.

```

```

p += (arc_ccd_bbdata.dx - 1);

for (i=0; i<n; i++)
{
    // set equal to n+1th readout pixel
    // note the readout order on the right side of the
    // chip is right to left

    *(p-i) = *(p-n);
}

if(lr)
{
    // move the pointer to the beginning of the
    // data for this quadrant.

    // quad 4 is the 3rd chunk of data

    p = dat + 2 *
    (arc_ccd_bbdata.dx * arc_ccd_bbdata.dy);

    // the n zeroed pixels are in the lower right
    // corner of the roi.

    p += (arc_ccd_bbdata.dx * arc_ccd_bbdata.dy)
    - 1;

    for (i=0; i<n; i++)
    {
        // set equal to n+1th readout pixel
        // note the readout order on the right side of the
        // chip is right to left

        *(p-i) = *(p-n);
    }
}

return;
} /* arc_ccd_rt_pixel_doctor() */

/*****
/* arc_ccd_rt_set_channels() */
/* */
/* User callable function which gets the channel_list array */
/* from the DATA_IN_FIFO. */
*****/
int arc_ccd_rt_set_channels(unsigned char arg)
{
    int err;

    // get the channel list

    if ( (err=rtf_get(DATA_IN_FIFO, arc_ccd_channel_list,
    (SPEC_SIZE+1)*sizeof(*arc_ccd_channel_list))) < 0)
        return err;

    return 0;
} /* arc_ccd_rt_set_channels() */

/*****
/* arc_ccd_rt_calculate_frame_rate() */
/* */
/* Called by the callback function. Figures frame rate from the frame */
/* number and frame time every NUMBER_OF_FRAMES_PER_FRAME_RATE_CALC... */
/* frames. Sets the value of arccd_frame_rate directly. */
/* */
*****/
void arc_ccd_rt_calculate_frame_rate(void)
{
    // n = frame number
    // t = frame time in millisec
    // i = number of times the function has been called

    static uint32_t old_n=0;
    static CHARA_TIME old_t=0;
    static uint32_t n;

```

```

static CHARA_TIME t;
static int i=0;
CHARA_TIME dt;
uint32_t dn;

if(++i >= NUMBER_OF_FRAMES_PER_FRAME_RATE_CALCULATION)
{
t = arc_ccd_frame_time;
n = arc_ccd_frame_number;

dn = n - old_n;
dt = t - old_t;

if(dt > 0)
{
arc_ccd_frame_rate = 1000.0 * (float)dn / (float)dt;
}
else
arc_ccd_frame_rate = -1;

// set up for next time

old_n = n;
old_t = t;
i=0;
}

return;
} /* arc_ccd_rt_calculate_frame_rate() */

/*****
/* cleanup_module() */
/* */
/* This code cleans up the module and the printer port itself. */
*****/

void cleanup_module(void)
{
//deregister callback function from camera driver

rt_astropci_reg_callback(0);

//remove commands

chara_remove_command(ARC_CCD_RT_SET_ROW);
chara_remove_command(ARC_CCD_RT_GET_ROW);
chara_remove_command(ARC_CCD_RT_REQ_PS);
chara_remove_command(ARC_CCD_RT_REQ_FRAME);
chara_remove_command(ARC_CCD_RT_SYNC_BB);
chara_remove_command(ARC_CCD_RT_CHECK_BB);
chara_remove_command(ARC_CCD_RT_SET_BIAS);
chara_remove_command(ARC_CCD_RT_SET_SPEC_FLAT);
chara_remove_command(ARC_CCD_RT_CHECK_FLAG);
chara_remove_command(ARC_CCD_RT_TOGGLE_FLAG);
chara_remove_command(ARC_CCD_RT_SET_SUPPRESS_PIXELS);
chara_remove_command(ARC_CCD_RT_GET_SUPPRESS_PIXELS);
chara_remove_command(ARC_CCD_RT_REQ_SPECTRUM);
chara_remove_command(ARC_CCD_RT_SET_PS_MEAN_NUM);
chara_remove_command(ARC_CCD_RT_REQ_PURGE_PS_BUF);
chara_remove_command(ARC_CCD_RT_INIT_FFT);
chara_remove_command(ARC_CCD_RT_REQ_PS_STATS);
chara_remove_command(ARC_CCD_RT_START_RECORD);
chara_remove_command(ARC_CCD_RT_END_RECORD);
chara_remove_command(ARC_CCD_RT_TWEAK_PID);
chara_remove_command(ARC_CCD_RT_NEW_TARGET_FREQ);
chara_remove_command(ARC_CCD_RT_SET_FLAG_TRUE);
chara_remove_command(ARC_CCD_RT_SET_FLAG_FALSE);
chara_remove_command(ARC_CCD_RT_SETUP_DITHER_SCAN);
chara_remove_command(ARC_CCD_RT_ACTIVATE_DITHER_SCAN);
chara_remove_command(ARC_CCD_RT_REQ_CHANNELS);
chara_remove_command(ARC_CCD_RT_SET_CHANNELS);
chara_remove_command(ARC_CCD_RT_SET_PS_FLAT);
chara_remove_command(ARC_CCD_RT_REQ_VD_STATS);

} /* cleanup_module() */

```

B.3 The Non-Real-Time Control Code

B.3.1 arc_ccd.c

```

/*****
/* arc_ccd.c */
/*
/* Main routine for the central scrutinizer. */
*****/
/*
/*          CHARA ARRAY USER INTERFACE */
/*          Based on the SUSI User Interface */
/* In turn based on the CHIP User interface */
/*
/*          Center for High Angular Resolution Astronomy
/*          Mount Wilson Observatory, CA 91001, USA */
/*
/* Telephone: 1-626-796-5405
/* Fax      : 1-626-796-6717
/* email    : theo@chara.gsu.edu
/* WWW      : http://www.chara.gsu.edu
/*
/* (C) This source code and its associated executable
/* program(s) are copyright.
/*
*****/
/*
/* Author : Theo ten Brummelaar
/* Date   : January 2000
/* Modified from Dalsa to Arc_ccd by Chad Ogden, Oct 2002
*****/
extern "C" {
#include "arc_ccd_test.h"
}

void arc_ccd_open_function(void);
void arc_ccd_close_function(void);

int main(int argc, char **argv)
{
    if (argc > 1)
    {
        default_display_machine = argv[1];
    }
    else
    {
        default_display_machine = NULL;
    }

    /* Title page. */

    ui_clear_screen();
    put_line("");
    center_line("ARC-CCD 1.0");
    put_line("");
    center_line("The CHARA Array");
    center_line("Center for High Angular Resolution Astronomy");
    center_line("Mount Wilson Observatory, CA 91001, USA");
    put_line("");
    center_line("Telephone: 1-626-796-5405");
    center_line("Fax: 1-626-796-6717");
    center_line("email: theo@chara.gsu.edu");
    center_line("WWW: http://www.mtwilson.edu");
    put_line("");
    center_line("(C) This executable program is copyright.");
    wait_for_title();

    /* Initialize the user interface */

    TITLE = "ARC-CCD 1.0";
    initialise_ui("arc_ccd_menus.ini", "arc_ccd_help.ini", "arcccd");
    set_user_close_function(arc_ccd_close);
    set_user_open_function(arc_ccd_open);

    /* Try and open up the FIFOs */

    if (!open_rt_fifos()) error(FATAL,"Failed to open FIFOs");

    /* Setup background job(s) */

```

```

setup_arc_ccd_background_jobs();

    /* Set up the client port */

    setup_standard_clock_messages();

/* Initilaize camera */

// open_driver();
if (init_arc_ccd() != 0)
exit(-1);

/* Let's go! */

start_ui(); /* Should never return from here. */

exit(0);

} /* main() */

```

B.3.2 arc_ccd_background.c

```

/*****
/* arc_ccd_background.c */
*/
/* Background jobs for central scrutinizer. */
/*****
*/
/*
/*          CHARA ARRAY USER INTERFACE */
/*          Based on the SUSI User Interface */
/* In turn based on the CHIP User interface */
/*
/*          Center for High Angular Resolution Astronomy
/*          Mount Wilson Observatory, CA 91001, USA */
/*
/* Telephone: 1-626-796-5405
/* Fax       : 1-626-796-6717
/* email     : theo@chara.gsu.edu
/* WWW       : http://www.chara.gsu.edu
/*
/* (C) This source code and its associated executable
/* program(s) are copyright.
/*
/*****
*/
/* Author : Theo ten Brummelaar
/* Date   : JAN 2000 */
/* modified by Chad Ogden, Oct 2002 from Dalsa to Arc_ccd */
/*****

extern "C" {
#include "arc_ccd_test.h"
}

/*****
/* setup_arc_ccd_background_jobs() */
/* */
/* Guess what this does. */
/*****

void setup_arc_ccd_background_jobs(void)
{
background_add(get_vd_stats);
background_add(get_ps_stats);
    background_add(error_status);
    background_add(arc_ccd_linux_time_status);
    background_add(arc_ccd_rt_time_status);
    background_add(arc_ccd_status);

} /* setup_arc_ccd_background_jobs() */

/*****
/* unsetup_arc_ccd_background_jobs() */
/* */
/* Guess what this does. */
/*****

```

```

void unsetup_arc_ccd_background_jobs(void)
{
background_del(get_vd_stats);
background_del(get_ps_stats);
    background_del(error_status);
    background_del(arc_ccd_linux_time_status);
    background_del(arc_ccd_rt_time_status);
    background_del(arc_ccd_status);

} /* unsetup_arc_ccd_background_jobs() */

/*****
/* arc_ccd_linux_time_status() */
/* */
/* Displays the linux system time in the status window. */
*****/

int arc_ccd_linux_time_status(void)
{
long current_time;
struct tm *now;

time(&current_time);
now = localtime(&current_time);
wstandout(status_window);
mvwaddstr(status_window,0,0,"Local Tm: ");
wstandend(status_window);
    wprintw(status_window," %2d:%2d:%02d",
        now->tm_hour,now->tm_min,now->tm_sec);

return NOERROR;

} /* arc_ccd_linux_time_status() */

/*****
/* arc_ccd_rt_time_status() */
/* */
/* Displays the rt system time in the status window. */
*****/

int arc_ccd_rt_time_status(void)
{
rt_time_struct now;
ap_time_struct t;
char s[20];

if (get_rt_time(&now) != 0) return ERROR;

wstandout(status_window);
mvwaddstr(status_window,1,0,"CHARA Tm: ");
wstandend(status_window);
t = rt_to_ap_time(now.time_now);
if (t.sign == -1)
strcpy(s,"-");
else
strcpy(s," ");
    sprintf(s+1,"%02ld:%02ld:%02ld",t.hrs,t.min,t.sec);
wprintw(status_window,"%s",s);

wstandout(status_window);
mvwaddstr(status_window,2,0,"Now      : ");
wstandend(status_window);
wprintw(status_window,"%9d",now.time_now);

wstandout(status_window);
mvwaddstr(status_window,3,0,"Lost Tks: ");
wstandend(status_window);
wprintw(status_window,"%9d",now.lost_ticks);

wstandout(status_window);
mvwaddstr(status_window,4,0,"Lost Sec: ");
wstandend(status_window);
wprintw(status_window,"%9d",now.lost_seconds);

return NOERROR;

} /* arc_ccd_rt_time_status() */

/*****
/* arc_ccd_status() */
*****/

```

```

/* */
/* Displays the Arc_ccd parameters in the status window. */
/*****

int arc_ccd_status(void)
{
    wstandout(status_window);
    mvwaddstr(status_window,5,0,"BB Orgn : ");
    wstandend(status_window);
    wprintw(status_window,"%4d,%4d",bbdata.x,bbdata.y);

    wstandout(status_window);
    mvwaddstr(status_window,6,0,"BB Size : ");
    wstandend(status_window);
    wprintw(status_window,"%4d,%4d",bbdata.dx,bbdata.dy);

    wstandout(status_window);
    mvwaddstr(status_window,7,0,"EXPOSURE: ");
    wstandend(status_window);
    wprintw(status_window,"%9d",exposure);

    wstandout(status_window);
    mvwaddstr(status_window,8,0,"FAKE ATM: ");
    wstandend(status_window);
    if (vd_par.use_offsets)
        wprintw(status_window,"      ON");
    else
        wprintw(status_window,"      OFF");

    wstandout(status_window);
        mvwaddstr(status_window,0,20,"USE BIAS: ");
        wstandend(status_window);
    if (use_bias)
        wprintw(status_window,"      YES");
    else
        wprintw(status_window,"      NO");

    wstandout(status_window);
        mvwaddstr(status_window,1,20,"FLAT PS : ");
        wstandend(status_window);
    if (use_ps_flat)
        wprintw(status_window,"      YES");
    else
        wprintw(status_window,"      NO");

    wstandout(status_window);
        mvwaddstr(status_window,2,20,"BIAS SET: ");
        wstandend(status_window);
    if (bias_has_been_set)
        wprintw(status_window,"      YES");
    else
        wprintw(status_window,"      NO");

    wstandout(status_window);
        mvwaddstr(status_window,3,20,"PSFLTSET: ");
        wstandend(status_window);
    if (ps_flat_has_been_set)
        wprintw(status_window,"      YES");
    else
        wprintw(status_window,"      NO");

    wstandout(status_window);
    mvwaddstr(status_window,4,20,"PS MEAN : ");
    wstandend(status_window);
    wprintw(status_window,"%9d",ps_mean_num);

    wstandout(status_window);
    mvwaddstr(status_window,5,20,"PEAK BIN: ");
    wstandend(status_window);
    wprintw(status_window,"%9.1f",ps_peak.bin);

    wstandout(status_window);
    mvwaddstr(status_window,6,20,"PEAK SNR: ");
    wstandend(status_window);
    wprintw(status_window,"%9.6g",ps_peak.snr);

    wstandout(status_window);
    mvwaddstr(status_window,7,20,"PEAK SIG: ");
    wstandend(status_window);

```



```

wprintw(status_window,"%9.1f",ps_peak.sigma);

wstandout(status_window);
mvwaddstr(status_window,8,20,"PEAK PHS: ");
wstandend(status_window);
wprintw(status_window,"%9.1f",ps_peak.phase);

wstandout(status_window);
mvwaddstr(status_window,0,40,"SERVO : ");
wstandend(status_window);
if(ps_servo_on)
wprintw(status_window,"      ON");
else
wprintw(status_window,"      OFF");

wstandout(status_window);
mvwaddstr(status_window,1,40,"ERRSIG : ");
wstandend(status_window);
wprintw(status_window,"%9.1f",ps_peak.errrsignal);

wstandout(status_window);
mvwaddstr(status_window,2,40,"TRACKSIG: ");
wstandend(status_window);
wprintw(status_window,"%9.0f",ps_peak.tracksignal);

wstandout(status_window);
mvwaddstr(status_window,3,40,"SQ WAVE : ");
wstandend(status_window);
if(ps_servo_squarewave_on)
wprintw(status_window,"      ON");
else
wprintw(status_window,"      OFF");

wstandout(status_window);
mvwaddstr(status_window,4,40,"DITH IDX: ");
wstandend(status_window);
wprintw(status_window,"%9d",vd_par.sweep_index);

wstandout(status_window);
mvwaddstr(status_window,5,40,"SERVO P : ");
wstandend(status_window);
wprintw(status_window,"%9.3f",ps_servo_p);

wstandout(status_window);
mvwaddstr(status_window,6,40,"SERVO I : ");
wstandend(status_window);
wprintw(status_window,"%9.3f",ps_servo_i);

wstandout(status_window);
mvwaddstr(status_window,7,40,"SERVO D : ");
wstandend(status_window);
wprintw(status_window,"%9.3f",ps_servo_d);

wstandout(status_window);
mvwaddstr(status_window,8,40,"SERVO G : ");
wstandend(status_window);
wprintw(status_window,"%9.3f",ps_servo_gain);

wstandout(status_window);
mvwaddstr(status_window,0,60,"FRAME Hz: ");
wstandend(status_window);
wprintw(status_window,"%9.3f",ps_peak.frame_rate);

wstandout(status_window);
mvwaddstr(status_window,1,60,"FRAME ms: ");
wstandend(status_window);
wprintw(status_window,"%9.3f",1000.0/ps_peak.frame_rate);

wstandout(status_window);
mvwaddstr(status_window,2,60,"VD sweep: ");
wstandend(status_window);
if(vd_par.sweep_on)
wprintw(status_window,"      ON");
else
wprintw(status_window,"      OFF");

wstandout(status_window);
mvwaddstr(status_window,3,60,"VD auto : ");
wstandend(status_window);
if(vd_par.sweep_automatic)

```

```

wprintw(status_window,"      ON");
else
wprintw(status_window,"      OFF");

wstandout(status_window);
mvwaddstr(status_window,4,60,"VDOffset: ");
wstandend(status_window);
if(vd_par.use_offsets)
wprintw(status_window,"      ON");
else
wprintw(status_window,"      OFF");

wstandout(status_window);
mvwaddstr(status_window,5,60,"VDswp um: ");
wstandend(status_window);
wprintw(status_window,"%9.3f",vd_par.sweep_um);

wstandout(status_window);
mvwaddstr(status_window,6,60,"VDswp dc: ");
wstandend(status_window);
wprintw(status_window,"%9d",vd_par.sweep_dac);

wstandout(status_window);
mvwaddstr(status_window,7,60,"Dith DAC: ");
wstandend(status_window);
wprintw(status_window,"%9d",vd_par.dac_curr);

wstandout(status_window);
mvwaddstr(status_window,8,60,"Req DAC: ");
wstandend(status_window);
wprintw(status_window,"%9d",vd_par.dac_req);

background_process_server_socket(shut_server);

return NOERROR;
} /* arc_ccd_status() */

```

B.3.3 arc_ccd_bias.c

```

/*****
/* arc_ccd_bias.c */
*/
/*
/* Routines to read/write info to the realtime process. */
*****/
/*
/*          CHARA ARRAY USER INTERFACE */
/*          Based on the SUSI User Interface */
/* In turn based on the CHIP User interface */
/*
/*          Center for High Angular Resolution Astronomy
/*          Mount Wilson Observatory, CA 91001, USA */
/*
/* Telephone: 1-626-796-5405
/* Fax       : 1-626-796-6717
/* email    : theo@chara.gsu.edu
/* WWW      : http://www.chara.gsu.edu
/*
/* (C) This source code and its associated executable
/* program(s) are copyright.
/*
*****/
/*
/* Author : Chad Ogden
/* Date   : Oct 2002 */
*****/

extern "C" {
#include "arc_ccd.h"
}

// globals
bool use_bias = FALSE;
bool bias_has_been_set = FALSE;
long int *bias = NULL;

/*****
/* free_bias() */
*****/

```

```

/* */
/* If bias buffer is not NULL, frees it. */
/*****

int free_bias(void)
{
    if (bias != NULL)
    {
        free(bias);
        bias = NULL; //Is this necessary after free()?
    }

    // make sure use_bias is set to FALSE

    use_bias = FALSE;
    set_rt_flag(FLAG_ID_USE_BIAS,FALSE);

    // and arc_ccd_rt bias_has_bben_set flag must be FALSE
    bias_has_bben_set = FALSE;
    set_rt_flag(FLAG_ID_BIAS_HAS_BEEN_SET,FALSE);

    // kill the flat while you're at it, since the bias has changed.

    use_spec_flat = FALSE;
    set_rt_flag(FLAG_ID_USE_SPEC_FLAT,FALSE);

    spec_flat_has_bben_set = FALSE;
    set_rt_flag(FLAG_ID_SPEC_FLAT_HAS_BEEN_SET,FALSE);

    use_ps_flat = FALSE;
    set_rt_flag(FLAG_ID_USE_PS_FLAT,FALSE);

    ps_flat_has_bben_set = FALSE;
    set_rt_flag(FLAG_ID_PS_FLAT_HAS_BEEN_SET,FALSE);

    return NOERROR;
} // free_bias()

/*****
/* toggle_bias() */
/* */
/* Toggles boolean use_bias flag. */
/*****

int toggle_bias(void)
{
    if(bias_has_bben_set)
    {

        use_bias = !use_bias;

        set_rt_flag(FLAG_ID_USE_BIAS,use_bias);

        return NOERROR;
    }
    else
    {
        return error(ERROR,"Bias frame has not been set. Make a bias first.");
    }
} // toggle_bias()

/*****
/* make_bias() */
/* */
/* Takes as an argument the number of frames to use in the bias */
/* calculation. Adds them together and averages, puts these values */
/* in the bias buffer. */
/*****

int make_bias(int frames)
{
    int i,loop, breakout,pixels;
    float *add_buf;
    uint16_t *temp_buf;

    // Turn off bias

    use_bias = FALSE;

```

```

set_rt_flag(FLAG_ID_USE_BIAS,FALSE);

if(bbquadrant<1 || bbquadrant >4)
pixels = bbdata.dx * bbdata.dy * 4;
else
pixels = bbdata.dx * bbdata.dy;

//free old bias if there is anything there

if (bias != NULL) free(bias);

//allocate memory for bias

if( ( bias = (long int *) malloc(pixels*sizeof(long int)) ) == NULL )
{
return error(ERROR,
"Unable to allocate memory for bias buffer.");
}

//malloc add_buf

if( (add_buf = (float *) malloc(pixels*sizeof(float))) == NULL)
{
free(bias); //clean up to avoid memory leak
return error(ERROR,"Failed to get memory for adding buffer.");
}

//initialize add_buf

for(i=0; i<pixels; i++)
{
*(add_buf+i) = 0.0;
}

//run this loop to acquire frame, add to add_buf

loop=0;
breakout=0;
while(loop<frames)
{
message(system_window,"Bias frame %d",loop+1);

//temp_buf assigned to buffer returned by get_bb()

temp_buf = get_bb();

if(temp_buf != NULL)
{
for(i=0; i<pixels; i++)
{
*(add_buf+i) += (float)( *(temp_buf+i) );
}

//increment loop to show that another frame
//has been added to add_buf

loop++;

//display frees temp_buf when it's done.

display_bb(NULL, temp_buf);
}
else
{
//this is to prevent getting stuck in an endless loop
//if get_bb() isn't returning anything.
breakout++;

if(breakout > ALLOWABLE_MAKE_BIAS_DROPPED_FRAMES)
{
//cleanup and exit. temp_buf is already NULL.
free(add_buf);
free(bias);
werase(system_window);
wrefresh(system_window);
return error(ERROR,
"Too many frames dropped when making bias. Aborted.");
}
}
}
}
//end while loop which obtains and adds frames

```

```

// Clean display
werase(system_window);
wrefresh(system_window);

//divide each pixel by number of frames to get average value
for(i=0; i<pixels; i++)
{
*(add_buf+i) /= (float)frames;
}

//transfer values in add_buf to bias;
for(i=0; i<pixels; i++)
{
*(bias+i) = (long int)( *(add_buf+i) );
}

//free add_buf
free(add_buf);

//malloc some space for temp_buf
if( (temp_buf = (uint16_t *)malloc( pixels*sizeof(uint16_t) ) ) == NULL)
{
return error(ERROR,"Failed to allocate memory for temp_buf.");
}

// copy bias into temp_buf
for(i=0; i<pixels; i++)
{
*(temp_buf + i) = (uint16_t)*(bias + i);
}

// displaying temp_buf will free it at display completion
display_bb(NULL, temp_buf);

//send bias frame to arc_ccd_rt side
write_rt_data(bias,pixels*sizeof(long int));
send_rt_command(ARC_CCD_RT_SET_BIAS,0);
bias_has_been_set = TRUE;
set_rt_flag(FLAG_ID_BIAS_HAS_BEEN_SET,TRUE);

// Turn on bias
use_bias = TRUE;
set_rt_flag(FLAG_ID_USE_BIAS,TRUE);

return NOERROR;
}

// make_bias()

//Bias application for the spectrum has been moved to arc_ccd_rt.c
// This applies to displayed ROIs.

/*****
/* apply_bias() */
/* */
/* Takes pointer to buffer containing data from latest bb image. */
/* Subtracts bias from data, pegging values at zero which would come */
/* out negative. Checks for use_bias flag, as well as whether bias */
/* is NULL. */
*****/

int apply_bias(uint16_t *data)
{
int pixels,i;
long int temp;
//If use_bias isn't true, do nothing.

if (use_bias != TRUE) return NOERROR; // Is this a good return value?

//If bias==NULL, demand make_bias().

```



```

#define DEBUG_PS_MEAN_MESSAGES

#define DISPLAY_PS_DATA_INSTEAD_OF_FRINGE_ENVELOPE

// MAX_NUM_IN_OBSERVING_SWEEP is defined in vis_dither.h as # of positions
// both up and down. 4096. That's 2048 in one direction.

#define MAX_SCAN_FFT_SIZE MAX_NUM_IN_OBSERVING_SWEEP/2 /* Data forced to be <=4*DISPLAY_X */
#define DISPLAY_X MAX_SCAN_FFT_SIZE/4 // that's 512
#define DISPLAY_Y 200
#define MIN_DATA_SIZE 200
#define MAX_SCAN_PS_SIZE (MAX_SCAN_FFT_SIZE/2)+1

// How many times through the main loop before executing a background task
#define MAIN_LOOP_BACKGROUND_FREQUENCY 50

// How many times through main loop before displaying stats which change rapidly
#define MAIN_LOOP_FAST_STATS_FREQUENCY 100

// Comment this out to turn off the loop counter display
// #define DISPLAY_LOOP_COUNTER

// The number of channels whose data record_channels will keep track of and fft
// for display
#define NUM_CHANNELS_DISPLAYABLE 2

// The max number of dither scans whose ps can be averaged together on the fly
#define MAX_NUM_SCANS_IN_PS_MEAN 40

// Uncomment this to use a leaky filter instead of a ring buffer for the ps
// mean
// #define PS_MEAN_LEAKY_FILTER

// Maximum number of spectra in a stack. The most you'd want is all 7
// subapertures of the fiber injector, both sides of the beam splitter.
#define MAX_SPECTRA_PER_CHANNEL 14

// channel_list[0] tells how many entries.
int channel_list[SPEC_SIZE+1];

// Keep this define to have the display colors go blue-green-yellow-red
// instead of blue-cyan-green-yellow-red

#define DISP_COLOR_SKIP_CYAN

// This controls the range of the rgb values. from 0 to RGB_COLOR_SCALE
// Since we're using this with the nsimpleX library, which uses Xcolor,
// which uses unsigned short ints for rgb values, the range should go to
// 65535

#define RGB_COLOR_SCALE 65535

#define FFT_FILTER_WIDTH_SCALE 2

// This is the color of the lines representing the control channel filter width
// on the scan power spectrum plots
// I've decided on grey which is GreyScale[NUM_COLORS/2]

#define FILTER_LINE_COLOR NUM_COLORS/2

// For the charalib fft filter, I just use RGB_COLOR_SCALE/2 for each color
// to get grey.

static XImage *up_image = NULL;
static XImage *dn_image = NULL;
static XImage *ps_image = NULL;
static XImage *env_image = NULL;
//static XImage *high_snr_image = NULL;

/*****
/* get_channels()

```

```

/*
/* Gets channel data from arc_ccd_rt side. Returns pointer to
/* the data. It is the responsibility of the function calling this
/* one to free the pointer when done with it.
/*
/*****

float *get_channels(void)
{
    float *chan;
    time_t start;

    int channel_data_size = spec_row[0] * channel_list[0] * sizeof(float);

    if(spec_row[0] < 1)
    {
        error(ERROR,"No spectral rows defined yet.");
        return NULL;
    }

    if ((chan = (float *) malloc(channel_data_size)) == NULL )
    {
        error(ERROR,"Unable to allocate memory for channels.");
        return NULL;
    }

    // send request to arc_ccd_rt side for spectrum data
    send_rt_command(ARC_CCD_RT_REQ_CHANNELS,0);

    // Wait for rt side to put spec data into DATA_OUT_FIFO
    start=time(NULL);

    //wait for the readout time (MICROSEC_PER_PIXEL*pixels)
    // + exposure time in ms (arg must be in microsec so * 1000)
    //      usleep((bbdata.dx * bbdata.dy * MICROSEC_PER_PIXEL
    //      + (exposure * 1000) );

    while( !char_waiting(fd_data_out) )
    {
        if( time(NULL) > start + 1 )
        {
            error(ERROR,"Timed out waiting for frame data.");
            if(chan != NULL)
                free(chan);
            return NULL;
        }
        usleep(FIFO_POLLING_INTERVAL);
    }

    // read channel data from DATA_OUT_FIFO

    if (!read_arc_data((char *)chan,channel_data_size))
    {
        error(ERROR,"Failed to get channel data.");
        if(chan != NULL)
            free(chan);
        return NULL;
    }
    return chan;
} // get_channels()

/*****
/* set_channel_list() */
/* */
/* Sends channel mask data to the rt side. */
/* */
/*****

int set_channel_list(void)
{
    // turn off interrupts?

    send_rt_command(RTCCD_INTS_OFF,0);

    // send the list

    if( write_rt_data(channel_list,(SPEC_SIZE+1)*sizeof(*channel_list)) !=

```



```

(SPEC_SIZE+1) * sizeof(*channel_list) )
return -2;

// send the command to the rt side

if( !send_rt_command(ARC_CCD_RT_SET_CHANNELS,0) )
return -1;

// turn interrupts back on?

send_rt_command(RTCCD_INTS_ON,0);

return NOERROR;
} /* set_channel_list() */

/*****
/* call_set_channel_list() */
/* */
/* User callable function to setup channel list & send copies to the */
/* rt side. */
/* */
/* Command line args: <channel number> <channel number> ... */
/* */
*****/

int call_set_channel_list(int argc, char **argv)
{
char    s[81], s2[81];
int    i,j,temp;
int    n_channels;
int    channel_sum = 0;
bool    redundant = FALSE;
int    temp_channel_list[SPEC_SIZE];

// pause interrupts?

// look for args

if(argc > 1)
{
n_channels = argc - 1;

// check the range on the requested number of channels

if(n_channels < 1 || n_channels > SPEC_SIZE )
{
return error(ERROR,
"Invalid number of channels requested:%d.",
n_channels);
}

// start reading in the list of channel numbers

for(i=0; i<n_channels; i++)
{
// read the next arg into the channel list

sscanf(argv[i+1], "%d", &temp);

// check each value for range

if(temp >= 1 && temp < SPEC_SIZE)
{
// check each value for redundancy

redundant = FALSE;
for(j=0; j<channel_sum; j++)
{
if(temp == temp_channel_list[j])
redundant = TRUE;
}

if(!redundant)
{
temp_channel_list[channel_sum] = temp;
channel_sum++;
}
}
}
}

```

```

} // done reading in channel list from command line

// set channel list

channel_list[0] = channel_sum;

for(i=1; i<=channel_sum; i++)
channel_list[i] = temp_channel_list[i-1];

}

// ask the user if no info on command line

else
{
// take and display a spectrum
//>>>>>>>>do this later
error(ERROR,"List channel numbers on the command line for now");
}

for(i=1; i<=channel_list[0]; i++)
{
if(i==1)
sprintf(s," %2d", channel_list[i]);
else
{
sprintf(s2," %2d", channel_list[i]);
strcat(s,s2);
}
}

message(system_window,"%2d Channels: %s",channel_list[0],s);

return set_channel_list();

} /* call_set_channel_list() */

/*****
/* get_disp_color() */
/* */
/* This code is adapted from Paul Bourke's website: */
/* astronomy.swin.edu.au/~pbourke/colour/colourramp/source1.c */
/* */
/* vmax: High end of input domain. This is mapped to red. */
/* vmin: Low end of input domain. This is mapped to blue. */
/* v: input value desired for mapping onto the blue -> red color ramp. */
/* c: pointer to COLOR structure, defined at top of file. */
/* COLOR structure holds 3 doubles: r, g, b. */
/* */
/* This function takes value v in the domain vmin..vmax and maps it */
/* to the appropriate RGB values to fit in the range from blue to red. */
/* The ramp goes linearly through the following vertices of the RGB */
/* parameter cube: */
/* blue:(0,0,1) -> cyan:(0,1,1) -> green:(0,1,0) -> yellow:(1,1,0) -> */
/* red:(1,0,0). */
/* The mapped RGB values for v are then written into COLOR structure c. */
/* Returns 0 if successful, error if dv == 0. */
*****/

int get_disp_color(double v, double vmin, double vmax, COLOR *c, double scale)
{
double dv;

// set rgb to white

c->r = 1;
c->g = 1;
c->b = 1;

// force v inside vmin..vmax domain

if(v < vmin)
v = vmin;
if(v > vmax)
v = vmax;

// dv is the spread between vmin and vmax

dv = vmax - vmin;

```

```

// check that dv != 0

if(dv == 0)
return error(ERROR,"vmax = vmin in get_disp_color. Mapping aborted.");

#ifdef DISP_COLOR_SKIP_CYAN

// The cyan was bugging me. This code breaks dv up into 3 equal
// regions. The first third goes directly from blue to green.
// blue -> green -> yellow -> red
// (0,0,1) -> (0,1,0) -> (1,1,0) -> (1,0,0)
// So the first region has the b going 1..0
// as the green simultaneously goes 0..1.
// It traces a straight line across the r=0 face of the rgb cube.

// for the first 1/3 of dv, go from blue to green
// (1,0,0) to (0,1,0) so let b go 1..0 and g go 0..1

if(v < (vmin + 0.333333 * dv))
{
c->r = 0;
c->b = 1 + 3 * (vmin - v) / dv;
c->g = 3 * (v - vmin) / dv;
}

// for the second third of dv, go from green to yellow
// (0,1,0) to (1,1,0) so just let r go 0..1

else if(v < (vmin + 0.666666 * dv))
{
c->r = 3 * (v - vmin - 0.666666 * dv) / dv;
c->b = 0;
}

// for the last third of dv, go from yellow to red
// (1,1,0) to (1,0,0) so let g go 1..0

else
{
c->g = 1 + 3 * (vmin + 0.666666 * dv - v) / dv;
c->b = 0;
}

#else

// This is the default. Go along the vertices of the rgb cube from
// blue -> cyan -> green -> yellow -> red
// (0,0,1) -> (0,1,1) -> (0,1,0) -> (1,1,0) -> (1,0,0)

// for the first 1/4 of dv, go from blue to cyan
// (0,0,1) to (0,1,1) so just let the g value go 0..1

if(v < (vmin + 0.25 * dv))
{
c->r = 0;
c->g = 4 * (v - vmin) / dv;
}

// for the second 1/4 of dv, go from cyan to green
// (0,1,1) to (0,1,0) so just let the b value go 1..0

else if(v < (vmin + 0.5 * dv))
{
c->r = 0;
c->b = 1 + 4 * (vmin + 0.25 * dv - v) / dv;
}

// for the third 1/4 of dv, go from green to yellow
// (0,1,0) to (1,1,0) so just let the r value go 0..1

else if(v < (vmin + 0.75 * dv))
{
c->r = 4 * (v - vmin - 0.5 * dv) / dv;
c->b = 0;
}

// for the last 1/4 of dv, go from yellow to red
// (1,1,0) to (1,0,0) so just let the g value go 1..0

```

```

else
{
c->g = 1 + 4 * (vmin + 0.75 * dv - v) / dv;
c->b = 0;
}

#endif

// now scale the colors
c->r *= scale;
c->g *= scale;
c->b *= scale;

return 0;

} // get_disp_color()

/*****
/* save_channels() */
/* */
/* Gets spectrum data and saves them to a file. The file will consist */
/* of a series of unsigned bytes. */
/* Data types recorded: */
/* 1. Power spectrum statistics (always) */
/* 2. Power Spectra (with -ps flag at command line) */
/* 3. Spectra (with -spec flag at command line) */
*****/

int save_channels(int argc, char **argv)
{
char display_name[256];
char s[81];
char temp[81];
char winname[81];
int i,j,k;
#ifdef USE_CHARALIB_FFT
int l;
#endif

// If you want to keep track of more than 1 stack of spectra
// you'll have to put ANOTHER index on scan_raw_data.
// (a stack is a group whose power spectra and fringe envelopes
// can be averaged together. Typically a stack corresponds to
// a particular pair of beam combiner outputs, i.e. B5-B6)

#ifdef USE_CHARALIB_FFT
float scan_raw_data[NUM_CHANNELS_DISPLAYABLE]
[MAX_SPECTRA_PER_CHANNEL]
[MAX_SCAN_FFT_SIZE];
// in this case the filtered scan data is written into the fft array
float scan_fft_data[NUM_CHANNELS_DISPLAYABLE]
[MAX_SPECTRA_PER_CHANNEL]
[MAX_SCAN_FFT_SIZE];
float scan_ps_data[NUM_CHANNELS_DISPLAYABLE]
[MAX_SPECTRA_PER_CHANNEL]
[MAX_SCAN_PS_SIZE];
float scan_envelope[NUM_CHANNELS_DISPLAYABLE]
[MAX_SPECTRA_PER_CHANNEL]
[MAX_SCAN_FFT_SIZE*2];
#else
float *scan_raw_data[NUM_CHANNELS_DISPLAYABLE]
[MAX_SPECTRA_PER_CHANNEL];
// in this case there is a separate set of pointers for the
// filtered scan data
float *scan_filt_data[NUM_CHANNELS_DISPLAYABLE]
[MAX_SPECTRA_PER_CHANNEL];
float *scan_fft_data[NUM_CHANNELS_DISPLAYABLE]
[MAX_SPECTRA_PER_CHANNEL];
float *scan_ps_data[NUM_CHANNELS_DISPLAYABLE]
[MAX_SPECTRA_PER_CHANNEL];
float *scan_envelope[NUM_CHANNELS_DISPLAYABLE]
[MAX_SPECTRA_PER_CHANNEL];

//set all these pointers to NULL to start out

for(i=0;i<NUM_CHANNELS_DISPLAYABLE; i++)
for(j=0;j<MAX_SPECTRA_PER_CHANNEL; j++)
{
scan_raw_data[i][j] = NULL;

```

```

scan_filt_data[i][j] = NULL;
scan_fft_data[i][j] = NULL;
scan_ps_data[i][j] = NULL;
scan_envelope[i][j] = NULL;
}

char fft_subtract_mean = 1;
int scan_fringe_position[NUM_CHANNELS_DISPLAYABLE]
[MAX_SPECTRA_PER_CHANNEL];
float scan_fringe_weight[NUM_CHANNELS_DISPLAYABLE]
[MAX_SPECTRA_PER_CHANNEL];
#endif

float scan_raw_avg[NUM_CHANNELS_DISPLAYABLE]
[MAX_SPECTRA_PER_CHANNEL];

int fft_start;
int loop_counter;
bool up = TRUE;

int ps_zero_bins = 80;
int ps_zero_bins_inc = 10;

// scan_ps_mean[0][j][k] holds the power for bin k of channel # found
// in disp_channels[0][j] averaged over scans_in_ps_mean scans.
// scan_ps_mean[i][j][k] holds the power for bin k for a particular
// scan, it's a rolling buffer in index i, with the running mean
// at index 0.

float scan_ps_mean[MAX_NUM_SCANS_IN_PS_MEAN+1]
[NUM_CHANNELS_DISPLAYABLE]
[MAX_SCAN_PS_SIZE];
int scan_ps_mean_index = 0;

//>>>>>>>>>>>>>>>>>>.need on_the_fly ability to change this & rezero ps mean

int scans_in_ps_mean = MAX_NUM_SCANS_IN_PS_MEAN;

// this flag tells the program to treat scan_ps_mean[i][*][*] as if
// it were 0 until the scan_ps_mean index has gone through a
// full cycle.
// This effectively ignores the old values in the ps_mean array
// which is necessary when starting up a fresh ps mean.

bool flush_ps_mean = TRUE;

float scan_envelope_mean[NUM_CHANNELS_DISPLAYABLE]
[MAX_SCAN_FFT_SIZE];
float channel_data[MAX_NUM_SIMULTANEOUS_SPECTRA][SPEC_SIZE];

// This array of color structures holds the display colors for
// all of the channels in channel_list[].
// The 0 element is unused so that the indices in this array
// can match up with those in channel_list[]

COLOR chan_colors[SPEC_SIZE+1];

// disp_channels[0][0] tells how many channels to display
// channel numbers given in subsequent slots [0][n].
// slot [1][0] unused.
// slots [1][n] give index in channel_list[] of channel number in
// disp_channels[0][n]
int disp_channels[2][NUM_CHANNELS_DISPLAYABLE+1];

// disp_stack tells which stack in spec_stack to display.
// default to the first stack

int disp_stack = 1;

int control_slot = 1;

char *filename,*p;
bool found_dir;
bool record_data = FALSE;
bool rec_chan = TRUE;
bool rec_vd = TRUE;
bool data_received = TRUE;
bool loop = TRUE;

```

```

bool filter_scan_fft = TRUE;
bool average_fringe_envelopes_across_channels = FALSE;
bool display_filtered_scans = TRUE;
bool internal_fringes = FALSE;
float passes_scale = 1.0;
float tempf;
char *buffer,*pbuffer,*rtdata = NULL;
FILE *output = NULL;
int frames;
int c;
char rec_arg = 0;
struct tm *gmt_out;
time_t now;

Window up_window;
Window dn_window;
Window ps_window;
Window env_window;

char *up_picture = NULL;
char *dn_picture = NULL;
char *ps_picture = NULL;
char *env_picture = NULL;
char *temp_picture = NULL;

// how many positions in dither up or down
int dith_steps=((int)*vis_dith_sweep_um)/2;
int channel_size = channel_list[0]
* sizeof(channel_data[0][0]);
int frame_data_size = spec_row[0] * channel_size;
int data_size = frame_data_size +
sizeof(frame_time) +
sizeof(frame_number) +
sizeof(vd_par);
int buf_size = data_size * dith_steps;

#ifdef DEBUG_SAVE_CHANNELS_MESSAGES
error(MESSAGE,"dith_steps = %d", dith_steps);
error(MESSAGE,"channel_size = %d",channel_size);
error(MESSAGE,"data_size = %d",data_size);
error(MESSAGE,"buf_size = %d",buf_size);

#endif

// fft filter stuff
// index corresponds to channel in disp_channels[0][i]
// 0 index currently unused.

int fft_filter_center[NUM_CHANNELS_DISPLAYABLE+1];
int fft_filter_halfwidth[NUM_CHANNELS_DISPLAYABLE+1];
#ifdef USE_CHARALIB_FFT
// This is the filter which will be applied in frequency space to the
// data after it has been fourier transformed.
float *filter[NUM_CHANNELS_DISPLAYABLE];
float *freqs;
#else
int fft_filter_start[NUM_CHANNELS_DISPLAYABLE+1];
int fft_filter_stop[NUM_CHANNELS_DISPLAYABLE+1];
#endif

float up_min = 0;
float up_max = 0;
float dn_min = 0;
float dn_max = 0;
float ps_min = 0;
float ps_max = 0;
float env_min = 0;
float env_max = 0;

#ifdef PS_MEAN_LEAKY_FILTER
float ps_mean_decay = 0.99;
float ps_element_scale = .01;
#endif

float dummy_scan[MAX_SCAN_FFT_SIZE];
for(i=0; i<MAX_SCAN_FFT_SIZE; i++)
dummy_scan[i] = (float)i;
#ifdef __NCURSES_H

```

```

MEVENT mouse; /* A mouse event. */
#endif

#ifdef USE_CHARALIB_FFT
// figure out lowest power of 2 which is bigger than dith_steps
if(dith_steps < 0 || dith_steps > MAX_SCAN_FFT_SIZE)
{
    sprintf(s,"%d dither steps outside acceptable range.",
    dith_steps);
    return error(ERROR,s);
}

i=1;
loop = TRUE;

while(loop)
{
    // check if 2^i is bigger than dith_steps
    if((1<<i) >= dith_steps)
    loop = FALSE;
    else
    i++;
}

int dynamic_fft_size = (1<<i);
int dynamic_ps_size = (dynamic_fft_size / 2) + 1;
#endif

// is display_ps running?
if (dps_running)
{
    return error(MESSAGE,"Can not save data while power spectrum is being displayed.");
}

no_socket();

/* Check out the command line */

filename = NULL;

if (argc > 1)
{
    record_data = ( *argv[1] == 'Y' || *argv[1] == 'y');
}
else
{
    record_data = ask_yes_no("", "Save data to a file?");
}

if(record_data)
{
    if (argc > 2)
    {
        filename = argv[2];
    }
    else
    {
        sprintf(s,"                                channels.dat");
        if (quick_edit("Filename",s,s,NULL,STRING) == KEY_ESC)
            return NOERROR;
        filename = s;
    }

    while(*filename==' ' && *filename != 0) filename++;
    if (*filename == 0)
    {
        return error(ERROR,"Null file name.");
    }
}

display_filtered_scans =
!ask_yes_no("", "Display unfiltered scan data?");

internal_fringes = ask_yes_no("", "Are these internal fringes?");
if(internal_fringes)
    passes_scale = 2.0;

```

```

//>>>>>>>>>fix this later to handle n display channels and smart channel selection
if(channel_list[0] <= 0 )
{
return error(ERROR,"Channel list not set.  Aborted.");
}
else
{
// pick up to two channels in channel list for display
disp_channels[0][0] = channel_list[0];
if(disp_channels[0][0] > 2) disp_channels[0][0] = 2;

// pick the last two channels

for(i=1; i<=disp_channels[0][0]; i++)
{
disp_channels[0][i] = channel_list[channel_list[0]-i+1];
disp_channels[1][i] = channel_list[0]-i+1;
}
}

#ifdef DEBUG_SAVE_CHANNELS_MESSAGES

error(MESSAGE,"Calculating filter stuff...");

#endif

// figure out the fft filter position and width

//filter center at freq bin
// dither_length/channel_lambda
// dither data length:
// SWEEP_AMPLITUDE * 2 (onsky)
// (fft_points/up dith points)

#ifdef USE_CHARALIB_FFT
// note that a sample time of 1/# data points
// makes for 1 second of data, so bin number
// = frequency in Hz.
// This is how to get around using frequency
// and get straight to bin number.
freqs = fft_freqs(dynamic_fft_size, (1/(float)dynamic_fft_size));
/*
#ifdef DEBUG_CHARALIB_FFT_MESSAGES
error(MESSAGE,"freqs: %f to %f",freqs[0],freqs[dynamic_fft_size/2]);
#endif
*/

#endif

for(i=1; i<=disp_channels[0][0]; i++)
{
tempf = SWEEP_AMPLITUDE *
passes_scale * 2.0 * (float)MAX_SCAN_FFT_SIZE /
(float)dith_steps;
fft_filter_center[i] = (int)(
tempf
* channel_wn[
disp_channels[0][i]+
spec_row_offset
]
) ;

// 1/2 width in bins of filter
// window

fft_filter_halfwidth[i] = (int)(
tempf * channel_delta_wn * FFT_FILTER_WIDTH_SCALE);

#ifdef USE_CHARALIB_FFT
// Calculate filters

filter[i] = make_bandpass_filter(freqs,
dynamic_fft_size,
fft_filter_center[i],
fft_filter_halfwidth[i] * 2);

#ifdef DEBUG_CHARALIB_FFT_MESSAGES
/*
error(MESSAGE,

```



```

"Filter %d: max %f at freq %d, 0 bin: %f, %dbin: %f", i,
filter[i][ fft_filter_center[i] ],
fft_filter_center[i],
filter[i][0],
dynamic_fft_size/2,
filter[i][dynamic_fft_size/2]);
*/
#endif

#else
// start and stop aren't used with the charalib stuff since
// it uses a gaussian filter with a center and FWHM

// apply limits which make sense

fft_filter_start[i] =
fft_filter_center[i] -
fft_filter_halfwidth[i];

fft_filter_stop[i] =
fft_filter_center[i] +
fft_filter_halfwidth[i];

if(fft_filter_start[i] < 1)
fft_filter_start[i] = 1;
else if(fft_filter_start[i] >
MAX_SCAN_FFT_SIZE/2)
fft_filter_start[i] =
MAX_SCAN_FFT_SIZE/2;

if(fft_filter_stop[i] < 1)
fft_filter_stop[i] = 1;
if(fft_filter_stop[i] >
MAX_SCAN_FFT_SIZE/2)
fft_filter_stop[i] =
MAX_SCAN_FFT_SIZE/2;
/*
error(MESSAGE,"chan: %3d lam: %4.3f tempf: %6.1f \n cent: %5d hw: %5d start: %5d stop: %5d",
disp_channels[0][i],
1.0/channel_wn[disp_channels[0][i]+spec_row_offset],
tempf, fft_filter_center[i], fft_filter_halfwidth[i],
fft_filter_start[i], fft_filter_stop[i] );
*/
#endif
}

#ifdef DEBUG_SAVE_CHANNELS_MESSAGES

error(MESSAGE,"Done calculating filter stuff...");

#endif
// set the display colors for the channels in channel_list
// channel_list[0]: # channels in list
// low channel limit: channel_list[i]
// high channel limit: channel_list[channel_list[0]]
// v,vmin,vmac,color pointer

// the highest channel number in the channel list will be red,
// the lowest channel number will be blue, all other channels
// linearly interpolated. This insures a wide color spread when 2
// adjacent channels are being displayed.
// If only 1 channel, then white.

if(channel_list[0] == 1)
{
chan_colors[1].r = chan_colors[1].g = chan_colors[1].b =
RGB_COLOR_SCALE;
}
else for(i=1; i<= channel_list[0]; i++)
{
get_disp_color(channel_list[i],channel_list[1],
channel_list[channel_list[0]],
&(chan_colors[i]), RGB_COLOR_SCALE);
}

// set flags for what will be recorded
if(rec_chan)
rec_arg = ( rec_arg | (1 << 3) );
if(rec_vd)
rec_arg = ( rec_arg | (1 << 4) );

```

```

if(use_bias)
rec_arg = ( rec_arg | (1 << 5) );
if(use_spec_flat)
rec_arg = ( rec_arg | (1 << 6) );
if(use_ps_flat)
rec_arg = ( rec_arg | (1 << 7) );

/*
 * How big is each dither ?
 * Maximum scan size is 4*DISPLAY_X.
 * Minimum is DISPLAY_X. If dither is off this is forced,
 * if dither is on a warning is issued.
 */

if (dith_steps < MIN_DATA_SIZE)
{
    message(system_window,
            "Scan is very short, the display may not keep up.");
    ui_print_log(
            "Scan is very short, the display may not keep up.");
}

if (dith_steps <= 0)
{
    return error(ERROR,"No dither set, that we know of.");
}

if (dith_steps > 4*DISPLAY_X)
{
    flush_data(0,NULL);
    if (display)
    {
        XFlush(theDisplay);
        quitX();
    }
    return error(ERROR,
            "Dith steps up %d > %d, try a smaller dither.",
            data_size, 4*DISPLAY_X);
}

/* Where do we imbed things for FFT use? */

// fftstart tells where in the data array to stop 0 padding
// and start inserting data

fft_start = (MAX_SCAN_FFT_SIZE - dith_steps)/2;

if (fft_start < 0)
{
    flush_data(0,NULL);
    if (display)
    {
        XFlush(theDisplay);
        quitX();
    }
    return error(ERROR,"Scan FFT size too big, try a smaller dither.");
}

/* Create the windows and the names */

sprintf(s,"FRINGE");

strcpy(winname,s);
strcat(winname," UP");
up_window = openWindow(winname,theWidth-2*DISPLAY_X-15,5,
DISPLAY_X, DISPLAY_Y);
strcpy(winname,s);
strcat(winname," DOWN");
dn_window = openWindow(winname,theWidth-2*DISPLAY_X-15,
DISPLAY_Y+40, DISPLAY_X, DISPLAY_Y);

// high_snr_window = openWindow("HIGH SNR",theWidth-DISPLAY_X-5, 5,
// DISPLAY_X, DISPLAY_Y);
strcpy(winname,s);
strcat(winname," ENVELOPE");
env_window = openWindow(winname, theWidth-DISPLAY_X-5, 5, DISPLAY_X,
DISPLAY_Y);

```



```

{
if (default_display_machine == NULL)
{
sprintf(display_name,"%s:0.0",
get_machine_name(display_name));
}
else
{
sprintf(display_name,"%s:0.0",
default_display_machine);
}
if (quick_edit("Display name",display_name,
display_name,NULL,STRING) == KEY_ESC)
return NOERROR;

if (strcmp(display_name,"null") == 0)
{
if (initX(NULL) == 0)
{
return error(ERROR,"Failed to open display.");
}
}
else
{
if (initX(display_name) == 0)
{
return error(ERROR,
"Failed to open display %s.",
display_name);
}
}
}

#ifdef DEBUG_SAVE_CHANNELS_MESSAGES
message(system_window,"Finished checking if display is open.");
#endif

if(record_data)
{
// Write the gmt

now = time(NULL);
gmt_out = gmtime(&now);

if (gmt_out == NULL)
loop = FALSE;

else
if( fwrite(gmt_out, sizeof(*gmt_out), 1, output) != 1)
loop = FALSE;

// write the exposure time

if (fwrite(&exposure, sizeof(exposure), 1, output) != 1)
loop = FALSE;

// write the rec_arg, which shows
// whether ps and spec data will be
// written in addition to the ps stats. Also tells whether bias
// and flats were used (if so, included in file)

if (fwrite(&rec_arg, sizeof(rec_arg), 1, output) != 1)
loop = FALSE;

//write the bounding box data

if (fwrite(&bbdata, sizeof(SROI), 1, output) != 1)
loop = FALSE;

//write the row number used for the spectrum

if(fwrite(spec_row, sizeof(*spec_row),
(MAX_NUM_SIMULTANEOUS_SPECTRA+1),output) !=
(MAX_NUM_SIMULTANEOUS_SPECTRA + 1)

```

[illegible]

```

        mvwaddstr(sub_main_window,5,44,"Dith um = ");
        mvwaddstr(sub_main_window,6,44,"Dith DAC= ");
        mvwaddstr(sub_main_window,7,44,"Dith ON = ");

// channel list and won't change, so just
// write them now.

// j holds the cursor starting position
j=17;
for(i=0; i<channel_list[0] ;i++)
{
    mvwprintw(sub_main_window,1,j+3*i,"%2d ",channel_list[i+1]);
}

    wrefresh(sub_main_window);

werase(status_window);
wrefresh(status_window);

#ifdef DEBUG_SAVE_CHANNELS_MESSAGES
message(system_window,"Finished setting up sub_main_window.");
#endif

#ifdef USE_CHARALIB_FFT
// allocate memory for the raw data

// i indexes the display channels, j indexes the spectral rows
for(i=0; i<disp_channels[0][0]; i++)
for(j=0; j<spec_row[0]; j++)
{
    scan_raw_data[i][j] = vector(1,dith_steps);
}

// remember to free these when you're done.
#else
// fill the fft and ps arrays with zeros

// i indexes which channel, of theose whose data are being
// processed for display

for(i=0;i<disp_channels[0][0]; i++)
{
    // j indexes which spectral row
    for(j=0; j<spec_row[0]; j++)
    {
        // k indexes the chronological data points for
        // a given channel and spectral row

        for(k=0; k<MAX_SCAN_FFT_SIZE; k++)
            scan_raw_data[i][j][k]=0.0;

        for(k=0; k<MAX_SCAN_PS_SIZE; k++)
            scan_ps_data[i][j][k]=0.0;
    }
}
#endif

/* Start looping */

background_off(0,NULL);

// send command to rt side to start recording

recording = TRUE;

freeze_stats_query = TRUE;

// only start recording on the rt side if you're not faking data
// for debugging
// there needs to be some freeing of windows, etc. if this fails

if ( !send_rt_command(ARC_CCD_RT_START_RECORD,rec_arg) )
return error(ERROR,"Unable to get rt side to start recording.");

```

```

if ( !sweep_reset() )
return error(ERROR,"Unable to reset dither sweep.");

#ifdef DEBUG_SAVE_CHANNELS_MESSAGES
message(system_window,
"RT side set to record. Starting recording loop.");

#endif
loop = TRUE;
loop_counter = 0;
frames = 1;
pbuffer = buffer;
while(loop)
{
#ifdef DISPLAY_LOOP_COUNTER
if(loop_counter % MAIN_LOOP_FAST_STATS_FREQUENCY == 0)
{
message(system_window,"%s Frame %d",
CONT_TEXT,frames);
}
#endif
frames++;

/* Get the data */

#ifdef DEBUG_SAVE_CHANNELS_MESSAGES
message(system_window,"%d Getting data from FIFO...",
loop_counter);

#endif

// read channel data for this frame from DATA_OUT_FIFO

//vis dither stats
if( !read_arc_data( (char *)&vd_par , sizeof(vd_par) ) )
{
data_received = FALSE;
message(system_window,
"Failed to get frame dither data.");
}

#ifdef DEBUG_SAVE_CHANNELS_MESSAGES
error(MESSAGE,"%d Got %d bytes of dither data from FIFO.",
loop_counter, sizeof(vd_par));

#endif

// frame time
if( !read_arc_data( (char *)&frame_time, sizeof(frame_time) ) )
{
data_received = FALSE;
message(system_window,
"Failed to get frame time.");
}

// frame number
if( !read_arc_data( (char *)&frame_number,
sizeof(frame_number) ) )
{
data_received = FALSE;
message(system_window,
"Failed to get frame number.");
}

// channel data
for(j=0; j<spec_row[0]; j++)
{
if ( !read_arc_data( (char *)&(channel_data[j][0]),
channel_size) )
{
data_received = FALSE;
message(system_window,

```

```

"Failed to get channel data.");
}
}
#ifdef DEBUG_SAVE_CHANNELS_MESSAGES
error(MESSAGE,"%d Got %d bytes of channel data from FIFO.",
loop_counter, channel_size * spec_row[0]);
#endif

// if data ok, add to buffer and display array
if (data_received)
{
#ifdef DEBUG_SAVE_CHANNELS_MESSAGES
message(system_window,
"%d Success. Putting data into buffer.",
loop_counter);
#endif

// put data into buffer

//frame time
rtdata = (char *) &frame_time;

for(i=0; (unsigned int)i < sizeof(frame_time); i++)
*pbuffer++ = *rtdata++;

//frame number
rtdata = (char *) &frame_number;

for(i=0; (unsigned int)i < sizeof(frame_number); i++)
*pbuffer++ = *rtdata++;

//dither stats
rtdata = (char *)&vd_par;

for(i=0; (unsigned int)i < sizeof(vd_par); i++)
*pbuffer++ = *rtdata++;

//channel data
for(j=0; j<spec_row[0]; j++)
{
rtdata = (char *) &(channel_data[j][0]);

for(i=0; i<channel_size; i++)
{
*pbuffer++ = *rtdata++;
}
}

#ifdef DEBUG_SAVE_CHANNELS_MESSAGES
message(system_window,
"%d Copying data into processing arrays.",
loop_counter);
#endif
//put channel data into processing and display arrays

//i steps through channel numbers in the
// disp_channels[][]

for(i=1; i<=disp_channels[0][0]; i++)
{
// j steps through the spectra in a stack
// spec_stack[0][disp_stack] is the # of spectra
// in stack # disp_stack.
// spec_stack[1][disp_stack] is the beginning
// index of the stack on spec_row[].
// disp_channels[1][i] is the index of that
// channel in the channel_list[].

// data are ordered in
// channel_data[row][channel]
// in order given by spec_row[] for rows
// and channel_list[] for channels.

```



```

// vd_par.sweep_index controls the data point
// time indexing.
// It goes from 1 to 2*dith_steps

for(j=0; j<spec_stack[0][disp_stack]; j++)
{
#ifdef DEBUG_SAVE_CHANNELS_MESSAGES
message(system_window,
"%d i= %d j=%d ilimit = %d jlimit = %d spec = %d chanindex = %d chan = %d cd = %f",
loop_counter, i, j, disp_channels[0][0],
spec_stack[0][disp_stack]-1,
spec_stack[1][disp_stack]-1+j,
disp_channels[1][i],
channel_list[disp_channels[1][i]],
channel_data
[ spec_stack[1][disp_stack]-1+j ]
[ disp_channels[1][i]-1 ] );
#endif

#ifdef USE_CHARALIB_FFT
//this version of scan_raw_data are
// vectors going from 1..dith_steps
// hence the +1 on the loop_counter.

scan_raw_data[i-1][j][loop_counter+1] =
channel_data[
spec_stack[1][disp_stack]-1+j]
[ disp_channels[1][i]-1 ];

#else
scan_raw_data[i-1][j]
[loop_counter+fft_start] =
channel_data
[ spec_stack[1][disp_stack]-1
+j ]
[ disp_channels[1][i]-1 ];

// do the average

if(loop_counter == 0)
scan_raw_avg[i-1][j] =
scan_raw_data[i-1][j]
[loop_counter+fft_start];
else
scan_raw_avg[i-1][j] +=
scan_raw_data[i-1][j]
[loop_counter+fft_start];

if(loop_counter == dith_steps-1)
scan_raw_avg[i-1][j] /=
(float)dith_steps;
#endif
} // end j loop for copying channel_data
} // end i loop for copying data from channel_data
#ifdef DEBUG_SAVE_CHANNELS_MESSAGES
message(system_window,
"%d Done copying data to processing arrays.",
loop_counter);
#endif

// any stats which change quickly and must be
// displayed several times in 1 scan go here

if(loop_counter % MAIN_LOOP_FAST_STATS_FREQUENCY == 0)
{
mvwprintw(sub_main_window,6,32,"%10d",
vd_par.dac_curr);
mvwprintw(sub_main_window,7,32,"%10d",
vd_par.sweep_index);
mvwprintw(sub_main_window,5,53,"%10.2f",
vd_par.sweep_um);
// note that this is a value from the
// dac sweep array.
// the dac_curr value is more accurate.
mvwprintw(sub_main_window,6,53,"%10.0f",

```

```
vd_par.sweep_dac);  
if(vd_par.sweep_on)  
{  
mvwprintw(sub_main_window,7,53,  
           "      ON");  
}  
else  
{  
mvwprintw(sub_main_window,7,53,  
           "      OFF");  
}  
  
wrefresh(sub_main_window);  
}  
} // end if data recieved  
  
// data not recieved  
else  
{  
  
#ifdef DEBUG_SAVE_CHANNELS_MESSAGES  
error(ERROR,"%d Failed. Aborting recording loop.",  
loop_counter);  
  
#endif  
  
loop = FALSE;  
}  
  
//>>>>>>>>>> STUFF TO DO AT THE END OF A DITHER SWEEP  
  
//if the buffer is full or exiting the loop, write data  
if( ((pbuffer - buffer)+1 >= buf_size)  
|| (loop == FALSE))  
{  
  
#ifdef DEBUG_SAVE_CHANNELS_MESSAGES  
message(system_window,"%d Writing buffer to file...",  
loop_counter);  
  
#endif  
if(record_data)  
{  
    if (fwrite(buffer, sizeof(char),  
pbuffer - buffer , output) !=  
(unsigned int)(pbuffer - buffer) )  
    {  
        loop = FALSE;  
        message(system_window,  
            "Failed to write data to %s.",filename);  
    }  
}  
// reset pbuffer to start filling  
// the buffer again.  
  
pbuffer = buffer;  
  
#ifdef DEBUG_SAVE_CHANNELS_MESSAGES  
message(system_window,"%d Buffer reset.", loop_counter);  
  
#endif  
}  
  
// if at the top or bottom of a sweep, process data & display  
// remember,  sweep index goes from 1 to 2xdith_steps  
  
if( loop_counter + 1 >= dith_steps)  
{  
#ifdef DEBUG_SAVE_CHANNELS_MESSAGES  
message(system_window,"%d Processing scan data.",  
loop_counter);  
  
#endif  
  
//>>>>process scan data  
  
// This moves through the ring buffer of ps_mean.  
// If you're doing a leaky filter,  
// it could stay put.
```

```

#ifndef PS_MEAN_LEAKY_FILTER
scan_ps_mean_index++;
#else
// do catch that 0 index, so you don't write over
// the running mean
if(scan_ps_mean_index == 0)
scan_ps_mean_index++;
#endif

// process the scan data into ffts

// i indexes which channel, of those whose data
// are being processed for display
// channel number is
// disp_channels[0][i]

for(i=0;i<disp_channels[0][0]; i++)
{
// j indexes which spectral row
// in the first spec stack

for(j=0; j<spec_stack[0][1]; j++)
{
#ifdef USE_CHARALIB_FFT

//free up memory since these are about
//to have new memory allocated to them

if(scan_fft_data[i][j] != NULL)
{
free_vector(scan_fft_data[i][j],
1,dynamic_fft_size);
scan_fft_data[i][j] = NULL;
}

if(scan_filt_data[i][j] != NULL)
{
free_vector(
scan_filt_data[i][j],
1,dith_steps);
scan_filt_data[i][j] = NULL;
}

if(scan_envelope[i][j] != NULL)
{
free_vector(
scan_envelope[i][j],
1,dynamic_fft_size);
scan_envelope[i][j] = NULL;
}

if(scan_ps_data[i][j] != NULL)
{
free_vector(scan_ps_data[i][j],
0,dynamic_ps_size-1);
scan_ps_data[i][j] = NULL;
}

#ifdef DEBUG_CHARALIB_FFT_MESSAGES
// for(k=1; k<=dith_steps; k++)
// error(MESSAGE,"Raw data point %d: %f",k,
// *(scan_raw_data[i][j]+k) );

message(system_window,"i=%d j=%d Starting fft of raw scan data.", i, j);
#endif
// this function allocates memory
// and returns the fft of the data
// note that its data will
// start at index 1, not 0.

scan_fft_data[i][j] =
fft_fringe(
scan_raw_data[i][j],
dith_steps,
dynamic_fft_size,
fft_subtract_mean
);
#ifdef DEBUG_CHARALIB_FFT_MESSAGES

```

```

message(system_window,
"i=%d j=%d Starting power spectrum from ft data.",i,j);
#endif
//////// get the power spectrum
// this one's indices are going to
// start at 0, as is normally
// returned by ps_fringe().
// Note that we take the power
// spectrum BEFORE filtering
// so we can display what we filtered
// out.

scan_ps_data[i][j] =
ps_fringe(
    scan_fft_data[i][j],
    dynamic_fft_size);

#ifdef DEBUG_CHARALIB_FFT_MESSAGES

if(scan_fft_data[i][j] == NULL)
error(ERROR,"scan_fft_data came up null! ");

// else for(k=1; k<=dynamic_fft_size;k++)
// error(MESSAGE,"fft point %d: %f", k,
// *(scan_fft_data[i][j]+k));

// error(MESSAGE,"dith_steps: %d  fft_size: %d",dith_steps,
// dynamic_fft_size);

message(system_window,"i=%d j=%d Starting filter of ft data.",i,j);

#endif
// apply the filter

if(filter_scan_fft)
{
use_filter(
    scan_fft_data[i][j],
    filter[i+1],
    dynamic_fft_size);
}
/*
* #ifdef DEBUG_CHARALIB_FFT_MESSAGES
* message(system_window,
* "i=%d j=%d Starting gdt to get envelope info.",i,j);
* #endif
* // run the gdt to get info for envelope
*
* We don't really need to do this.  envelope_fringe calculates
* the possible fringe position.
* gdt_fringe_track(
* scan_fft_data[i][j],
* dith_steps,
* dynamic_fft_size,
* 1/(float)dynamic_fft_size,
* fft_filter_center[i+1],
* fft_filter_halfwidth[i+1]*2,
* &scan_fringe_weight[i][j],
* &scan_fringe_position[i][j]);
*/

#ifdef DEBUG_CHARALIB_FFT_MESSAGES
message(system_window,
"i=%d j=%d Starting scan_envelope() of ft data.",i,j);
#endif
//////// get the fringe envelope
// the indices start at 1

scan_envelope[i][j] =
envelope_fringe(
    scan_fft_data[i][j],
    dith_steps,
    dynamic_fft_size,
    &scan_fringe_position[i][j]
);

#ifdef DEBUG_CHARALIB_FFT_MESSAGES
message(system_window,"i=%d j=%d Starting envelope mean.",i,j);

```

```

#endif
//envelope mean
// note the mean k indices start at 0,

if(j==0)
{
    for(k=0;k<dith_steps;k++)
    scan_envelope_mean
    [i][k] =
    scan_envelope[i][j][k+1];
}
else
{
    for(k=0;k<dith_steps;k++)
    scan_envelope_mean
    [i][k] +=
    scan_envelope[i][j][k+1];
}

// divide envelope mean by number
// of spectra

if(j==spec_stack[0][1]-1)
{
    for(k=0; k<dith_steps;k++)
    scan_envelope_mean
    [i][k] /=
    (float)spec_stack[0][1];
}

#ifdef DEBUG_CHARALIB_FFT_MESSAGES
message(system_window,"i=%d j=%d Starting fringe weight.",i,j);
#endif

// calculate the fringe weight

scan_fringe_weight[i][j] =
fringe_weight(
    scan_ps_data[i][j]-1,
    freqs,
    dynamic_fft_size,
    fft_filter_center[i+1],
    fft_filter_halfwidth[i+1]/2);

#ifdef DEBUG_CHARALIB_FFT_MESSAGES
message(system_window,"i=%d j=%d Zeroing ps bins.",i,j);
#endif
//what about zeroing bins?

for(k=0; k<ps_zero_bins; k++)
scan_ps_data[i][j][k] = 0.0;

// shift the ps_mean_index to
// the next position and subtract
// what's there so the average
// stays accurate

if(scan_ps_mean_index >
scans_in_ps_mean)
{
    scan_ps_mean_index = 1;

    //flush ps mean can turn off
    //after the index has gone
    //through a full cycle

    flush_ps_mean = FALSE;
}

#ifdef DEBUG_PS_MEAN_MESSAGES
message(system_window,
"%d Ps mean index %d.",
loop_counter,
scan_ps_mean_index);
#endif

#endif
// ps means
if(j==0)

```

```

{
for(k=0; k<dynamic_ps_size;
k++)
{
// don't do any subtracting from the mean if you're doing a leaky filter
#ifdef PS_MEAN_LEAKY_FILTER
// subtract old values
// from running mean

if(!flush_ps_mean)
scan_ps_mean[0][i][k]
-= scan_ps_mean
[scan_ps_mean_index]
[i][k]/
(float)
(scans_in_ps_mean);
#endif
// put new values

scan_ps_mean
[scan_ps_mean_index]
[i][k] =
scan_ps_data[i][j][k];
}
}
else
{
for(k=0; k<dynamic_ps_size;
k++)
scan_ps_mean
[scan_ps_mean_index]
[i][k] +=
scan_ps_data[i][j][k];
}

// divide mean by number of spectra

if(j==spec_stack[0][1]-1)
{
for(k=0; k<dynamic_ps_size;
k++)
{
#ifdef PS_MEAN_LEAKY_FILTER
// whittle down what's
// in the running
// mean and add a
// scaled version of
// the latest ps

scan_ps_mean[0][i][k]
=
ps_mean_decay *
scan_ps_mean[0][i][k]
+
(scan_ps_mean
[scan_ps_mean_index]
[i][k]
* ps_element_scale);
#else
scan_ps_mean
[scan_ps_mean_index]
[i][k] /=
(float)spec_stack[0][1];
// add to running mean

scan_ps_mean[0][i][k]
+=
(scan_ps_mean
[scan_ps_mean_index]
[i][k]
/
(float)
(scans_in_ps_mean))
;
#endif
}
}
}

```

```

#ifdef DEBUG_CHARALIB_FFT_MESSAGES
message(system_window,
"i=%d j=%d Starting ifft to get filtered scan from ft data.",
i,j);
#endif

///// get the filtered scan data
// by inverse fft
// note that data starts at index 1
// because of the pointer
// returned by invfft_fringe().

scan_filt_data[i][j] =
invfft_fringe(
    scan_fft_data[i][j],
    dith_steps,
    dynamic_fft_size);

// This concludes the calculation of displayable quantities by using
// the charalib fft functions. The next block of code is the older
// way, using static-sized arrays and direct calls to numerical recipes
// fft functions.
#else
// this method uses lots of permanent arrays and calls
// numerical recipes routines directly.
// Filtered scan data is written directly into the scan_fft_data array.
// Have had problems with the power spectra being buggy.

// k indexes the chronological
// data points for
// a given channel and spectral row

// leading zeros
for(k=0; k<fft_start; k++)
scan_raw_data[i][j][k]=0.0;

//following zeros
for(k=fft_start+loop_counter+1;
k<MAX_SCAN_FFT_SIZE; k++)
scan_raw_data[i][j][k]=0.0;

// the new way, subtract the mean
// and normalize to -1..1
for(k=fft_start;
k<=fft_start+loop_counter; k++)
{
/* With normalization... this caused a smegmentation fault...
if(scan_raw_avg[i][j] != 0)
scan_raw_data[i][j][k] =
(scan_raw_data[i][j][k] -
scan_raw_avg[i][j]) /
scan_raw_avg[i][j];
*/

scan_raw_data[i][j][k] -=
scan_raw_avg[i][j];
}

// copy to fft data
for(k=0; k<MAX_SCAN_FFT_SIZE; k++)
scan_fft_data[i][j][k] =
scan_raw_data[i][j][k];

//fft
realft( &(ampscan_fft_data[i][j][0])-1,
MAX_SCAN_FFT_SIZE, 1);

//0 bin power
scan_ps_data[i][j][0] =
scan_fft_data[i][j][0] *
scan_fft_data[i][j][0];

//Nyquist bin power
scan_ps_data[i][j]
[MAX_SCAN_FFT_SIZE/2] =
scan_fft_data[i][j][1] *
scan_fft_data[i][j][1];

```

```

// the rest of the ps bins
for(k=1; k<MAX_SCAN_FFT_SIZE/2; k++)
{
    l = k<<1;
    scan_ps_data[i][j][k] =
        scan_fft_data[i][j][l] *
        scan_fft_data[i][j][l] +
        scan_fft_data[i][j][l+1] *
        scan_fft_data[i][j][l+1]
}

// compute total power before filtering?

//fft data filtering
if(filter_scan_fft)
{
    // This next bit of zeroing
    // could be done with one
    // run through the array and
    // some ifs, but i'm thinking
    // the 2 for loops only going
    // through parts of the array
    // to be zeroed is faster

    // Zero all bins below the
    // filter window
    // bin 0 is 0 freq
    // bin 1 is nyq freq
    // freq n real is in bin 2*n

    for(k=2; k<fft_filter_start[i+1]*2
    ;k++)
        scan_fft_data[i][j][k]
        = 0.0;

    // Zero all bins above the
    // filter window
    // bin fft_filter_stop ends in
    // bin 2*filter_stop + 1
    // and shouldn't be zeroed

    for(k=2*
        (fft_filter_stop[i+1] + 1);
        k< MAX_SCAN_FFT_SIZE;
        k++)
        scan_fft_data[i][j][k]
        = 0.0;

    // get the nyquist bin if needed

    if(fft_filter_start[i+1] <=
        MAX_SCAN_FFT_SIZE/2 &&
        fft_filter_stop[i+1] >=
        MAX_SCAN_FFT_SIZE/2)
        scan_fft_data[i][j][1]
        =0.0;
    } // done filtering except for IFFT

    // copy the filtered
    // data to the envelope fft
    // and leave the negative
    // frequencies at 0. Double
    // the values as you copy
    // them... we're setting up a
    // Hilbert transform to pull
    // out the fringe envelope.

    for(k=0; k<MAX_SCAN_FFT_SIZE; k++)
        scan_envelope[i][j][k]
        =scan_fft_data[i][j][k]
        *2.0;

    // now move the nyquist bin
    // where it belongs in the
    // envelope fft data

    scan_envelope[i][j]

```



```

[MAX_SCAN_FFT_SIZE] =
scan_envelope[i][j]
[1];
scan_envelope[i][j][1] =
0.0;

// now inverse transform to get
// filtered scan data

four1(
    &(scan_envelope[i][j][0])-1,
    (unsigned long)MAX_SCAN_FFT_SIZE
    *2,
    -1);

// pull out the real data
// points and move them
// to the front of
// the array. The real
// data is in bins 0,2,4,etc.

for(k=1;k<MAX_SCAN_FFT_SIZE;k++)
{
    scan_envelope[i][j][k]=
    scan_envelope[i][j]
    [2*k];
}

//envelope mean

if(j==0)
{
    for(k=0;k<MAX_SCAN_FFT_SIZE;k++)
    scan_envelope_mean
    [i][k] =
    scan_envelope[i][j][k];
}
else
{
    for(k=0;k<MAX_SCAN_FFT_SIZE;k++)
    scan_envelope_mean
    [i][k] +=
    scan_envelope[i][j][k];
}

// divide mean by number
// of spectra

if(j==spec_stack[0][1]-1)
{
    for(k=0; k<MAX_SCAN_FFT_SIZE;k++)
    scan_envelope_mean
    [i][k] /=
    (float)spec_stack[0][1];
}

// finish scan filtering by ifft
// now that the fft data has been
// copied for the envelope stuff

realft( &(scan_fft_data[i][j][0])-1,
MAX_SCAN_FFT_SIZE, -1);
// the result must be
// multiplied by 2/npoints

float ifftscale =
    2.0/MAX_SCAN_FFT_SIZE;

for(k=0; k<MAX_SCAN_FFT_SIZE; k++)
scan_fft_data[i][j][k]
*= ifftscale;

//what about zeroing bins?

for(k=0; k<ps_zero_bins; k++)
scan_ps_data[i][j][k] = 0.0;

```

```

// shift the ps_mean_index to
// the next position and subtract
// what's there so the average
// stays accurate

if(scan_ps_mean_index >
scans_in_ps_mean)
{
scan_ps_mean_index = 1;

//flush ps mean can turn off
//after the index has gone
//through a full cycle

flush_ps_mean = FALSE;
}

#ifdef DEBUG_PS_MEAN_MESSAGES
message(system_window,
"%d Ps mean index %d.",
loop_counter,
scan_ps_mean_index);

#endif
// ps means
if(j==0)
{
for(k=0; k<MAX_SCAN_PS_SIZE;
k++)
{
// don't do any subtracting from the mean if you're doing a leaky filter
#ifdef PS_MEAN_LEAKY_FILTER
// subtract old values
// from running mean

if(!flush_ps_mean)
scan_ps_mean[0][i][k]
-= scan_ps_mean
[scan_ps_mean_index]
[i][k]/
(float)
(scans_in_ps_mean);
#endif
// put new values

scan_ps_mean
[scan_ps_mean_index]
[i][k] =
scan_ps_data[i][j][k];
}
}
else
{
for(k=0; k<MAX_SCAN_PS_SIZE;
k++)
scan_ps_mean
[scan_ps_mean_index]
[i][k] +=
scan_ps_data[i][j][k];
}

// divide mean by number of spectra

if(j==spec_stack[0][1]-1)
{
for(k=0; k<MAX_SCAN_PS_SIZE;
k++)
{
#ifdef PS_MEAN_LEAKY_FILTER
// whittle down what's
// in the running
// mean and add a
// scaled version of
// the latest ps

scan_ps_mean[0][i][k]
=
ps_mean_decay *
scan_ps_mean[0][i][k]
+

```

```

(scan_ps_mean
[scan_ps_mean_index]
[i][k]
* ps_element_scale);
#else
scan_ps_mean
[scan_ps_mean_index]
[i][k] /=
(float)spec_stack[0][1];

// add to running mean

scan_ps_mean[0][i][k]
+=
(scan_ps_mean
[scan_ps_mean_index]
[i][k]
/
(float)
(scans_in_ps_mean))
;
#endif
}
}

#endif // this ends the chunk of code that doesn't use the charalib fft stuff

} // end j loop, spectra for each channel

} // end i loop, channels being displayed

#ifdef DEBUG_SAVE_CHANNELS_MESSAGES
message(system_window,"%d Scan data processed.",
loop_counter);

#endif
// make plots

// dith_steps is # of positions in 1 direction
// dither sweep. sweep_index covers the entire dither
// position array, which is up + down positions,
// 2*dith_steps positons. If index > dith_steps,
// then we're in the downward part of the dither array.

// going up

if(up)
{
#ifdef DEBUG_SAVE_CHANNELS_MESSAGES
message(system_window,
"%d Making up, envelope plots.",
loop_counter);

#endif
//make plot for:
//up dither
//envelope mean

for(i=0; i<disp_channels[0][0]; i++)
{
// chan_colors[i] is
// the color structure for the
// channel at index i in the channel
// list.
// Since disp_channels is a subset of
// those listed in channel_list,
// we use disp_channels[i][i] to give
// us the index in the channel list of
// the channel we're displaying.

set_plot_color(
(int)chan_colors[
disp_channels[i][i+1]
].r,
(int)chan_colors[
disp_channels[i][i+1]
].g,
(int)chan_colors[
disp_channels[i][i+1]

```

```

].b
);

// reset these to force autoscaling
// for each channel

up_min = 0;
up_max = 0;
env_min = 0;
env_max = 0;
ps_min = 0;
ps_max = 0;

// data from first channel in list

if(i==0)
{
#ifdef DEBUG_CHARALIB_FFT_MESSAGES
error(MESSAGE,
      "0 Going up, i=%d", i);
#endif

if(display_filtered_scans)
// filtered scan data up

up_picture = plot_funct(
    DISPLAY_X, DISPLAY_Y,
#ifdef USE_CHARALIB_FFT
    scan_filt_data[i][0],
#else
    &(scan_fft_data[i][0]
    [fft_start])-1,
#endif
    dith_steps,
    &up_min,&up_max);

else
// raw scan data up
up_picture = plot_funct(
    DISPLAY_X, DISPLAY_Y,
#ifdef USE_CHARALIB_FFT
    scan_raw_data[i][0],
#else
    &(scan_raw_data[i][0]
    [fft_start])-1,
#endif
    dith_steps,
    &up_min,&up_max);

// filtered fringe envelope

#ifdef DISPLAY_PS_DATA_INSTEAD_OF_FRINGE_ENVELOPE
env_picture = plot_funct(
    DISPLAY_X, DISPLAY_Y,
    &(scan_ps_data[i][0][0])-1,
    // &(scan_ps_mean[scan_ps_mean_index][i][0])-1,
#ifdef USE_CHARALIB_FFT
    dynamic_ps_size,
#else
    MAX_SCAN_PS_SIZE,
#endif
    &ps_min, &ps_max);
#else // this ends the display of ps data instead of the fringe envelope

env_picture = plot_funct(
    DISPLAY_X, DISPLAY_Y,
#ifdef USE_CHARALIB_FFT
    &(scan_envelope_mean[i][0])-1,
#else
    &(scan_envelope_mean[i]
    [fft_start])-1,
#endif
    dith_steps,
    &env_min, &env_max);

#endif // this ends the else display fringe envelope mean
}

// add other channel data to pictures

```

```

else
{
#ifdef DEBUG_CHARALIB_FFT_MESSAGES
error(MESSAGE,
">0 Going up, i=%d", i);
#endif

if(display_filtered_scans)
// filtered scan data up
temp_picture = plot_funct(
DISPLAY_X, DISPLAY_Y,
#ifdef USE_CHARALIB_FFT
scan_filt_data[i][0],
#else
&(scan_fft_data[i][0]
[fft_start])-1,
#endif
dith_steps,
&up_min, &up_max);

else
// raw scan data up
temp_picture = plot_funct(
DISPLAY_X, DISPLAY_Y,
#ifdef USE_CHARALIB_FFT
scan_raw_data[i][0],
#else
&(scan_raw_data[i][0]
[fft_start])-1,
#endif
dith_steps,
&up_min, &up_max);

merge_plots(up_picture,
temp_picture, DISPLAY_X,
DISPLAY_Y);

free(temp_picture);

// filtered fringe envelope

#ifdef DISPLAY_PS_DATA_INSTEAD_OF_FRINGE_ENVELOPE
temp_picture = plot_funct(
DISPLAY_X, DISPLAY_Y,
&(scan_ps_data[i][0][0])-1,
// &(scan_ps_mean[scan_ps_mean_index][i][0])-1,
#ifdef USE_CHARALIB_FFT
dynamic_ps_size,
#else
MAX_SCAN_PS_SIZE,
#endif
&ps_min, &ps_max);

#else // this ends the display of ps_data instead of the fringe envelope

temp_picture = plot_funct(
DISPLAY_X, DISPLAY_Y,
#ifdef USE_CHARALIB_FFT
&(scan_envelope_mean[i][0])-1,
#else
&(scan_envelope_mean[i]
[fft_start])-1,
#endif
dith_steps,
&env_min, &env_max);

#endif // this ends the else display fringe envelope mean

merge_plots(env_picture,
temp_picture, DISPLAY_X,
DISPLAY_Y);

free(temp_picture);

} // done if not 1st display channel

} //done stepping through display channels
#endif USE_CHARALIB_FFT

```

```

//>>>>>>>.kludge display of filter lines on kludge ps disp

//overlay the filter lines
//for the control channel

// this is the x coordinate in
// units of the window size

tempf = (float)fft_filter_start[control_slot] *
(float)DISPLAY_X /
(float)(MAX_SCAN_PS_SIZE+1);
/*
* error(MESSAGE,"start = %f, tempf = %f %d",
* fft_filter_start[control_slot],
* tempf, (int)tempf);
*/
// the - width makes it draw
// the line away from filter center
// (the -x direction for the filter
// start line

overlay_line(DISPLAY_X,DISPLAY_Y,env_picture,
(int)tempf, 0,DISPLAY_Y,-1,0,
FILTER_LINE_COLOR);

tempf = (float)fft_filter_stop[control_slot] *
(float)DISPLAY_X /
(float)(MAX_SCAN_PS_SIZE+1);

overlay_line(DISPLAY_X,DISPLAY_Y,env_picture,
(int)tempf, 0,DISPLAY_Y,1,0,
FILTER_LINE_COLOR);
#endif
}

//going down

else
{
#ifdef DEBUG_SAVE_CHANNELS_MESSAGES
message(system_window,
"%d Making dn, ps plots.",
loop_counter);

#endif
//make plot for :
//dn dither
//ps mean

for(i=0; i<disp_channels[0][0]; i++)
{
// chan_colors[i] is
// the color structure for the
// channel at index i in the channel
// list.
// Since disp_channels is a subset of
// those listed in channel_list,
// we use disp_channels[i][i] to give
// us the index in the channel list of
// the channel we're displaying.

set_plot_color(
(int)chan_colors[
disp_channels[i][i+1]
].r,
(int)chan_colors[
disp_channels[i][i+1]
].g,
(int)chan_colors[
disp_channels[i][i+1]
].b
);

// reset these to force autoscaling
// for each channel

dn_min = 0;
dn_max = 0;
ps_min = 0;
ps_max = 0;

```

```

// data from first channel in list

if(i==0)
{
#ifdef DEBUG_CHARALIB_FFT_MESSAGES
error(MESSAGE,
      "0 Going down, i=%d", i);
#endif

if(display_filtered_scans)
// filtered scan data dn
dn_picture = plot_funct(
    DISPLAY_X, DISPLAY_Y,
#ifdef USE_CHARALIB_FFT
    scan_filt_data[i][0],
#else
    &(scan_fft_data[i][0]
    [fft_start])-1,
#endif
    dith_steps,
    &dn_min, &dn_max);

else
// unfiltered scan data dn
dn_picture = plot_funct(
    DISPLAY_X, DISPLAY_Y,
#ifdef USE_CHARALIB_FFT
    scan_raw_data[i][0],
#else
    &(scan_raw_data[i][0]
    [fft_start])-1,
#endif
    dith_steps,
    &dn_min, &dn_max);

// ps running mean

ps_picture = plot_funct(
    DISPLAY_X, DISPLAY_Y,
    &(scan_ps_mean[scan_ps_mean_index][i][0])-1,
    // &(scan_ps_data[i][0][0])-1,
#ifdef USE_CHARALIB_FFT
    dynamic_ps_size,
#else
    MAX_SCAN_PS_SIZE,
#endif
    &ps_min, &ps_max);
#ifdef USE_CHARALIB_FFT
// display filter in ps space

ps_min = 0;
ps_max = 0;

set_plot_color(
    RGB_COLOR_SCALE/2,
    RGB_COLOR_SCALE/2,
    RGB_COLOR_SCALE/2
);

temp_picture = plot_funct(
    DISPLAY_X, DISPLAY_Y,
    filter[i+1] - 1,
    dynamic_ps_size,
    &ps_min, &ps_max);
merge_plots(ps_picture,
temp_picture,
DISPLAY_X, DISPLAY_Y);

free(temp_picture);

#endif
}

// add other channel data to pictures

else
{
#ifdef DEBUG_CHARALIB_FFT_MESSAGES
error(MESSAGE,

```

```

    ">0 Going down, i=%d", i);
#endif

if(display_filtered_scans)
// filtered scan data dn
temp_picture = plot_funct(
    DISPLAY_X, DISPLAY_Y,
#ifdef USE_CHARALIB_FFT
    scan_filt_data[i][0],
#else
    &(scan_fft_data[i][0]
    [fft_start])-1,
#endif
    dith_steps,
    &dn_min, &dn_max);

else
// unfiltered scan data dn
temp_picture = plot_funct(
    DISPLAY_X, DISPLAY_Y,
#ifdef USE_CHARALIB_FFT
    scan_raw_data[i][0],
#else
    &(scan_raw_data[i][0]
    [fft_start])-1,
#endif
    dith_steps,
    &dn_min, &dn_max);

merge_plots(dn_picture,
    temp_picture, DISPLAY_X,
    DISPLAY_Y);

free(temp_picture);

// ps running mean

temp_picture = plot_funct(
    DISPLAY_X, DISPLAY_Y,
    &(scan_ps_mean[scan_ps_mean_index][i][0])-1,
//    &(scan_ps_data[i][0][0])-1,
#ifdef USE_CHARALIB_FFT
    dynamic_ps_size,
#else
    MAX_SCAN_PS_SIZE,
#endif
    &ps_min, &ps_max);

merge_plots(ps_picture,
    temp_picture, DISPLAY_X,
    DISPLAY_Y);

free(temp_picture);

} // done if not first disp channel

} // done stepping through all disp channels
#ifdef USE_CHARALIB_FFT
// Draw filter lines only if not using the
// charalib filter

//overlay the filter lines
//for the control channel

// this is the x coordinate in
// units of the window size

tempf = (float)fft_filter_start[control_slot] *
(float)DISPLAY_X /
(float)(MAX_SCAN_PS_SIZE+1);

// the - width makes it draw
// the line away from filter center
// (the -x direction for the filter
// start line

overlay_line(DISPLAY_X,DISPLAY_Y,ps_picture,
(int)tempf, 0,DISPLAY_Y,-1,0,
FILTER_LINE_COLOR);

```



```

tempf = (float)fft_filter_stop[control_slot] *
(float)DISPLAY_X /
(float)(MAX_SCAN_PS_SIZE+1);

overlay_line(DISPLAY_X,DISPLAY_Y,ps_picture,
(int)tempf, 0,DISPLAY_Y,1,0,
FILTER_LINE_COLOR);
#endif
} // done generating displays for down dither

//update display
#ifdef DEBUG_SAVE_CHANNELS_MESSAGES
message(system_window,
"%d Updating display quantities.",
loop_counter);
#endif

// channel list won't change, so they're
// displayed previously.

//>>>>kludge. Only built to handle 2 simultaneous channel displays

// display channels
if(control_slot==1)
{
mvwprintw(sub_main_window,2,21,">%3d <",
disp_channels[0][1]);
mvwprintw(sub_main_window,2,27," %3d ",
disp_channels[0][2]);
}
else if(control_slot == 2)
{
mvwprintw(sub_main_window,2,21," %3d ",
disp_channels[0][1]);
mvwprintw(sub_main_window,2,27,">%3d <",
disp_channels[0][2]);
}
else
{
mvwprintw(sub_main_window,2,21,"?%3d ?",
disp_channels[0][1]);
mvwprintw(sub_main_window,2,27,"?%3d ?",
disp_channels[0][2]);
}

// lambda of display channels (nm)
// == 1/wn(1/um) * 1000

mvwprintw(sub_main_window,2,51,"%6.1f",
1000.0/channel_wn[
disp_channels[0][1]+spec_row_offset
]);
mvwprintw(sub_main_window,2,58,"%6.1f",
1000.0/channel_wn[
disp_channels[0][2]+spec_row_offset
]);

// mvwprintw(sub_main_window,3,11,"%10.2f",fringe_max);
mvwprintw(sub_main_window,4,11,"%10.2f",
scan_raw_avg
[disp_channels[1][control_slot]
[0]]);
// mvwprintw(sub_main_window,5,11,"%10.2f",ps_max);
if(display_filtered_scans)
{
mvwprintw(sub_main_window,6,11,"      ON");
}
else
{
mvwprintw(sub_main_window,6,11,"      OFF");
}
if(average_fringe_envelopes_across_channels)
{
mvwprintw(sub_main_window,7,11,"      ON");
}
else

```

```

{
mvwprintw(sub_main_window,7,11,"      OFF");
}

mvwprintw(sub_main_window,3,32,"%10d",
fft_filter_center[control_slot]);
mvwprintw(sub_main_window,4,32,"%10d",
fft_filter_halfwidth[control_slot]*2);
mvwprintw(sub_main_window,5,32,"%10d",
scans_in_ps_mean);
mvwprintw(sub_main_window,6,32,"%10d",
vd_par.dac_curr);
mvwprintw(sub_main_window,7,32,"%10d",
vd_par.sweep_index);

mvwprintw(sub_main_window,3,53,"%10.2f",
ps_peak.frame_rate);
// mvwprintw(sub_main_window,4,53,"%10.2f",fringe_freq);
mvwprintw(sub_main_window,5,53,"%10.2f",
vd_par.sweep_um);
// note that this is a value from the dac sweep array.
// the dac_curr value is more accurate.

mvwprintw(sub_main_window,6,53,"%10.0f",
vd_par.sweep_dac);
if(vd_par.sweep_on)
{
mvwprintw(sub_main_window,7,53,"      ON");
}
else
{
mvwprintw(sub_main_window,7,53,"      OFF");
}

wrefresh(sub_main_window);

#ifdef DEBUG_SAVE_CHANNELS_MESSAGES
message(system_window,
"%d Checking for keyboard hit.",
loop_counter);
#endif

// check for key hit.

if (kbhit())
{
c = get_command();
while(kbhit())
get_command();
#ifdef __NCURSES_H
if (c == KEY_MOUSE)
{
getmouse(&mouse);

if (mouse.bstate == BUTTON1_CLICKED)
{
if (mouse.y == SYSTEM_BEGIN_Y
&& mouse.x <
(signed int)
(strlen(CONT_TEXT)+2) )
{
c = KEY_ESC;
}
}
}

#endif

switch(c)
{
// change which display channel
// is being controlled

case '1':
case '!':
control_slot = 1;
break;

case '2':

```

```

case '@':
control_slot = 2;
break;

// change fringe channel to display

case 'C':
// check to see if we're
// already at the highest
// index in channel_list

if (disp_channels[1][control_slot]
< channel_list[0])
{
// this is the index
// of the current
// display channel
// in the channel_list
// disp_channels[1]
// [control_slot];

// set the disp channel

disp_channels[0]
[control_slot] =
channel_list[
++disp_channels[1]
[control_slot]
];

// so we moved the
// index of that disp
// channel up by one,
// then looked up the
// new channel number
// in channel_list.

//set the new filter
//values for the disp
//channel

tempf =
SWEEP_AMPLITUDE *
2.0 *
passes_scale *
MAX_SCAN_FFT_SIZE /
(float)dith_steps;

fft_filter_center
[control_slot] =
(int)(
tempf
* channel_wn[
disp_channels[0]
[control_slot]+
spec_row_offset
]
) ;

// 1/2 width in
// bins of filter
// window

fft_filter_halfwidth
[control_slot] =
(int)(
tempf *
channel_delta_wn *
FFT_FILTER_WIDTH_SCALE);

#ifdef USE_CHARALIB_FFT
//recalculate the
//filter

filter[control_slot] =
make_bandpass_filter(
freqs,
dynamic_fft_size,
fft_filter_center
[control_slot],

```

```

        fft_filter_halfwidth
        [control_slot] * 2
    );
#else
// apply limits
// which make sense

fft_filter_start
[control_slot] =
fft_filter_center
[control_slot] -
fft_filter_halfwidth
[control_slot];

fft_filter_stop
[control_slot] =
fft_filter_center
[control_slot] +
fft_filter_halfwidth
[control_slot];

if(fft_filter_start
[control_slot] < 1)
    fft_filter_start
    [control_slot] = 1;

else if(
    fft_filter_start
    [control_slot] >
    MAX_SCAN_FFT_SIZE/2)
    fft_filter_start
    [control_slot] =
    MAX_SCAN_FFT_SIZE/2;

if(fft_filter_stop
[control_slot] < 1)
    fft_filter_stop
    [control_slot] = 1;

if(fft_filter_stop
[control_slot] >
    MAX_SCAN_FFT_SIZE/2)
    fft_filter_stop
    [control_slot] =
    MAX_SCAN_FFT_SIZE/2;
#endif
}
break;

case 'c':
// check to see if we're
// already at the lowest
// index in channel_list

if (disp_channels[1][control_slot]
> 1)
{
// this is the index
// of the current
// display channel
// in the channel_list
// disp_channels[1]
// [control_slot];

// set the disp channel

disp_channels[0]
[control_slot] =
channel_list[
    --disp_channels[1]
    [control_slot]
];

// so we moved the
// index of that disp
// channel down by one,
// then looked up the
// new channel number
// in channel_list.

```

```

//set the new filter
//values for the disp
//channel

tempf =
    SWEEP_AMPLITUDE *
    2.0 *
    passes_scale *
    MAX_SCAN_FFT_SIZE /
    (float)dith_steps;

fft_filter_center
[control_slot] =
(int)(
    tempf
    * channel_wn[
        disp_channels[0]
        [control_slot]+
        spec_row_offset
    ]
) ;

// 1/2 width in
// bins of filter
// window

fft_filter_halfwidth
[control_slot] =
(int)(
    tempf *
    channel_delta_wn *
    FFT_FILTER_WIDTH_SCALE);
#ifdef USE_CHARALIB_FFT
//recalculate the
//filter

filter[control_slot] =
    make_bandpass_filter(
        freqs,
        dynamic_fft_size,
        fft_filter_center
        [control_slot],
        fft_filter_halfwidth
        [control_slot] * 2
    );
#else
// apply limits
// which make sense

fft_filter_start
[control_slot] =
fft_filter_center
[control_slot] -
fft_filter_halfwidth
[control_slot];

fft_filter_stop
[control_slot] =
fft_filter_center
[control_slot] +
fft_filter_halfwidth
[control_slot];

if(fft_filter_start
[control_slot] < 1)
    fft_filter_start
    [control_slot] = 1;

else if(
    fft_filter_start
    [control_slot] >
    MAX_SCAN_FFT_SIZE/2)
    fft_filter_start
    [control_slot] =
    MAX_SCAN_FFT_SIZE/2;

if(fft_filter_stop
[control_slot] < 1)
    fft_filter_stop
    [control_slot] = 1;

```

```

if(fft_filter_stop
   [control_slot] >
       MAX_SCAN_FFT_SIZE/2)
    fft_filter_stop
    [control_slot] =
    MAX_SCAN_FFT_SIZE/2;
#endif
}
break;

// Change the ps filter center bin

case 'P':
if(fft_filter_center
   [control_slot] +
       FFT_FILTER_WIDTH_SCALE <
       MAX_SCAN_FFT_SIZE -
       fft_filter_halfwidth
       [control_slot] - 1)
{
    fft_filter_center
    [control_slot] +=
    FFT_FILTER_WIDTH_SCALE;

#ifdef USE_CHARALIB_FFT
//recalculate the
//filter

filter[control_slot] =
    make_bandpass_filter(
        freqs,
        dynamic_fft_size,
        fft_filter_center
        [control_slot],
        fft_filter_halfwidth
        [control_slot] * 2
    );
#else
//recalculate filter
//start and stop

fft_filter_start
[control_slot] =
    fft_filter_center
    [control_slot] -
    fft_filter_halfwidth
    [control_slot];

fft_filter_stop
[control_slot] =
    fft_filter_center
    [control_slot] +
    fft_filter_halfwidth
    [control_slot];
#endif
}
break;

case 'p':
if(fft_filter_center
   [control_slot] -
       FFT_FILTER_WIDTH_SCALE >
       fft_filter_halfwidth
       [control_slot])
{
    fft_filter_center
    [control_slot] -=
    FFT_FILTER_WIDTH_SCALE;

#ifdef USE_CHARALIB_FFT
//recalculate the
//filter

filter[control_slot] =
    make_bandpass_filter(
        freqs,
        dynamic_fft_size,
        fft_filter_center
        [control_slot],

```

```

        fft_filter_halfwidth
        [control_slot] * 2
    );
#else
//recalculate filter
//start and stop

fft_filter_start
[control_slot] =
fft_filter_center
[control_slot] -
fft_filter_halfwidth
[control_slot];

fft_filter_stop
[control_slot] =
fft_filter_center
[control_slot] +
fft_filter_halfwidth
[control_slot];
#endif
}
break;

// Change the ps filter width

case 'W':
if(fft_filter_halfwidth
[control_slot] +
fft_filter_center
[control_slot]
+ FFT_FILTER_WIDTH_SCALE <
MAX_SCAN_FFT_SIZE - 1
&&
fft_filter_center
[control_slot] -
fft_filter_halfwidth
[control_slot] -
FFT_FILTER_WIDTH_SCALE > 1)
{
fft_filter_halfwidth
[control_slot] =
fft_filter_halfwidth
[control_slot] +
FFT_FILTER_WIDTH_SCALE;

#ifdef USE_CHARALIB_FFT
//recalculate the
//filter

filter[control_slot] =
make_bandpass_filter(
    freqs,
    dynamic_fft_size,
    fft_filter_center
    [control_slot],
    fft_filter_halfwidth
    [control_slot] * 2
);
#else
//recalculate filter
//start and stop

fft_filter_start
[control_slot] =
fft_filter_center
[control_slot] -
fft_filter_halfwidth
[control_slot];

fft_filter_stop
[control_slot] =
fft_filter_center
[control_slot] +
fft_filter_halfwidth
[control_slot];
#endif
}
break;

```

```

case 'w':
if(fft_filter_halfwidth
   [control_slot] -
   FFT_FILTER_WIDTH_SCALE> 1)
{
fft_filter_halfwidth
[control_slot]=
fft_filter_halfwidth
[control_slot] -
FFT_FILTER_WIDTH_SCALE;

#ifdef USE_CHARALIB_FFT
//recalculate the
//filter

filter[control_slot] =
make_bandpass_filter(
    freqs,
    dynamic_fft_size,
    fft_filter_center
    [control_slot],
    fft_filter_halfwidth
    [control_slot] * 2
    );
#else
//recalculate filter
//start and stop

fft_filter_start
[control_slot] =
fft_filter_center
[control_slot] -
fft_filter_halfwidth
[control_slot];

fft_filter_stop
[control_slot] =
fft_filter_center
[control_slot] +
fft_filter_halfwidth
[control_slot];
#endif
}
break;

// change number of power spectra
// in mean
// no need to do a flush.

case 'N':
if(scans_in_ps_mean <
    MAX_NUM_SCANS_IN_PS_MEAN)
scans_in_ps_mean++;
break;

case 'n':
if(scans_in_ps_mean > 1)
scans_in_ps_mean--;
break;

// Flush the ps mean

case 'X':
case 'x':
for(i=0;
    i<NUM_CHANNELS_DISPLAYABLE;
    i++)
for(j=0; j<MAX_SCAN_PS_SIZE;
    j++)
{
scan_ps_mean[0][i][j]
= 0.0;
}
flush_ps_mean = TRUE;
scan_ps_mean_index = 0;
break;

case 'U':
case 'u':
display_filtered_scans = FALSE;

```



```

break;

case 'F':
case 'f':
display_filtered_scans = TRUE;
break;

case 'Z':
if(ps_zero_bins <
MAX_SCAN_PS_SIZE-
ps_zero_bins_inc)
ps_zero_bins +=
ps_zero_bins_inc;
break;

case 'z':
if(ps_zero_bins-
ps_zero_bins_inc > 0)
ps_zero_bins-=
ps_zero_bins_inc;
break;

/* Not enabling use of the arcccd managed dither scan.
* // Start arcccd dither scan
* // with current settings
* case '>':
* case ',':
* activate_dither_scan(
* (unsigned char)1);
* break;
*
* // Restart arcccd dither scan
* // with current settings
* case '<':
* case ',':
* activate_dither_scan(
* (unsigned char)2);
* break;
*
* //Pause arcccd dither scan
* case '?':
* case '/':
* activate_dither_scan(
* (unsigned char)0);
* break;
*/

#ifdef SPEAK_TO_OPLE
// Move ople down max
case 'J':
break;

// Move ople down min
case 'j':
break;

// Move ople up max
case 'K':
break;

// Move ople up min
case 'k':
break;
#endif

// stop looping

case KEY_ESC:
loop = FALSE;
break;

default:
break;
} // end kbhit switch(c)

} // end if(kbhit)

////////display scan

```

```

#ifdef DEBUG_SAVE_CHANNELS_MESSAGES
message(system_window,
"%d Displaying data.", loop_counter);
#endif
// update local display windows

// check for screwed up picture pointers

if( ( up &&
(up_picture == NULL || env_picture == NULL) )
    || ( !up &&
(dn_picture == NULL || ps_picture == NULL) )
    )
{
loop = FALSE;
error(ERROR,
"Failed to allocate memory for a picture. Aborting.");
}

// pitures aren't null, go ahead with display.
else
{
// going up display:
// up data
// envelope mean

if(up)
{
// destroy old images

if(up_image != NULL)
XDestroyImage(up_image);
if(env_image != NULL)
XDestroyImage(env_image);

// create images

up_image = XCreateImage(theDisplay,
theVisual, theDepth, ZPixmap,
0, up_picture, DISPLAY_X,
DISPLAY_Y, theBitmapPad, 0);
env_image = XCreateImage(theDisplay,
theVisual, theDepth, ZPixmap,
0, env_picture, DISPLAY_X,
DISPLAY_Y, theBitmapPad, 0);

// put & map images

XPutImage(theDisplay, up_window, theGC,
up_image,0,0,0,0, DISPLAY_X,
DISPLAY_Y);
XMapWindow(theDisplay, up_window);

XPutImage(theDisplay, env_window, theGC,
env_image,0,0,0,0, DISPLAY_X,
DISPLAY_Y);
XMapWindow(theDisplay, env_window);
}

// going down display:
// down data
// ps mean
else
{
// destroy old images

if(dn_image != NULL)
XDestroyImage(dn_image);
if(ps_image != NULL)
XDestroyImage(ps_image);

// create images

dn_image = XCreateImage(theDisplay,
theVisual, theDepth, ZPixmap,
0, dn_picture, DISPLAY_X,
DISPLAY_Y, theBitmapPad, 0);
ps_image = XCreateImage(theDisplay,

```

```

theVisual, theDepth, ZPixmap,
0, ps_picture, DISPLAY_X,
DISPLAY_Y, theBitmapPad, 0);

// put & map images

XPutImage(theDisplay, dn_window, theGC,
dn_image,0,0,0,0, DISPLAY_X,
DISPLAY_Y);
XMapWindow(theDisplay, dn_window);

XPutImage(theDisplay, ps_window, theGC,
ps_image,0,0,0,0, DISPLAY_X,
DISPLAY_Y);
XMapWindow(theDisplay, ps_window);

}

// make sure the result is displayed
XFlush(theDisplay);

} // done displaying pictures
#ifdef USE_CHARALIB_FFT
// free up data calculated by this sweep
for(i=0;i<disp_channels[0][0]; i++)
for(j=0; j<spec_stack[0][1]; j++)
{
if(scan_fft_data[i][j] != NULL)
{
free_vector(scan_fft_data[i][j],
1,dynamic_fft_size);
scan_fft_data[i][j] = NULL;
}

if(scan_filt_data[i][j] != NULL)
{
free_vector(scan_filt_data[i][j],
1,dith_steps);
scan_filt_data[i][j] = NULL;
}

if(scan_envelope[i][j] != NULL)
{
free_vector(scan_envelope[i][j],
1,dynamic_fft_size);
scan_envelope[i][j] = NULL;
}

if(scan_ps_data[i][j] != NULL)
{
free_vector(scan_ps_data[i][j],
0,dynamic_ps_size-1);
scan_ps_data[i][j] = NULL;
}
}
#endif
// reset loop counter
// it will be incremented momentarily.
//
loop_counter = -1;

// Switch direction that we think the dither is going.

up = !up;

} // end of if(step_count >= dith_steps)
// this ends the loop called at each end of the dither sweep

//increment loop counter

#ifdef DEBUG_SAVE_CHANNELS_MESSAGES
message(system_window,"Loop %d complete.",loop_counter);
#endif
loop_counter++;

/* Don't run a background. There are background jobs which request data
* from the rt side and will gum up the data_out_fifo.

```

```

*
* // run 1 background process
* // maybe do this much less often than every loop
*
* if(loop_counter % MAIN_LOOP_BACKGROUND_FREQUENCY == 0)
* background();
*/

} // end of while(loop)

// tell rt side to stop sending data

#ifdef DEBUG_SAVE_CHANNELS_FAKE_DATA
send_rt_command(ARC_CCD_RT_END_RECORD,0);
#endif

// purge RT_FIFOs
flush_data(0,NULL);

recording = FALSE;

freeze_stats_query = FALSE;

//restart background processes
background_on(0,NULL);

// cleanup
free(buffer);

#ifdef USE_CHARALIB_FFT
// free memory for the raw data

// i indexes the display channels, j indexes the spectral rows
for(i=0; i<disp_channels[0][0]; i++)
for(j=0; j<spec_row[0]; j++)
{
if(scan_raw_data[i][j] != NULL)
{
free(scan_raw_data[i][j]);
scan_raw_data[i][j] = NULL;
}
}

#endif

XDestroyWindow(theDisplay, up_window);
XDestroyWindow(theDisplay, dn_window);
XDestroyWindow(theDisplay, ps_window);
XDestroyWindow(theDisplay, env_window);
XFlush(theDisplay);

werase(system_window);
wrefresh(system_window);

if(record_data)
fclose(output);

return NOERROR;

} /* save_channels() */

```



```

// Specify the base ROI, which is reflected in all quadrants
if(single_quad_mode)
// 1 quad mode, ul quad
bbquadrant = 2;
else
//4 quad mode
bbquadrant = 0;

if(single_quad_mode)
{
if (set_bb(0, 0, CCD_X_PIXELS / 2, CCD_Y_PIXELS / 2) != 0)
goto Error;
}

else
{
#ifdef USE_CCD_OVERSCAN
if (set_bb(0, 0, CCD_X_PIXELS / 2, CCD_Y_PIXELS / 2) != 0)
#else
if (set_bb(USEFUL_X, USEFUL_Y, USEFUL_DX / 2, USEFUL_DY / 2) != 0)
#endif
goto Error;
}

// if not using the API (the old way of doing things, speak directly
// to the RTCCD module. The API should be used. This code is outdated.
#else
/* Turn off interrupts */

send_rt_command(RTCCD_INTS_OFF,0);

/* Default to full chip */

#ifdef USE_CCD_OVERSCAN

bbdata.x = 0;
bbdata.dx = CCD_X_PIXELS;
bbdata.y = 0;
bbdata.dy = CCD_Y_PIXELS;

#else

bbdata.x = USEFUL_X;
bbdata.dx = USEFUL_DX;
bbdata.y = USEFUL_Y;
bbdata.dy = USEFUL_DY;

#endif

//send bbdata to camera driver

write_rt_data(&bbdata,sizeof(SROI));
send_rt_command(RTCCD_SET_ROI,0);

#endif // end code for speaking directly to the RTCCD module

/* Other globals */

frame_time = 0;
frame_number = 0;

for(i=0; i<MAX_NUM_SIMULTANEOUS_SPECTRA+1; i++)
spec_row[i] = -1;

// make sure arc_ccd_rt flags are same initial values as user side

set_rt_flag(FLAG_ID_BIAS_HAS_BEEN_SET,bias_has_been_set);
set_rt_flag(FLAG_ID_USE_BIAS,use_bias);
set_rt_flag(FLAG_ID_PS_FLAT_HAS_BEEN_SET,ps_flat_has_been_set);
set_rt_flag(FLAG_ID_SPEC_FLAT_HAS_BEEN_SET,spec_flat_has_been_set);
set_rt_flag(FLAG_ID_USE_SPEC_FLAT,use_spec_flat);
set_rt_flag(FLAG_ID_USE_PS_FLAT,use_ps_flat);
set_rt_flag(FLAG_ID_USE_FFT,use_fft);
set_rt_flag(FLAG_ID_USE_LPT_TOGGLE,use_lpt_toggle);

#ifdef USE_VIS_DITHER_SWEEP_POINTERS

//set pointers for vis_dither sweep data

```

```

if( (vis_dith_sweep_dac = vis_dither_sweep_dac_ptr()) == NULL)
{
error(ERROR,"Unable to get address of vis_dither_sweep_dac.");
goto Error;
}

if( (vis_dith_sweep_um = vis_dither_sweep_um_ptr()) == NULL)
{
error(ERROR,"Unable to get address of vis_dither_sweep_um.");
goto Error;
}
#endif

#ifdef USE_RTCCD_API // if speaking directly to rtccd module
/* Turn on interrupts */

send_rt_command(RTCCD_INTS_ON,0);
#endif

return 0;

#ifdef USE_RTCCD_API
Error:
rtccdAPI_close();
return -1;
#endif

} /* init_arc_ccd() */

/*****
/* open_driver() */
/* */
/* Issues the RTCCD_OPEN command and turns interrupts on with */
/* RTCCD_INTS_ON. */
/* */
*****/

int open_driver(void)
{
#ifdef USE_RTCCD_API // the old way, speaking directly to rtccd mod.

if(!send_rt_command(RTCCD_OPEN,0)) return -1;

if(!send_rt_command(RTCCD_INTS_ON,0)) return -1;
return 0;

#else // the current way, speaking to the rtccd api.

if(clear_bb(0,NULL) != 0) return -1;

return 0;

#endif

} //open_driver()

/*****
/* close_driver() */
/* */
/* Issues the RTCCD_CLOSE command and turns interrupts off with */
/* RTCCD_INTS_OFF. */
/* */
*****/

int close_driver(void)
{
#ifdef USE_RTCCD_API // the old way, speaking directly to rtccd mod.

if(!send_rt_command(RTCCD_INTS_OFF,0)) return -1;

if(!send_rt_command(RTCCD_CLOSE,0)) return -1;
return 0;

#else // the current way, speaking to the rtccd api

rtccdAPI_close();

return 0;

```

```

#endif

} // close_driver()

/*****
 * reset_arc_ccd() */
 * */
 * Closes and reopens arc ccd vis the rtccd api. Must be called before */
 * changing bounding boxes, as only one is allowed per session. */
 * Must be followed by : */
 * rtccdAPI_SetBaseROI() */
 * rtccdAPI_SetExposureTime() */
 * rtccdAPI_SelectROITransferMode() */
 * rtccdAPI_StartExposures() */
 * */
 * These functions are covered in the set_bb() function which calls */
 * set_exposure(). */
 *****/

int reset_arc_ccd(void)
{
    // close arc_ccd if it's open

    rtccdAPI_close();

    // Initialize rtccdAPI and FIFO system

    if (rtccdAPI_open("RT_FIFO", 94, 80) != 0) goto Error;

    // Initialize camera system and load PCI and timing files

    if (rtccdAPI_InitCameraSystem(ARC_CCD_LOAD_FILES) != 0) goto Error;

    // Initialize ROI system

    if (rtccdAPI_InitROISystem(1,1) != 0) goto Error;

    return 0;

    // execute this code if one of the function calls didn't work
Error:
    rtccdAPI_close();
    return -1;
} /* reset_arc_ccd() */

// this function sends directly to the rtccd module. Now all
// communication should be with the rtccd api, so this function is obsolete.

#ifndef USE_RTCCD_API

/*****
 * send_ascii_command() */
 * */
 * Sends an ascii command to the arc ccd driver. */
 * See Voodoo and Device Driver Programmer's Reference Manual, Section */
 * V for instruction on formatting the data array. */
 * */
 *****/

int send_ascii_command(unsigned int *data)
{
    //this array will hold the reply. It's also the same size as the
    //data array to be sent.

    unsigned int reply[CMD_DATA_COUNT];

    //structure defined in rtccd.h

    CMD_ARGS command;

    // 3 characters for reply
    char c1,c2,c3;

    //turn off interrupts
    send_rt_command(RTCCD_INTS_OFF,0);

    flush_data(0,NULL);

```



```

//set up and send command data, tells camera its getting ascii command.
command.Command = ASTROPCI_COMMAND;
command.Arg0 = 0;
command.Arg1 = 1;
if(write_rt_data( &command,sizeof(command) ) != sizeof(command))
return -2;

//send ascii data, an array of 6 long ints as defined in
//Voodoo and Device Driver Programmer's reference manual, sec. V
if(write_rt_data( data, sizeof(reply) ) != sizeof(reply))
return -2;

//send command REQ_CMD so driver will process the data written into
//the DataInFIFO
if(!send_rt_command(RTCCD_REQ_CMD,0)) return -1;

//read reply from DataOutFIFO
if( !read_arc_data( (char *)reply,sizeof(reply) ) )
return -2;

//chad: write something later to write this reply data to the client

// convert reply into characters
c1 = (char)((reply[0] & 0x00FF0000) >> 16);
c2 = (char)((reply[0] & 0x0000FF00) >> 8);
c3 = (char)(reply[0] & 0x000000FF);

//print out reply
message(system_window,"Response from ARC CCD: %c%c%c",c1,c2,c3);

//turn interrupts back on
send_rt_command(RTCCD_INTS_ON,0);

return 0;

} //send_ascii_command()

#endif

/*****
 * set_exposure() */
 * */
 * Attempts to set the exposure for ARC CCD */
 * Returns -1 if it fails to send the command, -2 if it fails to */
 * send the data. */
 * Returns -3 if it is not ready to arm. */
 * Returns 0 if it worked. */
 *****/

int set_exposure(unsigned int exposure) // In milliseconds
{
#ifdef USE_RTCCD_API
if( rtccdAPI_SetExposureTime((int)exposure) != 0 )
{
return -1;
}
else
{
return 0;
}
#else
int test;

//set up the data array, as described in Voodoo and device
//driver programmer's manual, section V. array of
//6 32 bit unsigned ints

unsigned int cmd_data[CMD_DATA_COUNT];

//free the bias, so a new one will be required
free_bias();

cmd_data[0] = 0x00000203; //aka (TIM_ID << 8) | 3

```

```

cmd_data[1] = 0x00534554; //aka the SET command
cmd_data[2] = exposure;
cmd_data[3] = 0;
cmd_data[4] = 0;
cmd_data[5] = 0;

if( ( test = send_ascii_command(cmd_data) ) != 0) return test;

return 0;
#endif
} /* set_params() */

// this function is outdated.

#ifdef USE_RTCCD_API

/*****
/* set_bb() */
/* */
/* Attempts to set the bounding box for ARC CCD */
/* Returns -1 if it fails to send the command, -2 if it fails to */
/* send the data. */
/* Returns 0 if it worked. */
*****/

int set_bb(int x, int y, int dx, int dy)
{
    // free the bias, so a new one will be required
    free_bias();

    // turn off interrupts
    send_rt_command(RTCCD_INTS_OFF,0);

    // Format bb data as a SR0I structure
    // SR0I bbdata is a global defined in arc_ccd.h

    // SR0I wants lower left x and y coords, and dx, dy.
    /*
    * // lower left x = upper left x
    *
    * bbdata.x = ulx;
    *
    * // lower left y = lower right y
    *
    * bbdata.y = lry;
    *
    * // dx = lower right x - upper left x +1. (So if indices same, dx = 1)
    *
    * bbdata.dx = lrx - ulx + 1;
    *
    * // dy = upper left y - lower right y + 1.
    *
    * bbdata.dy = uly - lry + 1;
    */

    bbdata.x = x;
    bbdata.y = y;
    bbdata.dx = dx;
    bbdata.dy = dy;

    /* IS this possible? */

    if (bbdata.x < 0 || bbdata.x >= CCD_X_PIXELS ||
        bbdata.y < 0 || bbdata.y >= CCD_Y_PIXELS ||
        bbdata.dy <= 0 || bbdata.dx <= 0 ||
        (bbdata.x + bbdata.dx) > CCD_X_PIXELS ||
        (bbdata.y + bbdata.dy) > CCD_Y_PIXELS)
    {
        error(ERROR,
            "Non-physical bounding box\nx=%d y=%d dx=%d dy=%d.",
            bbdata.x, bbdata.y, bbdata.dx, bbdata.dy);
    }

    /* Default to full chip */

#ifdef USE_CCD_OVERSCAN
    bbdata.x = 0;
    bbdata.dx = CCD_X_PIXELS;
    bbdata.y = 0;

```

```

bbdata.dy = CCD_Y_PIXELS;
#else
// Default to usefule area of chip, skip overscans

bbdata.x = USEFUL_X;
bbdata.dx = USEFUL_DX;
bbdata.y = USEFUL_Y;
bbdata.dy = USEFUL_DY;
#endif
//write roi data to camera driver

write_rt_data(&bbdata,sizeof(SROI));
send_rt_command(RTCCD_SET_ROI,0);

//write roi data to arc_ccd_rt side

write_rt_data(&bbdata,sizeof(SROI));
send_rt_command(ARC_CCD_RT_SYNC_BB,0);

send_rt_command(RTCCD_INTS_ON,0);

return ERROR;
}

//write roi data to camera driver

if (write_rt_data(&bbdata,sizeof(SROI)) != sizeof(SROI))
return -2;
if(!send_rt_command(RTCCD_SET_ROI,0)) return -1;

//sync up arc_ccd_rt side roi data

if (write_rt_data(&bbdata,sizeof(SROI)) != sizeof(SROI))
return -2;
if(!send_rt_command(ARC_CCD_RT_SYNC_BB,0)) return -1;

//turn interrupts back on

send_rt_command(RTCCD_INTS_ON,0);

return 0;

} /* set_bb() */

#else // this is the current version which speaks to the rtccd api

/*****
/* set_bb() */
/* */
/* Attempts to set the bounding box for ARC CCD */
/* Returns -1 if it fails to send the command, -2 if it fails to */
/* send the data. */
/* Returns 0 if it worked. */
*****/

int set_bb(int x, int y, int dx, int dy)
{
// free the bias, so a new one will be required

free_bias();

// turn off interrupts

/* send_rt_command(RTCCD_INTS_OFF,0);
*/
flush_data(0,NULL);

if(reset_arc_ccd())
{
error(ERROR,"Failure while trying to reset arcccd. set_bb aborted.");
return -1;
}

// Format bb data as a SROI structure
// SROI bbdata is a global defined in arc_ccd.h

// SROI wants lower left x and y coords, and dx, dy.

/* // lower left x = upper left x
*

```

```

* bbdata.x = ulx;
*
* // lower left y = lower right y
*
* bbdata.y = lry;
*
* // dx = lower right x - upper left x +1. (So if indices same, dx = 1)
*
* bbdata.dx = lrx - ulx + 1;
*
* // dy = upper left y - lower right y + 1.
*
* bbdata.dy = uly - lry + 1;
*/

bbdata.x = x;
bbdata.y = y;
bbdata.dx = dx;
bbdata.dy = dy;

/* IS this possible? */

if (bbdata.x < 0 || bbdata.x >= CCD_X_PIXELS/2 ||
    bbdata.y < 0 || bbdata.y >= CCD_Y_PIXELS/2 ||
    bbdata.dy <= 0 || bbdata.dx <= 0 ||
    (bbdata.x + bbdata.dx) > CCD_X_PIXELS/2 ||
    (bbdata.y + bbdata.dy) > CCD_Y_PIXELS/2)
{
    error(ERROR,
        "Non-physical bounding box\nx=%d y=%d dx=%d dy=%d.",
        bbdata.x, bbdata.y, bbdata.dx, bbdata.dy);

    // Default to usefule area of chip, skip overscans

    bbdata.x = USEFUL_X;
    bbdata.dx = USEFUL_DX/2;
    bbdata.y = USEFUL_Y;
    bbdata.dy = USEFUL_DY/2;
    bbquadrant = 0;

    //write roi data to camera

    if (rtccdAPI_SetBaseROI(bbdata.x,bbdata.y,bbdata.dx,
        bbdata.dy) != 0)
    {
        error(ERROR,"Unable to send bounding box data to the RTCCD API.");
        return -1;
    }

    // send the remaining necessary commands to the camera

    set_exposure(exposure);
    rtccdAPI_SelectROITransferMode();
    rtccdAPI_StartExposures();

    //write roi data to arc_ccd_rt side

    write_rt_data(&bbdata,sizeof(SROI));
    write_rt_data(&bbquadrant,sizeof(bbquadrant));
    send_rt_command(ARC_CCD_RT_SYNC_BB,0);

    return ERROR;
}

// force the number of rows to be even (arc_ccd limitation)

if (bbdata.dy % 2 != 0)
{
    // if the bottom of the bb is right up against the
    //top of the quadrant

    if(bbdata.y >= (CCD_Y_PIXELS/2) - 1)
    {
        bbdata.y --;
        bbdata.dy ++;
    }

    // if the top of the bb is up against the top of the quadrant

    else if(bbdata.y + bbdata.dy >= (CCD_Y_PIXELS/2))

```

```

bbdata.dy--;

// otherwise you can just increase dy by one.

else
bbdata.dy++;
}

//write roi data to camera driver

if(rtccdAPI_SetBaseROI(bbdata.x,bbdata.y,bbdata.dx,
bbdata.dy) != 0)
{
error(ERROR,"Unable to send bounding box data to the RTCCD API.");
return -1;
}

// send the remaining necessary commands to the camera
if(set_exposure(exposure) != 0) return -1;
if(rtccdAPI_SelectROITransferMode() != 0) return -1;
if(rtccdAPI_StartExposures() != 0) return -1;

//sync up arc_ccd_rt side roi data
if (write_rt_data(&bbdata,sizeof(SROI)) != sizeof(SROI))
return -2;
if (write_rt_data(&bbquadrant,sizeof(bbquadrant))
!= sizeof(bbquadrant)) return -2;
if(!send_rt_command(ARC_CCD_RT_SYNC_BB,0)) return -1;

/* // turn on interrupts
*
* send_rt_command(RTCCD_INTS_ON,0);
*/
return 0;

} /* set_bb() */

#endif

/*****
* get_bb() */
*/
/* Attempts to get data from inside bounding box from the realtime side.*/
/* Will always assume that the array size is that stored in the global */
/* SROI bbdata. */
/* Returns a pointer to memory with the data in it which is allocated */
/* inside this function. The caller function should free this memory. */
/* Returns NULL if things do not go well. */
*****/

uint16_t *get_bb(void)
{
uint16_t *buf;
// uint16_t *iobuf;
// int i,iarraysize;
int abort = 0;
time_t start;
int arraysize,num_pixels;

// if using all 4 quadrants

if(bbquadrant == 0)

num_pixels = 4 * bbdata.dx * bbdata.dy;

// else only using 1 quadrant

else

num_pixels = bbdata.dx * bbdata.dy;

// arraysize = num_pixels * sizeof(long int);
// iarraysize = num_pixels * sizeof(uint16_t);

arraysize = num_pixels * sizeof(uint16_t);

/* OK, allocate the memory */

if ((buf = (uint16_t *) malloc(arraysize)) == NULL )

```

```

{
error(ERROR,"Failed to get memory.");
return NULL;
}

/* if ((iobuf = (uint16_t *) malloc(ioarraysize)) == NULL )
* {
* free(buf);
* error(ERROR,"Failed to get memory.");
* return NULL;
* }
*/
/* Send the command */

if (!send_rt_command(ARC_CCD_RT_REQ_FRAME, 0))
{
error(ERROR,"Failed to send command.");
free(buf);
return NULL;
}

/* Wait for driver to put frame data into DATA_OUT_FIFO */

start=time(NULL);

//wait for the readout time (MICROSEC_PER_PIXEL*pixels)
// + exposure time in ms (arg must be in microsec so * 1000)

usleep((bbdata.dx * bbdata.dy * MICROSEC_PER_PIXEL)
+ (exposure * 1000) );

while( !char_waiting(fd_data_out) )
{
if( time(NULL) > start + 1 )
{
error(ERROR,"Timed out waiting for frame data.");
free(buf);
return NULL;
}
usleep(FIFO_POLLING_INTERVAL);
}

/* Get frame time */

if (!read_arc_data( (char *)&frame_time, sizeof(frame_time) ) )
abort = 1;

/* Get frame number */

if ( !read_arc_data( (char *)&frame_number, sizeof(frame_number) ) )
abort = 1;

/* Get the data */

if ( !read_arc_data( (char *)buf,arraysize ) )
{
error(ERROR,"Failed to get data.");
abort = 1;
}

if (abort == 1)
{
free(buf);
// free(iobuf);
return NULL;
}

/* // old way
* if (use_bias)
* // apply bias and cast into buf as long int
* {
*
* for (i=0; i< (num_pixels); i++)
* *(buf+i) = ( *(iobuf+i) ) - (*(bias+i));
* }
*
* else
* // no bias, cast into buf as long int

```

```

* {
* for (i=0; i< (num_pixels); i++)
* *(buf+i) = *(iobuf+i) ;
* }
*/

// new way

if(use_bias)
{
if(apply_bias(buf) != 0)
{
free(buf);
error(ERROR,"Unsuccesful bias application.");
return NULL;
}
}

// free(iobuf);

/* That is all */

return buf;

} /* get_bb() */

/*****
/* set_rt_flag()
/*
/* Sets boolean flag on rt side. First arg is the flag id constant, */
/* defined in arc_ccd.h. Next argument is the boolean which the rt flag */
/* will be set to. */
*****/

int set_rt_flag(unsigned char flag_id,bool value)
{
bool test;

// turn off interrupts

// send_rt_command(RTCCD_INTS_OFF,0);

// now make sure that arc_ccd_rt side has same setting

//old way
// write_rt_data(&flag_id,sizeof(flag_id));
// send_rt_command(ARC_CCD_RT_CHECK_FLAG,0);

// new way uses arg to send flag id.

send_rt_command(ARC_CCD_RT_CHECK_FLAG,flag_id);

read_arc_data((char *)&test,sizeof(test));

// do this if the rt flag does not match the requested value

if (test != value)
{

// old way
// write_rt_data(&flag_id,sizeof(flag_id));
// send_rt_command(ARC_CCD_RT_TOGGLE_FLAG,0);

// new way. send flag id via arg

send_rt_command(ARC_CCD_RT_TOGGLE_FLAG,flag_id);

read_arc_data((char *)&test,sizeof(test));

// if the rt side returns a new value for the flag which
// doesn't match the requested value

if(test != value)
return error(ERROR,
"Error while setting rt flag. Value doesn't match.");

}

// if the rt flag already matches the requested value,
// nothing needs to be done.

```

```

//turn interrupts back on

// send_rt_command(RTCCD_INTS_ON,0);

return NOERROR;

} // set_rt_flag()

/*****
/* set_rt_flag_blind() */
/*
/* Sets boolean flag on rt side. First arg is the flag id constant, */
/* defined in arc_ccd.h. Next argument is the boolean which the rt flag */
/* will be set to. Doesn't check its work, but doesn't use fifo. */
*****/

int set_rt_flag_blind(unsigned char flag_id,bool value)
{

//call rt set true command if requested value true

if(value)

send_rt_command(ARC_CCD_RT_SET_FLAG_TRUE,flag_id);

// call rt set false if requested value false

else

send_rt_command(ARC_CCD_RT_SET_FLAG_FALSE,flag_id);

return NOERROR;

} // set_rt_flag()

/*****
/* get_ps_stats() */
/*
/* Gets power spectrum data from arc_ccd_rt side. Writes the data */
/* directly into the ps_peak structure. */
*****/

int get_ps_stats(void)
{
    time_t start;

// if freeze_stats_query flag is true, don't do anything

if (freeze_stats_query)
return NOERROR;

// flush_data(0,NULL);

// send request to arc_ccd_rt side for ps data

    send_rt_command(ARC_CCD_RT_REQ_PS_STATS,0);

    // Wait for rt side to put ps data into DATA_OUT_FIFO

    start=time(NULL);

    //wait for the readout time (MICROSEC_PER_PIXEL*pixels)
    // + exposure time in ms (arg must be in microsec so * 1000)

//    usleep((bbdata.dx * bbdata.dy * MICROSEC_PER_PIXEL)
//            + (exposure * 1000) );

    while( !char_waiting(fd_data_out) )
    {
        if( time(NULL) > start + 1 )
        {
            return error(ERROR,"Timed out waiting for ps data.");
        }
        usleep(FIFO_POLLING_INTERVAL);
    }

    // read ps data from DATA_OUT_FIFO

    if (!read_arc_data((char *)&ps_peak,sizeof(ps_peak) ) )

```



```

    {
        return error(ERROR,
"Failed to get power spectrum data.");
    }

return NOERROR;
} // get_ps_stats()

/*****
/* get_vd_stats() */
/*
/* Gets vis_dither data from arc_ccd_rt side. Writes the data */
/* directly into the vd_par structure. */
*****/

int get_vd_stats(void)
{
    time_t start;

// if freeze_stats_query flag is true, don't do anything
if (freeze_stats_query)
return NOERROR;

// flush_data(0,NULL);

// send request to arc_ccd_rt side for ps data
    send_rt_command(ARC_CCD_RT_REQ_VD_STATS,0);

// Wait for rt side to put ps data into DATA_OUT_FIFO
    start=time(NULL);

//wait for the readout time (MICROSEC_PER_PIXEL*pixels)
// + exposure time in ms (arg must be in microsec so * 1000)
//      usleep((bbdata.dx * bbdata.dy * MICROSEC_PER_PIXEL
//      + (exposure * 1000) );

    while( !char_waiting(fd_data_out) )
    {
        if( time(NULL) > start + 1 )
        {
            return error(ERROR,"Timed out waiting for vs_dith data.");
        }
        usleep(FIFO_POLLING_INTERVAL);
    }

// read ps data from DATA_OUT_FIFO
    if (!read_arc_data((char *)&vd_par,sizeof(vd_par) ) )
    {
        return error(ERROR,
"Failed to get vis_dither data.");
    }

return NOERROR;
} // get_vd_stats()

/*****
/* req_vis_dither_goto() */
/*
/* Sends request to dither mirror driver, asking for a given dac */
/* on the dither mirror. */
*****/
int req_vis_dither_goto(int target)
{
// turn off interrupts
send_rt_command(RTCCD_INTS_OFF,0);

// put requested dac into FIFO

if( write_rt_data(&target,sizeof(target)) != sizeof(target) )
return -2;

//send request command to vis_dither module

```

```

if( !send_rt_command(VIS_DITHER_GOTO,0) )
return -1;

//turn interrupts back on
send_rt_command(RTCCD_INTS_ON,0);

return 0;

} // req_vis_dither_goto()

/*****
/* req_vis_dither_move_rel() */
/* */
/* Sends request to dither mirror driver, asking for a change in the */
/* dac position of the dither mirror. */
*****/
int req_vis_dither_move_rel(int change)
{
// turn off interrupts
send_rt_command(RTCCD_INTS_OFF,0);

// put requested dac into FIFO

if( write_rt_data(&change,sizeof(change)) != sizeof(change) )
return -2;

//send request command to vis_dither module

if( !send_rt_command(VIS_DITHER_MOVE_RELATIVE,0) )
return -1;

//turn interrupts back on
send_rt_command(RTCCD_INTS_ON,0);

return 0;

} // req_vis_dither_move_rel()

/*****
/* set_ps_servo_target() */
/* */
/* Sends request to arc_ccd_rt module for new ps_servo target fringe */
/* frequency. */
*****/
int set_ps_servo_target(float target)
{
// turn off interrupts
send_rt_command(RTCCD_INTS_OFF,0);

// put requested target into FIFO

if( write_rt_data(&target,sizeof(target)) != sizeof(target) )
return -2;

//send request command to arc_ccd_rt module

if( !send_rt_command(ARC_CCD_RT_NEW_TARGET_FREQ,0) )
return -1;

//turn interrupts back on
send_rt_command(RTCCD_INTS_ON,0);

return 0;

} // set_ps_servo_target()

/*****
/* tweak_ps_servo_pid() */
/* */
/* Passes values to the arc_ccd_rt module to change the ps_servo params:*/
/* P,I,D,Gain. */
*****/
int tweak_ps_servo_pid(int param_id, float value)
{
// turn off interrupts
send_rt_command(RTCCD_INTS_OFF,0);

// put requested value into FIFO

```

```

        if( write_rt_data(&value,sizeof(value)) != sizeof(value) )
            return -2;

        //send request command to arc_ccd_rt module

        if( !send_rt_command(ARC_CCD_RT_TWEAK_PID,(unsigned char)param_id) )
            return -1;

    // update user side copy of altered parameter

    switch (param_id)
    {
    case TWEAK_PS_SERVO_P:
        ps_servo_p = value;
        break;
    case TWEAK_PS_SERVO_I:
        ps_servo_i = value;
        break;
    case TWEAK_PS_SERVO_D:
        ps_servo_d = value;
        break;
    case TWEAK_PS_SERVO_GAIN:
        ps_servo_gain = value;
        break;
    default:
        return error(ERROR,"Invalid parameter chosen. Possible inconsistency between user and rt parameter values!");
    }

    //turn interrupts back on
    send_rt_command(RTCCD_INTS_ON,0);

    return 0;
} // tweak_ps_servo_pid()

/*****
/* toggle_ps_servo() */
/*
/* Turn fringe tracking ps_servo on and off. Called by */
/* call_toggle_ps_servo() */
*****/
int toggle_ps_servo(bool value)
{
    int err = 0;

    // if recording data, use set_rt_flag_blind, which doesn't use the fifo
    if(recording)
        err = set_rt_flag_blind(FLAG_ID_PS_SERVO_ON,value);
    else
        err = set_rt_flag(FLAG_ID_PS_SERVO_ON,value);

    if( err != NOERROR)
        return err;

    // if successful, let the user side copy show the change

    else
        ps_servo_on = value;

    return err;
} // call_toggle_ps_servo()

/*****
/* toggle_ps_servo_squarewave() */
/*
/* Turn squarewave on/off in the ps_servo. For tuning. Called by */
/* call_toggle_ps_servo_squarewave(). */
*****/
int toggle_ps_servo_squarewave(bool value)
{
    int err = 0;

    // if recording data, use set_rt_flag_blind, which doesn't use the fifo

```

```

if(recording)

err = set_rt_flag_blind(FLAG_ID_SQUAREWAVE_ON,value);

else

err = set_rt_flag(FLAG_ID_SQUAREWAVE_ON,value);

    if( err != NOERROR)
        return err;

    // if successful, let the user side copy show the change

    else
        ps_servo_squarewave_on = value;

    return err;

} // call_toggle_ps_servo_squarewave()

/*****
/* setup_dither_scan() */
/* */
/* Sends dither scan parameters to the rt side. Called by */
/* call_setup_dither_scan(). */
*****/
int setup_dither_scan(ARCCCD_DITH_SCAN scan)
{
// turn off interrupts
// send_rt_command(RTCCD_INTS_OFF,0);

// put scan parameter structure into FIFO

if( write_rt_data(&scan,sizeof(scan)) != sizeof(scan) )
return -2;

//send setup command to arc_ccd_rt module

if( !send_rt_command(ARC_CCD_RT_SETUP_DITHER_SCAN,0) )
return -1;

//turn interrupts back on
// send_rt_command(RTCCD_INTS_ON,0);

return 0;

} /* setup_dither_scan()*/

/*****
/* activate_dither_scan() */
/* */
/* Tells rt side to start/stop/restart the current dither scan. */
/* Takes an unsigned char as its argument. */
/* arg = 1: turn scan on */
/* arg = 2: reset scan to beginning and turn on */
/* else: turn scan off. Can be resumed with arg = 1 later. */
*****/
int activate_dither_scan(unsigned char arg)
{

//send setup command to arc_ccd_rt module

if( !send_rt_command(ARC_CCD_RT_ACTIVATE_DITHER_SCAN,arg) )
return -1;

return 0;

} /* activate_dither_scan()*/

```

B.3.6 arc_ccd_control.c

```

/*****
/* arc_ccd_control.c */
/*
/* Functions to read and do stuff with the Arc_ccd camera. */
*/

```

```

/*****
/*
/*          CHARA ARRAY USER INTERFACE */
/*          Based on the SUSI User Interface */
/* In turn based on the CHIP User interface */
/*
/*          Center for High Angular Resolution Astronomy
/*          Mount Wilson Observatory, CA 91001, USA */
/*
/* Telephone: 1-626-796-5405
/* Fax       : 1-626-796-6717
/* email     : theo@chara.gsu.edu
/* WWW       : http://www.chara.gsu.edu
/*
/* (C) This source code and its associated executable
/* program(s) are copyright.
/*
/*****
/*
/* Author : Theo ten Brummelaar
/* Date   : Feb 2000
/* Modified from Dalsa to Arc_ccd by Chad Ogden, Oct 2002
/*****/

extern "C" {
#include <chara.h>
#include "arc_ccd.h"
}

//define DEBUG_SAVE_SPECTRUM_MESSAGES

//define DEBUG_SAVE_SPECTRUM_FAKE_DATA

#define SAVE_SPECTRUM_OUTPUT_BUFFER_ENTRIES 1000
#define SAVE_SPECTRUM_DATA_TIMEOUT 2

// uncomment this to force save specrum to quit after the buffer fills once.
// This is a kludge to avoid crashing while the buffer writes to the file.

//define SAVE_SPECTRUM_1_BUFFER_ONLY

#define SAVE_SPECTRUM_LOOPS_PER_UPDATE 4
// the number of possible quantities to display in save_spectrum()
// should be same as the index of the last scp_id[index]
// defined in that function

#define NUM_OF_SAVE_SPEC_SCOPE_OPTIONS 13

// which spectral channel is usually displayed in the save_spectrum scopes?
// try the "middle one.

#define DEFAULT_SPEC_DISP_CHANNEL1 SPEC_SIZE/2-1
#define DEFAULT_SPEC_DISP_CHANNEL2 SPEC_SIZE/2

//normalization factors fpr proper display of variables on scopes.
// normalized to = 1

#define BIN_NORMALIZATION PS_SIZE
#define POWER_NORMALIZATION 5000.0
#define SNR_NORMALIZATION 30.0
#define SIGMA_NORMALIZATION 10.0
#define HEIGHT_NORMALIZATION 1000.0
#define WIDTH_NORMALIZATION 30.0
#define SKEW_NORMALIZATION 10.0
#define SKEW_OFFSET 0.5
#define ERRSIGNAL_NORMALIZATION (float)2*(PS_SIZE-1)
#define ERRSIGNAL_OFFSET 0.5
#define TRACKSIGNAL_NORMALIZATION (float)2*(PS_SIZE-1)
#define TRACKSIGNAL_OFFSET 0.5
#define DITH_DAC_NORMALIZATION (float)(RTDAC_UPPER_LIMIT-RTDAC_LOWER_LIMIT)
#define PHASE_NORMALIZATION (1/180.0)
#define PHASE_OFFSET 0.5
#define FRINGE_OFFSET 0.5

#ifdef SPEAK_TO_OPPLER

#define LARGE_OPLE_INCREMENT 75
#define SMALL_OPLE_INCREMENT 10

#endif

```

```

char    *default_display_machine = NULL;
static XImage *bbImage = NULL;
static Window bbWindow; /* Window for displaying things */
static time_t bb_start_time;
static int last_bb_dx = 1;
static int last_bb_dy = 1;
static int last_bb_quadrant = 0;
bool single_quad_mode = TRUE;
static float bb_frames=0;
bool bb_movie_running = FALSE;
static int pixmult = 1;
static int last_pixmult = 1;
char *bias_types[3] = {"OFF", "ON ", NULL};
static long int min_frame, max_frame;
static int min_x, max_x;
static int min_y, max_y;
static float picture_min = 0.0;
static float picture_max = 4000.0;
static char picture_display_type = LIN;

// user side copies of arc_ccd_rt flags
bool use_fft = TRUE;
bool use_lpt_toggle = FALSE;
bool ps_servo_on = FALSE;
bool ps_servo_squarewave_on = FALSE;
bool recording = FALSE;

float ps_servo_target = PS_SERVO_TARGET_FREQ;
float ps_servo_p = PS_SERVO_P;
float ps_servo_i = PS_SERVO_I;
float ps_servo_d = PS_SERVO_D;
float ps_servo_gain = PS_SERVO_GAIN;

float ps_servo_tweak_increment = 0.1;

int spec_shut = -1;

char    *dith_scan_commands[4] = {"STOP", "START", "RESET", NULL};
char *picture_display_types[3] = {"LIN", "LOG", "EXP"};

/*****
/* call_set_exposure() */
/* */
/* Tries to set exposure on the Arc_ccd camera and get a single frame. */
*****/

int call_set_exposure(int argc, char **argv)
{
    char s[81];

    /* Now look at the arguments */

    if (argc > 1)
    {
        sscanf(argv[1], "%d", &exposure);
    }
    else
    {
        sprintf(s, "%5d", exposure);
        if (quick_edit("Exposure in milliseconds",
                      s, s, NULL, INTEGER) == KEY_ESC) return NOERROR;

        sscanf(s, "%d", &exposure);
    }
    clean_command_line();
}

/* Now set it up */

if (set_exposure(exposure) != 0)
{
    return error(ERROR, "Failed to set exposure.");
}

return NOERROR;
} /* call_set_exposure() */

```

```

/*****
/* clear_window() */
/* */
/* Cleans up X windows stuff. */
*****/

int clear_window(int argc, char **argv)
{

    if (bb_movie_running)
        return error(ERROR,"Can not clear while movie running.");

    if (bbImage != NULL)
    {
        XDestroyImage(bbImage);
        bbImage = NULL;
        XDestroyWindow(theDisplay, bbWindow);
    }

    quitX();
    return NOERROR;

} /* clear_window() */

/*****
/* arc_ccd_close() */
/* */
/* Called by UI when closing. */
*****/

void arc_ccd_close(void)
{
    req_vis_dither_goto(0);
    clear_window(0,NULL);
    close_driver();
    close_server_socket(shut_server);

} /* arc_ccd_close() */

/*****
/* arc_ccd_open() */
/* */
/* Called by UI when opening. */
*****/

void arc_ccd_open(void)
{
    clear_window(0,NULL);
    // open_driver();

    /* open connection with shutter server */

    shut_server = open_shutter_server();
    switch(shut_server)
    {
        case -1: error(FATAL,"Failed to open SHUTTER port.\n");
        break;

        case -2: error(FATAL,
            "Failed to add standard SHUTTER jobs.\n");
        break;

        case -3: error(FATAL,"Failed to add local SHUTTER jobs.\n");
        break;

        case -4: error(FATAL,
            "Failed to send SHUTTER name request.\n");
        break;

        case -5: error(FATAL,"SHUTTER server not responding.\n");
        break;

        default: break;
    }

    // get shutter id for spectrograph shutter

    if( (spec_shut = shutter_id("SPEC")) < 0 )
        error(ERROR, "Failed to get shutter id for SPEC");

```

```

background_start(0, NULL);

} /* arc_ccd_open() */

// no longer a valid function because the arc_ccd hardware requires that
// any bounding box be symmetrical in all 4 quadrants.

#ifdef USE_RTCCD_API

/*****
/* call_set_bb() */
/* */
/* Tries to set bounding box on the Arc_ccd camera. */
*****/

int call_set_bb(int argc, char **argv)
{
    char s[81];
    int ulx,uly,lrx,lry;
    int x,y,dx,dy;
    int rows;
    uint16_t *buffer;
    time_t start;

    if (bb_movie_running)
    {
        bb_background_movie(0,NULL);
        error(MESSAGE,"Background movie stopped.");
    }

    /* If there are no arguments, maybe we can use the mouse */

    if (argc <= 1 && active_socket == -1 && bbImage != NULL)
    {
        // pixmult = 1;

        //reset bb to full ccd
        #ifdef USE_CCD_OVERSCAN
        if (set_bb(0, 0, CCD_X_PIXELS, CCD_Y_PIXELS) < 0)
        #else
        if (set_bb(USEFUL_X, USEFUL_Y, USEFUL_DX, USEFUL_DY) < 0)
        #endif
        {
            return error(ERROR,"Failed to reset bounding box.");
        }

        /* Get the frame */

        if ((buffer = get_bb()) == NULL)
        {
            return error(ERROR,"Failed to get bb data.");
        }

        display_bb(NULL, buffer);

        #ifdef USE_CCD_OVERSCAN
        rows = CCD_Y_PIXELS;
        #else
        rows = USEFUL_DY;
        #endif

        message(system_window,
            "Left click mouse on upper left corner.");
        if (!get_mouse(bbWindow, pixmult, &ulx, &uly))
        {
            werase(system_window);
            wrefresh(system_window);
            return error(ERROR,"Failed to get upper left corner.");
        }

        /* Wait for a bit */

        working();
        start = time(NULL);
        while(time(NULL) == start) background();
        not_working();

    do
    {

```



```

message(system_window,
"Left click mouse on lower right corner.");
if (!get_mouse(bbWindow, pixmult, &lrx, &lry))
{
werase(system_window);
wrefresh(system_window);
return error(ERROR,
"Failed to get lower right corner.");
}

} while(lrx == ulx && lry == uly);

/*
 * The get_mouse function does things for NRC arrays
 * and reverses the sense of y to make zero at the bottom.
 * We need to undo this for this routine.
 */

ulx -= 1;
lrx -= 1;
uly -= 1;
lry -= 1;

x = ulx;
y = lry;
dx = lrx-x+1;
dy = uly-y+1;

#ifdef USE_CCD_OVERSCAN

// This is for when the default bb is not the whole chip.
// the window coords are 0 when at lower left corner of bb

x += (USEFUL_X);
y += (USEFUL_Y);
ulx += (USEFUL_X);
uly += (USEFUL_Y);
lrx += (USEFUL_X);
lry += (USEFUL_Y);
#endif

//these two lines flip the y-axis.
//uly = rows - uly;
//lry = rows - lry;

message(system_window,
"bb coords: Origin:%d,%d Size: %d,%d",x,y,dx,dy);
}
else
{
if (argc > 1)
{
sscanf(argv[1], "%d", &x);
}
else
{
sprintf(s, "%5d", bbdata.x);
if (quick_edit("Lower left corner X",
s, s, NULL, INTEGER) == KEY_ESC) return NOERROR;

sscanf(s, "%d", &x);
clean_command_line();
}

if (argc > 2)
{
sscanf(argv[2], "%d", &y);
}
else
{
sprintf(s, "%5d", bbdata.y);
if (quick_edit("Lower left corner Y",
s, s, NULL, INTEGER) == KEY_ESC) return NOERROR;

sscanf(s, "%d", &y);
clean_command_line();
}

if (argc > 3)
{

```

```

    sscanf(argv[3], "%d", &dx);
}
else
{
    sprintf(s, "%5d", bbdata.dx);
    if (quick_edit("Box width (dx)",
    s, s, NULL, INTEGER) == KEY_ESC) return NOERROR;

    sscanf(s, "%d", &dx);
    clean_command_line();
}

if (argc > 4)
{
    sscanf(argv[4], "%d", &dy);
}
else
{
    sprintf(s, "%5d", bbdata.dy);
    if (quick_edit("Box height (dy)",
    s, s, NULL, INTEGER) == KEY_ESC) return NOERROR;

    sscanf(s, "%d", &dy);
    clean_command_line();
}

ulx = x;
lry = y;
lrx = x + dx - 1;
uly = y + dy - 1;
}

if (set_bb(x, y, dx, dy) < 0)
{
    return error(ERROR, "Failed to set bounding box.");
}

message(system_window, "New Bounding box: Origin: (%d,%d) Size: (%d,%d)",
x, y, dx, dy);

return NOERROR;
} /* call_set_bb() */

#else // this is the current version of the function which uses the rtccd api.
/*****
/* call_set_bb() */
/* */
/* Tries to set bounding box on the Arc_ccd camera. */
*****/

int call_set_bb(int argc, char **argv)
{
    char s[81];
    int ulx, uly, lrx, lry;
    int x, y, dx, dy;
    int rows;
    uint16_t *buffer;
    bool loop = TRUE;
    int c;

    // set default values

    x=y=ulx=uly=lrx=lry=0;
    dx = dy = 1;

    clean_command_line();

    if (bb_movie_running)
    {
        bb_background_movie(0, NULL);
        error(MESSAGE, "Background movie stopped.");
    }

    if (single_quad_mode)
        return set_single_quad_bb(argc, argv);

    /* If there are no arguments, use graphic frames to select bb*/

```

```

if (argc <= 1 && active_socket == -1 && bbImage != NULL)
{
    // pixmult = 1;

    //reset bb to full ccd
    bbquadrant = 0;

#ifdef USE_CCD_OVERSCAN
    if (set_bb(0, 0, CCD_X_PIXELS/2, CCD_Y_PIXELS/2) < 0)
    #else
    if (set_bb(USEFUL_X, USEFUL_Y,
    USEFUL_DX/2, USEFUL_DY/2) < 0)
    #endif
    {
        return error(ERROR,
        "Failed to reset bounding box.");
    }

    /* Get the frame */

    if ((buffer = get_bb()) == NULL)
    {
        return error(ERROR, "Failed to get bb data.");
    }

#ifdef USE_CCD_OVERSCAN
    rows = CCD_Y_PIXELS/2;
    #else
    rows = USEFUL_DY/2;
    #endif

    message(system_window,
             "X,x,Y,y to move bb outer corner. <Space> to accept.");

    // this loop moves the starting corner of the overlays around
    while(loop)
    {
        // display the chip with overlay frames

        display_bb_overlay(NULL, buffer, ulx, uly, lrx, lry);

        // get a character from the keyboard
        c = getchar();

        switch(c)
        {
            case 'X':
#ifdef USE_CCD_OVERSCAN
                if(x < (CCD_X_PIXELS/2)-1)
            #else
                if(x < (USEFUL_DX/2)-1)
            #endif
            {
                x++;
                ulx=x;
                lrx= x+dx-1;
            }
            break;
            case 'x':
                if(x > 0)
                {
                    x--;
                    ulx=x;
                    lrx=x+dx-1;
                }
                break;
            case 'Y':
#ifdef USE_CCD_OVERSCAN
                if(y < (CCD_Y_PIXELS/2)-1)
            #else
                if(y < (USEFUL_DY/2)-1)
            #endif
            {
                y++;
                uly=y+dy-1;

```

```

lry=y;
}
break;
case 'y':
if(y > 0)
{
y--;
uly=y+dy-1;
lry=y;
}
break;
case KEY_SPACE:
case KEY_ENTER:
case ']'':
loop = FALSE;
break;
case KEY_ESC:
case 'q':
free(buffer);
return -1;
break;
default:
break;
}
}

// reset the loop indicator
loop = TRUE;

message(system_window,
        "X,x,Y,y to move the lower left quad bb lower right corner. <Space> to accept.");

// this loop moves the final corner of the overlays around
while(loop)
{
// display the chip with overlay frames

display_bb_overlay(NULL, buffer,ulx,uly,lrx,lry);

// get a character from the keyboard
c = getchar();

switch(c)
{
case 'X':
#ifdef USE_CCD_OVERSCAN
if(x+dx-1 < (CCD_X_PIXELS/2)-1)
#else
if(x+dx-1 < (USEFUL_DX/2)-1)
#endif
{
dx++;
lrx=x+dx-1;
}
break;
case 'x':
if(dx > 1)
{
dx--;
lrx=x+dx-1;
}
break;
case 'Y':
if(y+dy-1 < (CCD_Y_PIXELS/2)-1)
{
dy++;
uly=y+dy-1;
}
break;
case 'y':
if(dy > 1)
{
dy--;
uly=y+dy-1;
}
break;
case KEY_SPACE:
case KEY_ENTER:
case ']'':

```

```

loop = FALSE;
break;
case KEY_ESC:
case 'q':
free(buffer);
return -1;
break;
default:
break;
}
}
free(buffer);

#ifdef USE_CCD_OVERSCAN

// This is for when the default bb is not the whole chip.
// the window coords are 0 when at lower left corner of bb

if(!single_quad_mode)
{
x += (USEFUL_X);
y += (USEFUL_Y);
ulx += (USEFUL_X);
uly += (USEFUL_Y);
lrx += (USEFUL_X);
lry += (USEFUL_Y);
}
#endif

//these two lines flip the y-axis.
//uly = rows - uly;
//lry = rows - lry;

message(system_window,
"bb coords: origin:%d,%d; size: %d,%d",x,y,dx,dy);
}
else
{
if (argc > 1)
{
sscanf(argv[1], "%d", &x);
}
else
{
sprintf(s, "%5d", bbdata.x);
if (quick_edit("Outer Corner x",
s, s, NULL, INTEGER) == KEY_ESC) return NOERROR;

sscanf(s, "%d", &x);
clean_command_line();
}

if (argc > 2)
{
sscanf(argv[2], "%d", &y);
}
else
{
sprintf(s, "%5d", bbdata.y);
if (quick_edit("Outer Corner y",
s, s, NULL, INTEGER) == KEY_ESC) return NOERROR;

sscanf(s, "%d", &y);
clean_command_line();
}

if (argc > 3)
{
sscanf(argv[3], "%d", &dx);
}
else
{
sprintf(s, "%5d", bbdata.dx);
if (quick_edit("Box width (dx)",
s, s, NULL, INTEGER) == KEY_ESC) return NOERROR;

sscanf(s, "%d", &dx);
clean_command_line();
}
}

```

```

if (argc > 4)
{
    sscanf(argv[4], "%d", &dy);
}
else
{
    sprintf(s, "%5d", bbdata.dy);
    if (quick_edit("Box height (dy)",
s,s,NULL,INTEGER) == KEY_ESC) return NOERROR;

    sscanf(s, "%d", &dy);
    clean_command_line();
}

}

// get the quadrant
sprintf(s, "%d", bbquadrant);
if(quick_edit("Quadrant (0=all,1-4 clockwise from lower left):",
s,s,NULL,INTEGER) == KEY_ESC) return NOERROR;
sscanf(s, "%d", &bbquadrant);
clean_command_line();

// check quadrant limits
if(bbquadrant < 0) bbquadrant = 0;
if(bbquadrant > 4) bbquadrant = 4;

// try to set the bounding box
if (set_bb(x,y,dx,dy) < 0)
{
    return error(ERROR,"Failed to set bounding box.");
}

message(system_window,"New Bounding box: Origin: (%d,%d) Size: (%d,%d)",
x,y,dx,dy);

return NOERROR;

} /* call_set_bb() */

/*****
/* set_single_quad_bb() */
/* */
/* Special case of set_bb called by call_set_bb() while single_quad_mode*/
/* is TRUE. Necessary because only the upper left quadrant of the */
/* detector is available in this mode. */
*****/

int set_single_quad_bb(int argc, char **argv)
{
    char    s[81];
    int     ulx,uly,lrx,lry,x,y,dx,dy;
    int     rows;
    uint16_t *buffer;
    bool loop = TRUE;
    int c;

    // set default values
    x=y=dx=dy=ulx=uly=lrx=lry=0;

    clean_command_line();

    if (bb_movie_running)
    {
        bb_background_movie(0,NULL);
        error(MESSAGE,"Background movie stopped.");
    }

    /* If there are no arguments, use graphic frames to select bb*/

    if (argc <= 1 && active_socket == -1 && bbImage != NULL)
    {
        //
        pixmult = 1;

        //reset bb to ul quadrant

```

```

bbquadrant = 2;

// #ifdef USE_CCD_OVERSCAN
// if (set_bb(0, 0, CCD_X_PIXELS/2, CCD_Y_PIXELS/2) < 0)
// #else
// if (set_bb(USEFUL_X, USEFUL_Y,
// USEFUL_DX/2, USEFUL_DY/2) < 0)
// #endif
{
    return error(ERROR,
        "Failed to reset bounding box.");
}

/* Get the frame */
if ((buffer = get_bb()) == NULL)
{
    return error(ERROR, "Failed to get bb data.");
}

// #ifdef USE_CCD_OVERSCAN
// rows = CCD_Y_PIXELS/2;
// #else
// rows = USEFUL_DY/2;
// #endif

// the outer corner of the bb is fixed at the outer
// corner of the chip

// these coords are for the master quadrant (11) but
// it is the ul quad which is actually read out
// in single quad mode.

x = 0;
y = 0;
dx = 20;
dy = 10;
ulx = x;
lry = y;
uly = y + dy - 1;
lrx = x + dx - 1;

// reset the loop indicator

loop = TRUE;

message(system_window,
    "X,x,Y,y to move the bb inside corner. <Space> to accept.");

// this loop moves the final corner of the overlays around
while(loop)
{
    // display the chip with overlay frames
    display_bb_overlay(NULL, buffer, ulx, uly, lrx, lry);

    // get a character from the keyboard
    c = getchar();

    switch(c)
    {
        case 'X':
            #ifdef USE_CCD_OVERSCAN
            if (x+dx-1 < (CCD_X_PIXELS/2)-1)
            #else
            if (x+dx+1 < USEFUL_X+(USEFUL_DX/2)-1)
            #endif
            {
                dx++;
            }
            break;
        case 'x':
            if (dx > 1)
            {

```

```

        dx--;
        lrx = x + dx - 1;
        }
        break;
        case 'Y':
            if(y+dy-1 < (CCD_Y_PIXELS/2)-1)
            {
                dy++;
            }
            break;
        case 'y':
            if(dy > 1)
            {
                dy--;
            }
            break;
        case KEY_SPACE:
        case KEY_ENTER:
            case ']':
                loop = FALSE;
                break;
            case KEY_ESC:
            case 'q':
                free(buffer);
                return -1;
                break;
            default:
                break;
        }
    }
    free(buffer);

#ifdef USE_CCD_OVERSCAN
    // This is for when the default bb is not the whole chip.
    // the window coords are 0 when at lower left corner of bb

    //          ulx += (USEFUL_X);
    //          uly += (USEFUL_Y);
    //          lrx += (USEFUL_X);
    //          lry += (USEFUL_Y);
//endif

    //these two lines flip the y-axis.
    //uly = rows - uly;
    //lry = rows - lry;

    message(system_window,
            "bb coords: origin %d,%d, size %d,%d",x,y,dx,dy);
}
else
{
    // set for ul quadrant
    bbquadrant = 2;

    x = 0;
    y = 0;

    if (argc > 1)
    {
        sscanf(argv[1], "%d", &dx);
    }
    else
    {
        sprintf(s, "%5d", bbdata.dx);
        if (quick_edit("Bounding box dx",
            s, s, NULL, INTEGER) == KEY_ESC) return NOERROR;

        sscanf(s, "%d", &dx);
        clean_command_line();
    }

    if (argc > 2)
    {
        sscanf(argv[2], "%d", &dy);
    }
    else
    {

```



```

        sprintf(s,"%5d",bbdata.dy);
        if (quick_edit("Bounding box dy",
            s,s,NULL,INTEGER) == KEY_ESC) return NOERROR;

        sscanf(s,"%d",&dy);
        clean_command_line();
    }

    ulx = x;
    lry = y;
    uly = y + dy - 1;
    lrx = x + dx - 1;

    }

    // try to set the bounding box

    if (set_bb(x,y,dx,dy) < 0)
    {
        return error(ERROR,"Failed to set bounding box.");
    }

    message(system_window,"New Bounding box: origin(%d,%d), size (%d,%d)",
        x,y,dx,dy);

    return NOERROR;
}

#endif

// this function is obsolete, ccd quadrants make it too much of a pain to use.

#ifndef USE_RTCCD_API

/*****
/* call_set_bb_row_64() */
/* */
/* Calls set_bb() to declare a 1 row bounding box, 64 pixels wide, */
/* centered on the chip. For picking out spectra without wasting extra */
/* readout pixels. Sets the power spectrum row as well. */
*****/

int call_set_bb_row_64(int argc, char **argv)
{
    char s[81];
    int ulx,uly;
    int rows;
    uint16_t *buffer;

    if (bb_movie_running)
    {
        bb_background_movie(0,NULL);
        error(MESSAGE,"Background movie stopped.");
    }

    /* If there are no arguments, maybe we can use the mouse */

    if (argc <= 1 && active_socket == -1 && bbImage != NULL)
    {
        // pixmult = 1;

        //reset bb to full ccd
        #ifdef USE_CCD_OVERSCAN
        if (set_bb(0, CCD_Y_PIXELS-1, CCD_X_PIXELS-1, 0) < 0)
        #else
        if (set_bb(USEFUL_ULX, USEFUL_ULY, USEFUL_LRX, USEFUL_LRY) < 0)
        #endif
        {
            return error(ERROR,"Failed to reset bounding box.");
        }

        /* Get the frame */

        if ((buffer = get_bb()) == NULL)
        {
            return error(ERROR,"Failed to get bb data.");
        }
    }
}

```

```

}

display_bb(NULL, buffer);

#ifdef USE_CCD_OVERSCAN
rows = CCD_Y_PIXELS;
#else
rows = USEFUL_DY;
#endif

message(system_window,
        "Left click mouse on desired row.");
if (!get_mouse(bbWindow, pixmult, &ulx, &uly))
{
    werase(system_window);
    wrefresh(system_window);
    return error(ERROR, "Failed to get upper left corner.");
}

/*
 * The get_mouse function does things for NRC arrays
 * and reverses the sense of y to make zero at the bottom.
 * We need to undo this for this routine.
 */

ulx -= 1;
uly -= 1;

// Set the bb at 64 pix wide, centered, at row uly.

#ifdef USE_CCD_OVERSCAN

// This is for when the default bb is not the whole chip.
// the window coords are 0 when at lower left corner of bb
ulx += (USEFUL_X + 1);
uly += (USEFUL_Y);
#endif

//these two lines flip the y-axis.
//uly = rows - uly;
//lry = rows - lry;

message(system_window,
        "bb coords: row %d from bottom, 64 pix wide, centered."
        , uly);
}
else
{
    if (argc > 1)
    {
        sscanf(argv[1], "%d", &uly);
    }
    else
    {
        sprintf(s, "%5d", bbdata.y);
        if (quick_edit("Row of spectrum (from bottom)",
            s, s, NULL, INTEGER) == KEY_ESC) return NOERROR;

        sscanf(s, "%d", &uly);
        clean_command_line();
    }
}

if (set_bb(SINGLE_ROW_BB_ULX, uly, SINGLE_ROW_BB_LRX, uly) < 0)
{
    return error(ERROR, "Failed to set bounding box.");
}

message(system_window,
        "New Bounding box: row %d from bottom, 64 pix wide, centered."
        , uly);

// now set the row of the spectrum while you're at it,
// because there's only 1 row in the bb: row 0.

```

```

return set_spec_row(0);
} /* call_set_bb_row_64() */

#endif

// this function is outdated. the new one must use the rtccd api.

#ifndef USE_RTCCD_API

/*****
/* call_set_bb_row_16() */
/* */
/* Calls set_bb() to declare a 1 row bounding box, 16 pixels wide. */
/* For picking out spectra without wasting extra */
/* readout pixels. Sets the power spectrum row as well. */
/* Picks the 16 pixels to the right of the one which the user selects. */
*****/

int call_set_bb_row_16(int argc, char **argv)
{
    char s[81];
    int ulx,uly;
    int rows;
    uint16_t *buffer;

    if (bb_movie_running)
    {
        bb_background_movie(0,NULL);
        error(MESSAGE,"Background movie stopped.");
    }

    /* If there are no arguments, maybe we can use the mouse */

    if (argc <= 1 && active_socket == -1 && bbImage != NULL)
    {
        // pixmult = 1;

        //reset bb to full ccd
        #ifdef USE_CCD_OVERSCAN
        if (set_bb(0, CCD_Y_PIXELS-1, CCD_X_PIXELS-1, 0) < 0)
        #else
        if (set_bb(USEFUL_ULX, USEFUL_ULY, USEFUL_LRX, USEFUL_LRY) < 0)
        #endif
        {
            return error(ERROR,"Failed to reset bounding box.");
        }

        /* Get the frame */

        if ((buffer = get_bb()) == NULL)
        {
            return error(ERROR,"Failed to get bb data.");
        }

        display_bb(NULL, buffer);

        #ifdef USE_CCD_OVERSCAN
        rows = CCD_Y_PIXELS;
        #else
        rows = USEFUL_DY;
        #endif

        message(system_window,
                "Left click mouse on the left end of the desired spectrum.");
        if (!get_mouse(bbWindow, pixmult, &ulx, &uly))
        {
            werase(system_window);
            wrefresh(system_window);
            return error(ERROR,"Failed to get upper left corner.");
        }

        /*
        * The get_mouse function does things for NRC arrays
        * and reverses the sense of y to make zero at the bottom.
        * We need to undo this for this routine.
        */

        ulx -= 1;

```

```

        uly -= 1;

// Set the bb at 16 pix wide, to the right of the pixel clicked

#ifndef USE_CCD_OVERSCAN

// This is for when the default bb is not the whole chip.
// the window coords are 0 when at lower left corner of bb
ulx += (USEFUL_X + 1);
uly += (USEFUL_Y);
#endif

//these two lines flip the y-axis.
//uly = rows - uly;
//lry = rows - lry;

}
else
{
    if (argc > 3 || argc == 2)
    {
        error(ERROR, "Usage: function [row] [leftmost pixel]");
        return -1;
    }

    if (argc == 3)
    {
        sscanf(argv[1], "%d", &uly);
        sscanf(argv[2], "%d", &ulx);
    }
    else
    {
        sprintf(s, "%5d", bbdata.y);
        if (quick_edit("Row of spectrum (from bottom)",
            s, s, NULL, INTEGER) == KEY_ESC) return NOERROR;

        sscanf(s, "%d", &uly);
        clean_command_line();
    }
}

// check to make sure that the requested spectrum doesn't run
// off the chip

if(ulx+15 > USEFUL_LRX)
{
    ulx = USEFUL_LRX - 15;

    error(MESSAGE, "To keep the bounding box on the chip, it will start at pixel # %d ", ulx);
}

message(system_window,
    "bb coords: row %d from bottom, starting at pixel # %d."
    , uly, ulx);

if (set_bb(ulx, uly, ulx+15, uly) < 0)
{
    return error(ERROR, "Failed to set bounding box.");
}

message(system_window,
    "New Bounding box: row %d from bottom, starting at pixel # %d"
    , uly, ulx);

// now set the row of the spectrum while you're at it,
// because there's only 1 row in the bb: row 0.

return set_spec_row(0);
}/* call_set_bb_row_16() */

#else // this is the current version which uses the rtccd api.

/*****
/* call_set_bb_row_16() */
/*
/* Calls set_bb() to declare a 1 row bounding box, 16 pixels wide. */
/* For picking out spectra without wasting extra */

```

```

/* readout pixels. Sets the power spectrum row as well. */
/* Picks the 16 pixels to the right of the one which the user selects. */
/*****

int call_set_bb_row_16(int argc, char **argv)
{
    char s[81];
    int ulx,uly;
    int rows;
    uint16_t *buffer;
    bool loop=TRUE;
    int c;

    // set default values

    ulx=0;
    uly=0;

    if (bb_movie_running)
    {
        bb_background_movie(0,NULL);
        error(MESSAGE,"Background movie stopped.");
    }

    /* If there are no arguments, maybe we can use the mouse */

    if (argc <= 1 && active_socket == -1 && bbImage != NULL)
    {
        // pixmult = 1;

        //reset bb to full ccd
        bbquadrant = 0;

#ifdef USE_CCD_OVERSCAN
        if (set_bb(0, CCD_Y_PIXELS/2 -1, CCD_X_PIXELS/2 -1, 0) < 0)
        #else
        if (set_bb(USEFUL_X, USEFUL_DY/2 -1, USEFUL_X+USEFUL_DX/2 -1,
        USEFUL_Y) < 0)
        #endif
        {
            return error(ERROR,"Failed to reset bounding box.");
        }

        /* Get the frame */

        if ((buffer = get_bb()) == NULL)
        {
            return error(ERROR,"Failed to get bb data.");
        }

#ifdef USE_CCD_OVERSCAN
        rows = CCD_Y_PIXELS/2;
        #else
        rows = USEFUL_DY/2;
        #endif

        message(system_window,
            "X,x,Y,y to move the bounding box outer corner. <Enter> to accept.");

        // this loop moves the starting corner of the overlays around
        while(loop)
        {
            // display the chip with overlay frames

            display_bb_overlay(NULL, buffer,ulx,uly,ulx+15,uly);

            // get a character from the keyboard
            c = getchar();

            switch(c)
            {
                case 'X':
#ifdef USE_CCD_OVERSCAN
                    if(ulx < (CCD_X_PIXELS/2)-16)
                    #else
                    if(ulx < (USEFUL_DX/2)-16)
                    #endif
                    {
                        ulx++;

```

```

}
break;
case 'x':
if(ulx > 0)
{
ulx--;
}
break;
case 'Y':
#ifdef USE_CCD_OVERSCAN
if(uly < (CCD_Y_PIXELS/2)-1)
#else
if(ulx < (USEFUL_DY/2)-1)
#endif
{
uly++;
}
break;
case 'y':
if(uly > 0)
{
uly--;
}
break;
case KEY_ENTER:
case ']':
loop = FALSE;
break;
case KEY_ESC:
case 'q':
free(buffer);
return -1;
break;
default:
break;
}
}

free(buffer);

#ifdef USE_CCD_OVERSCAN

// This is for when the default bb is not the whole chip.
// the window coords are 0 when at lower left corner of bb
ulx += (USEFUL_X);
uly += (USEFUL_Y);
#endif

//these two lines flip the y-axis.
//uly = rows - uly;
//lry = rows - lry;

}
else
{
if (argc > 3 || argc == 2)
{
error(ERROR,"Usage: function [row] [leftmost pixel]");
return -1;
}

if (argc == 3)
{
sscanf(argv[1],"%d",&uly);
sscanf(argv[2],"%d",&ulx);
}
else
{
sprintf(s,"%5d",bbdata.y);
if (quick_edit("Row of spectrum (from bottom)",
s,s,NULL,INTEGER) == KEY_ESC) return NOERROR;

sscanf(s,"%d",&uly);
clean_command_line();

sprintf(s,"%5d",bbdata.x);
if (quick_edit("Beginning pixel (from outer edge)",

```

```

s,s,NULL,INTEGER) == KEY_ESC) return NOERROR;

sscanf(s,"%d",&ulx);
clean_command_line();
}

}

// get the quadrant
sprintf(s,"%d",bbquadrant);
if(quick_edit("Quadrant (0=all,1-4 clockwise from lower left):",
s,s,NULL,INTEGER) == KEY_ESC) return NOERROR;
sscanf(s,"%d",&bbquadrant);
clean_command_line();

// check quadrant limits
if(bbquadrant < 0) bbquadrant = 0;
if(bbquadrant > 4) bbquadrant = 4;

// check to make sure that the requested spectrum doesn't run
// off the chip
if(ulx+16 > CCD_X_PIXELS/2)
{
ulx = CCD_X_PIXELS/2 - 16;

error(MESSAGE,"To keep the bounding box in the quadrant, it will start at pixel # %d ",ulx);
}

message(system_window,
"bb coords: row %d from bottom, starting at pixel # %d."
, uly,ulx);

// Set the bb at 16 pix wide, to the right of the pixel clicked
// the set_bb function will force it to be 2 rows high, but row 0 should
// still be the one selected by the user.

if (set_bb(ulx,uly,ulx+15,uly) < 0)
{
return error(ERROR,"Failed to set bounding box.");
}

message(system_window,
"New Bounding box: row %d from bottom, starting at pixel # %d. Bottom row automatically selected for spectrum."
, bbdata.y,bbdata.x);

// now set the row of the spectrum while you're at it,
// because there's only 1 row in the bb: row 0.

return set_spec_row(0,spec_row_offset);
//>>>>>>>>>note the offset could cause trouble

}/* call_set_bb_row_16() */

# endif

/*****
/* clear_bb() */
/* */
/* Forces bounding box to be full chip. */
*****/

int clear_bb(int argc, char **argv)
{
if(single_quad_mode)
{
bbquadrant = 2;

if (set_bb(0, 0, CCD_X_PIXELS/2, CCD_Y_PIXELS/2) < 0)
{
return error(ERROR,"Failed to reset bounding box.");
}
}
else
{
bbquadrant = 0;
#ifdef USE_CCD_OVERSCAN
if (set_bb(0, 0, CCD_X_PIXELS/2, CCD_Y_PIXELS/2) < 0)

```

```

#else
if (set_bb(USEFUL_X, USEFUL_Y, USEFUL_DX/2,
USEFUL_DY/2) < 0)
#endif
{
return error(ERROR,"Failed to reset bounding box.");
}

}

return NOERROR;
} /* clear_bb() */

/*****
/* single_bb() */
/* */
/* Tries to arm the Arc_ccd camera and get a single bb frame. This will */
/* be displayed on the X terminal using display_bb. */
*****/

int single_bb(int argc, char **argv)
{
uint16_t *buffer; /* Data area for images */
char s[80];

no_socket();

    /* Go get the data from the command line */

if (argc > 1)
{
sscanf(argv[1], "%d", &pixmult);
}
else
{
if (bbImage == NULL)
{
sprintf(s, "%4d", 5);
if (quick_edit("Pixel multiplier",
s, s, NULL, INTEGER) == KEY_ESC) return NOERROR;

sscanf(s, "%d", &pixmult);
clean_command_line();
}
}

/* Are the parameters valid? */

if (exposure < 0)
{
return error(ERROR, "Set up the exposure time first.");
}

/* Get the frame */

if ((buffer = get_bb()) == NULL)
{
return error(ERROR, "Failed to get bb data.");
}

if (argc > 2)
{
return display_bb(argv[2], buffer);
}
else
{
return display_bb(NULL, buffer);
}

} /* single_bb() */

/*****
/* display_bb_overlay() */
/* */
/* Tries to display the bb on the X term. If no window exists it */
/* will create one for you. */
/* Also two globals, min_frame and max_frame, are set to the minimum */
/* and maximum of this frame. */
/* Overlays a 4 axisymmetric frames over the image. For use in */

```



```

/* selecting bounding boxes. Doesn't free the buffer when it's done */
/* so that the buffer can be redisplayed several times, then freed. */
/*****

int display_bb_overlay(char *display, uint16_t *buffer, int ulx, int uly,
int lrx, int lry)
{
int col, row;
int colors;
char display_name[81];
uint16_t *buf;
char *picture;
float **values;
int i,j;

/* Which machine do we display this on ? */

if ((bbImage == NULL) || !display_is_open)
{
if (display != NULL)
{
strcpy(display_name,display);
}
else
{
if (default_display_machine == NULL)
{
sprintf(display_name,"%s:0.0",
get_machine_name(display_name));
}
else
{
sprintf(display_name,"%s:0.0",
default_display_machine);
}
if (quick_edit("Display name",display_name,
display_name,NULL,STRING) == KEY_ESC) return NOERROR;
}
clean_command_line();
}

/* Setup the row and col variables */
if(bbquadrant < 1 || bbquadrant > 4)
{
col = bbdata.dx*2;
row = bbdata.dy*2;
}
else
{
col = bbdata.dx;
row = bbdata.dy;
}

/* If the box size has changed we gotta redo the window */

if (col != last_bb_dx || row != last_bb_dy ||
pixmult != last_pixmult || last_bb_quadrant != bbquadrant)
{
clear_window(0,NULL);
last_bb_dx = col;
last_bb_dy = row;
last_pixmult = pixmult;
last_bb_quadrant = bbquadrant;
}

/* Try and open the display */

if (bbImage != NULL)
{
XDestroyImage(bbImage);
bbImage = NULL;
}
else
{
if (!display_is_open)
{
if (strcmpi(display_name,"null") == 0)
{
if ((colors = initX(NULL)) == 0)
{

```

```

return error(ERROR,
"Failed to open display.");
}
}
else
{
if ((colors = initX(display_name)) == 0)
{
return error(ERROR,
"Failed to open display %s.",
display_name);
}
}
}
//changed this from ccd_size.  openWindow args: name,x,y,width,height
// bbWindow = openWindow("ARC CCD BB",60+col,50,
// col*pixmult,row*pixmult);

// Window moved to under where the ps are displayed,
// to keep out of the way of other program displays

bbWindow = openWindow("ARC CCD BB",theWidth-525,480,
col*pixmult,row*pixmult);
}

/* Allocate memory */

values = matrix(1,col,1,row);

/* Now fill in the values */

buf = buffer;
min_frame = 65535;
max_frame = 0;
max_x = max_y = 0;
min_x = min_y = 0;

//note that the rows are entered into values[] [] in reverse order.
//the y axis of the picture will be inverted.
//make_picture will invert the y-axis again.

for(j=row; j>=1; j--)
for(i=1; i<=col; i++)
{
if (*buf > max_frame)
{
max_frame = *buf;
max_x = i;
max_y = j;
}
if (*buf < min_frame)
{
min_frame = *buf;
min_x = i;
min_y = j;
}
values[i][j] = (*buf++);
}

/* Create a picture */

picture = make_scaled_picture(col,row,pixmult,values,
picture_display_type,&picture_min,&picture_max);

/* Overlay frames */

if(single_quad_mode)
{
// upper left frame

/* overlay_rectangle(col*pixmult,row*pixmult, picture,
* ulx*pixmult, ((row-lry)*pixmult)-1,
* lrx*pixmult, ((row-uly)*pixmult)-1,
* pixmult/2, NUM_COLORS-1);
*/

overlay_rectangle(col*pixmult,row*pixmult, picture,
ulx*pixmult, uly*pixmult,
lrx*pixmult, lry*pixmult,
pixmult/2, NUM_COLORS-1);

```

```

}

// all 4 quads displayed

else
{
    // lower left frame

    overlay_rectangle(col*pixmult,row*pixmult, picture, ulx*pixmult,
        uly*pixmult, lrx*pixmult, lry*pixmult,pixmult/2,NUM_COLORS-1);

    // upper left frame

    overlay_rectangle(col*pixmult,row*pixmult, picture, ulx*pixmult,
        ((row-lry)*pixmult)-1, lrx*pixmult, ((row-uly)*pixmult)-1,
        pixmult/2, NUM_COLORS-1);

    // lower right frame

    overlay_rectangle(col*pixmult, row*pixmult, picture,
        ((col-lrx)*pixmult)-1, uly*pixmult, ((col-ulx)*pixmult)-1,
        lry*pixmult, pixmult/2, NUM_COLORS-1);

    // upper right frame

    overlay_rectangle(col*pixmult,row*pixmult, picture,
        ((col-lrx)*pixmult)-1, ((row-lry)*pixmult)-1,
        ((col-ulx)*pixmult)-1, ((row-uly)*pixmult)-1, pixmult/2,
        NUM_COLORS-1);
}

/* Display it */

bbImage = XCreateImage(theDisplay,theVisual,
    theDepth,ZPixmap,0, picture,col*pixmult,row*pixmult,
    theBitmapPad, 0);
XPutImage(theDisplay,bbWindow,theGC,bbImage,0,0,0,0,
    col*pixmult,row*pixmult);
XMapWindow(theDisplay, bbWindow);
XFlush(theDisplay);

/* Clean up memory, leave buffer. Must be freed when done */

free_matrix(values,1,col,1,row);

/* And go */

return NOERROR;

} /* display_bb_overlay() */

/*****
/* display_bb() */
/* */
/* Tries to display the bb on the X term. If no window exists it */
/* will create one for you. */
/* Also two globals, min_frame and max_frame, are set to the minimum */
/* and maximum of this frame. */
*****/

int display_bb(char *display, uint16_t *buffer)
{
    int col, row;
    int colors;
    char display_name[81];
    uint16_t *buf;
    char *picture;
    float **values;
    int i,j;

    /* Which machine do we display this on ? */

    if ((bbImage == NULL) || !display_is_open)
    {
        if (display != NULL)
        {
            strcpy(display_name,display);
        }
    }

```

```

else
{
if (default_display_machine == NULL)
{
sprintf(display_name,"%s:0.0",
get_machine_name(display_name));
}
else
{
sprintf(display_name,"%s:0.0",
default_display_machine);
}
if (quick_edit("Display name",display_name,
display_name,NULL,STRING) == KEY_ESC) return NOERROR;
}
clean_command_line();
}

/* Setup the row and col variables */

if(bbquadrant == 0)
{
col = bbdata.dx*2;
row = bbdata.dy*2;
}
else
{
col = bbdata.dx;
row = bbdata.dy;
}

/* If the box size has changed we gotta redo the window */

if (col != last_bb_dx || row != last_bb_dy ||
pixmapult != last_pixmapult || last_bb_quadrant != bbquadrant)
{
clear_window(0,NULL);
last_bb_dx = col;
last_bb_dy = row;
last_pixmapult = pixmapult;
last_bb_quadrant = bbquadrant;
}

/* Try and open the display */

if (bbImage != NULL)
{
XDestroyImage(bbImage);
bbImage = NULL;
}
else
{
if (!display_is_open)
{
if (strcmpi(display_name,"null") == 0)
{
if ((colors = initX(NULL)) == 0)
{
return error(ERROR,
"Failed to open display.");
}
}
else
{
if ((colors = initX(display_name)) == 0)
{
return error(ERROR,
"Failed to open display %s.",
display_name);
}
}
}
//changed this from ccd_size. openWindow args: name,x,y,width,height
// bbWindow = openWindow("ARC CCD BB",60+col,50,
// col*pixmapult,row*pixmapult);

// Window moved to under where the ps are displayed,
// to keep out of the way of other program displays
bbWindow = openWindow("ARC CCD BB",theWidth-525,480,

```

```

col*pixmult,row*pixmult);
}

/* Allocate memory */
values = matrix(1,col,1,row);

/* Now fill in the values */

buf = buffer;
min_frame = 65535;
max_frame = 0;
max_x = max_y = 0;
min_x = min_y = 0;

//note that the rows are entered into values[][] in reverse order.
//the y axis of the picture will be inverted.
//make_picture will invert the y-axis again.

for(j=row; j>=1; j--)
for(i=1; i<=col; i++)
{
if (*buf > max_frame)
{
max_frame = *buf;
max_x = i;
max_y = j;
}
if (*buf < min_frame)
{
min_frame = *buf;
min_x = i;
min_y = j;
}
values[i][j] = (*buf++);
}

/* Create a picture */

picture = make_scaled_picture(col,row,pixmult,values,
picture_display_type,&picture_min,&picture_max);

/* Display it */

bbImage = XCreateImage(theDisplay,theVisual,
theDepth,ZPixmap,0, picture,col*pixmult,row*pixmult,
theBitmapPad, 0);
XPutImage(theDisplay,bbWindow,theGC,bbImage,0,0,0,0,
col*pixmult,row*pixmult);
XMapWindow(theDisplay, bbWindow);
XFlush(theDisplay);

/* Clean up memory */

free(buffer);
free_matrix(values,1,col,1,row);

/* And go */

return NOERROR;
} /* display_bb() */

/*****
/* bb_movie() */
/* */
/* Run a movie of this stuff. Done by repeated calls to display_bb. */
*****/

int bb_movie(int argc, char **argv)
{
#ifdef __NCURSES_H
    MEVENT mouse;          /* A mouse event. */
#endif
    int key_stroke;

    // is display_ps running?

    if (dps_running)
    {

```

```

return error(MESSAGE,"Can not start movie while power spectrum is being displayed.");
}

no_socket();

/* Are we already doing this in the background? */
if (bb_movie_running)
{
return error(ERROR,"Already running movie in the background.");
}

// disable query of stats from rt side during movie
freeze_stats_query = TRUE;

        time(&bb_start_time);
bb_frames = 0;
werase(system_window);
mvwaddstr(system_window,0,0,CONT_TEXT);
while(TRUE)
{
/* Display the frame */

if (single_bb(0,NULL) != NOERROR) break;
bb_frames++;
mvwprintw(system_window,1,0,
"%5.1f F/S %3d (%3d,%3d) Max %3d (%3d,%3d) Min",
bb_frames/((float)time(NULL) - bb_start_time),
max_frame, max_x, max_y,
min_frame, min_x, min_y);
wrefresh(system_window);

/* Did someone type a key? */

if (kbhit())
{
key_stroke = get_command();
#ifdef __NCURSES_H
if (key_stroke == KEY_MOUSE)
{
getmouse(&mouse);

if (mouse.bstate == BUTTON1_CLICKED)
{
if (mouse.y == SYSTEM_BEGIN_Y &&
mouse.x < (int)strlen(CONT_TEXT)+2)
{
key_stroke = KEY_ESC;
}
}
}

#endif
if (key_stroke == KEY_ESC) break;
}

/* Run a background job */

background();
}

werase(system_window);
wrefresh(system_window);

// enable query of stats from rt side now that movie is over
freeze_stats_query = FALSE;

return NOERROR;
} /* bb_movie() */

/*****
/* bb_background_movie() */
/* */
/* Run a movie of BB. Done by repeated calls to single_bb. */
/* This version does most of the work as a background process and so */
/* may be slow. It does, however, allow you to do other things while */
/* the movie is displayed. */

```

```

/* First call toggles it on, the second one off. */
/*****

int bb_background_movie(int argc, char **argv)
{
    no_socket();

    if (dps_running)
    {
        error(MESSAGE,"Can not run background movie while power spectrum is being displayed.");
    }

    // turn off bg movie if it is already running

    if (bb_movie_running)
    {
        background_del(bb_movie_background_job);
        bb_movie_running = FALSE;
        message(system_window,"Average frame rate = %.1f",
            bb_frames/((float)time(NULL) - bb_start_time));
    }

    // start bg movie if it wasn't running already

    else
    {
        if(!dps_running)
        {
            clean_command_line();
            single_bb(0,NULL);
            background_add(bb_movie_background_job);
            bb_movie_running = TRUE;
            time(&bb_start_time);
            bb_frames = 0;
        }
    }

    return NOERROR;
} /* bb_background_movie() */

/*****
/* bb_movie_background_job() */
/* */
/* Background job used by background_movie process. */
/* If a window does not already exist it will use the default window. */
*****/

int bb_movie_background_job(void)
{
    uint16_t *buffer;
    char s[81],display[81];

    /* Is there a bb to get and display? */

    /* Go get the bb */

    if ((buffer = get_bb()) == NULL)
    {
        return error(ERROR,"Failed to get bb data.");
    }

    /* display it */

    if (default_display_machine == NULL)
    {
        sprintf(display,"%s:0.0",get_machine_name(s));
    }
    else
    {
        sprintf(display,"%s:0.0",default_display_machine);
    }
    display_bb(display, buffer);
    bb_frames++;

    return NOERROR;
} /* bb_movie_background_job() */

```

```

/*****
/* save_bb() */
/* */
/* Gets subframes and saves them to a file. The file will consist */
/* of a series of unsigned bytes. The first two are the x and y sizes */
/* then comes the data in the frame itself. */
*****/

int save_bb(int argc, char **argv)
{
    char s[81];
    char temp[81];
    char *filename = NULL;
    char *p = NULL;
    bool found_dir;
    uint16_t *buffer;
    FILE *output;
    int frames;
    int c;
    bool loop, frames_set, filename_set;
    int maxframes;
    uint16_t exp, x, y, dx, dy;

#ifdef __NCURSES_H
    MEVENT mouse;          /* A mouse event. */
#endif

    // is display_ps running?
    if (dps_running)
    {
        return error(MESSAGE, "Can not save data while power spectrum is being displayed.");
    }

    no_socket();

    // Set default number of exposures to record. 1 frame, no looping.
    loop = FALSE;
    maxframes = 1;

    //Set command line flags
    frames_set = filename_set = FALSE;

        /* Check out the command line */

        if (argc > 1)
        {
            filename = argv[1];
            filename_set = TRUE;
        }

    if (!frames_set)
    {
        clean_command_line();
        sprintf(s, "%5d", maxframes);
        if (quick_edit("Number of exposures to save",
                      s, s, NULL, INTEGER) == KEY_ESC) return NOERROR;

        sscanf(s, "%d", &maxframes);
        clean_command_line();
    }
    if (maxframes < 1)
    {
        error(MESSAGE, "Exposure number < 1 means loop.");
        loop = TRUE;
    }
    frames_set = TRUE;
}

    if (!filename_set)
    {
        filename = s;
        sprintf(s, "          frame.dat");
        if (quick_edit("Filename", s, s, NULL, STRING) == KEY_ESC)
            return NOERROR;
        filename_set = TRUE;
    }
}

```



```

        while(*filename==' ' && *filename != 0) filename++;
        if (*filename == 0)
        {
            return error(ERROR,"Null file name.");
        }

        /* OK, if no directory specified, put it in data directory. */
        for(found_dir = FALSE, p = filename; !found_dir && *p != 0; p++)
        {
            if (*p == '/') found_dir = TRUE;
        }

        if (!found_dir)
        {
            sprintf(temp,"%s%s",ARC_CCD_DATA_DIRECTORY,filename);
            filename = temp;
        }

/* OK, now we try and save the data */
        if (fopen(filename,"r") != NULL)
        {
            sprintf(s,"The file %s already exists.",filename);
            if (!ask_yes_no(s,"Shall we overwrite it?"))
            {
                return NOERROR;
            }
        }

        if ((output = fopen(filename,"w")) == NULL)
        {
            return error(ERROR,"Failed to open file %s.", filename);
        }

/* Are the parameters valid? */
        if (exposure < 0)
        {
            return error(ERROR,"Set up the exposure time first.");
        }

        exp = exposure;
        x = bbdata.x;
        y = bbdata.y;
        dx = bbdata.dx;
        dy = bbdata.dy;

        // write the exposure time and bounding box data

        if (fwrite(&exp, sizeof(exp), 1, output) != 1 ||
            fwrite(&x, sizeof(x), 1, output) != 1 ||
            fwrite(&y, sizeof(y), 1, output) != 1 ||
            fwrite(&dx, sizeof(dx), 1, output) != 1 ||
            fwrite(&dy, sizeof(dy), 1, output) != 1)
        {
            fclose(output);
            return error(ERROR,"Failed to write data to %s.",filename);
        }

/* Start looping */
        frames = 1;
        while(frames <= maxframes || loop == TRUE)
        {
            message(system_window,"%s Frame %d",CONT_TEXT,frames);

/* Get the frame */
            if ((buffer = get_bb()) == NULL)
            {
                fclose(output);
                werase(system_window);
                wrefresh(system_window);
                return error(ERROR,"Failed to get frame data.");
            }

            if ((int)fwrite(buffer, sizeof(uint16_t),
                bbdata.dx*bbdata.dy, output) != bbdata.dx*bbdata.dy)

```

```

{
fclose(output);
free(buffer);
werase(system_window);
wrefresh(system_window);
return error(ERROR,
"Failed to write data to %s.",filename);
}
free(buffer);

frames++;

if (kbhit())
{
c = get_command();
#ifdef __NCURSES_H
if (c == KEY_MOUSE)
{
getmouse(&mouse);

if (mouse.bstate == BUTTON1_CLICKED)
{
if (mouse.y == SYSTEM_BEGIN_Y &&
mouse.x < (int)strlen(CONT_TEXT)+2)
{
c = KEY_ESC;
}
}
}
}

#endif
if (c == KEY_ESC) break;
}

werase(system_window);
wrefresh(system_window);
fclose(output);
return NOERROR;

} /* save_bb() */

/*****
/* call_make_bias() */
/* */
/* Calls make_bias(). Argument is number of frames to use. */
*****/

int call_make_bias(int argc, char **argv)
{
char s[256];

int retval;

int frames = DEFAULT_FRAMES_IN_BIAS;

if (dps_running)
{
return error(MESSAGE,"Can not make bias while power spectrum is being displayed.");
}

if (argc > 1)
{
sscanf(argv[1],"%d",&frames);
}
else
{
sprintf(s,"%5d",frames);
if (quick_edit("Number of frames to use in bias",
s,s,NULL,INTEGER) == KEY_ESC) return NOERROR;

sscanf(s,"%d",&frames);
clean_command_line();
}

// close the spectrograph shutter for making a bias

send_shutter_close(spec_shut);

```

```

// wait for shutter
usleep(SHUTTER_WAIT_TIME);
// disable query of stats through RT FIFO
freeze_stats_query = TRUE;
//call make_bias to get bias frames from rt side
retval = make_bias(frames);
//enable query of stats through RT FIFO
freeze_stats_query = FALSE;
// open the spectrograph shutter again
send_shutter_open(spec_shut);
// wait for shutter ?
// usleep(SHUTTER_WAIT_TIME);
return retval;
} // call_make_bias()

/*****
 * call_toggle_bias() */
*/
/* Calls toggle_bias(). */
*****/

int call_toggle_bias(int argc, char **argv)
{
return toggle_bias();
} // call_toggle_bias

/*****
 * send_ints_on() */
*/
/* Turn on interrupts. */
*****/

int send_ints_on(int argc, char **argv)
{
send_rt_command(RTCCD_INTS_ON, 0);
return NOERROR;
} /* send_ints_on() */

/*****
 * send_ints_off() */
*/
/* Turn off interrupts. */
*****/

int send_ints_off(int argc, char **argv)
{
send_rt_command(RTCCD_INTS_OFF, 0);
return NOERROR;
} /* send_ints_off() */

/*****
 * toggle_use_fft() */
*/
/* Toggles the use_fft flag */
*****/

int toggle_use_fft(int argc, char **argv)
{
int err = 0;
use_fft = !use_fft;
err = set_rt_flag(FLAG_ID_USE_FFT, use_fft);
return err;
} // toggle_use_fft()

```

```

/*****
/* toggle_use_lpt() */
/* */
/* Toggles the use_lpt_toggle flag */
*****/

int toggle_use_lpt(int argc, char **argv)
{
    int err = 0;
    use_lpt_toggle = !use_lpt_toggle;
    err = set_rt_flag(FLAG_ID_USE_LPT_TOGGLE, use_lpt_toggle);
    return err;
} // toggle_use_lpt()

/*****
/* init_fft() */
/* */
/* Requests init of fft. */
*****/

int init_fft(int argc, char **argv)
{
    send_rt_command(ARC_CCD_RT_INIT_FFT, 0);
    return NOERROR;
} /* init_fft() */

/*****
/* save_spectrum() */
/* */
/* Gets spectrum data and saves them to a file. The file will consist */
/* of a series of unsigned bytes. */
/* Data types recorded: */
/* 1. Power spectrum statistics (always) */
/* 2. Power Spectra (with -ps flag at command line) */
/* 3. Spectra (with -spec flag at command line) */
*****/

int save_spectrum(int argc, char **argv)
{
    char display_name[256];
    char s[81];
    char temp[81];
    char *filename, *p;
    bool found_dir;
    bool record_data = FALSE;
    bool rec_ps = FALSE;
    bool rec_spec = FALSE;
    bool rec_vd = FALSE;
    bool data_received = TRUE;
    bool loop = TRUE;
    bool begin_with_dither = FALSE;
    char *buffer, *pbuffer, *rtdata = NULL;
    float *ps = NULL;
    float *spec = NULL;
    FILE *output = NULL;
    int buf_size, frame_size, spec_size, ps_size;
    int frames;
    int c;
    int i, j;
    char rec_arg = 1;
    struct tm *gmt_out;
    time_t now;
    int spec_disp_channel1 = DEFAULT_SPEC_DISP_CHANNEL1;
    int spec_disp_channel2 = DEFAULT_SPEC_DISP_CHANNEL2;
    char *scp_id[NUM_OF_SAVE_SPEC_SCOPE_OPTIONS+1];
    int loop_update = 0;
    float channel_disp_scale = 2.0;

#ifdef SPEAK_TO_OPLE
    int cart = -2;
#endif

    // from simpleX

    //first is bin, power, snr. second is height, width, skew
    //these will be the quantities associated with R,G,B for each scope.

```

```

scope scope_1, scope_2;

// vector of values for scope;

float *scp_1_values, *scp_2_values, *scp_values;

// int which keeps track of which value will be displayed in the scope.
// 1-10 = bin,pow,snr,height,width,skew,sigma,errsignal,tracksignal,
// dither position (dac)

int scp_1_disp = 12, scp_2_disp = 13;

#ifdef __NCURSES_H
    MEVENT mouse;          /* A mouse event. */
#endif

// dummy arrays for fake data when debugging

#ifdef DEBUG_SAVE_SPECTRUM_FAKE_DATA

PS_STATS fake_ps_stats;

float fake_ps[PS_SIZE];
float fake_spec[FFT_SIZE];

//set up the fake ps stats

fake_ps_stats.time = 1;
fake_ps_stats.bin = 1.0;
fake_ps_stats.height = 1.0;
fake_ps_stats.power = 1.0;
fake_ps_stats.width = 1.0;
fake_ps_stats.snr = 1.0;
fake_ps_stats.skew = 0.0;
fake_ps_stats.errsignal = 0.0;
fake_ps_stats.tracksignal = 0.0;
fake_ps_stats.sigma = 0.0;
fake_ps_stats.errsignal = 0.0;
fake_ps_stats.tracksignal = 0.0;
fake_ps_stats.dith_dac = 0;
fake_ps_stats.dith_um = 0.0;
fake_ps_stats.phase = 0.0;

// set up the fake ps as incrementally descending to 0

for(i=0; i < PS_SIZE; i++)

fake_ps[i] = PS_SIZE - i;

// set up the fake spectrum as incrementally ascending

for(i=0; i < FFT_SIZE; i++)

fake_spec[i] = i;

#endif

//set up names of scope ids
//they're all 10 character strings

// the #define NUM_OF_SAVE_SPEC_SCOPE_OPTIONS at the top of this file
// should equal the index of the last scp_id[index] on the list.
// currently = 12

scp_id[0] = "    None";
scp_id[1] = "    Bin";
scp_id[2] = "    Power";
scp_id[3] = "    SNR";
scp_id[4] = "    Sigma";
scp_id[5] = "    Height";
scp_id[6] = "    Width";
scp_id[7] = "    Skew";
scp_id[8] = "    Errsignal";
scp_id[9] = "    Tracksig";
scp_id[10] = "Dither Pos";
scp_id[11] = "    Phase";
scp_id[12] = "    Channel A";
scp_id[13] = "    Channel B";

// is display_ps running?

```

```

if (dps_running)
{
return error(MESSAGE,"Can not save data while power spectrum is being displayed.");
}

no_socket();

/* Check out the command line */

filename = NULL;

if (argc > 1)
{
record_data = ( *argv[1] == 'Y' || *argv[1] == 'y');
}
else
{
record_data = ask_yes_no("", "Save data to a file?");
}

if(record_data)
{
if (argc > 2)
{
rec_ps = (*argv[2] == 'Y' || *argv[2] == 'y');
}
else
{
rec_ps = ask_yes_no("We can save the power spectrum.",
"Should we do that?");
}

if (argc > 3)
{
rec_spec = (*argv[3] == 'Y' || *argv[3] == 'y');
}
else
{
rec_spec = ask_yes_no("We can save the spectrum.",
"Should we do that?");
}

if (argc > 4)
{
filename = argv[4];
}
else
{
sprintf(s, "ps.dat");
if (quick_edit("Filename",s,s,NULL,STRING) == KEY_ESC)
return NOERROR;
filename = s;
}

while(*filename==' ' && *filename != 0) filename++;
if (*filename == 0)
{
return error(ERROR,"Null file name.");
}
}

begin_with_dither = ask_yes_no("",
"Do you want to start a dither scan immediately?");

#ifdef SPEAK_TO_OPLE
// Set which ople cart will be controlled
if( (cart = get_dl_number(1,NULL)) < 0)
{
return error(ERROR,"Delay line not set. Exiting.");
}
#endif

// set flags for what will be recorded

if(rec_ps)
rec_arg = ( rec_arg | (1 << 1) );
if(rec_spec)
rec_arg = ( rec_arg | (1 << 2) );
if(rec_vd)

```

```

rec_arg = ( rec_arg | (1 << 4) );

if(use_bias)
rec_arg = ( rec_arg | (1 << 5) );
if(use_spec_flat)
rec_arg = ( rec_arg | (1 << 6) );
if(use_ps_flat)
rec_arg = ( rec_arg | (1 << 7) );

ps_size = PS_SIZE * sizeof(float);

spec_size = spec_row[0]* SPEC_SIZE * sizeof(float);

// Allocate memory for the output buffer
// first work out how many bytes per frame

frame_size = sizeof(PS_STATS);
if(rec_ps) frame_size += ps_size;
if(rec_spec) frame_size += spec_size;
buf_size = frame_size * SAVE_SPECTRUM_OUTPUT_BUFFER_ENTRIES;

if ((buffer = (char *) malloc(buf_size)) == NULL )
{
return error(ERROR,
"Unable to allocate memory for output buffer.");
}

// Allocate memory for spectrum and / or ps if they're being recorded

if (rec_ps)
{
if ((ps = (float *) malloc(ps_size)) == NULL )
{
free(buffer);
return error(ERROR,
"Unable to allocate memory for power spectrum.");
}
}

if (rec_spec)
{
if ((spec = (float *) malloc(spec_size)) == NULL )
{
free(buffer);
if(rec_ps) free(ps);
return error(ERROR,
"Unable to allocate memory for spectrum.");
}
}

#ifdef DEBUG_SAVE_SPECTRUM_MESSAGES
error(ERROR,"Finished allocating memory for buffer and spectra.\n");
#endif

if(record_data)
{
/* OK, if no directory specified, put it in data directory. */
for(found_dir = FALSE, p = filename; !found_dir && *p != 0; p++)
{
if (*p == '/') found_dir = TRUE;
}

if (!found_dir)
{
sprintf(temp,"%s%s",ARC_CCD_DATA_DIRECTORY,filename);
filename = temp;
}

/* OK, now we try and save the data */

if (fopen(filename,"r") != NULL)
{
sprintf(s,"The file %s already exists.",filename);
if (!ask_yes_no(s,"Shall we overwrite it?"))
{
return NOERROR;
}
}
}

```

```

}
}

if ((output = fopen(filename,"w")) == NULL)
{
return error(ERROR,"Failed to open file %s.", filename);
}
}
/* Are the parameters valid? */

if (exposure < 0)
{
return error(ERROR,"Set up the exposure time first.");
}

// create displays

#ifdef DEBUG_SAVE_SPECTRUM_MESSAGES
error(ERROR,"Setting up scopes...\n");
#endif

if (!display_is_open)
{
if (default_display_machine == NULL)
{
sprintf(display_name,"%s:0.0",
get_machine_name(display_name));
}
else
{
sprintf(display_name,"%s:0.0",
default_display_machine);
}
if (quick_edit("Display name",display_name,
display_name,NULL,STRING) == KEY_ESC)
return NOERROR;

if (strcmpi(display_name,"null") == 0)
{
if (initX(NULL) == 0)
{
return error(ERROR,"Failed to open display.");
}
}
else
{
if (initX(display_name) == 0)
{
return error(ERROR,
"Failed to open display %s.",
display_name);
}
}
}

#ifdef DEBUG_SAVE_SPECTRUM_MESSAGES
error(ERROR,"Finished checking if display is open...\n");
#endif

scope_1 = open_scope("Scope 1",theWidth-525,480,
150,50);

#ifdef DEBUG_SAVE_SPECTRUM_MESSAGES
error(ERROR,"Scope 1 opened.\n");
#endif

scope_2 = open_scope("Scope 2",theWidth-525,650,
150,50);

#ifdef DEBUG_SAVE_SPECTRUM_MESSAGES

```



```

error(ERROR,"Scope 2 opened.\n");

#endif

    if (add_signal(&scope_1,65535,0,0,0.0,1.0) != 1)
    {
        close_scope(&scope_1);
        free(buffer);
        if(ps != NULL)
            free(ps);
        if(spec != NULL)
            free(spec);
        if(record_data)
            fclose(output);
        quitX();
        exit(-2);
    }

#ifdef DEBUG_SAVE_SPECTRUM_MESSAGES
error(ERROR,"Scope 1 signal added.\n");
#endif

    if (add_signal(&scope_2,0,65535,0,0.0,1.0) != 1)
    {
        close_scope(&scope_2);
        free(buffer);
        if(ps != NULL)
            free(ps);
        if(spec != NULL)
            free(spec);
        if(record_data)
            fclose(output);
        quitX();
        exit(-2);
    }

#ifdef DEBUG_SAVE_SPECTRUM_MESSAGES
error(ERROR,"Scope 2 signal added.\n");
#endif

    scp_values = vector(1,NUM_OF_SAVE_SPEC_SCOPE_OPTIONS);

    scp_1_values = vector(1,1);
    scp_2_values = vector(1,1);

#ifdef DEBUG_SAVE_SPECTRUM_MESSAGES
error(ERROR,"Finished setting up scopes.\n");
#endif

    if(record_data)
    {
        // Write the gmt

        now = time(NULL);
        gmt_out = gmtime(&now);

        if (gmt_out == NULL)
            loop = FALSE;

        else
            if( fwrite(gmt_out, sizeof(*gmt_out), 1, output) != 1)
                loop = FALSE;

        // write the exposure time

        if( fwrite(&exposure, sizeof(exposure), 1, output) != 1)
            loop = FALSE;

        // write the rec_arg, which shows
        // whether ps and spec data will be
        // written in addition to the ps stats. Also tells whether bias
        // and flats were used (if so, included in file)

```

```

if (fwrite(&rec_arg, sizeof(rec_arg), 1, output) != 1)
loop = FALSE;

//write the bounding box data

if (fwrite(&bbdata, sizeof(SROI), 1, output) != 1)
loop = FALSE;

//write the row number used for the spectrum

if(fwrite(&(spec_row[0]), sizeof(spec_row[0])*(MAX_NUM_SIMULTANEOUS_SPECTRA+1),1,output) != 1)
loop = FALSE;

// write bias data

if(use_bias)
{
for(j=1; j<=spec_row[0]; j++)
{
if ( (int)fwrite( bias +
( spec_row[j] * bbdata.dx ),
sizeof(long int),bbdata.dx,
output)
!= bbdata.dx)
loop = FALSE;
}
}

// write spec flat data

if(use_spec_flat)
{
for(j=0; j<spec_row[0]; j++)
{
if (fwrite( &(spec_flat[j][0]), sizeof(float),
SPEC_SIZE,output) != SPEC_SIZE)
loop = FALSE;
}
}

// write ps flat data

if(use_ps_flat)
{
if (fwrite( ps_flat, sizeof(float),
PS_SIZE,output) != PS_SIZE)
loop = FALSE;
}

#ifdef DEBUG_SAVE_SPECTRUM_MESSAGES
printf("Finished writing initial file data.\n");
#endif
}

// Set up main window display for data gathering mode.

heading(heading_window,"COLLECTING DATA - TYPE <ESC> TO STOP",
"Press <1,2> to cycle quantity displayed by corresponding scope.");

active_window = sub_main_window;

if (record_data)
{
mvwprintw(sub_main_window,0,10,"Filename          = %s",
filename);
}

werase(sub_main_window);
mvwaddstr(sub_main_window,1,2," Bin = ");
mvwaddstr(sub_main_window,2,2," Power = ");
mvwaddstr(sub_main_window,3,2," SNR = ");
mvwaddstr(sub_main_window,4,2," Sigma = ");
mvwaddstr(sub_main_window,5,2,"Height = ");
mvwaddstr(sub_main_window,6,2," Width = ");
mvwaddstr(sub_main_window,7,2,"ChnScl = ");
mvwaddstr(sub_main_window,1,23,"Scope1 = ");
mvwaddstr(sub_main_window,2,23,"Scope2 = ");
mvwaddstr(sub_main_window,3,23," Errsg = ");

```

```

        mvwaddstr(sub_main_window,4,23,"Tracksg= ");
        mvwaddstr(sub_main_window,5,23,"DithDac= ");
        mvwaddstr(sub_main_window,6,23,"ChnlVal= ");
        mvwaddstr(sub_main_window,7,23,"SpecChn= ");
        mvwaddstr(sub_main_window,1,44," Servo = ");
        mvwaddstr(sub_main_window,2,44,"Target = ");
        mvwaddstr(sub_main_window,3,44," Prop = ");
        mvwaddstr(sub_main_window,4,44," Integ = ");
        mvwaddstr(sub_main_window,5,44," Diff = ");
        mvwaddstr(sub_main_window,6,44," Gain = ");
        mvwaddstr(sub_main_window,7,44," Incr = ");
        wrefresh(sub_main_window);

werase(status_window);
wrefresh(status_window);

#ifdef DEBUG_SAVE_SPECTRUM_MESSAGES
error(ERROR,"Finished setting up sub_main_window.\n");
#endif

/* Start looping */
background_off(0,NULL);

// send command to rt side to start recording
recording = TRUE;
freeze_stats_query = TRUE;

// only start recording on the rt side if you're not faking data
// for debugging

#ifdef DEBUG_SAVE_SPECTRUM_FAKE_DATA
send_rt_command(ARC_CCD_RT_START_RECORD,rec_arg);
#endif

#ifdef DEBUG_SAVE_SPECTRUM_MESSAGES
error(ERROR,"RT side set to record. Starting recording loop.\n");
#endif

if(begin_with_dither)
activate_dither_scan((unsigned char)2);

frames = 1;
pbuffer = buffer;
while(loop)
{
message(system_window,"%s Frame %d",CONT_TEXT,frames++);

/* Get the data */

#ifdef DEBUG_SAVE_SPECTRUM_MESSAGES
message(system_window,"Getting data from FIFO...\n");
#endif

#ifdef DEBUG_SAVE_SPECTRUM_FAKE_DATA
// point to the fake data
ps_peak = fake_ps_stats;

if(rec_ps)
ps = &fake_ps[0];

if(rec_spec)
spec = &fake_spec[0];
#else

// read ps data from DATA_OUT_FIFO

```

```

if (!read_arc_data((char *)&ps_peak,sizeof(PS_STATS)))
{
    data_received = FALSE;
    message(system_window,
"Failed to get power spectrum data.");
}

if(rec_ps)
{
    // read ps data from DATA_OUT_FIFO

    if (!read_arc_data((char *)ps,ps_size))
    {
        data_received = FALSE;
        message(system_window,"Failed to get ps data.");
    }
}

if(rec_spec)
{
    // read spec data from DATA_OUT_FIFO

    if (!read_arc_data((char *)spec,spec_size))
    {
        data_received = FALSE;
        message(system_window,"Failed to get spec data.");
    }
}

#endif

// if data ok, add to buffer and display array

if (data_received)
{
    #ifdef DEBUG_SAVE_SPECTRUM_MESSAGES
    message(system_window,"Success. Putting data into buffer.\n");
    #endif

    // put data into buffer

    rtdata = (char *) &ps_peak;

    for(i=0; i< (int)sizeof(PS_STATS); i++)
    {
        *pbuffer++ = *rtdata++;
    }

    if(rec_ps)
    {
        rtdata = (char *) ps;

        for(i=0; i< ps_size ; i++)
        {
            *pbuffer++ = *rtdata++;
        }
    }

    if(rec_spec)
    {
        rtdata = (char *) spec;

        for(i=0; i< spec_size; i++)
        {
            *pbuffer++ = *rtdata++;
        }
    }

    if(loop_update++ >= SAVE_SPECTRUM_LOOPS_PER_UPDATE)
    {
        // display data
        // everything should be normalized to 0..1

        scp_values[1] = ps_peak.bin / BIN_NORMALIZATION;
        scp_values[2] = ps_peak.power/ POWER_NORMALIZATION;
        scp_values[3] = ps_peak.snr / SNR_NORMALIZATION;
    }
}

```

```

scp_values[4] = ps_peak.sigma / SIGMA_NORMALIZATION;
scp_values[5] = ps_peak.height / HEIGHT_NORMALIZATION;
scp_values[6] = ps_peak.width / WIDTH_NORMALIZATION;
scp_values[7] = (ps_peak.skew / SKEW_NORMALIZATION)
+ SKEW_OFFSET;
scp_values[8] =
(ps_peak.errsignal / ERRSIGNAL_NORMALIZATION)
+ ERRSIGNAL_OFFSET;
scp_values[9] =
(ps_peak.tracksignal
/ TRACKSIGNAL_NORMALIZATION)
+ TRACKSIGNAL_OFFSET;
scp_values[10] = (float)ps_peak.dith_dac /
DITH_DAC_NORMALIZATION;
scp_values[11] = (ps_peak.phase / PHASE_NORMALIZATION)
+ PHASE_OFFSET;

// displaying the current data in a spectral channel requires
// the flat to be set and non-zero. The spectrum must also be recording.

if(rec_spec)
{
if(use_spec_flat)
{
scp_values[12] =
(*(spec+spec_disp_channel1)-1.0)
* channel_disp_scale / 2.0
+ FRINGE_OFFSET ;
}
else
{
if(spec_flat_has_been_set &&
(spec_flat[0][spec_disp_channel1]
> 0.0) )
{
scp_values[12] =
(
(*(spec+spec_disp_channel1))
/ spec_flat[0][spec_disp_channel1]
- 1.0
) * channel_disp_scale / 2.0
+ FRINGE_OFFSET;
}
else
scp_values[12] = 0;
}
}
else
scp_values[12] = 0;

if(rec_spec)
{
if(use_spec_flat)
{
scp_values[13] =
(*(spec+spec_disp_channel2)-1.0)
* channel_disp_scale / 2.0
+ FRINGE_OFFSET ;
}
else
{
if(spec_flat_has_been_set &&
(spec_flat[0][spec_disp_channel2]
> 0.0) )
{
scp_values[13] =
(
(*(spec+spec_disp_channel2))
/ spec_flat[0][spec_disp_channel2]
- 1.0
) * channel_disp_scale / 2.0
+ FRINGE_OFFSET;
}
else
scp_values[13] = 0;
}
}
else
scp_values[13] = 0;

```

```

scp_1_values[1] = scp_values[scp_1_disp];
scp_2_values[1] = scp_values[scp_2_disp];

update_scope(scope_1, scp_1_values);
update_scope(scope_2, scp_2_values);

mvwprintw(sub_main_window, 1, 11, "%10.2f", ps_peak.bin);
mvwprintw(sub_main_window, 2, 11, "%10.2f", ps_peak.power);
mvwprintw(sub_main_window, 3, 11, "%10.2f", ps_peak.snr);
mvwprintw(sub_main_window, 4, 11, "%10.2f", ps_peak.sigma);
mvwprintw(sub_main_window, 5, 11, "%10.2f", ps_peak.height);
mvwprintw(sub_main_window, 6, 11, "%10.2f", ps_peak.width);
mvwprintw(sub_main_window, 7, 11, "%10.2f",
channel_disp_scale);
// mvwprintw(sub_main_window, 7, 11, "%10.2f", ps_peak.phase);

mvwprintw(sub_main_window, 1, 32, "%s", scp_id[scp_1_disp]);
mvwprintw(sub_main_window, 2, 32, "%s", scp_id[scp_2_disp]);
mvwprintw(sub_main_window, 3, 32, "%10.2f",
ps_peak.errsignal);
mvwprintw(sub_main_window, 4, 32, "%10.0f",
ps_peak.tracksignal);
mvwprintw(sub_main_window, 5, 32, "%10d",
ps_peak.dith_dac);
/* if (ps_servo_on)
* {
* mvwprintw(sub_main_window, 6, 32, "          ON");
* }
* else
* {
* mvwprintw(sub_main_window, 6, 32, "          OFF");
* }
*/
if (rec_spec)
{
mvwprintw(sub_main_window, 6, 32, "%10.0f",
*(spec+spec_disp_channel1));
}

mvwprintw(sub_main_window, 7, 32, "%5d,%4d",
spec_disp_channel1, spec_disp_channel2);

if (ps_servo_on)
{
mvwprintw(sub_main_window, 1, 53, "          ON");
}
else
{
mvwprintw(sub_main_window, 1, 53, "          OFF");
}
mvwprintw(sub_main_window, 2, 53, "%10.2f", ps_servo.target);
mvwprintw(sub_main_window, 3, 53, "%10.2f", ps_servo.p);
mvwprintw(sub_main_window, 4, 53, "%10.2f", ps_servo.i);
mvwprintw(sub_main_window, 5, 53, "%10.2f", ps_servo.d);
mvwprintw(sub_main_window, 6, 53, "%10.2f", ps_servo.gain);
mvwprintw(sub_main_window, 7, 53, "%10.2f",
ps_servo.tweak_increment);

wrefresh(sub_main_window);

#ifdef DEBUG_SAVE_SPECTRUM_MESSAGES
message(system_window, "Displays updated.\n");
#endif
// reset update counter
loop_update = 0;
} // end display if uptade counter reached
}

else
{
#ifdef DEBUG_SAVE_SPECTRUM_MESSAGES
error(ERROR, "Failed. Aborting recording loop.\n");
#endif
#endif

```

```

loop = FALSE;
}

// check for key hit.

if (kbhit())
{
c = get_command();
#ifdef __NCURSES_H
if (c == KEY_MOUSE)
{
getmouse(&mouse);

if (mouse.bstate == BUTTON1_CLICKED)
{
if (mouse.y == SYSTEM_BEGIN_Y &&
mouse.x < (int)strlen(CONT_TEXT)+2)
{
c = KEY_ESC;
}
}
}

}

#endif

switch(c)
{
// UCase = scope1, lcase = scope2

case '!':
scp_1_disp++;
if(scp_1_disp >
NUM_OF_SAVE_SPEC_SCOPE_OPTIONS)
scp_1_disp = 1;
break;

case '1':
scp_1_disp--;
if(scp_1_disp < 1)
scp_1_disp =
NUM_OF_SAVE_SPEC_SCOPE_OPTIONS;
break;

case '@':
scp_2_disp++;
if (scp_2_disp >
NUM_OF_SAVE_SPEC_SCOPE_OPTIONS)
scp_2_disp = 1;
break;

case '2':
scp_2_disp--;
if (scp_2_disp < 1)
scp_2_disp =
NUM_OF_SAVE_SPEC_SCOPE_OPTIONS;
break;

// change fringe channel to display

case 'X':
if (spec_disp_channel1 < SPEC_SIZE-1)
spec_disp_channel1++;
break;

case 'x':
if (spec_disp_channel1 > 0)
spec_disp_channel1--;
break;

case 'C':
if (spec_disp_channel2 < SPEC_SIZE-1)
spec_disp_channel2++;
break;

case 'c':
if (spec_disp_channel2 > 0)
spec_disp_channel2--;
break;

// change channel display scale

```

```

case 'Z':
if(channel_disp_scale < 10)
channel_disp_scale += 0.5;
break;

case 'z':
if(channel_disp_scale > 1.0)
channel_disp_scale -= 0.5;
break;

//increment ps_servo p parameter
case 'p':
ps_servo_p += ps_servo_tweak_increment;
if(tweak_ps_servo_pid(TWEAK_PS_SERVO_P
,ps_servo_p) != 0)
{
ps_servo_p -=
ps_servo_tweak_increment;
}
break;

//decrement ps_servo p parameter
case 'p':
ps_servo_p -= ps_servo_tweak_increment;
if(tweak_ps_servo_pid(TWEAK_PS_SERVO_P
,ps_servo_p) != 0)
{
ps_servo_p +=
ps_servo_tweak_increment;
}
break;

//increment ps_servo i parameter
case 'I':
ps_servo_i += ps_servo_tweak_increment;
if(tweak_ps_servo_pid(TWEAK_PS_SERVO_I
,ps_servo_i) != 0)
{
ps_servo_i -=
ps_servo_tweak_increment;
}
break;

//decrement ps_servo i parameter
case 'i':
ps_servo_i -= ps_servo_tweak_increment;
if(tweak_ps_servo_pid(TWEAK_PS_SERVO_I
,ps_servo_i) != 0)
{
ps_servo_i +=
ps_servo_tweak_increment;
}
break;

//increment ps_servo d parameter
case 'D':
ps_servo_d += ps_servo_tweak_increment;
if(tweak_ps_servo_pid(TWEAK_PS_SERVO_D
,ps_servo_d) != 0)
{
ps_servo_d -=
ps_servo_tweak_increment;
}
break;

//decrement ps_servo d parameter
case 'd':
ps_servo_d -= ps_servo_tweak_increment;
if(tweak_ps_servo_pid(TWEAK_PS_SERVO_D
,ps_servo_d) != 0)
{
ps_servo_d +=
ps_servo_tweak_increment;
}
break;

//increment ps_servo gain parameter
case 'G':
ps_servo_gain += ps_servo_tweak_increment;

```



```

if(tweak_ps_servo_pid(TWEAK_PS_SERVO_GAIN
,ps_servo_gain) != 0)
{
ps_servo_gain -=
ps_servo_tweak_increment;
}
break;

//decrement ps_servo gain parameter
case 'g':
ps_servo_gain -= ps_servo_tweak_increment;
if(tweak_ps_servo_pid(TWEAK_PS_SERVO_GAIN
,ps_servo_gain) != 0)
{
ps_servo_gain +=
ps_servo_tweak_increment;
}
break;

//increase increment size x10
case '=':
case '+':
ps_servo_tweak_increment *= 10.0;
break;

//decrease increment size x10
case '-':
case '_':
ps_servo_tweak_increment /= 10.0;
break;

//Toggle ps_servo
case 'S':
case 's':
if(!ps_servo_on)
toggle_ps_servo(TRUE);
else
toggle_ps_servo(FALSE);
break;

//Toggle ps_servo squarewave injection
case 'Q':
case 'q':
if(!ps_servo_squarewave_on)
toggle_ps_servo_squarewave(TRUE);
else
toggle_ps_servo_squarewave(FALSE);
break;

//Increment ps_servo target freq by 1
case 'T':
case 'B':
case 'F':
ps_servo_target += 1.0;
if(ps_servo_target > PS_SERVO_TARGET_MIN &&
ps_servo_target < PS_SERVO_TARGET_MAX)
{
if(set_ps_servo_target(
ps_servo_target) != 0)
ps_servo_target -= 1.0;
}
else
ps_servo_target -= 1.0;
break;

//Decrement ps_servo target freq by 1
case 't':
case 'b':
case 'f':
ps_servo_target -= 1.0;
if(ps_servo_target > PS_SERVO_TARGET_MIN &&
ps_servo_target < PS_SERVO_TARGET_MAX)
{
if(set_ps_servo_target(
ps_servo_target) != 0)
ps_servo_target += 1.0;
}
else
ps_servo_target += 1.0;
break;

```

```

// Start vis dither scan with current settings
case '>':
case '.':
activate_dither_scan((unsigned char)1);
break;

// Restart vis dither scan with current settings
case '<':
case ',':
activate_dither_scan((unsigned char)2);
break;

//Pause vis dither scan
case '?':
case '/':
activate_dither_scan((unsigned char)0);
break;

#ifdef SPEAK_TO_OPLE
// Move ople down max
case 'J':
incremental_manual_target(cart,
-LARGE_OPLE_INCREMENT);
break;

// Move ople down min
case 'j':
incremental_manual_target(cart,
-SMALL_OPLE_INCREMENT);
break;

// Move ople up max
case 'K':
incremental_manual_target(cart,
LARGE_OPLE_INCREMENT);
break;

// Move ople up min
case 'k':
incremental_manual_target(cart,
SMALL_OPLE_INCREMENT);
break;
#endif

// stop looping
case KEY_ESC:
loop = FALSE;
break;

default:
break;
} // end kbhit switch(c)

} // end if(kbhit)

//if the buffer is full or exiting the loop, write data
if( ((pbuffer - buffer + 1) >= buf_size)
|| (loop == FALSE))
{
if(record_data)
{
if ((int)fwrite(buffer, (int)sizeof(char),
pbuffer - buffer , output) !=
pbuffer - buffer )
{
loop = FALSE;
message(system_window,
"Failed to write data to %s.",filename);
}
}
// reset pbuffer to start filling
// the buffer again.

pbuffer = buffer;

```

```

#ifdef SAVE_SPECTRUM_1_BUFFER_ONLY
//kludge to end recording when buffer fills for first time
loop = FALSE;
#endif
}

}

// tell rt side to stop sending data

#ifndef DEBUG_SAVE_SPECTRUM_FAKE_DATA

send_rt_command(ARC_CCD_RT_END_RECORD,0);

#endif

// purge RT_FIFOs

flush_data(0,NULL);

recording = FALSE;

freeze_stats_query = FALSE;

//restart background processes

background_on(0,NULL);

// cleanup

if(rec_spec)
{
if(spec != NULL)
free(spec);
}
if(rec_ps)
{
if(ps != NULL)
free(ps);
}

if(buffer != NULL)
free(buffer);

free_vector(scp_values,1,6);
free_vector(scp_1_values,1,3);
free_vector(scp_2_values,1,3);
close_scope(&scope_1);
close_scope(&scope_2);

werase(system_window);
wrefresh(system_window);

if(record_data)

fclose(output);

return NOERROR;

} /* save_spectrum() */

/*****
/* read_arc_data() */
/*
/* Attempts to read in arc ata. If it takes more than 2 seconds
/* we time out and give up. Returns TRUE if all went well.
*****/

bool read_arc_data(char *data, int n)
{
    time_t start;
    int i;
    char c;

    start = time(0);
    for(i = 0; i < n; i++)
    {
        while (read_rt_data(&c,1) != 1)

```

```

        {
            if (time(0) > start+2) return FALSE;
        }
        *data++ = c;
    }

    return TRUE;
} /* read_arc_data() */

#ifndef USE_RTCCD_API
// Obsolete function whose work is now done by the rtccd api.

/*****
/* load_arc_ccd() */
/* */
/* User callable. Closes FIFOS, calls script load_arc_cdd then opens */
/* FIFOs. */
*****/

int load_arc_ccd(int argc, char **argv)
{
    background_stop(0, NULL);
    close_rt_fifos();
    system("clear");
    nl();
    system("/usr/local/bin/load_arc_cdd");
    nonl();
    sleep(2);
    wrefresh(curscr);
    if (!open_rt_fifos()) error(FATAL,"Failed to open FIFOs");
    background_start(0, NULL);
    return NOERROR;
} /* load_arc_ccd() */

#endif

/*****
/* call_req_vis_dither_goto() */
/* */
/* Tries to request that the visible dither mirror be sent a dac */
/* position. */
*****/

int call_req_vis_dither_goto(int argc, char **argv)
{
    int dac_target = 0;

    char s[81];

    /* Now look at the arguments */

    if (argc > 1)
    {
        sscanf(argv[1], "%d", &dac_target);
    }
    else
    {
        sprintf(s, "%5d", dac_target);
        if (quick_edit("Dither position in dac",
                      s, s, NULL, INTEGER) == KEY_ESC) return NOERROR;

        sscanf(s, "%d", &dac_target);
        clean_command_line();
    }

    /* Now set it up */

    if (req_vis_dither_goto(dac_target) != 0)
    {
        return error(ERROR, "Failed to request dither mirror position.");
    }

    return NOERROR;
} /* call_req_vis_dither_goto() */

```

```

/*****
/* call_req_vis_dither_move_rel() */
/* */
/* Tries to request that the visible dither mirror be sent a dac */
/* position. */
*****/

int call_req_vis_dither_move_rel(int argc, char **argv)
{
    int dac_change = 0;

    char s[81];

    /* Now look at the arguments */

    if (argc > 1)
    {
        sscanf(argv[1], "%d", &dac_change);
    }
    else
    {
        sprintf(s, "%5d", dac_change);
        if (quick_edit("Dither position change in dac",
                      s, s, NULL, INTEGER) == KEY_ESC) return NOERROR;

        sscanf(s, "%d", &dac_change);
        clean_command_line();
    }

    /* Now set it up */

    if (req_vis_dither_move_rel(dac_change) != 0)
    {
        return error(ERROR, "Failed to request dither mirror position.");
    }

    return NOERROR;
} /* call_req_vis_dither_move_rel() */

/*****
/* call_set_ps_servo_target() */
/* */
/* Tries to request a target ps_servo fringe frequency. */
*****/

int call_set_ps_servo_target(int argc, char **argv)
{
    float new_target;

    char s[81];

    new_target = ps_servo_target;

    /* Now look at the arguments */

    if (argc > 1)
    {
        sscanf(argv[1], "%f", &new_target);
    }
    else
    {
        sprintf(s, "%5f", new_target);
        if (quick_edit("Fringe Tracker target fringe frequency",
                      s, s, NULL, FLOAT) == KEY_ESC) return NOERROR;

        sscanf(s, "%f", &new_target);
        clean_command_line();
    }

    // Check that it is within the acceptable range. If not, force it.

    if (new_target < PS_SERVO_TARGET_MIN)
        new_target = PS_SERVO_TARGET_MIN;

    else if (new_target > PS_SERVO_TARGET_MAX)
        new_target = PS_SERVO_TARGET_MAX;

```

```

/* Now set it up */

if ( set_ps_servo_target(new_target) != 0)
{
    return error(ERROR,
        "Failed to set new ps_servo target fringe frequency.");
}

// if it worked, let the user side copy of ps_servo_target show the change.

else
    ps_servo_target = new_target;

return NOERROR;

} /* call_set_ps_servo_target() */

/*****
/* call_toggle_ps_servo() */
/* */
/* The user can call this with no argument to simply toggle the ps_servo,*/
/* or can specify ON (also on,On,1, and anything beginning with t or T) */
/* or OFF (also Off, off, 0, or anything beginning with f or F). */
*****/
int call_toggle_ps_servo(int argc, char**argv)
{
    int err = 0;

    char c;

    bool new_value;

    // look at the arguments

    if (argc > 1)
    {
        // set c = to first char in argument string

        c = *argv[1];

        // anything starting with t is taken to mean true, also the # 1
        // and the word ON

        if ( c == 'T' || c == 't' || c == 1 || argv[1] == "on"
            || argv[1] == "On" || argv[1] == "ON")

            new_value = TRUE;

        else if ( c == 'F' || c == 'f' || c == 0 || argv[1] == "off"
            || argv[1] == "Off" || argv[1] == "OFF")

            new_value = FALSE;

        else
            return error(ERROR,
                "Argument passed to call_toggle_ps_servo invalid. ON or OFF. ");
    }

    // if no argument, then the new value is to toggle the old value

    else new_value = !ps_servo_on;

    // call set_rt_flag() to implement the change.

    if ( (err = toggle_ps_servo(new_value)) != NOERROR)
        return err;

    return err;

} // call_toggle_ps_servo()

/*****
/* call_toggle_ps_servo_squarewave() */
/* */
/* The user can call this with no argument to simply toggle the ps_servo*/
/* tuning squarewave,*/
/* or can specify ON (also on,On,1, and anything beginning with t or T) */
/* or OFF (also Off, off, 0, or anything beginning with f or F). */
*****/

```

```

int call_toggle_ps_servo_squarewave(int argc, char**argv)
{
    int err = 0;

    char c;

    bool new_value;

    // look at the arguments
    if (argc > 1)
    {
        // set c = to first char in argument string
        c = *argv[1];

        // anything starting with t is taken to mean true, also the # 1
        // and the word ON

        if ( c == 'T' || c == 't' || c == 1 || argv[1] == "on"
            || argv[1] == "On" || argv[1] == "ON")

            new_value = TRUE;

        else if ( c == 'F' || c == 'f' || c == 0 || argv[1] == "off"
            || argv[1] == "Off" || argv[1] == "OFF")

            new_value = FALSE;

        else
            return error(ERROR,
                "Argument passed to call_toggle_ps_servo invalid. ON or OFF. ");
    }

    // if no argument, then the new value is to toggle the old value
    else new_value = !ps_servo_squarewave_on;

    // call set_rt_flag() to implement the change.

    if( (err = toggle_ps_servo_squarewave(new_value)) != NOERROR)
        return err;

    return err;
} // call_toggle_ps_servo_squarewave()

/*****
/* call_tweak_ps_servo_pid() */
/* */
/* The user can change ps_servo p,i,d,or gain from the command line. */
/* Can include on the command line arg p,i,d,g for which parameter, and */
/* a number for the new value. Will prompt for this info if needed. */
*****/
int call_tweak_ps_servo_pid(int argc, char **argv)
{
    char s[81];
    char c;
    int which_par;
    float new_value;
    int err = 0;

    clean_command_line();

    // look at the arguments

    // if user designated which ps_servo parameter to tweak

    if (argc > 1)

        c = *argv[1];

    // if no args on command line, ask which parameter to tweak

    else

    {
        sprintf(s,"p");
        if (quick_edit("Which parameter?",s,s,NULL,STRING)
            == KEY_ESC) return NOERROR;
    }

```

```

sscanf(s,"%c",&c);
clean_command_line();

}

switch(c)
{
case 'P':
case 'p':
which_par = TWEAK_PS_SERVO_P;
new_value = ps_servo_p;
break;
case 'I':
case 'i':
which_par = TWEAK_PS_SERVO_I;
new_value = ps_servo_i;
break;
case 'D':
case 'd':
which_par = TWEAK_PS_SERVO_D;
new_value = ps_servo_d;
break;
case 'G':
case 'g':
which_par = TWEAK_PS_SERVO_GAIN;
new_value = ps_servo_gain;
break;
default:
return error(ERROR,
"Parameter must be p,i,d,or g. Aborted.");
}

// this happens if the user also stated a new value for the parameter.

if (argc > 2)
{
sscanf(argv[2],"%f",&new_value);
}

else
{
    sprintf(s,"%5f",new_value);
    if (quick_edit("New parameter value",
        s,s,NULL,FLOAT) == KEY_ESC) return NOERROR;

    sscanf(s,"%f",&new_value);
    clean_command_line();
}

err = tweak_ps_servo_pid(which_par,new_value);

return err;
} // call_tweak_ps_servo_pid()

/*****
* call_setup_dither_scan() */
*/
/* User callable function to set up parameters for a vis dither scan. */
/* The scan can be activated within the save_spectrum function. */
*****/
int call_setup_dither_scan(int argc, char **argv)
{
ARCCCD_DITH_SCAN scan;
int temp;
char s[81];

// give the scan the default values

scan.lower_limit = RTDAC_LOWER_LIMIT;
scan.upper_limit = RTDAC_UPPER_LIMIT;
scan.dx = MAX_DIFF;
scan.on = FALSE;
scan.going_up = TRUE;
scan.dx_index = 0;
scan.scan_index = 0;
scan.total_scans = 0;

// need dither start, end, step size, and number of dithers.

```



```

if(argc == 5)
{
// first is the lower limit
sscanf(argv[1],"%d", &(scan.lower_limit) );
// second is the upper limit
sscanf(argv[2],"%d", &(scan.upper_limit) );
// third is the step size
sscanf(argv[3],"%d", &(scan.dx) );
// fourth is the number of scans
sscanf(argv[4],"%d", &(scan.total_scans) );
}

// if only one parameter supplied, it's the number of dithers.
else if(argc == 2)
sscanf(argv[1],"%d", &(scan.total_scans) );
// if no parameters supplied, ask.
else if(argc == 1)
{
// ask for the lower limit
sprintf(s,"          %d",scan.lower_limit);
if (quick_edit("Lower scan limit",s,s,NULL,INTEGER) == KEY_ESC)
return NOERROR;
sscanf(s,"%d", &(scan.lower_limit) );
// ask for the upper limit
sprintf(s,"          %d",scan.upper_limit);
if (quick_edit("Upper scan limit",s,s,NULL,INTEGER) == KEY_ESC)
return NOERROR;
sscanf(s,"%d", &(scan.upper_limit) );
// ask for the step size
sprintf(s,"          %d",scan.dx);
if (quick_edit("Scan step size",s,s,NULL,INTEGER) == KEY_ESC)
return NOERROR;
sscanf(s,"%d", &(scan.dx) );
// ask for the number of scans
sprintf(s,"          %d",scan.total_scans);
if (quick_edit("Number of scans",s,s,NULL,INTEGER) == KEY_ESC)
return NOERROR;
sscanf(s,"%d", &(scan.total_scans));
}

// otherwise print usage
else
{
error(ERROR,"Usage: function <start> <end> <dx> <scans>");
return -1;
}

```

```

//check the validity of the lower limit
if(scan.lower_limit < RTDAC_LOWER_LIMIT
|| scan.lower_limit > RTDAC_UPPER_LIMIT)
{
error(ERROR,
"Lower limit request of %d is out of bounds (%d .. %d).",
scan.lower_limit, RTDAC_LOWER_LIMIT,
RTDAC_UPPER_LIMIT);

scan.lower_limit = RTDAC_LOWER_LIMIT;
}

//check the validity of the upper limit
if(scan.upper_limit < RTDAC_LOWER_LIMIT
|| scan.upper_limit > RTDAC_UPPER_LIMIT)
{
error(ERROR,
"Upper limit request of %d is out of bounds (%d .. %d).",
scan.upper_limit, RTDAC_LOWER_LIMIT,
RTDAC_UPPER_LIMIT);

scan.upper_limit = RTDAC_UPPER_LIMIT;
}

//check the limits against each other
if(scan.lower_limit == scan.upper_limit)
{
error(ERROR,
"The lower and upper scan limits must be different!");

scan.lower_limit = RTDAC_LOWER_LIMIT;
scan.upper_limit = RTDAC_UPPER_LIMIT;
}

if(scan.lower_limit > scan.upper_limit)
{
error(ERROR,
"The lower limit should be LOWER than the upper limit. "
);

// switch the two values
temp = scan.lower_limit;
scan.lower_limit = scan.upper_limit;
scan.upper_limit = temp;
}

// check the step size
if(scan.dx > (scan.upper_limit - scan.lower_limit))
{
error(ERROR,
"The step size is larger than the scan range.");

if(MAX_DIFF > (scan.upper_limit - scan.lower_limit) )

scan.dx = 1;

else

scan.dx = MAX_DIFF;
}

// check the number of scans
if(scan.total_scans < 1)
{
error(MESSAGE,
"If total scans is < 1, endless scans requested.");

scan.total_scans = 0;
}

return setup_dither_scan(scan);

```

```

} /* call_setup_dither_scan() */

/*****
/* call_activate_dither_scan() */
***/
/* User callable function to activate a vis dither scan. */
/*****
int call_activate_dither_scan(int argc, char **argv)
{
    unsigned char arg;
    char s[81];
    int comm=0;

    // did the user specify start,stop, or reset?

    if(argc == 2)
    {
        strcpy(s,argv[1]);
    }

    else if(argc == 1)
    {
        if (quick_edit("start,stop,or reset?",
            dith_scan_commands[(int) comm],
            &comm,
            dith_scan_commands,
            ENUMERATED)
            == KEY_ESC)

        return -1;
    }

    // if the command line wasn't entered right, give usage

    else
    {
        error(ERROR,"Usage: function <start,stop,reset>");

        return -1;
    }

    if( strcmp( argv[1], "start" ) == 0
    || strcmp( argv[1], "START" ) == 0
    || strcmp( argv[1], "on" ) == 0
    || strcmp( argv[1], "ON" ) == 0 )

    arg = 1;

    else if( strcmp( argv[1], "reset" ) == 0
    || strcmp( argv[1], "RESET" ) == 0 )

    arg = 2;

    // default behavior is to stop the scan.

    else

    arg = 0;

    return activate_dither_scan(arg);
} /* call_activate_dither_scan()*/

/*****
/* set_picture_scale() */
***/
/* Sets globals picture_min and picture_max. */
/*****
int set_picture_scale(int argc, char **argv)
{
    char s[80];

    if (argc > 1)
    {
        sscanf(argv[1], "%i", &picture_min);
    }
    else

```

```

{
    sprintf(s,"%6.0f",picture_min);
    if (quick_edit("Picture scale minimum",
                  s,s,NULL,FLOAT) == KEY_ESC) return NOERROR;
    sscanf(s, "%f", &picture_min);
}

if (argc > 2)
{
    sscanf(argv[2], "%f", &picture_max);
}
else
{
    sprintf(s,"%6.0f",picture_max);
    if (quick_edit("Picture scale maximum",
                  s,s,NULL,FLOAT) == KEY_ESC) return NOERROR;
    sscanf(s, "%f", &picture_max);
}

return NOERROR;

} /* set_picture_scale() */

/*****
/* set_picture_display_type() */
** */
/* Sets global picture_display_type to LIN, LOG or EXP */
*****/

int set_picture_display_type(int argc, char **argv)
{
    char temp;
    int idx = 0;

    if (argc > 1)
    {
        // LIN, EXP, LOG are character values defined in nsimpleX.h

        // if the user asked for LIN... or lin...

        if(strncmp(argv[1],"LIN",3) == 0 ||
           strncmp(argv[1],"lin",3) == 0)
            temp = LIN;

        // if the user asked for LOG... or log...

        else if(strncmp(argv[1],"LOG",3) == 0 ||
                strncmp(argv[1],"log",3) == 0)
            temp = LOG;

        // if the user asked for EXP... or exp...

        else if(strncmp(argv[1],"EXP",3) == 0 ||
                strncmp(argv[1],"exp",3) == 0)
            temp = EXP;

        else
            return error(ERROR,
                        "Display type requested must be LIN, LOG, or EXP.");

    }
    else
    {
        clean_command_line();

        // Get the user's preference by having them selesct from a list
        if(quick_edit("Picture Display Type",
                    picture_display_types[idx],
                    &idx, picture_display_types, ENUMERATED) == KEY_ESC)
            return -1;

        clean_command_line();

        // assign temp's value by looking at the list index of the
        // user's selection
        // the list is defined at the top of this file.

        if(idx==2)
            temp = EXP;
        else if(idx == 1)

```

```

temp = LOG;
else
temp = LIN;
}

picture_display_type = temp;

return NOERROR;
} /* set_picture_display_type() */

```

B.3.7 arc_ccd_spect.c

```

//define SKIP_LOOP
/*****
/* arc_ccd_spect.c */
/*
/* Routines for looking at the dalsa data in the spectrograph. */
/*****
/*
/*          CHARA ARRAY USER INTERFACE */
/*          Based on the SUSI User Interface */
/*          In turn based on the CHIP User interface */
/*
/*          Center for High Angular Resolution Astronomy
/*          Mount Wilson Observatory, CA 91001, USA */
/*
/* Telephone: 1-626-796-5405
/* Fax       : 1-626-796-6717
/* email     : theo@chara.gsu.edu
/* WWW       : http://www.chara.gsu.edu
/*
/* (C) This source code and its associated executable
/* program(s) are copyright.
/*
/*****
/*
/* Author : Theo ten Brummelaar
/* Date   : 1998-1999
/* Modified from Dalsa to Arc_ccd by Chad Ogden, Oct 2002 */
/*****

extern "C" {
#include "arc_ccd.h"
#include <chara.h>
//include <dalsa.h>
#include <nrc.h>
}

// switch to keep compiler from looking at old code

#define OUTDATED_CODE

// comment this out to turn off debug messages.

//define DEBUG_MESSAGES

// comment this out to normalize the flat when making it

//define NORMALIZE_SPEC_FLAT_WHEN_MAKING

static XImage *ps_image = NULL;
static XImage *data_image = NULL;
float spec_flat[MAX_NUM_SIMULTANEOUS_SPECTRA][SPEC_SIZE];
float ps_flat[PS_SIZE];
static int num_pix_zero = 0;

// spec_row[0] tells how many rows, with row numbers starting at spec_row[1]
int spec_row[MAX_NUM_SIMULTANEOUS_SPECTRA+1];
int spec_row_offset = DEFAULT_SPEC_ROW_OFFSET;

// spec_stack[0][0] tells how many stacks. the first spec_stack[0][1] entries
// of spec_row[] make up the first stack. the next spec_stack[0][2] entries
// of spec_row[] make up the second stack, etc.
// Note the spec_row[] entries start at index 1, as spec_row[0] is
// the total number of rows.
// spec_stack[1][0] is unused.
// spec_stack[1][n] gives the index in spec_row[] where stack n starts.

```

```

int spec_stack[2][MAX_SPECTRUM_STACKS+1];

static bool auto_range_display = FALSE;
static float max_range = ARC_CCD_WELL_DEPTH;
static float spec_disp_max = ARC_CCD_WELL_DEPTH;
static float spec_disp_min = 0.0;
static float spect_cutoff=1.5;
int ps_mean_num = 1;
bool dps_running = FALSE;

// user side copies of arc_ccd_rt flags
bool use_spec_flat = FALSE;
bool spec_flat_has_been_set = FALSE;
bool use_ps_flat = FALSE;
bool ps_flat_has_been_set = FALSE;

// no longer using these dark routines
static bool dark_data = FALSE;
static float data_dark[FFT_SIZE];
static float ps_bias[PS_SIZE];
static bool debias_ps = FALSE;

//no longer in use. All fft done on arc_ccd_rt side
#ifdef OUTDATED_CODE

/*****
/* bb_power_spec() */
/* */
/* Get's the current bounding box, get's */
/* the row of data, calculates the power */
/* spectrum of that row. */
/* Returns this power spectrum. */
/* FFT always done in FFT_SIZE sized array so result always 0-PS_SIZE */
/* long. */
/* If no_fft set to TRUE no FFT is done and the result is FFT_SIZE long.*/
/* Returns NULL if something goes wrong. */
/* IF the no_fft flag is TRUE it simply returns the data untouched. */
*****/

float *bb_power_spec(bool no_fft)
{
    float *ps;
    float data[FFT_SIZE];
    int start_i,i,j;
    uint16_t *bb;
    int size;
    float total;

    /* What size is the thing? */

    size = bbdata.dx;
    start_i = (FFT_SIZE-size)/2;

    /* Go get the bounding box */

    if ((bb = get_bb()) == NULL) return NULL;

    /* OK, Create the data */

    for(i=0; i<start_i; i++) data[i] = 0;
    for(i=start_i+size; i<FFT_SIZE; i++) data[i] = 0;
    if (use_spec_flat)
    {
        total = 0.0;
        if (dark_data)
        {
            for(i=0; i<size; i++)
            {
                j = start_i + i;
                if ((data[j] = *(bb+spec_row*size+i) - data_dark[j]) < 0.0)
                data[j] = 0.0;
                total += data[j];
            }
        }
        else
        {
            for(i=0; i<size; i++)
            {
                total += (data[start_i+i] = *(bb+spec_row*size+i));
            }
        }
    }
}

```

```

    }

    for(i=0; i<size; i++)
    {
        j = i + start_i;
        if (data_flat_field[j] == 0)
        {
            data[j] = 0.0;
        }
        else
        {
            data[j] = (float)data[j]/
                (total*data_flat_field[j])-1.0;
        }
    }
}
else
{
    if (dark_data) for(i=0; i<size; i++)
    {
        j = start_i + i;
        if ((data[j] = (float)*(bb+spec_row*size+i) - data_dark[j])<0.0)
        data[j] = 0.0;
    }
    else for(i=0; i<size; i++)
    {
        data[i+start_i] = (float)*(bb+spec_row*size+i);
    }
}

free(bb);

/* If we are not doing an FFT just return the data */
if (no_fft)
{
    ps = vector(0,FFT_SIZE-1);
    for(i=0;i<FFT_SIZE;i++) ps[i] = data[i];
    return ps;
}

/* Do the FFT on the data */
realft(data-1, FFT_SIZE, 1);

/* Now pull out the power spectrum (forcing first few to be zero) */
ps = vector(0,PS_SIZE);
ps[0] = data[0] * data[0];
ps[PS_SIZE] = data[1]*data[1];
for(i=1;i<PS_SIZE;i++)
{
    j = i<<1;
    ps[i] = data[j]*data[j] + data[j+1] * data[j+1];
}
if (debias_ps)
{
    for(i=0;i<129;i++) ps[i] -= ps_bias[i];
}
if (num_pix_zero > 0) for(i=0; i<num_pix_zero; i++) ps[i] = 0.0;

return ps;
} /* bb_power_spec() */

#endif

/*****
/* get_power_spec() */
/*
/* Gets power spectrum data from arc_ccd_rt side. Returns pointer to */
/* the data. It is the responsibility of the function calling this */
/* one to free the pointer when done with it. */
*****/

float *get_power_spec(void)
{
    int size;
    float *ps;

```

```

time_t start;

size = PS_SIZE * sizeof(float);

if ((ps = (float *) malloc(size)) == NULL )
{
    error(ERROR,"Unable to allocate memory for power spectrum.");
    return NULL;
}

// send request to arc_ccd_rt side for ps data
send_rt_command(ARC_CCD_RT_REQ_PS,0);

    // Wait for rt side to put ps data into DATA_OUT_FIFO
    start=time(NULL);

        //wait for the readout time (MICROSEC_PER_PIXEL*pixels)
        // + exposure time in ms (arg must be in microsec so * 1000)

        usleep((bbdata.dx * bbdata.dy * MICROSEC_PER_PIXEL)
+ (exposure * 1000) );

        while( !char_waiting(fd_data_out) )
        {
            if( time(NULL) > start + 1 )
            {
                error(ERROR,"Timed out waiting for frame data.");
                free(ps);
                return NULL;
            }
            usleep(FIFO_POLLING_INTERVAL);
        }

// read ps data from DATA_OUT_FIFO
/* Get the data from FIFO 1 byte at a time or there may be trouble reading.
* if ((i=read_rt_data(ps,size)) != size)
*/

if (!read_arc_data((char *)ps, size))
{
    error(ERROR,"Failed to get ps data.");
    free(ps);
    return NULL;
}

#ifdef DEBUG_MESSAGES
error(MESSAGE,"First 3 bins of ps recieved: %f %f %f", *ps,
*(ps+1),*(ps+2));
#endif

return ps;

} // get_power_spec()

/*****/
/* get_spectrum() */
/* */
/* Gets spectrum data from arc_ccd_rt side. Returns pointer to */
/* the data. It is the responsibility of the function calling this */
/* one to free the pointer when done with it. */
/*****/

float *get_spectrum(void)
{
    int i;
    int size;
    float *spec;
    time_t start;

    size = FFT_SIZE * sizeof(float);

    if(spec_row[0] < 0)
    {
        error(ERROR,"No spectral rows defined yet.");
        return NULL;
    }

```



```

}

if ((spec = (float *) malloc(size*spec_row[0])) == NULL )
{
    error(ERROR,"Unable to allocate memory for spectrum.");
    return NULL;
}

// send request to arc_ccd_rt side for spectrum data
send_rt_command(ARC_CCD_RT_REQ_SPECTRUM,0);

    // Wait for rt side to put spec data into DATA_OUT_FIFO
start=time(NULL);

    //wait for the readout time (MICROSEC_PER_PIXEL*pixels)
    // + exposure time in ms (arg must be in microsec so * 1000)

//      usleep((bbdata.dx * bbdata.dy * MICROSEC_PER_PIXEL)
// + (exposure * 1000) );

    while( !char_waiting(fd_data_out) )
    {
        if( time(NULL) > start + 1 )
        {
            error(ERROR,"Timed out waiting for frame data.");
        }
    }

if(spec != NULL)
    free(spec);

    return NULL;
}
    usleep(FIFO_POLLING_INTERVAL);
}

// read spec data from DATA_OUT_FIFO
for(i=0; i<spec_row[0]; i++)
{
    if (!read_arc_data((char *) (spec+i*FFT_SIZE),size))
    {
        error(ERROR,"Failed to get spectrum data.");
        if(spec != NULL)
            free(spec);

        return NULL;
    }
}
return spec;

} // get_spectrum()

/*****
/* display_power_spec() */
/* */
/* Calls bb_power_spec and then displays the result in a window. */
/* Returns the power spectrum or NULL on error. */
/* If the bias is set to NULL it is ignored, if set to point to */
/* a vector of numbers these will be subtracted before display. This */
/* only happens if you are doing the power spectrum. no_fft bool like */
/* bb_power_spec call. */
*****/

float *display_power_spectrum(Window win, int width,
int height, bool no_fft)
{
    float *ps;
    char *picture;
    float min = spec_disp_min;
    float max = spec_disp_max;

    /* Create the picture */

    if (no_fft)
    {
        /* First, get the spectrum */

```

```

if ((ps = get_spectrum()) == NULL) return NULL;

if (auto_range_display)
{
    min = 0.0;
    max = 0.0;
}
else if (use_spec_flat)
{
    max = 1.05;
    min = -1.05;
}

// next block of code made obsolete by use of spec_disp_min
// and max global to this file

/* else
 * {
 *   spec_disp_max = max_range;
 *   spec_disp_min = 0.0;
 * }
 */
if (data_image != NULL) XDestroyImage(data_image);
if ((picture=plot_funct(width,height,ps-1,
FFT_SIZE,&min,&max))==NULL)
{

if( (ps) != NULL)
free(ps);

return NULL;
}
data_image = XCreateImage(theDisplay,theVisual,
theDepth,ZPixmap,0,
picture,width, height, theBitmapPad, 0);
XPutImage(theDisplay,win,theGC,data_image,0,0,0,0,width,height);
}
else
{

/* First, get the power spectrum */

if ((ps = get_power_spec()) == NULL) return NULL;

min = 0.0;
max = 0.0;
if (ps_image != NULL) XDestroyImage(ps_image);
if ((picture=plot_funct(width,height,ps-1,PS_SIZE,
&min,&max))==NULL)
{
    free(ps);
    return NULL;
}
ps_image = XCreateImage(theDisplay,theVisual,
theDepth,ZPixmap,0,
picture, width, height, theBitmapPad,0);
XPutImage(theDisplay, win, theGC,ps_image,0,0,0,0,width,height);
}

/* Now display it */

XMapWindow(theDisplay, win);
XFlush(theDisplay);

return ps;

} /* display_power_spectrum() */

/*****
/* call_display_power_spec() */
/* */
/* User callable version of display_power_spec. */
*****/

int call_display_power_spec(int argc, char **argv)
{
    Window ps_win;
    Window data_win;

```

```

char display_name[81];
int c;
float *ps;
#ifdef __NCURSES_H
    MEVENT mouse;          /* A mouse event. */
#endif

dps_running = TRUE;

/* Stop background movie of ccd if it is running */
if(bb_movie_running) bb_background_movie(0,NULL);

no_socket();

/* Warn the user about setting the row. */
if (spec_row[0] < 1)
error(MESSAGE,
"You may want to set the row.");

/* Create the windows */
if (display_is_open)
{
ps_win = openWindow("PS",theWidth-525,480, 512, 100);
data_win = openWindow("DATA",theWidth-525,610, 512, 100);
}

message(system_window,CONT_TEXT);
#ifdef SKIP_LOOP
while(1)
{
#ifdef
if (!display_is_open)
{
if (argc > 2)
{
strcpy(display_name,argv[2]);
}
else
{
if (default_display_machine == NULL)
{
sprintf(display_name,"%s:0.0",
get_machine_name(display_name));
}
else
{
sprintf(display_name,"%s:0.0",
default_display_machine);
}
if (quick_edit("Display name",display_name,
display_name,NULL,STRING) == KEY_ESC)
return NOERROR;
}
}

if (strcmpi(display_name,"null") == 0)
{
if (initX(NULL) == 0)
{
return error(ERROR,
"Failed to open display.");
}
}
else
{
if (initX(display_name) == 0)
{
return error(ERROR,
"Failed to open display %s.",
display_name);
}
}

/* ReCreate the windows */

ps_win = openWindow("PS",theWidth-525,480,512,100);
data_win = openWindow("DATA",theWidth-525,610,512,100);

```

```

}

background();
process_message_socket();

/* Are the parameters valid? */

if (exposure < 0)
{
return error(ERROR,"Set up the exposure time first.");
}

/* Get the frame */

if ((ps=display_power_spectrum(ps_win,512,100,FALSE))
==NULL)
{
clean_command_line();
XDestroyWindow(theDisplay, ps_win);
XDestroyWindow(theDisplay, data_win);
werase(system_window);
wrefresh(system_window);
return error(ERROR,"Failed to display power spectrum.");
}
free(ps);

if ((ps=display_power_spectrum(data_win,512,100,TRUE))
==NULL)
{
clean_command_line();
XDestroyWindow(theDisplay, ps_win);
XDestroyWindow(theDisplay, data_win);
werase(system_window);
wrefresh(system_window);
return error(ERROR,"Failed to display spectrum.");
}
free(ps);

if (kbhit())
{
c = get_command();
#ifdef __NCURSES_H
if (c == KEY_MOUSE)
{
getmouse(&mouse);

if (mouse.bstate == BUTTON1_CLICKED)
{
if (mouse.y == SYSTEM_BEGIN_Y &&
mouse.x < (int)strlen(CONT_TEXT)+2)
{
c = KEY_ESC;
}
}
}
#endif
if (c == KEY_ESC)
{
clean_command_line();
XDestroyWindow(theDisplay, ps_win);
XDestroyWindow(theDisplay, data_win);
werase(system_window);
wrefresh(system_window);
dps_running = FALSE;
return NOERROR;
}

command_processor(c);
}

#ifdef SKIP_LOOP
}
#endif

} /* call_display_power_spec() */

//no longer in use

/*****
/* power_spec_bias() */

```

```

/* */
/* calculates the power spectrum bias data. */
/*****

int power_spec_bias(int argc, char **argv)
{
    int i,j;
    float *ps;
    float n;
    char s[81];

    no_socket();

    /* Analyses the command line */

    if (argc > 1)
    {
        sscanf(argv[1], "%f", &n);
    }
    else
    {
        sprintf(s, "    10");
        if (quick_edit("Number of frames to average", s, s, NULL, INTEGER)
        == KEY_ESC) return NOERROR;
        sscanf(s, "%f", &n);
        clean_command_line();
    }

    /* First we set it to zero */

    for(i=0; i< PS_SIZE; i++) ps_bias[i] = 0;

    /* Now we cycle through things and make the average */

    for(i=0; i<n; i++)
    {
        if (exposure < 0)
        {
            return error(ERROR, "Set up the exposure time first.");
        }

        /* Get the frame */

        message(system_window, "Frame = %d", i+1);

        if ((ps = get_power_spec()) == NULL)
        {
            for(i=0; i< PS_SIZE; i++) ps_bias[i] = 0;
            return error(ERROR, "Failed... Bias zero.");
        }

        for(j=0; j<PS_SIZE; j++) ps_bias[j] += ps[j];

        free(ps);

        background();
    }

    for(i=0; i<PS_SIZE; i++) ps_bias[i] /= n;

    werase(system_window);
    wrefresh(system_window);

    return NOERROR;
} /* power_spec_bias() */

// no longer in use.  remove this functionality

/*****
/* toggle_ps_bias() */
/* */
/* Toggle on/off whether to use the PS bias or not. */
/*****

int toggle_ps_bias(int argc, char **argv)
{
    debias_ps = !debias_ps;

```

```

if (debias_ps)
{
message(system_window,"Using power spectrum bias frame.");
}
else
{
message(system_window,"Not using power spectrum bias frame.");
}

return NOERROR;

} /* toggle_ps_bias() */

/*****
 * set_num_pix_zero() */
 */
/* Sets the number of pixel to force to zero in the power spectrum. */
*****/

int set_num_pix_zero(int argc, char **argv)
{
int new_num_pix_zero;
char s[81];

if (argc > 1)
{
sscanf(argv[1],"%d",&new_num_pix_zero);
}
else
{
sprintf(s,"%3d",num_pix_zero);
if (quick_edit("Number of pixels to zero",s, s,NULL,INTEGER)
== KEY_ESC) return NOERROR;
sscanf(s,"%d",&new_num_pix_zero);
clean_command_line();
}

if (new_num_pix_zero < 0 || new_num_pix_zero > PS_SIZE)
return error(ERROR,"Number of pixels must be bewteen 0 and %d.",
PS_SIZE);

num_pix_zero = new_num_pix_zero;

// send this to the arc_ccd_rt side

if (write_rt_data(&num_pix_zero,sizeof(num_pix_zero))
!= sizeof(num_pix_zero))
{
return error(ERROR,"Unable to write num_pix_zero to rt side.");
}

// send the command

send_rt_command(ARC_CCD_RT_SET_SUPPRESS_PIXELS,0);

return NOERROR;

} /* set_num_pix_zero() */

/*****
 * call_set_spec_row() */
 */
/* Sets which row to look for the spectrum. If an argument is given */
/* that's what is used. If no argument it looks for the brightest */
/* pixel and asks if you want to use that row. */
/* Always checks to see if this operation is actually possible. */
/* Command line inputs: #rows <row numbers> <offset.>*/
*****/

int call_set_spec_row(int argc, char **argv)
{
uint16_t *bb,max[MAX_NUM_SIMULTANEOUS_SPECTRA],max_temp,*p;
int max_row[MAX_NUM_SIMULTANEOUS_SPECTRA];
int size_x, size_y;
int i,j,k,l;
int row[MAX_NUM_SIMULTANEOUS_SPECTRA+1];
int row_offset = spec_row_offset;
int temp;
bool max_filed = FALSE;
char s[81],s2[81];

```

```

// kill the flat if there is one
use_spec_flat = FALSE;
set_rt_flag(FLAG_ID_USE_SPEC_FLAT,FALSE);

spec_flat_has_been_set = FALSE;
set_rt_flag(FLAG_ID_SPEC_FLAT_HAS_BEEN_SET,FALSE);

use_ps_flat = FALSE;
set_rt_flag(FLAG_ID_USE_PS_FLAT,FALSE);

ps_flat_has_been_set = FALSE;
set_rt_flag(FLAG_ID_PS_FLAT_HAS_BEEN_SET,FALSE);

/* How big is the bounding box? */
size_x = bbdata.dx;
size_y = bbdata.dy;

/* Analyses the command line */

if (argc > 1)
{
sscanf(argv[1],"%d",&row[0]);
}

// get the number of rows if the user has not specified
else
{
clean_command_line();
sprintf(s,"%3d",DEFAULT_NUM_OF_SPECTRA);
if (quick_edit("Number of spectral rows",s, s,NULL,INTEGER)
    == KEY_ESC) return NOERROR;
sscanf( s, "%d", &(row[0]) );
clean_command_line();
}

// if the user input the row numbers and the offset
if (argc == row[0] + 3)
{
//get the row numbers
for(i=1; i<=row[0]; i++)
{
//get the next command line value
sscanf(argv[i+1],"%d",&row[i]);
}

// get the offset
sscanf(argv[i+row[0]+1],"%d",&temp);
}

//if the user input the row numbers but no offset
else if(argc == row[0] + 2)
{
//get the row numbers
for(i=1; i<=row[0]; i++)
{
//get the next command line value
sscanf(argv[i+1],"%d",&row[i]);
}
}

else if(ask_yes_no("", "To you want me to pick the brightest rows?"))
{
/* Go get the bounding box */

if ((bb = get_bb()) == NULL)
return error(ERROR,"Failed to get bounding box.");

// This is the old way of finding the max pixel.
// It only can handle 1 row.
// /* Find the maximum pixel */

```

```

//
// max_y = 0;
// for(max = 0, p = bb, j=0; j<size_y; j++)
// for(i=0; i<size_x; i++)
// {
// if (*p > max)
// {
// max = *p;
// max_y = j;
// }
// p++;
// }

// zero out the arrays for ranking rows in order of brightness

for(i=0; i<MAX_NUM_SIMULTANEOUS_SPECTRA; i++)
{
max_row[i] = -1;
max[i] = 0;
}

// this for loop finds the row[0] brightest pixel rows and ranks them
// row max pixel values in descending order in max[]
// row numbers of these pixels in max_row[]

for(j=0; j<size_y; j++)
{
// move the pointer to the beginning of the row
p = bb+j*size_x;

//set the row max to zero
max_temp = 0;

// move along the row, finding the max pixel value
for(i=0; i<size_x; i++)
{
if(*p > max_temp)
{
max_temp = *p;
}

p++;
} //done finding row max

//now check the row maximum vs the list
max_filed = FALSE;
for(k=0; k<row[0]; k++)
{
if(max_temp > max[k] && !max_filed)
{
//cascade max values down the array
// same with max_row array
// and set max_filed to TRUE

for(l=row[0]-1; l>k; l--)
{
max[l]=max[l-1];
max_row[l]=max_row[l-1];
}
max[k]=max_temp;
max_row[k]=j;
max_filed=TRUE;
}
} //done checking row max vs. max array
} // done ranking row[0] rows according to brightest pixel

for(i=0; i<row[0]; i++)
{
clean_command_line();
sprintf(s,"%3d",max_row[i]);
sprintf(s2,
"Row #%d,ranked according to pixel brightness",
i+1);
if (quick_edit(s2,s, s,NULL,INTEGER)
== KEY_ESC) return NOERROR;
sscanf(s,"%d",&(row[i+1]));
clean_command_line();
}
}

```



```

clean_command_line();
sprintf(s,"%3d",row_offset);
if (quick_edit("Offset from outside edge of chip",
s, s,NULL,INTEGER) == KEY_ESC) return NOERROR;
sscanf(s,"%d",&row_offset);
        clean_command_line();
}

//otherwise use defaults
else
{
    //>>>..kludge
    row[0] = 2;
    row[1] = 4;
    row[2] = 6;
    row[3] = 8;
    row[4] = 10;
    row[5] = 12;
    row[6] = 14;
}

/* OK, is the answer reasonable? */
for(i=1; i<=row[0]; i++)
{
    if (row[i] < 0 || row[i] > (size_y - 1) )
    {
        return error(ERROR,"Row value outside of bounding box.");
    }
}
if (row_offset + SPECTRUM_WIDTH > bbdata.dx || row_offset < 0)
{
    return error(ERROR,
"Row offset puts spectrum window outside quadrant.");
}

//>>>>>do this properly later, for more stacks than 1
//set spectrum stacks

spec_stack[0][0] = 1;
spec_stack[0][1] = row[0];
spec_stack[i][1] = 1;

// call set_spec_row()

if( ( i=set_spec_row(&row[0],row_offset) ) == NOERROR)
{
    for(j=1; j<=row[0]; j++)
    {
        if(j==1)
            sprintf(s," %2d", row[j]);
        else
        {
            sprintf(s2," %2d", row[j]);
            strcat(s,s2);
        }
    }

    message(system_window,"%2d Rows: %s Offset: %d", row[0],s,
spec_row_offset);
}

return i;
} /* call_set_spec_row() */

/*****
/* set_spec_row() */
/* */
/* Sends spectrum row to the rt side. */
*****/

int set_spec_row(int *row, int row_offset)
{
    int i;

        // Send it to the arc_ccd_rt side

        if (write_rt_data(row,sizeof(row)*(MAX_NUM_SIMULTANEOUS_SPECTRA+1))
!= sizeof(row)*(MAX_NUM_SIMULTANEOUS_SPECTRA+1))
        {

```

```

        return error(ERROR,"Unable to write row array to rt side.");
    }

    if (write_rt_data(&row_offset,sizeof(row_offset)) != sizeof(row_offset))
    {
        return error(ERROR,"Unable to write row_offset to rt side.");
    }
    // send the command

    send_rt_command(ARC_CCD_RT_SET_ROW,0);

/* Set the new value */

spec_row_offset = row_offset;

// copy the values in row[], which is temporary, into spec_row[],
// which is the one everything else will refer to.

for(i=0; i<=MAX_NUM_SIMULTANEOUS_SPECTRA; i++)
{
*(spec_row+i)=*(row+i);
}

    return NOERROR;

}/* set_spec_row()*/

/*****
/* make_spec_flat() */
/* */
/* Calculates the flat field for the data. */
*****/

int make_spec_flat(int argc, char **argv)
{
    int i,j,k;
    float *spec;
    float n;
    char s[81];
    float total[MAX_NUM_SIMULTANEOUS_SPECTRA];
    bool old_use_fft;

    no_socket();

/* Analyses the command line */

    if (argc > 1)
    {
        sscanf(argv[1],"%f",&n);
    }
    else
    {
        sprintf(s,"      20");
        if (quick_edit("Number of frames to average",s, s,NULL,INTEGER)
        == KEY_ESC) return NOERROR;
        sscanf(s,"%f",&n);
        clean_command_line();
    }

    if(n<1)
    {
        return error(ERROR,"Number of flat frames must be > 0.");
    }

/* First we set it to zero */

    for(j=0; j<spec_row[0]; j++)
    {
        for(i=0; i< FFT_SIZE; i++) spec_flat[j][i] = 0;
    }

/* Now we cycle through things and make the average */

    use_spec_flat = FALSE;
    set_rt_flag(FLAG_ID_USE_SPEC_FLAT,FALSE);

    // turn off FFTs for flatting
    old_use_fft = use_fft;
    use_fft = FALSE;
    set_rt_flag(FLAG_ID_USE_FFT,FALSE);

```

```

for(k=0; k<n; k++)
{
/* Are the parameters valid? */

if (exposure < 0)
{
return error(ERROR,"Set up the exposure time first.");
}

/* Get the frame */

message(system_window,"Frame = %d",k+1);

if ((spec = get_spectrum()) == NULL)
{
for(j=0; j<spec_row[0]; j++)
{
for(i=0; i< FFT_SIZE; i++) spec_flat[j][i] = 0;
}
return error(ERROR,"Failed... Flat field zero.");
}

for(j=0; j<spec_row[0]; j++)
{
for(i=0; i<FFT_SIZE; i++) spec_flat[j][i] +=
*(spec+j*FFT_SIZE+i);
}
free(spec);

background();
}

for(j=0; j<spec_row[0]; j++)
total[j] = 0;

#ifdef NORMALIZE_SPECTRUM_FLAT_WHEN_MAKING
for(j=0; j<spec_row[0]; j++)
{
for(i=0; i<FFT_SIZE; i++)
{
total[j] += spec_flat[j][i];
}
}

for(j=0; j<spec_row[0]; j++)
{
for(i=0; i<FFT_SIZE; i++)
{
if (spec_flat[j][i] <= n*spect_cutoff)
spec_flat[j][i] = 0.0;
else
spec_flat[j][i] /= total;
}
}
#else //skip normalization, just average.
for(j=0; j<spec_row[0]; j++)
{
for(i=0; i<FFT_SIZE; i++)
spec_flat[j][i] /= n;
}
#endif

//turn FFT back to value before function was called
use_fft = old_use_fft;
set_rt_flag(FLAG_ID_USE_FFT,use_fft);

// send spectrum flat to rt side

write_rt_data(spec_flat,spec_row[0]*FFT_SIZE*sizeof(float));
send_rt_command(ARC_CCD_RT_SET_SPEC_FLAT,0);

spec_flat_has_been_set = TRUE;
set_rt_flag(FLAG_ID_SPEC_FLAT_HAS_BEEN_SET,TRUE);

use_spec_flat = TRUE;
set_rt_flag(FLAG_ID_USE_SPEC_FLAT,TRUE);

message(system_window,"Using data flat field frame.");

```

```

werase(system_window);
wrefresh(system_window);

return NOERROR;

} /* make_spec_flat() */

/*****
/* toggle_spec_flat() */
/* */
/* Toggle on/off whether to use the flat or not. */
*****/

int toggle_spec_flat(int argc, char **argv)
{
if(spec_flat_has_been_set)
{
use_spec_flat = !use_spec_flat;
set_rt_flag(FLAG_ID_USE_SPEC_FLAT,use_spec_flat);

if (use_spec_flat)
{
message(system_window,"Using spectrum flat field frame.");
}
else
{
message(system_window,
"Not using spectrum flat field frame.");
}
return NOERROR;
}
else
{
return error(ERROR,"Spectrum flat has not been set. Make flat first.");
}

} /* toggle_spec_flat() */

/*****
/* make_ps_flat() */
/* */
/* Calculates the flat field for the power spectrum data. */
*****/

int make_ps_flat(int argc, char **argv)
{
int i,j;
float *ps;
float n;
char s[81];
float total;

no_socket();

/* Analyses the command line */

if (argc > 1)
{
sscanf(argv[1],"%f",&n);
}
else
{
sprintf(s,"      20");
if (quick_edit("Number of frames to average",s, s,NULL,INTEGER)
== KEY_ESC) return NOERROR;
sscanf(s,"%f",&n);
clean_command_line();
}

if(n<1)
{
return error(ERROR,"Number of flat frames must be > 0.");
}

/* First we set it to zero */

for(i=0; i< PS_SIZE; i++) ps_flat[i] = 0;

```

```

/* Now we cycle through things and make the average */

use_ps_flat = FALSE;
set_rt_flag(FLAG_ID_USE_PS_FLAT,FALSE);

for(i=0; i<n; i++)
{
/* Are the parameters valid? */

if (exposure < 0)
{
return error(ERROR,"Set up the exposure time first.");
}

/* Get the frame */

message(system_window,"Frame = %d",i+1);

if ((ps = get_power_spec()) == NULL)
{
for(i=0; i< PS_SIZE; i++) ps_flat[i] = 0;
return error(ERROR,"Failed... PS flat zeroed.");
}

for(j=0; j<PS_SIZE; j++) ps_flat[j] += ps[j];

free(ps);

background();
}

total = 0;

//average.
for(i=0; i<PS_SIZE; i++)
ps_flat[i] /= n;

// send ps flat to rt side

write_rt_data(ps_flat,PS_SIZE*sizeof(float));
send_rt_command(ARC_CCD_RT_SET_PS_FLAT,0);

ps_flat_has_been_set = TRUE;
set_rt_flag(FLAG_ID_PS_FLAT_HAS_BEEN_SET,TRUE);

use_ps_flat = TRUE;
set_rt_flag(FLAG_ID_USE_PS_FLAT,TRUE);

message(system_window,"Using power spectrum flat.");

werase(system_window);
wrefresh(system_window);

return NOERROR;
}

// make_ps_flat()

/*****
/* toggle_ps_flat() */
**/
/* Toggle on/off whether to use the power spectrum flat or not. */
*****/

int toggle_ps_flat(int argc, char **argv)
{
if(ps_flat_has_been_set)
{
use_ps_flat = !use_ps_flat;
set_rt_flag(FLAG_ID_USE_PS_FLAT,use_ps_flat);

if (use_ps_flat)
{
message(system_window,"Using power spectrum flat.");
}
else
{
message(system_window,

```

```

"Not using power spectrum flat.");
}
return NOERROR;
}
else
{
return error(ERROR,
"Power spectrum flat has not been set. Make flat first.");
}

} /* toggle_ps_flat() */

// this functionality is no longer valid. Darking is taken care of
// by the "bias" functions on the arc_ccd_rt side.

/*****
/* calc_data_dark() */
/* */
/* Calculates the dark for the data. */
*****/

int calc_data_dark(int argc, char **argv)
{
int i,j;
float *ps;
float n;
char s[81];
bool old_use_spec_flat;
no_socket();

/* Analyses the command line */

if (argc > 1)
{
sscanf(argv[1],"%f",&n);
}
else
{
sprintf(s,"      20");
if (quick_edit("Number of frames to average",s, s,NULL,INTEGER)
== KEY_ESC) return NOERROR;
sscanf(s,"%f",&n);
clean_command_line();
}

/* First we set it to zero */

for(i=0; i< FFT_SIZE; i++) data_dark[i] = 0;

/* Now we cycle through things and make the average */

old_use_spec_flat = use_spec_flat;
dark_data = FALSE;
use_spec_flat = FALSE;
set_rt_flag(FLAG_ID_USE_SPEC_FLAT,FALSE);
for(i=0; i<n; i++)
{
/* Are the parameters valid? */

if (exposure < 0)
{
return error(ERROR,"Set up the exposure time first.");
}

/* Get the frame */

message(system_window,"Frame = %d",i+1);

if ((ps = get_power_spec()) == NULL)
{
for(i=0; i< FFT_SIZE; i++) data_dark[i] = 0;
return error(ERROR,"Failed... Dark zero.");
}

for(j=0; j<FFT_SIZE; j++) data_dark[j] += ps[j];

free(ps);

background();
}

```

```

for(i=0; i< FFT_SIZE; i++) data_dark[i] /= (float)n;
dark_data = TRUE;
use_spec_flat = old_use_spec_flat;
set_rt_flag(FLAG_ID_USE_SPEC_FLAT,use_spec_flat);
message(system_window,"Using data dark frame.");

werase(system_window);
wrefresh(system_window);

return NOERROR;

} /* calc_data_dark() */

// no longer valid

/*****
/* toggle_data_dark() */
*/
/* Toggle on/off whether to use the PS bias or not. */
*****/

int toggle_data_dark(int argc, char **argv)
{
dark_data = !dark_data;

if (dark_data)
{
message(system_window,"Using data dark frame.");
}
else
{
message(system_window,"Not using data dark frame.");
}

return NOERROR;

} /* toggle_data_dark() */

/*****
/* toggle_auto_range() */
*/
/* Toggle on/off whether to auto range the display. */
*****/

int toggle_auto_range(int argc, char **argv)
{
auto_range_display = !auto_range_display;

if (auto_range_display)
{
message(system_window,"Auto display range ON.");
}
else
{
message(system_window,"Auto display range OFF.");
}

return NOERROR;

} /* toggle_auto_range() */

//is this redundant?

/*****
/* set_spect_cutoff() */
*/
/* Sets the number of pixel to force to zero in the power spectrum. */
*****/

int set_spect_cutoff(int argc, char **argv)
{
float new_spect_cutoff;
char s[81];

if (argc > 1)
{
sscanf(argv[1],"%f",&new_spect_cutoff);
}
else
{

```

```

sprintf(s,"%5f",spect_cutoff);
if (quick_edit("Spectrum intensity cutoff",s, s,NULL,FLOAT)
== KEY_ESC) return NOERROR;
sscanf(s,"%f",&new_spect_cutoff);
        clean_command_line();
}

if (spect_cutoff < 0 || spect_cutoff > 254)
    return error(ERROR,"Spectrum cutoff must be bewteen 0 and 254.");

spect_cutoff = new_spect_cutoff;

return NOERROR;

} /* set_spect_cutoff() */

/*****
/* set_ps_mean_num() */
/* */
/* Sets the number of power spectra to use in the mean power spectrum. */
*****/

int set_ps_mean_num(int argc, char **argv)
{
    int new_ps_mean_num;
    char s[81];

    if (argc > 1)
    {
        sscanf(argv[1],"%d",&new_ps_mean_num);
    }
    else
    {
        sprintf(s,"%5d",ps_mean_num);
        if (quick_edit("Number of power spectra to use in mean:"
,s, s,NULL,INTEGER) == KEY_ESC)
            return NOERROR;
        sscanf(s,"%d",&new_ps_mean_num);
        clean_command_line();
    }

    if (new_ps_mean_num < 1 || new_ps_mean_num > MAX_POW_SPECTRA_IN_MEAN)
        return error(ERROR,
"Number of spectra must be bewteen 1 and %d. Try again.",
MAX_POW_SPECTRA_IN_MEAN);

    ps_mean_num = new_ps_mean_num;

    // send this to the arc_ccd_rt side

    if (write_rt_data(&ps_mean_num,sizeof(ps_mean_num))
!= sizeof(ps_mean_num))
    {
        return error(ERROR,"Unable to write ps_mean_num to rt side.");
    }

    // send the command

    send_rt_command(ARC_CCD_RT_SET_PS_MEAN_NUM,0);

    return NOERROR;

} /* set_ps_mean_num() */

/*****
/* request_purge_ps() */
/* */
/* Ask rt side to purge the ps_buf. */
*****/

int request_purge_ps(int argc, char **argv)
{
    send_rt_command(ARC_CCD_RT_REQ_PURGE_PS_BUF,0);
    return 0;

} // request_purge_ps()

/*****
/* change_spec_display_range() */

```



```

/* */
/* Function which changes the display range for the spec display window.*/
/*****
int change_spec_display_range(float min, float max)
{
    float temp;

    // make sure both values make sense
    if (min < 0)
    {
        min = 0.0;
        error(MESSAGE,
        "Requested min display value out of range. Reset to $f", min);
    }

    if (max < 0)
    {
        max = 0.0;
        error(MESSAGE,
        "Requested max display value out of range. Reset to $f", max);
    }

    if (max > max_range)
    {
        max = max_range;
        error(MESSAGE,
        "Requested max display value out of range. Reset to $f", max);
    }

    if (min > max_range)
    {
        min = max_range;
        error(MESSAGE,
        "Requested min display value out of range. Reset to $f", min);
    }

    if (min > max)
    {
        temp = min;
        min = max;
        max = temp;
    }

    if (min == max)
    {
        min = 0.0;
        max = max_range;
        error(MESSAGE,"Min must not equal max.  Values reset to full range.");
    }

    spec_disp_min = min;
    spec_disp_max = max;

    return NOERROR;
} /* change_spec_display_range() */

/*****
/* call_change_spec_display_range() */
/* */
/* User callable function to change the display range for the spec */
/* window. One parameter on command line assumes it's the max value. */
/*****
int call_change_spec_display_range(int argc, char **argv)
{
    float min,max;
    char s[81];

    if (argc < 2)
    {
        // ask for min

        sprintf(s,"%f",spec_disp_min);

        if ( quick_edit("Min display value: ",s,s,NULL,FLOAT)
        == KEY_ESC )
            return NOERROR;

        sscanf(s,"%f",&min);

```

```

// ask for max

sprintf(s,"%f",spec_disp_max);

if ( quick_edit("Max display value: ",s,s,NULL,FLOAT)
== KEY_ESC )
return NOERROR;

sscanf(s,"%f",&max);
}

else if (argc == 2)
{
if(strcmp(argv[1],"reset") == 0)
{
min = 0.0;
max = max_range;
}

else
{
min = spec_disp_min;
sscanf(argv[1],"%f",&max);
}
}

else if (argc == 3)
{
sscanf(argv[1],"%f",&min);
sscanf(argv[2],"%f",&max);
}

else
{
error(ERROR,"usage: function <<min>> <max>");
return -1;
}

return change_spec_display_range(min,max);
} /* call_change_spec_display_range */

```

B.4 The Float ARCTAN Lookup Function For Real-Time Use

B.4.1 f_arctanlookup.h

```

/*****
/* f_arctanlookup.h */
*/
/*****
#include "f_tantable.h"

//This controls whether arctanlookup is using outputting radians or degrees.
//If radians, Max_ANGLE = M_PI/2.0. M_PI is defined in f_tantable.h
//If degrees, MAX_ANGLE = 180.
#define ARCTANLOOKUP_MAX_ANGLE M_PI/2.0

float f_arctanlookup(float y, float x);
float f_arctan_recurse(float x, int pow_2, int index);

```

B.4.2 f_arctanlookup.c

```

/*****
/* f_arctanlookup.c */
*/
/* f_arctanlookup(): Wrapper function which takes two float arguments,*/
/* y,x. Returns arctan(y/x) by calling f_arctan_recurse(). Because y */
/* and x are given separately, the range of arctanlookup() is -pi..pi */
/* instead of -pi/2..pi/2. */
*/
/* f_arctan_recurse(): */
/* Recursive function which uses the table f_tantable[] defined in */
/* f_tantable.h. The table of values is evenly spaced in the */

```

```

/* dependent variable, which means that the output values of */
/* arctan() are sampled evenly, but the input values are not. */
/* If the inputs were sampled evenly, the output would be under */
/* sampled for low valued inputs. */
/* f_tantable[] is of size 2^N where N=TAN_TABLE_POW2 defined in */
/* f_arctantable.h. The values stored in the table are the */
/* _input_ values to arctan(). The result is = */
/* = Pi/2 * index/(2^N -1). */
/* This function calls itself N times to execute a binary search */
/* of f_tantable[] to find which two input values x lies between. */
/* When it has found the number tantab[index] which x is closest */
/* to but greater than, it returns the value of */
/* Pi/2 * index/(2^N -1). */
/* */
/*****
/*
/*      Center for High Angular Resolution Astronomy
/* Georgia State University, Atlanta, GA 30303-3083, U.S.A.
/*
/* Telephone: 1-626-796-4130
/* Fax       : 1-626-796-6717
/* email    : ogden@chara-array.org
/* WWW      : http://www.chara.gsu.edu/~ogden/ogden.html
/* (C) This source code and its associated executable program(s) are
/* copyright.
/*
/*
/*****
/* Author: Chad Ogden */
/* Date: May 2002 */
/*****
#include "f_arctanlookup.h"

/*****
/* f_arctanlookup() */
/* */
/* Wrapper function for recursive arctan lookup function. Takes two */
/* arguments, y and x, and returns arctan(y/x). Is able to unwrap phase*/
/* from -Pi .. Pi. */
/*****

float f_arctanlookup(float y, float x)
{
float arg;
float theta;

//protect against divided by zero with y/x
//and cover situation where phasor is vertical, horizontal(<-save time)
if(x==0 && y>0) return M_PI/2.0;
else if(x==0 && y<0) return -M_PI/2.0;
else if (x<0 && y==0) return M_PI;
else if (x>=0 && y==0) return 0;
else arg = y/x;

//call recursive arctan lookup function
theta = f_arctan_recurse(arg,TAN_TABLE_POW2,0);

//figure out which quadrant the phasor x+iy is in.
//f_arctan_recurse will handle only the -pi/2..pi/2 range.
//need to wrap for quadrants when x is negative.
if(x<0)
{
//upper left quadrant, add pi.
if(y>0) return (theta + M_PI);

//lower left quadrant, subtract pi.
if(y<0) return (theta - M_PI);
}
//this happens if x is not negative, the right two quadrants.
//the recursive function normally gives back theta in this range.
return theta;
}

/*****
/* f_arctan_recurse() */
/* */
/* IMPORTANT: When you call this function from a program, use */
/* TAN_TABLE_POW2 as the second argument, and 0 as the third. */
/* */
/* finds where x sits in the f_tantable[] by determining its index. */

```

```

/* index is built one bit at a time, starting with highest order bit. */
/* */
/* Ex: if TAN_TABLE_POW2=6 then index will have 6 useful bits to */
/* describe where x could lie in the table. */
/* If x is greater than whatever is at testindex 100000, then index is */
/* set equal to 100000 and the function calls itself to try x vs. */
/* whatever is at testindex 110000. */
/* If x is less than what's at testindex 100000, then index remains */
/* 000000 and the function calls itself to compare x with what's at */
/* testindex 010000. */
/* This pattern is followed until index has been determined down to the */
/* least significant bit. */
/*****

float f_arctan_recurse(float x, int pow_2, int index)
{
//kluge to take care of arctan(-x)=-arctan(x)
int sign;

//The next power of 2 down from pow_2
int powstep=pow_2 -1;

//intstep = 2^powstep.
int intstep=1<<powstep;

//which array element to test x vs. this iteration.
int testindex=index+intstep;

//if x is negative, use arctan(-x)=-arctan(x)
if(x<0)
{
//change sign of x
x = -x;

//make sign out front negative
sign =-1;
}
else
{
//sign out front is normally positive
sign =1;
}

//if x is >= the value at testindex, set index=testindex.
if(x >= f_tantable[testindex])
{
index = testindex;
}

//if powstep <=0, then the lowest order bit of index has been
//determined and we're ready to return the answer.
if(powstep <= 0)
{
//arctan(x)=index/tablelength * max angle
//max angle is either Pi/2 or 180 depending on whether
//you want radians or degrees output. This is
//defined in f_arctanlookup.h
return ARCTANLOOKUP_MAX_ANGLE*(float)index/
(float)((1<<TAN_TABLE_POW2)-1);
}

//if powstep > 0, then index is not fully determined yet.
//call this function recursively to determine the next lower
//bit of index and return a value once the lowest order bit
//has been determined.
else
{
//sign is out front to handle arctan(-x)=-arctan(x)
return sign*f_arctan_recurse(x,powstep,index);
}

}/*end f_arctan_recurse()*/

```

B.4.3 f_mktantable.c

```

/*****
/* f_mktantable.c */

```

```

/* creates the file f_tantable.h, to be included as */
/* a float lookup table for the tan or arctan function. */
/* Trig functions aren't available in Linux kernel space, so this */
/* is necessary for modules which need tan() or arctan(). */
/* When executing f_mktantable, pass the table size at the */
/* command line. The table is the value of tan() from inputs of */
/* [0,Pi/2). */
/* Negative values of tan() can be found using tan(-a) = -tan(a) */
/* and by phase wrapping the input angle to (-Pi/2,Pi/2) */
/*****
/*
/*      Center for High Angular Resolution Astronomy
/*      Georgia State University, Atlanta, GA 30303-3083, U.S.A.
/*
/*      Telephone: 1-626-796-4130
/*      Fax       : 1-626-796-6717
/*      email    : ogden@chara-array.org
/*      WWW      : http://www.chara.gsu.edu/~ogden/ogden.html
/*      (C) This source code and its associated executable program(s) are
/*      copyright.
/*
*****/
/* */
/* Author: Chad Ogden, Based on code by Theo ten Brummelaar. */
/* Date: May 2002 */
*****/

#include <stdio.h>
#include <math.h>
#define FILENAME "f_tantable.h"
#define ENTRIES_PER_LINE 8
#define PI_DECIMAL_PLACES 20

int main(int argc, char **argv)
{
    int i,length,pow2;
    FILE *outfile;

    /*Prints usage if wrong number of arguments*/
    if (argc != 2)
    {
        fprintf(stderr,"usage:f_mktantable <table length power of 2>\n");
        exit(-1);
    }

    /*reads in first argument on command line after program name.*/
    sscanf(argv[1],"%d",&pow2);

    //Table length will be 2^pow2
    length = 1<<pow2;

    /*opens file FILENAME for writing/overwriting. */
    outfile = fopen(FILENAME,"w");

    /*if the file was unable to open, exits.*/
    if (outfile == NULL)
    {
        fprintf(stderr,"Can't open file for writing. Exiting program.");
        exit(-1);
    }

    fprintf(outfile,"/*****/\n");
    fprintf(outfile,"/*  */");
    fprintf(outfile,FILENAME);
    fprintf(outfile," */\n");
    fprintf(outfile,"/* Do not edit this file directly. */\n");
    fprintf(outfile,"/* It is generated by f_mktantable. */\n");
    fprintf(outfile,"/*****/\n\n");

    /* define length = (value stored in length) for programs using f_sintable.h */
    fprintf(outfile,"#define TAN_TABLE_POW2 %d\n",pow2);
    fprintf(outfile,"#define TAN_TABLE_LENGTH %d\n",length);
    fprintf(outfile,"#ifndef M_PI\n");
    fprintf(outfile,"#define M_PI %.*f\n",PI_DECIMAL_PLACES,M_PI);
    fprintf(outfile,"#endif\n");
    /*Begin definition of f_tantable[]={x0,x1,...,xlength-1}; */
    fprintf(outfile,"float f_tantable[]={");

    /*This loop makes entries for f_tantable[], from 0 to length-1/length*M_PI/2 */
    for (i=0; i<length; i++)

```

```

{
fprintf(outfile,"%f", (float)tan(((double)i)/((double)length)*M_PI/2.0));
/*Put commas after every number in the table except the last one */
if(i<length-1) fprintf(outfile," ");

/*Put a newline every ENTRIESPERLINE entries*/
if(i%ENTRIES_PER_LINE==0) fprintf(outfile,"\n");
}

/*End the definition of f_tantable[]={x0,x1,...,xlength-1}; */
fprintf(outfile,"};\n");
exit(0);
}

```

B.4.4 f_tantable.h

```

/*****
/* f_tantable.h */
/* Do not edit this file directly. */
/* It is generated by f_mktantable. */
*****/

#define TAN_TABLE_POW2 7
#define TAN_TABLE_LENGTH 128
#ifndef M_PI
#define M_PI 3.14159265358979311600
#endif
float f_tantable[]={0.000000,
0.012272, 0.024549, 0.036832, 0.049127, 0.061436, 0.073764, 0.086115, 0.098491,
0.110898, 0.123338, 0.135816, 0.148336, 0.160901, 0.173516, 0.186185, 0.198912,
0.211702, 0.224558, 0.237484, 0.250487, 0.263570, 0.276737, 0.289995, 0.303347,
0.316799, 0.330355, 0.344023, 0.357806, 0.371710, 0.385743, 0.399908, 0.414214,
0.428665, 0.443270, 0.458034, 0.472965, 0.488070, 0.503358, 0.518835, 0.534511,
0.550394, 0.566493, 0.582817, 0.599377, 0.616182, 0.633243, 0.650571, 0.668179,
0.686077, 0.704279, 0.722799, 0.741651, 0.760848, 0.780408, 0.800345, 0.820679,
0.841426, 0.862606, 0.884239, 0.906347, 0.928952, 0.952079, 0.975753, 1.000000,
1.024850, 1.050333, 1.076481, 1.103330, 1.130916, 1.159278, 1.188459, 1.218503,
1.249460, 1.281382, 1.314323, 1.348344, 1.383510, 1.419891, 1.457562, 1.496606,
1.537110, 1.579173, 1.622897, 1.668399, 1.715803, 1.765247, 1.816880, 1.870868,
1.927394, 1.986659, 2.048886, 2.114322, 2.183246, 2.255964, 2.332824, 2.414214,
2.500574, 2.592402, 2.690266, 2.794813, 2.906786, 3.027043, 3.156580, 3.296558,
3.448340, 3.613536, 3.794063, 3.992224, 4.210802, 4.453202, 4.723629, 5.027339,
5.370990, 5.763142, 6.214988, 6.741452, 7.362888, 8.107786, 9.017303, 10.153171,
11.612399, 13.556669, 16.277008, 20.355467, 27.150171, 40.735485, 81.483238};

```

B.5 The Data File Reader, arc_ccd_read.c

```

/*****
/* arc_ccd_read.c */
/*
/* Reads a file created by arc_ccd routine save_spectrum() and spits */
/* it out as ASCII. */
*****/
/*
/* CHARA ARRAY USER INTERFACE */
/* Based on the SUSI User Interface */
/* In turn based on the CHIP User interface */
/*
/* Center for High Angular Resolution Astronomy */
/* Mount Wilson Observatory, CA 91001, USA */
/*
/* Telephone: 1-626-796-5405 */
/* Fax : 1-626-796-6717 */
/* email : theo@chara.gsu.edu */
/* WWW : http://www.chara.gsu.edu */
/*
/* (C) This source code and its associated executable */
/* program(s) are copyright. */
*****/
/*
/* Author : Theo ten Brummelaar, modified from irread.c to */
/* arc_ccd_read.c by Chad Ogden */
/* Date : July 1999, mod March 2003 */
*****/

```

```

#include "arc_ccd.h"
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>
#include <time.h>

// #define DEBUG_MESSAGES

// use the following only in conjunction with arc_ccd_testbuffer

// #define DEBUG_BUFFER

int main(int argc, char **argv)
{
    int fp;
    int i,j;
    struct tm gmt_now;
    int year;
    long int time_start,time_stop;
    long int time_mid,time_range;
    bool time_mid_read = FALSE;
    bool time_range_read = FALSE;

    // data read from file

    int exp_time;
    char rec_arg;
    long int *f_bias=NULL;
    float *f_spec=NULL;
    float *f_spec_flat=NULL;
    float f_ps[PS_SIZE];
    float f_ps_flat[PS_SIZE];
    SROI f_bb;
    PS_STATS f_stats;
    VIS_DITH f_vd;
    int row[MAX_NUM_SIMULTANEOUS_SPECTRA + 1];
    int chan_list[SPEC_SIZE + 1];
    float *f_chan=NULL;
    CHARA_TIME fr_time;
    uint32_t fr_num;

    // behavior flags

    bool rec_ps_stats = TRUE;
    bool rec_ps = FALSE;
    bool rec_spec = FALSE;
    bool rec_chan = FALSE;
    bool rec_vd = FALSE;
    bool used_spec_flat = FALSE;
    bool used_ps_flat = FALSE;
    bool used_bias = FALSE;
    bool write_ps_stats = TRUE;
    bool write_ps = FALSE;
    bool write_spec = FALSE;
    bool write_vd = FALSE;
    bool write_chan = FALSE;
    bool columnar_output = FALSE;

    /* Check command line */

    if (argc < 2)
    {
        fprintf(stderr,"usage: %s file {-ps,-spec} {time time_range}\n",
            argv[0]);
        exit(-1);
    }

    // check each argv for flags and data. argv[1] is filename

    for(i=2; i<argc; i++)
    {
        if( strcmp( argv[i],"-ps" ) == 0)
            write_ps = TRUE;

        else if( strcmp( argv[i],"-spec" ) == 0)

```

```

write_spec = TRUE;

else if( strcmp( argv[i], "-nostats" ) == 0)
write_ps_stats = FALSE;

else if( strcmp( argv[i], "-dither" ) == 0)
write_vd = TRUE;

else if( strcmp( argv[i], "-chan" ) == 0)
{
// if channel data was recorded, ps stats weren't
write_chan = TRUE;
write_ps_stats = FALSE;
}
else if( strcmp( argv[i], "-channels" ) == 0)
write_chan = TRUE;

else if( strcmp( argv[i], "-columnar" ) == 0)
columnar_output = TRUE;

else if (!time_mid_read)
{
sscanf(argv[i], "%ld", &time_mid);
time_mid_read = TRUE;
}

else if (!time_range_read)
{
sscanf(argv[i], "%ld", &time_range);
time_range_read = TRUE;
}

else
{
fprintf(stderr,
"usage: %s file {-ps, -spec} {time time_range}\n",
argv[0]);
exit(-1);
}
}

if(!time_mid_read)
time_mid = 100000000L;

if(!time_range_read)
time_range = 100000000L;

time_start = time_mid - time_range;
time_stop = time_mid + time_range;

/* Open the file */
if ((fp = open(argv[1], O_RDONLY)) == -1)
{
fprintf(stderr, "Failed to open file %s:\n\n%s",
argv[1], strerror(errno));
exit(-1);
}

#ifdef DEBUG_BUFFER

/* Read in the gmt */
if (read(fp, &gmt_now, sizeof(gmt_now)) != sizeof(gmt_now))
{
fprintf(stderr, "Error reading GMT from file.\n");
exit(-2);
}

// Read in the exposure time
if (read(fp, &exp_time, sizeof(exp_time)) != sizeof(exp_time))
{
fprintf(stderr, "Error reading exposure time from file.\n");
exit(-2);
}

// Read in character whose bits are recording status flags

```



```

if (read(fp, &rec_arg, sizeof(char)) != sizeof(char))
{
fprintf(stderr,"Error reading recording status from file.\n");
exit(-2);
}

// set recording status flags

// bit 0 = recorded power spectrum stats
if( (rec_arg & 1) == 1)
rec_ps_stats = TRUE;
else
rec_ps_stats = FALSE;

// bit 1 = recorded power spectrum data
if( ((rec_arg >> 1) & 1) == 1)
rec_ps = TRUE;
else
rec_ps = FALSE;

// bit 2 = recorded spectrum data
if( ((rec_arg >> 2) & 1) == 1)
rec_spec = TRUE;
else
rec_spec = FALSE;

// bit 3 = recorded channel data
if( ((rec_arg >> 3) & 1) == 1 )
rec_chan = TRUE;
else
rec_chan = FALSE;

// bit 4 = recorded vis_dither data
if( ((rec_arg >> 4) & 1) == 1)
rec_vd = TRUE;
else
rec_vd = FALSE;

// bit 5 = debiased data, bias included in file
if( ((rec_arg >> 5) & 1) == 1)
used_bias = TRUE;
else
used_bias = FALSE;

// bit 6 = flatted spectrum data, spec flat included in file
if( ((rec_arg >> 6) & 1) == 1)
used_spec_flat = TRUE;
else
used_spec_flat = FALSE;

// bit 7 = flatted power spectrum data, ps flat included in file
if( ((rec_arg >> 7) & 1) == 1)
used_ps_flat = TRUE;
else
used_ps_flat = FALSE;

// finished setting recording status flags

#ifdef DEBUG_MESSAGES

fprintf(stderr,"PS stats recorded:  %d\n", (int)rec_ps_stats);
fprintf(stderr,"Write PS statistics: %d\n", (int)write_ps_stats);
fprintf(stderr,"PS recorded:      %d\n", (int)rec_ps);
fprintf(stderr,"Write PS data:      %d\n", (int)write_ps);
fprintf(stderr,"Spectrum recorded:  %d\n", (int)rec_spec);
fprintf(stderr,"Write spectrum data: %d\n", (int)write_spec);
fprintf(stderr,"Channels recorded: %d\n", (int)rec_chan);
fprintf(stderr,"Write channel data:  %d\n", (int)write_chan);
fprintf(stderr,"Vis dither recorded: %d\n", (int)rec_vd);
fprintf(stderr,"Write dither data:   %d\n", (int)write_vd);
fprintf(stderr,"Bias recorded:      %d\n", (int)used_bias);
fprintf(stderr,"Spec flat recorded: %d\n", (int)used_spec_flat);

```

```

fprintf(stderr,"PS flat recorded:          %d\n",(int)used_ps_flat);

#endif

//read in bounding box data

if (read(fp, &f_bb, sizeof(SROI)) != sizeof(SROI))
{
fprintf(stderr,"Error reading bounding box parameters.\n");
exit(-2);
}

//read in spectrum row

if (read(fp, &(row[0]), sizeof(row[0])*(MAX_NUM_SIMULTANEOUS_SPECTRA+1)) != sizeof(row[0])*(MAX_NUM_SIMULTANEOUS_SPECTRA+1))
{
fprintf(stderr,"Error reading spectrum row.\n");
exit(-2);
}

// check that the number of rows is viable

if (row[0] < 1 || row[0] > f_bb.dy)
{
fprintf(stderr,"Number of rows: %d, invalid.",row[0]);
exit(-1);
}

// read in channel data if it was recorded

if(rec_chan)
if(read( fp,&(chan_list[0]),sizeof(chan_list[0])*(SPEC_SIZE+1) ) !=
   sizeof(chan_list[0])*(SPEC_SIZE+1) )
{
fprintf(stderr,"Error reading channel list.\n");
exit(-2);
}

/* Read in the bias data */

if(used_bias)
{
// allocate memory for bias data

if( (f_bias = malloc(row[0] * f_bb.dx * sizeof(long int)))
== NULL )
{
fprintf(stderr,"Unable to allocate memory for bias data.\n");
exit(-2);
}

// read bias data from file

if( read( fp, f_bias, row[0] * f_bb.dx * sizeof(long int) )
!= row[0] * f_bb.dx * sizeof(long int) )
{
fprintf(stderr,"Failed to read bias data.\n");
free(f_bias);
exit(-3);
}

// Read in the spec flat data

if(used_spec_flat)
{
// allocate memory for spec flat data
if( (f_spec_flat = malloc( row[0] * SPEC_SIZE * sizeof(float) )
) == NULL)
{
fprintf(stderr,"Unable to allocate memory for spec flat.\n");
if(f_bias != NULL)
free(f_bias);
exit(-2);
}

if( read( fp, f_spec_flat, row[0] * SPEC_SIZE * sizeof(float) )
!= row[0] * SPEC_SIZE * sizeof(float) )
{
fprintf(stderr,"Failed to read flat data.\n");
}
}

```

```

    if(f_bias != NULL)
    free(f_bias);
    if(f_spec_flat != NULL)
    free(f_spec_flat);
    exit(-3);
}

}

// Read in the ps flat data

if(used_ps_flat)
{
    if( read( fp, f_ps_flat, PS_SIZE * sizeof(float) )
    != PS_SIZE * sizeof(float) )
    {
        fprintf(stderr,"Failed to read ps flat data.\n");
        if(f_bias != NULL)
        free(f_bias);
        if(f_spec_flat != NULL)
        free(f_spec_flat);
        exit(-3);
    }
}

/* Note... not really Y2K cool.... */

year = gmt_now.tm_year + 1900;
if (year < 1950) year+=100;

/* Put out header information */

printf("# FILE: %s\n",argv[1]);
printf("# GMT DATE      : %d\n",year);
printf("# GMT MONTH     : %d\n",gmt_now.tm_mon+1);
printf("# GMT DAY       : %d\n",gmt_now.tm_mday);
printf("# GMT DAY OF YEAR : %d\n",gmt_now.tm_yday);
printf("# DAYLIGHT SAVE? : %d\n",gmt_now.tm_isdst);
printf("# BB LOWER LEFT XY: %d %d\n", f_bb.x, f_bb.y);
printf("# BB DX DY      : %d %d\n", f_bb.dx, f_bb.dy);
printf("# NUM OF SPEC ROWS: %d\n",row[0]);
printf("# SPEC ROW IN BB : ");
for(i=1; i<=row[0]; i++)
printf("%d ", row[i]);
printf("\n");
if(rec_chan)
{
    printf("# NUM CHANNELS : %d\n", chan_list[0]);
    printf("# CHANNEL   : ");
    for(i=1; i<=chan_list[0]; i++)
    printf("%d ", chan_list[i]);
    printf("\n");
}
printf("# EXPOSURE TIME   : %d ms\n",exp_time);
#endif
// If not writing ps stats, time will be the only stat header written

printf("#   TIME");

// If recording channels, frame number will be written also

if(rec_chan)
printf(" FRAME_NUM");

if(write_ps_stats)
{
    printf("      BIN      HEIGHT    POWER    WIDTH    SNR      SIGMA    SKEW    ERRSIG    TRACKSIG    DITH_POS  \n");
}

else

printf("\n");

// allocate memory for the spectra

if( (f_spec = (float *)malloc( row[0] * SPEC_SIZE * sizeof(float) )
) == NULL)
{

```

```

fprintf(stderr,"Unable to allocate memory for spec flat.\n");
if(f_bias != NULL)
free(f_bias);
if(f_spec_flat != NULL)
free(f_spec_flat);
exit(-2);
}

// allocate memory for the channel data

if( ( f_chan = (float *)malloc( row[0] * chan_list[0] *
sizeof(float) ) ) == NULL)
{
fprintf(stderr,
"Unable to allocate memory for channel data.\n");
if(f_bias != NULL)
free(f_bias);
if(f_spec_flat != NULL)
free(f_spec_flat);
if(f_spec != NULL)
free(f_spec);
exit(-2);
}

/* And spit out the data */

while(1)
{
/* Try and read in next set of data */
// if using save_channels, the time and frame number
// were written first
if(rec_chan)
{
if (read(fp, &fr_time, sizeof(fr_time))
!= sizeof(fr_time)) break;
if (read(fp, &fr_num, sizeof(fr_num))
!= sizeof(fr_num)) break;
}
if(rec_ps_stats)
{
if (read(fp, &f_stats, sizeof(PS_STATS))
!= sizeof(PS_STATS)) break;
}
if(rec_vd)
{
if(read(fp, &f_vd, sizeof(f_vd)) != sizeof(f_vd)) break;
}

if(rec_ps)
{
if (read(fp, f_ps, PS_SIZE * sizeof(float))
!= PS_SIZE * sizeof(float)) break;
}

if(rec_spec)
{
if (read( fp, f_spec, row[0] * SPEC_SIZE * sizeof(float)
) != row[0] * SPEC_SIZE * sizeof(float) ) break;
}

if(rec_chan)
{
if(read( fp, f_chan, row[0] * chan_list[0] *
sizeof(float)) != row[0] * chan_list[0] *
sizeof(float)) break;
}

/* Is it withint the time range we want */

// Not sure this worked.

// if (f_stats.time < time_start) continue;
// if (f_stats.time > time_stop) break;

```

```

/* Spit out the data */

// if not writing ps stats, time will still be written
// if recording channels, time doesn't come from
// ps stats structure

if(!rec_chan)
printf("%8ld",f_stats.time);
else
{
printf("# FRAME_TIME FRAME_NUM\n");
printf("%8ld %8ld", fr_time,fr_num);
}

if(write_ps_stats)
{
printf("%8.1f %8.2f %8.2f %8.2f %8.2f %8.2f %8.2f %8.0f %10d\n",
f_stats.bin, f_stats.height,
f_stats.power, f_stats.width, f_stats.snr,
f_stats.sigma, f_stats.skew, f_stats.errsignal,
f_stats.tracksignal,f_stats.dith_dac);
}

else

printf("\n");

if(write_vd && rec_vd)
{
printf("# Vis Dither data: (dac_curr dac_req sweep_um sweep_dac sweep_index sweep_on sweep_auto offset offset_index use_offsets)\n",
printf("%d %d %8.4f %8.0f %d %d %d %8.0f %d %d\n",
f_vd.dac_curr, f_vd.dac_req, f_vd.sweep_um,
f_vd.sweep_dac, f_vd.sweep_index, (f_vd.sweep_on & 1),
(f_vd.sweep_automatic & 1), f_vd.offset,
f_vd.offset_index, (f_vd.use_offsets & 1));
}

if(write_ps && rec_ps)
{
printf("# Power Spectrum data:\n");

for(i=0;i<PS_SIZE;i++)
{
printf("%f ",f_ps[i]);

if(columnar_output)
printf("\n");
}

printf("\n");
}

if(write_spec && rec_spec)
{
printf("# Spectrum data:\n");

// this loop steps through the spectra
for(j=0;j<row[0];j++)
{
// this loop steps through the pixels
// in each spectrum
for(i=0;i<SPEC_SIZE;i++)
{
printf("%f ",
*(f_spec+j * SPEC_SIZE+i)
);

if(columnar_output)
printf("\n");
}

printf("\n");
}

if(write_chan && rec_chan)
{

```

```

printf("# Channel Data:\n");

// this loop steps through the spectra
for(j=0; j<row[0]; j++)
{
    // this loop steps through the channels
    for(i=0; i<chan_list[0]; i++)
    {
        printf("%f ",
            *(f_chan+j*chan_list[0]+i)
        );
    }

    printf("\n");
}

}

//print out flat and bias data

if(used_bias)
{
    printf("# Bias data:\n");

    for(j=0; j<row[0]; j++)
    {
        for(i=0; i<f_bb.dx; i++)
        {
            printf("%ld ", *(f_bias + f_bb.dx*j + i) );
        }

        printf("\n");
    }
}

if(used_spec_flat)
{
    printf("# Spectrum flat data:\n");

    for(j=0; j<row[0]; j++)
    {
        for(i=0; i<SPEC_SIZE; i++)
        {
            printf("%f ", *(f_spec_flat + j * SPEC_SIZE + i) );
        }

        printf("\n");
    }
}

if(used_ps_flat)
{
    printf("# Power Spectrum flat data:\n");

    for(i=0; i<PS_SIZE; i++)
    {
        printf("%f ", f_ps_flat[i] );
    }

    printf("\n");
}

/* That is all */

if(f_bias != NULL)
    free(f_bias);
if(f_spec_flat != NULL)
    free(f_spec_flat);
if(f_spec != NULL)
    free(f_spec);
close(fp);
exit(0);

} /* main() */

```

Appendix C

VIS Dither Control Code (on CD-ROM)

C.1 The Header File, vis_dither.h

```

/*****
/* vis_dither.h */
*/
/* Header file for the vis_dither controller. */
/* This newer version is for the 16 bit PCI */
/* based ADLINK Technology 6216V card and is the default. */
/*****
/*
/*      Center for High Angular Resolution Astronomy
/* Georgia State University, Atlanta GA 30303-3083, U.S.A.
/*
/*
/* Telephone: 1-626-796-5405
/* Fax      : 1-626-796-6717
/* email    : theo@chara.gsu.edu
/* WWW      : http://www.chara.gsu.edu/~theo/theo.html
/*
/* (C) This source code and its associated executable
/* program(s) are copyright.
/*
/*****
/*
/* Author : Chad Ogden, Based on dither.h by Theo ten Brummelaar
/* Date   : May 2003
/*****

#ifndef __DITHER__
#define __DITHER__

/*****
/* The stuff for both MODULES and User Interfaces */
/*****

/* Edit this out if you wish to use the CHARA fast handler instead */

#define USE_RTL_SCHED

#include <chara.h>
#include <clock.h>
#include <linux/types.h>
#include <unistd.h>

// DATA_DIRECTORY is defined in chara.h
// This is where the log data will be automatically written.

#define VIS_DITHER_LOG_FILE DATA_DIRECTORY "vis_dither.log"

/*
 * The commands
 */

#define VIS_DITHER_GOTO (FIRST_VISDITHER_COMMAND + 0)
#define VIS_DITHER_MOVE_RELATIVE (FIRST_VISDITHER_COMMAND + 1)
#define VIS_DITHER_QUERY_POSITION (FIRST_VISDITHER_COMMAND + 2)
#define VIS_DITHER_QUERY_TARGET (FIRST_VISDITHER_COMMAND + 3)
#define VIS_DITHER_LOAD_OFFSETS (FIRST_VISDITHER_COMMAND + 4)
#define VIS_DITHER_LOAD_SWEEP (FIRST_VISDITHER_COMMAND + 5)
#define VIS_DITHER_QUERY_STATS (FIRST_VISDITHER_COMMAND + 6)

```

```

#define VIS_DITHER_SET_FLAG_TRUE (FIRST_VISDITHER_COMMAND + 7)
#define VIS_DITHER_SET_FLAG_FALSE (FIRST_VISDITHER_COMMAND + 8)
#define VIS_DITHER_SWEEP_RESET (FIRST_VISDITHER_COMMAND + 9)
#define VIS_DITHER_CHANGE_DAMPING (FIRST_VISDITHER_COMMAND + 10)

// Biggest step that the handler will allow the dither mirror to
// make per cycle.

#ifdef USE_RTL_SCHED

// This one is for when the dither handler is called by the rtl
// scheduler
#define VIS_DITHER_TASK_RATE 20000 /* Hz */
#define RTL_SCHED_MS_RATE (VIS_DITHER_TASK_RATE/1000)

#define MAX_DIFF (40<<3) / (RTL_SCHED_MS_RATE)
// MAX_DIFF = 320 / 20 = 16
#else

// This was for when the handler was called every ms
#define MAX_DIFF (40<<3)

#endif

#define RTDAC_UPPER_LIMIT 32767
#define RTDAC_LOWER_LIMIT 0

#define AMPLIFIER_GAIN 15.0
#define DAC_GAIN (10.0/32767)
#define TOTAL_GAIN (AMPLIFIER_GAIN*DAC_GAIN) // = 4.577E-3

/* These from 130 Hz file */

#define pos_to_volts_up(x) ( -3.8706450 \
                          + 1.1599582*x \
                          - 0.0031450815*x*x \
                          + 7.6368467e-05*x*x*x)

#define pos_to_volts_down(x) ( -6.9031688 \
                              + 1.9197402*x \
                              - 0.013533760*x*x \
                              + 0.00010366182*x*x*x)

#define pos_to_volts_linear(x) (1.46534*x)
#define volts_to_pos_linear(x) (x/1.46534)

// old version v to dac
// #define volts_to_dac(x) ((short int)(x / TOTAL_GAIN + 0.5))

// new version of v to dac leaves out short int conversion. do later.
// I don't think that 0.5 is in the right place. Should probably be in
// dac_to_volts.
// #define volts_to_dac(x) (x / TOTAL_GAIN + 0.5)
#define volts_to_dac(x) (x / TOTAL_GAIN)
#define dac_to_volts(x) (x * TOTAL_GAIN)

// this comes out to mean 320.10 dac steps per 1 um.

// the fakephase script puts out 10,000 samples. That's 10 s worth
// of 1 ms samples. That's the maximum number of samples to use in the
// array of phase offsets.

#define MAX_NUM_IN_OFFSETS_ARRAY 10000

// the number of passes in delay to or from the dither mirror. If there are
// 4 passes, only 1/4 of the delay needs to be introduced in order to
// produce the same phase shift as would be seen from the atmospheric piston.

#define OPTICAL_PATH_LAB_PASSES 4.0

// The max number of steps to allow in a real dither.
// If 4096 in one direction, 8192 up and down.
// If 2048 in one direction, 4096 up and down.

#define MAX_NUM_IN_OBSERVING_SWEEP 4096

// the maximum number of positions in a full dither sweep, up and back

```



```

// plus the zero bin

#define MAX_NUM_IN_SWEEP_ARRAY (MAX_NUM_IN_OBSERVING_SWEEP + 1)

//These define the limits for fringe wavelength,
// scanning fringe frequency, and the 6db cutoff

#define FRINGE_LAMBDA_MAX 3.0
#define FRINGE_LAMBDA_MIN 0.45
#define FRINGE_FREQ_MAX 500.0
#define FRINGE_FREQ_MIN 41.0
#define FRINGE_6DB_MAX 100.0
#define FRINGE_6DB_MIN 2.0
#define FRINGE_SAMPLING_MIN 2.0

//These define where (in microns) the dither mirror starts and stops its sweep.

#define SWEEP_START          4.2336592
#define SWEEP_STOP           92.414582
#define SWEEP_AMPLITUDE      (SWEEP_STOP - SWEEP_START)
#define FILTER_WIDTH         0.2

// This is the default wavelength for calculating dithers.  In microns

#define DEFAULT_WAVELENGTH .8

// some constants for use when setting flags on the rt side

#define FLAG_ID_USE_OFFSETS 1
#define FLAG_ID_SWEEP_ON 2
#define FLAG_ID_SWEEP_AUTOMATIC 3

// make sure booleans are defined

#ifndef bool
typedef unsigned char bool;
#endif
#ifndef TRUE
#define TRUE 1
#endif
#ifndef FALSE
#define FALSE 0
#endif

struct vis_dither_parameters {
int dac_curr;
int dac_req;
float sweep_um;
float sweep_dac;
// note that sweep_index starts at 1 not 0
int sweep_index;
bool sweep_on;
bool sweep_automatic;
float offset;
int offset_index;
bool use_offsets;
};

typedef struct vis_dither_parameters VIS_DITH;

/*****
/* The stuff for only MODULES */
*****/

#ifdef MODULE

#ifdef USE_RTL_SCHED
#include <rtl_sched.h>
#endif

/* Macros */

/* prototypes */

int init_module(void);
#ifdef USE_RTL_SCHED
void vis_dither_handler(int arc);
#else
void vis_dither_handler(void);

```

```

#endif
int vis_dither_goto(unsigned char arg);
int vis_dither_rt_goto(int dac);
int vis_dither_move_relative(unsigned char arg);
int vis_dither_rt_move_relative(int rel_move);
int vis_dither_query_position(unsigned char arg);
int vis_dither_rt_query_position(void);
int vis_dither_query_target(unsigned char arg);
int vis_dither_rt_query_target(void);
int vis_dither_query_stats(unsigned char arg);
VIS_DITH vis_dither_rt_query_stats(void);
int vis_dither_load_offsets(unsigned char arg);
int vis_dither_load_sweep(unsigned char arg);
float *vis_dither_rt_sweep_um_ptr(void);
short int *vis_dither_rt_sweep_dac_ptr(void);
int vis_dither_set_flag_true(unsigned char arg);
int vis_dither_set_flag_false(unsigned char arg);
VIS_DITH vis_dither_sweep_increment(void);
int vis_dither_sweep_reset(unsigned char arg);
int vis_dither_change_damping(unsigned char arg);
void cleanup_module(void);

#else

/*****
/* This stuff only for user interfaces */
*****/

#include <math.h>
#include <stdio.h>
#include <nsimpleX.h>
#include <nrc.h>
#include <charaui.h>
#include <string.h>
#include <filter.h>
#include <malloc.h>

/* Macros */

/* Globals */

//extern bool enable_offsets;

/* Prototypes */

int new_dither(int argc, char **argv);
int calculate_dither(float lambda, float sampling, float f6db, char *filename);
int load_dither_offsets(char *infile_name);
int call_load_dither_offsets(int argc, char **argv);
int set_vis_dith_flag(unsigned char flag_id, bool value);
int call_set_vis_dith_flag(int argc, char **argv, unsigned char flag_id);
int call_enable_dither_offsets(int argc, char **argv);
int call_enable_vis_dither_sweep(int argc, char **argv);
int call_enable_vis_dither_sweep_automatic(int argc, char **argv);
short int *vis_dither_sweep_dac_ptr(void);
float *vis_dither_sweep_um_ptr(void);
int sweep_reset(void);
int call_sweep_reset(int argc, char **argv);
int change_damping(char damp);
int call_change_damping(int argc, char **argv);

#endif
#endif

```

C.2 The Real-Time Linux Module, vis_dither_rt.c

```

/*****
/* vis_dither_rt.c */
*/
/* Fast handler to control rt_dac and callable function to request */
/* rt_dac dacs. */
*****/
/*
/* Center for High Angular Resolution Astronomy */
/* Georgia State University, Atlanta GA 30303-3083, U.S.A. */
*/

```

```

/* Telephone: 1-626-796-5405 */
/* Fax      : 1-626-796-6717 */
/* email    : theo@chara.gsu.edu */
/* WWW      : http://www.chara.gsu.edu/~theo/theo.html */
/*
/* (C) This source code and its associated executable */
/* program(s) are copyright. */
/*
/*****
/*
/* Author : Chad Ogden */
/* Date   : May 2003 */
*****/

#define MODULE

// Include header file

#include "vis_dither.h"
#include "../rtdac/rtdac.h"

#define DITHER_CHANNEL 0

// Globals

static VIS_DITH vis_dither_par;

// these arrays hold the positions in microns and dac for each step of the
// dither sweep. The [0] element tells how many steps there are in the
// full sweep, both up and down.

static float vis_dither_sweep_um[MAX_NUM_IN_SWEEP_ARRAY];
static short int vis_dither_sweep_dac[MAX_NUM_IN_SWEEP_ARRAY];

static float vis_dither_offsets[MAX_NUM_IN_OFFSETS_ARRAY];
static int vis_dither_num_offsets;
static float vis_dither_last_offset;
static int vis_dither_sub_ms_counter;

// this is used as a low-pass filter to slow down vis dither's maximum
// step size. It is applied by multiplying MAX_DIFF by
// 1/vis_dither_damping. Since MAX_DIFF is a small integer,
// probably 16 for the rtl_sched case and 320 otherwise, there's no point
// in having the damping term have any more resolution than an +int which
// is <= MAX_DIFF.

static int vis_dither_damping;

#ifdef USE_RTL_SCHED
static RT_TASK vis_dither_task;
#endif

/*****
/* init_module() */
/*
/* This code initializes the module and the printer port itself. */
*****/

int init_module(void)
{
    int i;
    #ifdef USE_RTL_SCHED
    RTIME now, period;
    #endif

    // initialise the phases array at all 0s
    for(i=0; i<MAX_NUM_IN_OFFSETS_ARRAY; i++)
        vis_dither_offsets[i] = 0.0;

    // set last_offset
    vis_dither_last_offset = 0.0;

    // set the damping term
    vis_dither_damping = 1;

    // set sub_ms_counter

```

```

vis_dither_sub_ms_counter = 0;

// initialize the dither sweep arrays with all 0s.
for(i=0; i<MAX_NUM_IN_SWEEP_ARRAY; i++)
{
    vis_dither_sweep_um[i] = 0.0;
    vis_dither_sweep_dac[i] = 0;
}

vis_dither_par.sweep_index = 0;
vis_dither_par.sweep_on = FALSE;
vis_dither_par.sweep_automatic = FALSE;
vis_dither_par.offset_index = 0;
vis_dither_par.use_offsets = FALSE;
vis_dither_par.sweep_um = 0.0;
vis_dither_par.sweep_dac = 0.0;

///// set up vis dither mirror

// We want a maximum of 10 volts and minimum of 0 volts
    if (RTDAC_SetAmplitudeLimits(RTDAC_LOWER_LIMIT, RTDAC_UPPER_LIMIT) != 0)
    {
        printk("dither_rt: Could not set DAC limits.\n");
        return -EINVAL;
    }

    // To start we disable the filter
    RTDAC_DisableFilter();

    // We should start at zero
vis_dither_par.dac_curr = 0;
vis_dither_par.dac_req = vis_dither_par.dac_curr;

    if (RTDAC_WriteAnalogOutput(DITHER_CHANNEL, 0) != 0)
    {
        printk("dither_rt: Could not set DAC to zero.\n");
        return -EINVAL;
    }

///// add local commands

if(chara_add_command(VIS_DITHER_GOTO,vis_dither_goto) < 0)
{
    printk("example: Could not get command %d.\n",
        VIS_DITHER_GOTO);
    return -EINVAL;
}

if(chara_add_command(VIS_DITHER_MOVE_RELATIVE,
vis_dither_move_relative) < 0)
{
    chara_remove_command(VIS_DITHER_GOTO);
    printk("example: Could not get command %d.\n",
        VIS_DITHER_MOVE_RELATIVE);
    return -EINVAL;
}

if(chara_add_command(VIS_DITHER_QUERY_POSITION,
vis_dither_query_position) < 0)
{
    chara_remove_command(VIS_DITHER_GOTO);
    chara_remove_command(VIS_DITHER_MOVE_RELATIVE);
    printk("example: Could not get command %d.\n",
        VIS_DITHER_QUERY_POSITION);
    return -EINVAL;
}

if(chara_add_command(VIS_DITHER_QUERY_TARGET,
vis_dither_query_target) < 0)
{
    chara_remove_command(VIS_DITHER_GOTO);
    chara_remove_command(VIS_DITHER_MOVE_RELATIVE);
    chara_remove_command(VIS_DITHER_QUERY_POSITION);
}

```

```

        printk("example: Could not get command %d.\n",
               VIS_DITHER_QUERY_TARGET);
        return -EINVAL;
    }

    if(chara_add_command(VIS_DITHER_LOAD_OFFSETS,
vis_dither_load_offsets) < 0)
    {
        chara_remove_command(VIS_DITHER_GOTO);
        chara_remove_command(VIS_DITHER_MOVE_RELATIVE);
        chara_remove_command(VIS_DITHER_QUERY_POSITION);
        chara_remove_command(VIS_DITHER_QUERY_TARGET);
        printk("example: Could not get command %d.\n",
               VIS_DITHER_LOAD_OFFSETS);
        return -EINVAL;
    }

    if(chara_add_command(VIS_DITHER_LOAD_SWEEP,
vis_dither_load_sweep) < 0)
    {
        chara_remove_command(VIS_DITHER_GOTO);
        chara_remove_command(VIS_DITHER_MOVE_RELATIVE);
        chara_remove_command(VIS_DITHER_QUERY_POSITION);
        chara_remove_command(VIS_DITHER_QUERY_TARGET);
        chara_remove_command(VIS_DITHER_LOAD_OFFSETS);
        printk("example: Could not get command %d.\n",
               VIS_DITHER_LOAD_SWEEP);
        return -EINVAL;
    }

    if(chara_add_command(VIS_DITHER_QUERY_STATS,
vis_dither_query_stats) < 0)
    {
        chara_remove_command(VIS_DITHER_GOTO);
        chara_remove_command(VIS_DITHER_MOVE_RELATIVE);
        chara_remove_command(VIS_DITHER_QUERY_POSITION);
        chara_remove_command(VIS_DITHER_QUERY_TARGET);
        chara_remove_command(VIS_DITHER_LOAD_OFFSETS);
        chara_remove_command(VIS_DITHER_LOAD_SWEEP);
        printk("example: Could not get command %d.\n",
               VIS_DITHER_QUERY_STATS);
        return -EINVAL;
    }

    if(chara_add_command(VIS_DITHER_SET_FLAG_TRUE,
vis_dither_set_flag_true) < 0)
    {
        chara_remove_command(VIS_DITHER_GOTO);
        chara_remove_command(VIS_DITHER_MOVE_RELATIVE);
        chara_remove_command(VIS_DITHER_QUERY_POSITION);
        chara_remove_command(VIS_DITHER_QUERY_TARGET);
        chara_remove_command(VIS_DITHER_LOAD_OFFSETS);
        chara_remove_command(VIS_DITHER_LOAD_SWEEP);
        chara_remove_command(VIS_DITHER_QUERY_STATS);
        printk("example: Could not get command %d.\n",
               VIS_DITHER_SET_FLAG_TRUE);
        return -EINVAL;
    }

    if(chara_add_command(VIS_DITHER_SET_FLAG_FALSE,
vis_dither_set_flag_false) < 0)
    {
        chara_remove_command(VIS_DITHER_GOTO);
        chara_remove_command(VIS_DITHER_MOVE_RELATIVE);
        chara_remove_command(VIS_DITHER_QUERY_POSITION);
        chara_remove_command(VIS_DITHER_QUERY_TARGET);
        chara_remove_command(VIS_DITHER_LOAD_OFFSETS);
        chara_remove_command(VIS_DITHER_LOAD_SWEEP);
        chara_remove_command(VIS_DITHER_QUERY_STATS);
        chara_remove_command(VIS_DITHER_SET_FLAG_TRUE);
        printk("example: Could not get command %d.\n",
               VIS_DITHER_SET_FLAG_FALSE);
        return -EINVAL;
    }

    if(chara_add_command(VIS_DITHER_SWEEP_RESET,
vis_dither_sweep_reset) < 0)
    {
        chara_remove_command(VIS_DITHER_GOTO);

```

```

chara_remove_command(VIS_DITHER_MOVE_RELATIVE);
chara_remove_command(VIS_DITHER_QUERY_POSITION);
chara_remove_command(VIS_DITHER_QUERY_TARGET);
chara_remove_command(VIS_DITHER_LOAD_OFFSETS);
chara_remove_command(VIS_DITHER_LOAD_SWEEP);
chara_remove_command(VIS_DITHER_QUERY_STATS);
chara_remove_command(VIS_DITHER_SET_FLAG_TRUE);
chara_remove_command(VIS_DITHER_SET_FLAG_FALSE);
    printk("example: Could not get command %d.\n",
           VIS_DITHER_SWEEP_RESET);
    return -EINVAL;
}

if(chara_add_command(VIS_DITHER_CHANGE_DAMPING,
vis_dither_change_damping) < 0)
{
    chara_remove_command(VIS_DITHER_GOTO);
    chara_remove_command(VIS_DITHER_MOVE_RELATIVE);
    chara_remove_command(VIS_DITHER_QUERY_POSITION);
    chara_remove_command(VIS_DITHER_QUERY_TARGET);
    chara_remove_command(VIS_DITHER_LOAD_OFFSETS);
    chara_remove_command(VIS_DITHER_LOAD_SWEEP);
    chara_remove_command(VIS_DITHER_QUERY_STATS);
    chara_remove_command(VIS_DITHER_SET_FLAG_TRUE);
    chara_remove_command(VIS_DITHER_SET_FLAG_FALSE);
    chara_remove_command(VIS_DITHER_SWEEP_RESET);
    printk("example: Could not get command %d.\n",
           VIS_DITHER_CHANGE_DAMPING);
    return -EINVAL;
}

///// set up vis dither handler
///// set up vis dither handler

#ifdef USE_RTL_SCHED
now = rt_get_time();
period = RT_TICKS_PER_SEC/VIS_DITHER_TASK_RATE;
rt_task_init(&vis_dither_task, vis_dither_handler, 0, 3000, 2);
    rt_task_make_periodic(&vis_dither_task, now+period, period);
#else
if (chara_set_fast_handler(vis_dither_handler) != 0)
{
    chara_remove_command(VIS_DITHER_GOTO);
    chara_remove_command(VIS_DITHER_MOVE_RELATIVE);
    chara_remove_command(VIS_DITHER_QUERY_POSITION);
    chara_remove_command(VIS_DITHER_QUERY_TARGET);
    chara_remove_command(VIS_DITHER_LOAD_OFFSETS);
    chara_remove_command(VIS_DITHER_LOAD_SWEEP);
    chara_remove_command(VIS_DITHER_QUERY_STATS);
    chara_remove_command(VIS_DITHER_SET_FLAG_TRUE);
    chara_remove_command(VIS_DITHER_SET_FLAG_FALSE);
    chara_remove_command(VIS_DITHER_SWEEP_RESET);
    chara_remove_command(VIS_DITHER_CHANGE_DAMPING);
    printk("dither_rt: Could not get fast handler.\n");
    return -EINVAL;
}
#endif

return 0;

} // init_module()

/*****
/* vis_dither_handler() */
/* */
/* Checks difference between requested dac and current dac sent */
/* to the dither mirror. Moves dither mirror up to MAX_DIFF */
/* steps towards the requested dac. Updates vis_dither_dac_curr. */
*****/

#ifdef USE_RTL_SCHED
void vis_dither_handler(int arc)
#else
void vis_dither_handler(void)
#endif
{
    int errsignal;
    int new_target;
    float max_step = 0.0;

```

```

#ifdef USE_RTL_SCHED
/* If we are using the RT scheduler we must loop forever */

while(TRUE) {
#ifdef
// if the dither sweep is on, increment the counter and set
// the requested target.

if(vis_dither_par.sweep_on)
{
if(vis_dither_par.sweep_automatic)
{
// check for the end of the array as you increment the
// sweep index

if(++vis_dither_par.sweep_index >=
vis_dither_sweep_dac[0] + 1)

vis_dither_par.sweep_index = 1;
}

// set the requested target

vis_dither_par.dac_req =
vis_dither_sweep_dac[vis_dither_par.sweep_index];
}

// errsignal tells you how far you need to go and in what
// direction to get to the requested dac

errsignal = vis_dither_par.dac_req - vis_dither_par.dac_curr;

// apply offset # (vis_dither_par.offset_index) to the errsignal
// if vis_dither_par.use_offsets is TRUE. Then increment vis_dither_par.offset_index.

if(vis_dither_par.use_offsets)
{
#ifdef USE_RTL_SCHED
// the sub_ms_counter must loop for 1 ms before going to the
// next index in the offset array. The RTL_SCHED runs at
// RTL_SCHED_MS_RATE per ms, so that's the number of times
// to loop before moving to the next offset index.

// if you've reached RTL_SCHED_MS_RATE loops, you've spent
// 1ms at this offset, so reset the loop counter and move
// to the next offset index.

// This makes for a smooth application of fake atmosphere
// with the smallest possible changes at the highest possible
// frequency.

if(++vis_dither_sub_ms_counter >= RTL_SCHED_MS_RATE)
{
vis_dither_sub_ms_counter = 0;

// put the current offset value into last_offset

vis_dither_last_offset = vis_dither_offsets[
vis_dither_par.offset_index];

// increment the offset index and check its limit
if(++vis_dither_par.offset_index >=
vis_dither_num_offsets)
vis_dither_par.offset_index = 0;
}

// Now interpolate between the current offset and the last,
// and apply this to the error signal
// last + (curr-last) * (sub_ms_count/ms_rate)

errsignal += (int)(
vis_dither_last_offset +
(vis_dither_offsets[vis_dither_par.offset_index] -
vis_dither_last_offset) *
( (float)vis_dither_sub_ms_counter /
(float)RTL_SCHED_MS_RATE )
);

```

```

#else
// if you've reached the last offset in the array,
// go back to 0.

if(++vis_dither_par.offset_index >= vis_dither_num_offsets)
vis_dither_par.offset_index = 0;

errrsignal += (int)vis_dither_offsets[vis_dither_par.offset_index];
#endif
}

// if we need to go + by more than the max step size, change
// errrsignal to the max step size

if(vis_dither_damping > 0)
max_step = (float) MAX_DIFF / (float) vis_dither_damping;
else
max_step = MAX_DIFF;

// if we're dead on, don't do anything

if (errrsignal == 0)
goto end_loop;

// otherwise check to see if errrsignal is too big

else if(errrsignal > max_step)
errrsignal = max_step;

// if we need to go - by more than the max step size, change
// errrsignal to the -max step size

else if (errrsignal < -(max_step))
errrsignal = -(max_step);

// new_target is where we'll send the dither mirror

new_target = vis_dither_par.dac_curr + errrsignal;

// check target limits

if(new_target > RTDAC_UPPER_LIMIT)
new_target = RTDAC_UPPER_LIMIT;

else if (new_target < RTDAC_LOWER_LIMIT)
new_target = RTDAC_LOWER_LIMIT;

// now send this new dac to the dither mirror

if (RTDAC_WriteAnalogOutput(DITHER_CHANNEL, new_target) != 0)
{
chara_printerr("Failed to set DAC in vis_dither_handler.");
goto end_loop;
}

// new_target is now the dac of the dither mirror.
// set vis_dither_dac_curr to this dac.

vis_dither_par.dac_curr = new_target;

// make sure the parameters structure is up to date
vis_dither_par.sweep_dac =
vis_dither_sweep_dac[vis_dither_par.sweep_index];

vis_dither_par.sweep_um =
vis_dither_sweep_um[vis_dither_par.sweep_index];

vis_dither_par.offset =
vis_dither_offsets[vis_dither_par.offset_index];

end_loop:

```



```

#ifdef USE_RTL_SCHED
    rt_task_wait();} /* Release control to the scheduler */
#else
return;
#endif

} // vis_dither_handler()

/*****
 * vis_dither_goto() */
 * */
 * Function by which the user side requests the dither mirror be sent */
 * to a given dac. The value of 'dac' is between 0 and 2^15. */
 * The function making the goto request is responsible for figuring */
 * out what actual position this dac represents. */
 *****/
int vis_dither_goto(unsigned char arg)
{
    int dac;
    int err;

    // get dac from FIFO

    if( (err = rtf_get(DATA_IN_FIFO,&dac,sizeof(dac)) ) < 0)
    {
        chara_printerr("Failed to get dac from DATA_IN_FIFO.");
        return err;
    }

    // check limits and force dac to be within them

    if(dac > RTDAC_UPPER_LIMIT)

        dac = RTDAC_UPPER_LIMIT;

    else if (dac < RTDAC_LOWER_LIMIT)

        dac = RTDAC_LOWER_LIMIT;

    vis_dither_par.dac_req = dac;

    return err;
} // vis_dither_goto()

/*****
 * vis_dither_rt_goto() */
 * */
 * Function by which other modules request the dither mirror be sent */
 * to a given dac. The value of 'dac' is position */
 * between 0 and 2^15. */
 * out what actual position this dac represents. */
 *****/
int vis_dither_rt_goto(int dac)
{
    // check limits and force dac to be within them

    if(dac > RTDAC_UPPER_LIMIT)

        dac = RTDAC_UPPER_LIMIT;

    else if (dac < RTDAC_LOWER_LIMIT)

        dac = RTDAC_LOWER_LIMIT;

    vis_dither_par.dac_req = dac;

    return 0;
} // vis_dither_rt_goto()

/*****
 * vis_dither_move_relative() */
 * */
 * Function by which the user side requests the dither mirror be moved */
 * by a given dac. The value of 'dac' is between 0 and 2^15. */
 * Keeps the change from sending the vis dither outside its set range. */
 *****/
int vis_dither_move_relative(unsigned char arg)
{

```

```

int rel_move;
int dac;
int err;

// get dac from FIFO

if( (err = rtf_get(DATA_IN_FIFO,&rel_move,sizeof(rel_move)) ) < 0)
{
chara_printerr("Failed to get dac from DATA_IN_FIFO.");
return err;
}

// dac is the new target with the relative move applied

dac = vis_dither_par.dac_req + rel_move;

// check limits and force dac to be within them

if(dac > RTDAC_UPPER_LIMIT)

dac = RTDAC_UPPER_LIMIT;

else if (dac < RTDAC_LOWER_LIMIT)

dac = RTDAC_LOWER_LIMIT;

vis_dither_par.dac_req = dac;

return err;
} // vis_dither_move_relative()

/*****
* vis_dither_rt_move_relative() */
*/
/* Function by which other modules request the dither mirror be moved */
/* by a given dac. The value of 'dac' is between 0 and 2^15. */
/* Keeps the change from sending the vis dither outside its set range. */
*****/
int vis_dither_rt_move_relative(int rel_move)
{
int dac;

// dac is the new target with the relative move applied

dac = vis_dither_par.dac_req + rel_move;

// check limits and force dac to be within them

if(dac > RTDAC_UPPER_LIMIT)

dac = RTDAC_UPPER_LIMIT;

else if (dac < RTDAC_LOWER_LIMIT)

dac = RTDAC_LOWER_LIMIT;

vis_dither_par.dac_req = dac;

return 0;
} // vis_dither_rt_goto()

/*****
* vis_dither_query_position() */
*/
/* Function by which the user side can ask the current dac position. */
/* Note that it tells where it is currently, not the target position. */
*****/
int vis_dither_query_position(unsigned char arg)
{
int err;

if ((err=rtf_put(DATA_OUT_FIFO, &(vis_dither_par.dac_curr), sizeof(vis_dither_par.dac_curr)))<0)
return err;

#ifdef DEBUG_MESSAGES

chara_printerr("I sent the dac target: %d.",vis_dither_dac_curr);

```

```

#endif
    return err;

} // vis_dither_query_position()

/*****
 * vis_dither_rt_query_position() */
/*
 * Function by which other modules can ask the current dac position.
 * Note that it tells where it is currently, not the target position.
 */
*****/
int vis_dither_rt_query_position(void)
{
    return vis_dither_par.dac_curr;
} // vis_dither_rt_query_position()

/*****
 * vis_dither_query_target() */
/*
 * Function by which the user side can ask the current dac target.
 * Note that it tells the TARGET, not where it is currently.
 */
*****/
int vis_dither_query_target(unsigned char arg)
{
    int err;

    if ((err=rtf_put(DATA_OUT_FIFO, &(vis_dither_par.dac_req),
sizeof(vis_dither_par.dac_req)))<0)
        return err;

#ifdef DEBUG_MESSAGES

        chara_printerr("I sent the dac target: %d.",vis_dither_par.dac_req);

#endif
    return err;
} // vis_dither_query_target()

/*****
 * vis_dither_rt_query_target() */
/*
 * Function by which other modules can ask the current dac target.
 * Note that it tells the TARGET, not where it is currently.
 */
*****/
int vis_dither_rt_query_target(void)
{
    return vis_dither_par.dac_req;
} // vis_dither_rt_query_target()

/*****
 * vis_dither_query_stats() */
/*
 * Function by which the user side can get a copy of the current
 * vis_dither_parameters structure.
 */
*****/
int vis_dither_query_stats(unsigned char arg)
{
    int err;

    if ((err=rtf_put(DATA_OUT_FIFO, &vis_dither_par,
sizeof(vis_dither_par)))<0)
        return err;

#ifdef DEBUG_MESSAGES

        chara_printerr("I sent the vis dither parameters.");

#endif
    return err;
} // vis_dither_query_stats()

/*****
 * vis_dither_rt_query_stats() */
/*
 * Function by which other modules can get a copy of the current
 * vis_dither_parameters structure.
 */
*****/

```

```

VIS_DITH vis_dither_rt_query_stats(void)
{
    return vis_dither_par;
} // vis_dither_rt_query_stats()

/*****
/* vis_dither_load_sweep */
/* */
/* User called function. Gets the size of the offsets array from the */
/* DATA_IN_FIFO, then reads that many floats from the DATA_IN_FIFO and */
/* writes them into the vis_dither_sweep[] arrays. */
*****/
int vis_dither_load_sweep(unsigned char arg)
{
    int err;

    // get the array size.

    if( (err = rtf_get(DATA_IN_FIFO, vis_dither_sweep_dac,
        sizeof(vis_dither_sweep_dac[0])) ) < 0)
    {
        chara_printerr("Failed to get array size from DATA_IN_FIFO.");
        return err;
    }

    // check vis_dither_sweep_dac[0] for size validity

    if(vis_dither_sweep_dac[0] < 1 || vis_dither_sweep_dac[0] > MAX_NUM_IN_SWEEP_ARRAY)
    {
        chara_printerr(
            "%d is an invalid number of elements for the sweep array."
            , vis_dither_sweep_dac[0]);
        vis_dither_sweep_dac[0] = 0;
        return -1;
    }

    // get the dac array data

    if( ( err =
        rtf_get( DATA_IN_FIFO, vis_dither_sweep_dac + 1,
        sizeof(vis_dither_sweep_dac[0]) *
        vis_dither_sweep_dac[0])
        )
        < 0)
    {
        chara_printerr("Failed to get dac sweep array from DATA_IN_FIFO.");
        return err;
    }

    // get the um array data

    if( ( err =
        rtf_get( DATA_IN_FIFO, vis_dither_sweep_um,
        sizeof(vis_dither_sweep_um[0]) *
        (vis_dither_sweep_um[0] + 1)
        )
        < 0)
    {
        chara_printerr("Failed to get um sweep array from DATA_IN_FIFO.");
        return err;
    }

    // set the index so it will increment to 1 at the beginning of the
    // next handler loop

    vis_dither_par.sweep_index = vis_dither_sweep_dac[0];

    return err;
} // vis_dither_load_sweep()

/*****
/* vis_dither_load_offsets */
/* */
/* User called function. Gets the size of the offsets array from the */
/* DATA_IN_FIFO, then reads that many floats from the DATA_IN_FIFO and */
/* writes them into the vis_dither_offsets[] array. */
*****/

```

```

/*****
int vis_dither_load_offsets(unsigned char arg)
{
    int err;

    // get the array size.

    if( (err = rtf_get(DATA_IN_FIFO,&vis_dither_num_offsets,sizeof(vis_dither_num_offsets)) ) < 0)
    {
        chara_printerr("Failed to get array size from DATA_IN_FIFO.");
        vis_dither_par.use_offsets = FALSE;
        return err;
    }

    // chech vis_dither_num_offsets for validity

    if(vis_dither_num_offsets < 1 || vis_dither_num_offsets > MAX_NUM_IN_OFFSETS_ARRAY)
    {
        chara_printerr(
            "%d is an invalid number of elements for the offsets array."
            , vis_dither_num_offsets);
        vis_dither_num_offsets = 0;
        vis_dither_par.use_offsets = FALSE;
        return -1;
    }

    // get the array data

    if( ( err =
        rtf_get( DATA_IN_FIFO, vis_dither_offsets,
        sizeof(vis_dither_offsets[0]) * vis_dither_num_offsets )
        )
        < 0)
    {
        chara_printerr("Failed to get data array from DATA_IN_FIFO.");
        vis_dither_par.use_offsets = FALSE;
        return err;
    }

    // set the index so it will increment to 0 at the beginning of the
    // next handler loop

    vis_dither_par.offset_index = vis_dither_num_offsets - 1;

    return err;
} // vis_dither_load_offsets()

/*****
/* vis_dither_rt_sweep_um_ptr() */
/* */
/* Callable from rt side only. Returns pointer to beginning of */
/* vis_dither_sweep_um[] array. */
/*****
float *vis_dither_rt_sweep_um_ptr(void)
{
    return vis_dither_sweep_um;
} /* *vis_dither_rt_sweep_um_ptr() */

/*****
/* vis_dither_rt_sweep_dac_ptr() */
/* */
/* Callable from rt side only. Returns pointer to beginning of */
/* vis_dither_sweep_dac[] array. */
/*****
short int *vis_dither_rt_sweep_dac_ptr(void)
{
    return vis_dither_sweep_dac;
} /* *vis_dither_rt_sweep_dac_ptr() */

/*****
/* vis_dither_set_flag_true() */
/* */
/* User side callable function to set various rt side flags to TRUE. */
/* Arg value is a FLAG_ID constant which defines which flag to change. */
/* FLAG_IDs defined in vis_dither.h */
/*****
int vis_dither_set_flag_true(unsigned char arg)
{
    switch(arg)

```

```

{
case FLAG_ID_USE_OFFSETS:
vis_dither_par.use_offsets = TRUE;
break;
case FLAG_ID_SWEEP_ON:
vis_dither_par.sweep_on = TRUE;
break;
case FLAG_ID_SWEEP_AUTOMATIC:
vis_dither_par.sweep_automatic = TRUE;
break;
default:
return -1;
}

return 0;
} // vis_dither_set_flag_true()

/*****
* vis_dither_set_flag_false() */
*/
/* User side callable function to set various rt side flags to FALSE. */
/* Arg value is a FLAG_ID constant which defines which flag to change. */
/* FLAG_IDs defined in vis_dither.h */
*****/
int vis_dither_set_flag_false(unsigned char arg)
{
switch(arg)
{
case FLAG_ID_USE_OFFSETS:
vis_dither_par.use_offsets = FALSE;
break;
case FLAG_ID_SWEEP_ON:
vis_dither_par.sweep_on = FALSE;
break;
case FLAG_ID_SWEEP_AUTOMATIC:
vis_dither_par.sweep_automatic = FALSE;
break;
default:
return -1;
}

return 0;
} // vis_dither_set_flag_false()

/*****
* vis_dither_sweep_increment() */
*/
/* Called by other rt modules, this function is used to manually */
/* increment the dither sweep counter. This only works if */
/* vis_dither_sweep_automatic is FALSE. Returns the vis_dither_par */
/* structure. */
*****/
VIS_DITH vis_dither_sweep_increment(void)
{
if(!vis_dither_par.sweep_automatic && vis_dither_par.sweep_on)
{
// check for the end of the array as you increment the
// sweep index

if(++(vis_dither_par.sweep_index) >=
vis_dither_sweep_dac[0] + 1)
vis_dither_par.sweep_index = 1;

}

vis_dither_par.sweep_um =
vis_dither_sweep_um[vis_dither_par.sweep_index];

vis_dither_par.sweep_dac =
vis_dither_sweep_dac[vis_dither_par.sweep_index];

return vis_dither_par;
} /* vis_dither_sweep_increment() */

/*****
* vis_dither_sweep_reset() */
*/
/* */
/* Callable by the user side, this function is used to manually */

```

```

/* reset the dither sweep counter. This only works if */
/* vis_dither_sweep_automatic is FALSE. Returns 0 if successful. */
/*****
int vis_dither_sweep_reset(unsigned char arg)
{
    vis_dither_par.sweep_index = vis_dither_sweep_dac[0];
    return 0;
} /* vis_dither_sweep_reset() */

/*****
/* vis_dither_change_damping() */
/* */
/* Callable by the user side, this function changes the damping term */
/* on the maximum allowable step size. */
/*****
int vis_dither_change_damping(unsigned char arg)
{
    int test = (int)arg;

    // apply limits

    if(test < 1) test = 1;
    else if(test > MAX_DIFF) test = MAX_DIFF;

    vis_dither_damping = test;

    return 0;
} /* vis_dither_change_damping() */

/*****
/* cleanup_module() */
/* */
/* This code cleans up the module and the printer port itself. */
/*****

void cleanup_module(void)
{
    //clean up vis dither handler

#ifdef USE_RTL_SCHED
    rt_task_delete(&vis_dither_task);
#else
    chara_unset_fast_handler();
#endif

    chara_remove_command(VIS_DITHER_GOTO);
    chara_remove_command(VIS_DITHER_MOVE_RELATIVE);
    chara_remove_command(VIS_DITHER_QUERY_POSITION);
    chara_remove_command(VIS_DITHER_QUERY_TARGET);
    chara_remove_command(VIS_DITHER_LOAD_OFFSETS);
    chara_remove_command(VIS_DITHER_LOAD_SWEEP);
    chara_remove_command(VIS_DITHER_QUERY_STATS);
    chara_remove_command(VIS_DITHER_SET_FLAG_TRUE);
    chara_remove_command(VIS_DITHER_SET_FLAG_FALSE);
    chara_remove_command(VIS_DITHER_SWEEP_RESET);
}

```

C.3 The Non-Real Time Control Code, vis_dither_control.c

```

/*****
/* vis_dither_control.c */
/*
/* Functions to interface with the vis_dither_rt module. */
/*****
/*
/* CHARA ARRAY USER INTERFACE */
/* Based on the SUSI User Interface */
/* In turn based on the CHIP User interface */
/*
/* Center for High Angular Resolution Astronomy
/* Mount Wilson Observatory, CA 91001, USA */
/*
/* Telephone: 1-626-796-5405
/* Fax : 1-626-796-6717
/*

```

```

/* email      : theo@chara.gsu.edu                                */
/* WWW        : http://www.chara.gsu.edu                          */
/*                                                     */
/* (C) This source code and its associated executable          */
/* program(s) are copyright.                                  */
/*                                                     */
/*****/
/*
/* Author : Chad Ogden                                         */
/* Date   : Aug 2003                                           */
/*****/

#include "vis_dither.h"

// arrays holding dither sweep positions. [0] element gives number n
// of positions in the full sweep, up and down. These start at element [1]

static short int sweep_dac[MAX_NUM_IN_SWEEP_ARRAY];
static float sweep_um[MAX_NUM_IN_SWEEP_ARRAY];

// for remembering dither sweep settings

float dither_lambda = 0;
float dither_frg_freq = 0;
float dither_amplitude = SWEEP_AMPLITUDE;
float dither_f6db = 0;
int max_dither_pos = 0;
float dither_range = 0.0;
bool internal_fringes = FALSE;
int damp_curr = 1;

// stuff for offsets added to the standard dither target position
// used to simulate atmospheric turbulence, etc.

float offset_data[MAX_NUM_IN_OFFSETS_ARRAY];
int num_data;

static VIS_DITH vd = {
0,
0,
0.0,
0.0,
0,
FALSE,
FALSE,
0.0,
0,
FALSE
};

// Uncomment this to enable debug messages

// #define DEBUG_DITHER_MESSAGES

/*****/
/* new_dither() */
/* */
/* User callable function for setting up a dither pattern. Includes */
/* hysteresis. */
/* */
/*****/
int new_dither(int argc, char **argv)
{
float lambda;
// float frg_freq;
float sampling;
float f6db;
char s[81], prompt[81], temp[81];

char *filename=NULL;
bool save_dither_data;
bool found_dir;
char *p;

/* Get the stuff from the command line */

if (argc > 1)
{
sscanf(argv[1], "%f", &lambda);
}

```



```

else
{
    sprintf(s, "          %.4f", DEFAULT_WAVELENGTH);
    clean_command_line();
    sprintf(prompt, "Wavelength (um)");
    if (quick_edit(prompt, s, s, NULL, FLOAT) == KEY_ESC)
        return NOERROR;
    sscanf(s, "%f", &lambda);
}

if (argc > 2)
{
    sscanf(argv[2], "%f", &sampling);
}
else
{
    strcpy(s, " 4.0");
    clean_command_line();
    if (quick_edit("Samples per fringe", s, s, NULL, FLOAT) == KEY_ESC)
        return NOERROR;
    sscanf(s, "%f", &sampling);
}

if (argc > 3)
{
    sscanf(argv[3], "%f", &f6db);
}
else
{
    strcpy(s, " 50.0");
    clean_command_line();
    if (quick_edit("-6db point (Hz)", s, s, NULL, FLOAT) == KEY_ESC)
        return NOERROR;
    sscanf(s, "%f", &f6db);
}

internal_fringes = ask_yes_no("", "Are these internal fringes?");
save_dither_data = ask_yes_no("", "Save dither data to a file?");

if(save_dither_data)
{
    sprintf(s, "%.1fx%.4fum_dither.dat", sampling, lambda);
    if (quick_edit("Filename", s, s, NULL, STRING) == KEY_ESC)
        return NOERROR;
    filename = s;
}

/* OK, if no directory specified, put it in data directory. */
for(found_dir = FALSE, p = filename; !found_dir && *p != 0; p++)
{
    if (*p == '/') found_dir = TRUE;
}

if (!found_dir)
{
    sprintf(temp, "%s%s", DATA_DIRECTORY, filename);
    filename = temp;
}
}

/* Are they reasonable numbers ? */

if (lambda < FRINGE_LAMBDA_MIN || lambda > FRINGE_LAMBDA_MAX)
{
    return error(ERROR, "Wavelength outside range %.1f nm to %.1f um.",
    FRINGE_LAMBDA_MIN, FRINGE_LAMBDA_MAX);
}

if (sampling < FRINGE_SAMPLING_MIN)
{
    return error(ERROR, "Fringe sampling must be > %.0f per fringe.",
    FRINGE_SAMPLING_MIN);
}

if (f6db < FRINGE_6DB_MIN || f6db > FRINGE_6DB_MAX)
{
    return error(ERROR, "-6dB point range %.0f to %.0f Hz.",
    FRINGE_6DB_MIN, FRINGE_6DB_MAX);
}

/* OK, calculate it */

```

```

if (calculate_dither(lambda,sampling, f6db, filename)
    != NOERROR)
    return error(ERROR,"Failed to calculate new dither.");

/* Put stuff into FIFO */
// write the sweep data in dac
    if (write_rt_data(sweep_dac,(sweep_dac[0]+1)*sizeof(sweep_dac[0])) !=
        (sweep_dac[0]+1)*sizeof(sweep_dac[0]))
    {
        return error(ERROR,"Failed to load data into FIFO.");
    }

// write the sweep data in microns
    if (write_rt_data(sweep_um,(sweep_um[0]+1)*sizeof(sweep_um[0])) !=
        (sweep_um[0]+1)*sizeof(sweep_um[0]))
    {
        return error(ERROR,"Failed to load data into FIFO.");
    }

/* Send the command */
    if (!send_rt_command(VIS_DITHER_LOAD_SWEEP, 0))
    {
// clean up the fifos from all of that data you just shoved in.
// flush_rt_data_out must be called before flush_rt_data_in.
flush_rt_data_out();
flush_rt_data_in();

        return error(ERROR,"Failed to send NEW_DITHER command.");
    }

/* Now set the bandpass filter to the same frequency */
/*
    if (set_bp_filter(frg_freq, FILTER_WIDTH, K_BAND) < 0)
    return error(ERROR,"Failed to send SET_BP_FILTER command.");
*
    if (set_lp_filter(5.0, K_BAND) < 0)
    return error(ERROR,"Failed to send SET_LP_FILTER command.");
*/
return NOERROR;
} /* new_dither() */

/*****
/* calculate_dither() */
/* Calculates the dither settings. Given the wavelength (um), fringe
/* frequency, amplitude (um) and the -6dB point it builds a triangle
/* wave, passes it through a low pass filter and converts it to
/* the right DAC values. Builds array data(0..n), where data(0)=n.
/* Fills sweep_um[] and sweep_dac[] arrays. */
/* This one include hysteresis.
*****/
int calculate_dither(float lambda, float sampling, float f6db, char *filename)
{
// float velocity;
// float volts;
// float frequency;
int n=0;
float *raw_data;
float *filt_data;
short int *data;
float inc;
int i;
digital_filter filter;
float tau,dt;
float min,max;
bool out_of_range;
short int last_data = pos_to_volts_up(SWEEP_START);
float min_pos,max_pos;
FILE *out = NULL;

char s[81];
int max_move;
#ifdef USE_RTL_SCHED
max_move = MAX_DIFF * VIS_DITHER_TASK_RATE/1000;
#else

```

```

max_move = MAX_DIFF;
#endif

    if (f6db < FRINGE_6DB_MIN)
    {
        f6db = FRINGE_6DB_MIN;
        error(WARNING, "6db Point forced to minimum of %.0f Hz.",
FRINGE_6DB_MIN);
    }

/* Theo's way, using fringe frequency.
* Data rate must be known for this to work.
* I could do this if I make vis dither get the data rate from arcccd.
*
*     // Calculate the velocity
*
*     velocity = frg_freq * lambda / 2.0;
*
*     // Now the frequency
*
*     frequency = 0.5*velocity / SWEEP_AMPLITUDE;
*
*     // How many ticks in the period ?
*
*     n = (int) ((float)TICKS_PER_SECOND/frequency + 0.5);
*/

// Number of samples in the whole dither
// total opd of sweep = sweep_amplitude * 2;
// opd of 1 data point = lambda / sampling
// n = total opd / opd per datum
// and remember n is for sweep up and down together, 2* 1 direction.

n = (int)((float)SWEEP_AMPLITUDE * 4.0 * sampling / lambda);
if(internal_fringes) n *= 2;

    /* It must be even... */

    if (n%2 != 0) n++;

// Check n.

if(n > MAX_NUM_IN_SWEEP_ARRAY)
return error(ERROR,
"Too many dither points requested. Use a longer wavelength or lower sampling.");

if(n > MAX_NUM_IN_OBSERVING_SWEEP)
error(WARNING, "Too many dither points to display. Display won't keep up.");

/* What's the increment ? */

    inc = SWEEP_AMPLITUDE/((float)(n>>1));

    /* Allocate memory */

    if ((raw_data = malloc(sizeof(float)*n+1)) == NULL ||
        (filt_data = malloc(sizeof(float)*n+1)) == NULL ||
        (data = malloc(sizeof(short int)*n+1)) == NULL)
    {
        return error(ERROR, "Out of memory.");
    }
    sweep_um[0] = sweep_dac[0] = data[0] = raw_data[0] = filt_data[0] = n;

    /* OK, fill out the array with triangle wave of positions */

    raw_data[1] = SWEEP_START;
    for (i=2; i<=n/2+1; i++) raw_data[i] = raw_data[i-1] + inc;
    for (i=n/2+2; i<=n; i++) raw_data[i] = raw_data[i-1] - inc;

/* Now we set up the filter */

    tau = sqrt(3) / (2.0 * M_PI * f6db);
    dt = 1.0 / TICKS_PER_SECOND;

    if (simple_lag(&filter, dt, tau) != f6db)
    {
        error(WARNING, "Something is wrong with the -6db point.");
    }

    /* OK, now we run everything through the filter */

```

```

    for(i=1; i<=n; i++) filt_data[i] = process_filter(filter, raw_data[i]);
    for(i=1; i<=n; i++) filt_data[i] = process_filter(filter, raw_data[i]);

    /* Now again and find the minimum and maximum */

    min = 1e32;
    max = -1e32;
    for(i=1; i<=n; i++)
    {
        if ((filt_data[i] = process_filter(filter, raw_data[i])) < min)
            min = filt_data[i];
        else if (filt_data[i] > max)
            max = filt_data[i];
    }

    /* Setup to copy this array into the global one */

    max_dither_pos = data[0];

    /* Calculate the full range of the dither */

    min_pos = 1e32;
    max_pos = 0.0;
    for(i=1; i<=n; i++)
    {
        if (filt_data[i] > max_pos) max_pos = filt_data[i];
        if (filt_data[i] < min_pos) min_pos = filt_data[i];
    }
    if (min_pos == 1e32 || max_pos == 0)
    {
        error(WARNING,"Dither range value ridiculous...set to zero");
        dither_range = 0.0;
    }
    else
    {
        dither_range = max_pos - min_pos;
    }
    if(internal_fringes) dither_range *= 2;
}

#ifdef DEBUG_DITHER_MESSAGES
error(MESSAGE,"calculate_dither(): About to check for NULL filename pre-save.");
#endif

// Try to open the file for recording the dither

// try to save the data
if(filename != NULL)
{
    // check for previous file of same name
    if(fopen(filename,"r") != NULL)
    {
        sprintf(s,"The file %s already exists.",
            filename);
        if(!ask_yes_no(s,"Shall we overwrite it?"))
        {
            filename = NULL;
        }
    }
}

// catch the guy who decided not to overwrite
if(filename != NULL)
{
}

// try to open the file

if ((out = fopen(filename,"w")) == NULL)
{
    return error(ERROR,"Failed to open file %s.",
        filename);
}

// write a little header
fprintf(out,"# Number of steps\n");
fprintf(out,"%d\n",n);
fprintf(out,"# Index\tum\tV\tDAC\n");
}

```

```

}
#ifdef DEBUG_DITHER_MESSAGES
error(MESSAGE,"calculate_dither(): Done opening file if possible.");
#endif

    /* Turn these positions into a voltage */

    out_of_range = FALSE;
    for(i=1; i<=n; i++)
    {
        /* Make the copy */

        sweep_um[i] = filt_data[i];

if(internal_fringes)
sweep_um[i] *= 2.0;

        /* Apply Neda's Hyteresis fit */

        if (i > n/2)
        {
            volts = pos_to_volts_up(filt_data[i]);
        }
        else
        {
            volts = pos_to_volts_down(filt_data[i]);
        }

        /* And convert to DAC number */

        sweep_dac[i] = volts_to_dac(volts);

// try to save the data

if(filename != NULL)
{
    fprintf(out,"%d\t%f\t%f\t%d\n",i, sweep_um[i],
        volts, sweep_dac[i]);
}

// done saving the data

        if (i != 1)
        {
            if (abs((float)(sweep_dac[i]-last_data)) > max_move)
            {
                free(raw_data);
                free(filt_data);
                free(data);
                max_dither_pos = 0;
                if(out != NULL)

                    return error(ERROR,
                        "Unreasonable jump from %d to %d at %d.",
                        data[i],last_data,i);
            }

            last_data = sweep_dac[i];
        }

#ifdef DEBUG_DITHER_MESSAGES
error(MESSAGE,"calculate_dither(): done calculating dither positions.");
#endif

if(out != NULL)
    fclose(out);

#ifdef DEBUG_DITHER_MESSAGES
error(MESSAGE,"calculate_dither(): out file closed.");
#endif

        if (abs((float)(sweep_dac[1]-last_data)) > max_move)
        {
            free(raw_data);

```

```

        free(filt_data);
        free(data);
        max_dither_pos = 0;

return error(ERROR,
    "Unreasonable jump from %d to %d at %d.",
        last_data,data[1],n);
    }
    if (out_of_range)
    {
        error(WARNING,"One or more values where outside range of 0-150V.");
    }

    /* Clean up the memory */

    clean_up_filter(filter);
    free(raw_data);
    free(filt_data);
/* Set globals for future reference */

    dither_lambda = lambda;
    //    dither_frg_freq = frg_freq;
    dither_amplitude = SWEEP_AMPLITUDE;
    dither_f6db = f6db;

    error(MESSAGE,"Dither range %f um over %d samples.",
        dither_range,max_dither_pos/2);

#ifdef DEBUG_DITHER_MESSAGES
error(MESSAGE,"calculate_dither(): done. returning.");
#endif

// purge rt FIFOs
flush_data(0,NULL);

    return NOERROR;
}/* calculate_dither() */

/*****
/* load_dither_offsets(char *infile_name) */
/* */
/* Takes input file name in string. Counts number of entries in file. */
/* Scans entries into the offset_data[] array. */
*****/
int load_dither_offsets(char *infile_name)
{
    int i, datacount=0;
    FILE *infile = NULL;
    FILE *outfile = NULL;

    // open the data file

    if( (infile = fopen(infile_name,"r")) == NULL)

return error(ERROR,"Unable to open file: %s",infile_name);

    // count the number of data as they are read in

    while(fscanf(infile, "%f", &(offset_data[datacount])) != EOF)
    {
        // Note that when this loop kicks out, datacount will
        // equal the number of elements used in the array,
        // from 0 to datacount-1.

        datacount ++;

        // if the data array is full,

        if(datacount >= MAX_NUM_IN_OFFSETS_ARRAY)

            break;
    }

    // close the input file

    fclose(infile);

    // open the output file for logging
    if( ( outfile = fopen(VIS_DITHER_LOG_FILE,"a") ) == NULL)

```

```

{
// oops didn't work
error(MESSAGE,"Failed to open log file to write offsets.");
}
// write the header of the offset data to the log file
else
{
fprintf(outfile,"# Dither Offset data calculated:\n");
fprintf(outfile,"# Model offset (um)\t DAC offset\n");
}

// convert the phases array into dac units
// phases are in microns.

for(i=0; i<datacount; i++)
{
// write the um offset to log file
if(outfile != NULL)
fprintf(outfile,"%f\t", offset_data[i]);

offset_data[i] = volts_to_dac(
pos_to_volts_linear(
offset_data[i] / OPTICAL_PATH_LAB_PASSES
)
); //>>>> double check that this works

// write the dac conversion to log file
if(outfile != NULL)
fprintf(outfile,"%f\n", offset_data[i]);
}

// close the log file
if(outfile != NULL)
{
fclose(outfile);
message(system_window,"Offset data written to log file.");
}
//write offsets data to fifo
if( write_rt_data( &datacount, sizeof(datacount)) != sizeof(datacount) )
return -2;

if( write_rt_data( &offset_data,sizeof(offset_data[0])*datacount )
!= sizeof(offset_data[0])*datacount )
return -2;

//call vis_dither_load_offsets
if ( !send_rt_command(VIS_DITHER_LOAD_OFFSETS,0) )
return -1;

return NOERROR;

} /* load_dither_offsets() */

/*****
/* call_load_dither_offsets() */
/* */
/* User callable function, calls load_dither_offsets(). Takes */
/* input file name as command line argument. */
*****/
int call_load_dither_offsets(int argc, char **argv)
{
int err;

if(argc != 2)
{
error(ERROR,"Usage: (function) filename");
return -1;
}

err = load_dither_offsets(argv[1]);

return err;

} /* call_load_dither_offsets() */

/*****
/* set_vis_dith_flag() */
/* */
/* Sets boolean flag on rt side. Sets boolean flag on rt side. */
/* First arg is the flag id constant, defined in vis_dither.h. */
*****/

```

```

/* Next argument is the boolean which the rt flag will be set to. */
/*****
int set_vis_dith_flag(unsigned char flag_id, bool value)
{
    //call rt set true command if requested value true

    if(value)
    {
        if( !send_rt_command(VIS_DITHER_SET_FLAG_TRUE,flag_id) )
        return -1;
    }

    // call rt set false if requested value false

    else
    {
        if( !send_rt_command(VIS_DITHER_SET_FLAG_FALSE,flag_id) )
        return -1;
    }

    return NOERROR;
} // set_vis_dith_rt_flag()

/*****
/* call_set_vis_dith_flag() */
/* */
/* General purpose front end for changing rt flg values. */
/* Requires a wrapper function which passes its argc and argv, and */
/* specifies which flag to change. */
/*****
int call_set_vis_dith_flag(int argc, char ** argv, unsigned char flag_id)
{
    bool state;
    int err;
    // this is if the user specified on or off

    if (argc == 2)
    {
        if( strcmp(argv[1],"on") == 0 || strcmp(argv[1],"ON") == 0
        || strcmp(argv[1],"1") == 0)

        state = TRUE;

        else if( strcmp(argv[1],"off") == 0
        || strcmp(argv[1],"OFF") == 0
        || strcmp(argv[1],"0") == 0)

        state = FALSE;

        else
        {
            return error(ERROR,
            "Usage: (function) (on,off,or no argument)");
        }
    }

    // this is if the user called the function with no argument,
    // meaning to toggle the enable state

    else if (argc == 1)
    {
        switch(flag_id)
        {
            case FLAG_ID_USE_OFFSETS:
                state = !vd.use_offsets;
                break;
            case FLAG_ID_SWEEP_ON:
                state = !vd.sweep_on;
                break;
            case FLAG_ID_SWEEP_AUTOMATIC:
                state = !vd.sweep_automatic;
                break;
            default:
                return -1;
        }
    }
}

```



```

// this is if the user didn't call it right

else
{
return error(ERROR,"Usage: (function) (on,off,or no argument)");
}

// call set_vis_dith_flag().

switch(flag_id)
{
case FLAG_ID_USE_OFFSETS:
if( (err=set_vis_dith_flag(FLAG_ID_USE_OFFSETS,state))
== NOERROR)
vd.use_offsets = state;
return err;
case FLAG_ID_SWEEP_ON:
if( (err=set_vis_dith_flag(FLAG_ID_SWEEP_ON,state))
== NOERROR)
vd.sweep_on = state;
return err;
case FLAG_ID_SWEEP_AUTOMATIC:
if( (err=
set_vis_dith_flag(FLAG_ID_SWEEP_AUTOMATIC,
state))
== NOERROR)
vd.sweep_automatic = state;
return err;
default:
return -1;
}

} /* call_set_vis_dith_flag()*/

/*****
/* call_enable_dither_offsets() */
/* */
/* User callable wrapper for call_set_vis_dith_flag(). Used to turn */
/* the use of rt dither offsets on/off from the user side. */
/* Command line arguments On and OFF, no argument toggles the state of */
/* enable_offsets. */
*****/
int call_enable_dither_offsets(int argc, char ** argv)
{
return call_set_vis_dith_flag(argc,argv,FLAG_ID_USE_OFFSETS);
} /* call_enable_dither_offsets()*/

/*****
/* call_enable_vis_dither_sweep() */
/* */
/* User callable wrapper for call_set_vis_dith_flag(). Used to turn */
/* on/off vis dither sweep from the user side. */
/* Command line arguments On and OFF, no argument toggles the state of */
/* sweep_on. */
*****/
int call_enable_vis_dither_sweep(int argc, char ** argv)
{
return call_set_vis_dith_flag(argc,argv,FLAG_ID_SWEEP_ON);
} /* call_enable_vis_dither_sweep()*/

/*****
/* call_enable_vis_dither_sweep_automatic() */
/* */
/* User callable wrapper for call_set_vis_dith_flag(). Used to turn */
/* on/off automatic incrementing of the vis dither sweep by the handler.*/
/* Command line arguments On and OFF, no argument toggles the state of */
/* sweep_automatic. */
*****/
int call_enable_vis_dither_sweep_automatic(int argc, char ** argv)
{
return call_set_vis_dith_flag(argc,argv,FLAG_ID_SWEEP_AUTOMATIC);
} /* call_enable_vis_dither_sweep_automatic()*/

/*****
/* vis_dither_sweep_dac_ptr() */
/* */
/* Called by other user side programs to access the address of the */

```

```

/* sweep_dac array. Pointers set to the return value of this function */
/* should be of the const short int *<name> type, as to be read-only. */
/*****
short int *vis_dither_sweep_dac_ptr(void)
{
return sweep_dac;
} /* vis_dither_sweep_dac_ptr() */

/*****
/* vis_dither_sweep_um_ptr() */
/* */
/* Called by other user side programs to access the address of the */
/* sweep_um array. Pointers set to the return value of this function */
/* should be of the const float *<name> type, as to be read-only. */
/*****
float *vis_dither_sweep_um_ptr(void)
{
return sweep_um;
} /* vis_dither_sweep_um_ptr() */

/*****
/* sweep_reset() */
/* */
/* Call this function to set the dither sweep index to 1 on the */
/* rt side. Don't worry about the discontinuity, the rt side */
/* checks and won't move faster than is good for the piezo. */
/*****
int sweep_reset(void)
{
return send_rt_command(VIS_DITHER_SWEEP_RESET,0);
} /* sweep_reset() */

/*****
/* call_sweep_reset() */
/* */
/* User callable wrapper for sweep_reset. */
/*****
int call_sweep_reset(int argc, char **argv)
{
if(!sweep_reset())
return error(ERROR,"Unable to reset vis dither sweep.");
else
{
message(system_window,"Vis dither sweep reset.");
return NOERROR;
}
} /* call_sweep_reset() */

/*****
/* change_damping() */
/* */
/* Call this function to set the damping term on the max step size */
/* on the rt side. */
/*****
int change_damping(char damp)
{
return send_rt_command(VIS_DITHER_CHANGE_DAMPING,damp);
} /* change_damping() */

/*****
/* call_change_damping() */
/* */
/* User callable wrapper for change_damping. Knows that the arg should */
/* be >0 and <= MAX_DIFF. */
/*****
int call_change_damping(int argc, char **argv)
{
char s[81], prompt[81];
int damp;

if (argc > 1)
{
sscanf(argv[1], "%d", &damp);
}
else
{
clean_command_line();
sprintf(s, "%d", damp_curr);
sprintf(prompt, "Damping term = 1/(1 to %d):", MAX_DIFF);
if (quick_edit(prompt,s,NULL,INTEGER) == KEY_ESC)

```

```

                                return NOERROR;
                                sscanf(s,"%d",&damp);
                                }

if(damp < 1) damp = 1;
else if(damp > MAX_DIFF) damp = MAX_DIFF;

if( !change_damping((char)damp) )
return error(ERROR,
"Unable to send new damping term to vis_dither_rt.");

else
{
damp_curr = damp;
return NOERROR;
}
} /* call_change_damping() */

```

References

- Anderson, J. A. (1920). Application of Michelson's interferometer method to the measurement of close double stars. *ApJ*, 51:263–275.
- Armstrong, J. T., Mozurkewich, D., Rickard, L. J., Hutter, D. J., Benson, J. A., Bowers, P. F., Elias, N. M., Hummel, C. A., Johnston, K. J., Buscher, D. F., Clark, J. H., Ha, L., Ling, L.-C., White, N. M., and Simon, R. S. (1998). The Navy Prototype Optical Interferometer. *ApJ*, 496:550.
- Assus, P., Choplin, H., Corteggiani, J. P., Cuot, E., Gay, J., Journet, A., Merlin, G., and Rabbia, Y. (1979). The CERGA infrared interferometer. *Journal of Optics*, 10:345–350.
- Aufdenberg, J. P., Hauschildt, P. H., Baron, E., Nordgren, T. E., Burnley, A. W., Howarth, I. D., Gordon, K. D., and Stansberry, J. A. (2002). The Spectral Energy Distribution and Mass-Loss Rate of the A-Type Supergiant Deneb. *ApJ*, 570:344–368.
- Bagnuolo, W. G. (1993). Chara preliminary engineering design report, appendix e: Diffraction effects. Technical report, Georgia State University Center for High Angular Resolution Astronomy.
- Bagnuolo, W. G., Taylor, S. F., McAlister, H. A., ten Brummelaar, T. A., and Gies, D. R. (2005). First Results from the CHARA Array. V. Binary Star Astrometry: the Case of 12 Persei. *AJ*. Submitted.
- Baldwin, J. E., Beckett, M. G., Boysen, R. C., Burns, D., Buscher, D. F., Cox, G. C., Haniff, C. A., Mackay, C. D., Nightingale, N. S., Rogers, J., Scheuer, P. A. G., Scott, T. R., Tuthill, P. G., Warner, P. J., Wilson, D. M. A., and Wilson, R. W. (1996). The first images from an optical aperture synthesis array: mapping of Capella with COAST at two epochs. *A&A*, 306:L13.
- Baldwin, J. E., Haniff, C. A., Mackay, C. D., and Warner, P. J. (1986). Closure phase in high-resolution optical imaging. *Nature*, 320:595–597.
- Basden, A. G., Haniff, C. A., Mackay, C. D., Bridgeland, M. T., Wilson, D. M., Young, J. S., and Buscher, D. F. (2004). A new photon-counting spectrometer for the COAST. In *New Frontiers in Stellar Interferometry, Proceedings of SPIE*

- Volume 5491. Edited by Wesley A. Traub. Bellingham, WA: The International Society for Optical Engineering, 2004., p.677, page 677.*
- Benson, J. A., Dyck, H. M., and Howell, R. R. (1995). Technique for obtaining visibility amplitudes from atmospherically disturbed interferograms. *Appl. Opt.*, 34:51–57.
- Berger, D. H. (2003). *Longitudinal Dispersion Compensation for a Long Baseline Optical Interferometer*. PhD thesis, Georgia State University.
- Berger, D. H., Gies, D. R., ten Brummelaar, T. A., Henry, T. J., Sturmann, J., Sturmann, L., Turner, N. H., Ridgway, S. T., Aufdenberg, J. P., and Mérand, A. (2005). First Results from the CHARA Array. IV. The interferometric Radii of Low-Mass Stars. *ApJ*. Submitted.
- Berger, J.-P., Haguenaue, P., Kern, P. Y., Rousselet-Perraut, K., Malbet, F., Gluck, S., Lagny, L., Schanen-Duport, I., Laurent, E., Delboulbe, A., Tatulli, E., Traub, W. A., Carleton, N., Millan-Gabet, R., Monnier, J. D., Pedretti, E., and Ragland, S. (2003). An integrated-optics 3-way beam combiner for IOTA. In *Interferometry for Optical Astronomy II. Edited by Wesley A. Traub . Proceedings of the SPIE, Volume 4838*, pages 1099–1106.
- Bessell, M. S. (1990). UBVRI passbands. *PASP*, 102:1181–1199.
- Blackwell, D. E., Petford, A. D., and Shallis, M. J. (1980). Use of the infra-red flux method for determining stellar effective temperatures and angular diameters - The stellar temperature scale. *A&A*, 82:249–252.
- Bohlin, R. C., and Gilliland, R. L. (2004). Hubble Space Telescope Absolute Spectrophotometry of Vega from the Far-Ultraviolet to the Infrared. *AJ*, 127:3508–3515.
- Bonneau, D., Koechlin, L., Oneto, J. L., and Vakili, F. . (1981). Stellar diameter measurements by two-telescope interferometry in optical wavelengths. *A&A*, 103:28–34.
- Born, M., and Wolf, E. (1998). *Principles of Optics, 7th edition*. Cambridge University Press.
- Bracewell, R. (1965). *The Fourier Transform and its applications*. McGraw-Hill Electrical and Electronic Engineering Series, New York.
- Brown, R. H., Davis, J., and Allen, L. R. (1967a). The stellar interferometer at Narrabri Observatory I. A. description of the instrument and the observational procedure. *MNRAS*, 137:375.

- Brown, R. H., Davis, J., Allen, L. R., and Rome, J. M. (1967b). The stellar interferometer at Narrabri Observatory-II. The angular diameters of 15 stars. *MNRAS*, 137:393.
- Buscher, D. (1988a). Optimizing a ground-based optical interferometer for sensitivity at low light levels. *MNRAS*, 235:1203–1226.
- Buscher, D. F. (1988b). *Getting the most out of C.O.A.S.T.* PhD thesis, Cambridge University.
- Cayrel de Strobel, G. (1960). Utilisation des intensités centrales des raies de Balmer pour déterminer les températures de surface, de brillance et le diamètre apparent de 33 étoiles. *Annales d'Astrophysique*, 23:278.
- Colavita, M., Akeson, R., Wizinowich, P., Shao, M., Acton, S., Beletic, J., Bell, J., Berlin, J., Boden, A., Booth, A., Boutell, R., Chaffee, F., Chan, D., Chock, J., Cohen, R., Crawford, S., Creech-Eakman, M., Eychaner, G., Felizardo, C., Gathright, J., Hardy, G., Henderson, H., Herstein, J., Hess, M., Hovland, E., Hrynevych, M., Johnson, R., Kelley, J., Kendrick, R., Koresko, C., Kurpis, P., Le Mignant, D., Lewis, H., Ligon, E., Lupton, W., McBride, D., Mennesson, B., Millan-Gabet, R., Monnier, J., Moore, J., Nance, C., Neyman, C., Niessner, A., Palmer, D., Reder, L., Rudeen, A., Saloga, T., Sargent, A., Serabyn, E., Smythe, R., Stomski, P., Summers, K., Swain, M., Swanson, P., Thompson, R., Tsubota, K., Tumminello, A., van Belle, G., Vasisht, G., Vause, J., Walker, J., Wallace, K., and Wehmeier, U. (2003). Observations of DG Tauri with the Keck Interferometer. *ApJ*, 592:L83–L86.
- Colavita, M. M., Wizinowich, P. L., and Akeson, R. L. (2004). Keck Interferometer status and plans. In *New Frontiers in Stellar Interferometry, Proceedings of SPIE Volume 5491. Edited by Wesley A. Traub. Bellingham, WA: The International Society for Optical Engineering, 2004*, page 454.
- Coude du Foresto, V., Borde, P. J., Merand, A., Baudouin, C., Remond, A., Perrin, G. S., Ridgway, S. T., ten Brummelaar, T. A., and McAlister, H. A. (2003). FLUOR fibered beam combiner at the CHARA array. In *Interferometry for Optical Astronomy II. Edited by Wesley A. Traub. Proceedings of the SPIE, Volume 4838*, pages 280–285.
- Coude Du Foresto, V., Perrin, G., Ruilier, C., Mennesson, B. P., Traub, W. A., and Lacasse, M. G. (1998). FLUOR fibered instrument at the IOTA interferometer. In *Proc. SPIE Vol. 3350, p. 856-863, Astronomical Interferometry, Robert D. Reasenberg; Ed.*, pages 856–863.
- Coude Du Foresto, V., Ridgway, S., and Mariotti, J.-M. (1997). Deriving object visibilities from interferograms obtained with a fiber stellar interferometer. *A&AS*, 121:379–392.

- Coulman, C. E. (1985). Fundamental and applied aspects of astronomical 'seeing'. *ARA&A*, 23:19–57.
- Cox, G. C. (1994). The COAST interferometer. In *IAU Symp. 158: Very High Angular Resolution Imaging*, page 163.
- Creech-Eakman, M. J., Buscher, D. F., Haniff, C. A., and Romero, V. D. (2004). The Magdalena Ridge Observatory Interferometer: a fully optimized aperture synthesis array for imaging. In *New Frontiers in Stellar Interferometry, Proceedings of SPIE Volume 5491. Edited by Wesley A. Traub. Bellingham, WA: The International Society for Optical Engineering, 2004*, page 405.
- Davis, J., and Tango, W. J. (1985). The Sydney University 11.4 M prototype stellar interferometer. *Proceedings of the Astronomical Society of Australia*, 6:34–38.
- Davis, J., Tango, W. J., Booth, A. J., Brummelaar, T. A. T., Minard, R. A., and Owens, S. M. (1999). The Sydney University Stellar Interferometer - I. The instrument. *MNRAS*, 303:773–782.
- Du-Forest, V. C., and Ridgway, S. T. (1992). Fluor - a Stellar Interferometer Using Single-Mode Fibers. In *High-Resolution Imaging by Interferometry*, page 731.
- Dyck, H. M., Benson, J. A., and Ridgway, S. T. (1993). IRMA - A prototype infrared Michelson stellar interferometer. *PASP*, 105:610–615.
- Fizeau, H. (1868). *C. R. Acad. Sci.*, 66:932.
- Ford, V. G., Lisman, P. D., Shaklan, S., White, M., and Hull, T. (2004). TPF Coronagraph: Pathway toward full mission architecture. *American Astronomical Society Meeting Abstracts*, 204.
- Fowles, G. R. (1989). *Introduction to Modern Optics*. Dover.
- Fresnel, A.-J. (1816). Mémoire sur la diffraction de la lumière, où l'on examine particulièrement le phénomène des franges colorées que présentent les ombres des corps éclairés par un point lumineux. *Annales de Chimie et de Physique*, 1:239–281.
- Fried, D. L. (1965). Statistics of a Geometric Representation of Wavefront Distortion. *Journal of the Optical Society of America (1917-1983)*, 55:1427.
- Glindemann, A., Albertsen, M., Andolfato, L., Avila, G., Ballester, P., Bauvir, B., Delplancke, F., Derie, F., Dimmler, M., Duhoux, P., di Folco, E., Frahm, R., Galliano, E., Gilli, B., Giordano, P. N., Gitton, P. B., Guisard, S., Housen, N., Hummel, C. A., Huxley, A., Karban, R., Kervella, P., Kiekebusch, M., Koehler, B., Leveque, S. A., Licha, T., Longinotti, A., McKay, D. J., Menardi, S., Monnet, G. J., Morel, S., Paresce, F., Percheron, I., Petr-Gotzens, M., Phan Duc, T.,

- Pott, J.-U., Puech, F., Rantakyro, F. T., Richichi, A., Sabet, C., Scales, K. L., Schoeller, M., Schuhler, N., van den Ancker, M., Vannier, M., Wallander, A., Witkowski, M., and Wilhelm, R. C. (2004). VLT technical advances: present and future. In *New Frontiers in Stellar Interferometry, Proceedings of SPIE Volume 5491*. Edited by Wesley A. Traub. Bellingham, WA: The International Society for Optical Engineering, 2004, page 447.
- Goodman, J. W. (1968). *Introduction to Fourier Optics*. McGraw-Hill.
- Hale, D. D. S., Bester, M., Danchi, W. C., Fitelson, W., Hoss, S., Lipman, E. A., Monnier, J. D., Tuthill, P. G., and Townes, C. H. (2000). The Berkeley Infrared Spatial Interferometer: A Heterodyne Stellar Interferometer for the Mid-Infrared. *ApJ*, 537:998–1012.
- Hanbury Brown, R., Davis, J., and Allen, L. R. (1974). The angular diameters of 32 stars. *MNRAS*, 167:121–136.
- Hanbury Brown, R., and Twiss, R. Q. (1956a). A test of a new type of stellar interferometer on Sirius. *Nature*, 178:1046.
- Hanbury Brown, R., and Twiss, R. Q. (1956b). Correlation Between Photons In Two Coherent Beams of Light. *Nature*, 178:27.
- Hill, J. M., and Salinari, P. (2004). The Large Binocular Telescope project. In *Proceedings of the SPIE, Volume 5489*, pages 603–614.
- Holfeltz, S. T., and Taff, L. G. (1992). HST FGS Calibration of Asteroid Angular Diameters. *Bulletin of the American Astronomical Society*, 24:739.
- Horton, A. J., Buscher, D. F., and Haniff, C. A. (2001). Diffraction losses in ground-based optical interferometers. *MNRAS*, 327:217–226.
- Hrynevych, M. (1992). *Diffraction Effects in Michelson Stellar Interferometry*. PhD thesis, University of Sydney.
- Hummel, C. A., Mozurkewich, D., Armstrong, J. T., Hajian, A. R., Elias, N. M., and Hutter, D. J. (1998). Navy Prototype Optical Interferometer Observations of the Double Stars Mizar A and Matar. *AJ*, 116:2536–2548.
- Huygens, C. (1690). *Traité de la Lumière*.
- Johnson, M. A., Betz, A. L., and Townes, C. H. (1974). Results from the 5.5 Meter Infrared Stellar Interferometer. *BAAS*, 6:450.

- Komolgorov, A. N. (1941). *The Local Structure of Turbulence in Incompressible Viscous Fluid for Very Large Reynolds' Numbers*, in *Turbulence: 1961*, Ed: Friedlander, S. K. and Topper, L., pages 151–155. Interscience Publishers, Inc., New York.
- Kotani, T., Nishikawa, J., Sato, K., Yoshizawa, M., Ohishi, N., Fukushima, T., Torii, Y., Matsuda, K., Kubo, K., Iwashita, H., and Suzuki, S. (2003). Long-baseline optical fiber interferometer instruments and science. In *Interferometry for Optical Astronomy II. Edited by Wesley A. Traub . Proceedings of the SPIE, Volume 4838*, pages 1370–1377.
- Labeyrie, A. (1970). Attainment of Diffraction Limited Resolution in Large Telescopes by Fourier Analysing Speckle Patterns in Star Images. *A&A*, 6:85.
- Labeyrie, A. (1975). Interference fringes obtained on VEGA with two optical telescopes. *ApJ*, 196:L71–L75.
- Labeyrie, A., Schumacher, G., Dugue, M., Thom, C., and Bourlon, P. (1986). Fringes obtained with the large 'boules' interferometer at CERGA. *A&A*, 162:359–364.
- Lawson, P. R. (2000a). Notes On The History Of Stellar Interferometry. In *Principles of Long Baseline Stellar Interferometry*, pages 325–332. JPL Publication 000-009 07/00, available online at olbin.jpl.nasa.gov.
- Lawson, P. R., editor (2000b). *Principles of Long Baseline Stellar Interferometry*. JPL Publication 000-009 07/00, available online at olbin.jpl.nasa.gov.
- Leggett, S. K., Mountain, C. M., Selby, M. J., Blackwell, D. E., Booth, A. J., Haddock, D. J., and Petford, A. D. (1986). The effective temperatures, diameters and luminosities of 22 bright stars by application of the infrared flux method. *A&A*, 159:217–222.
- Mason, B. D., Wycoff, G. L., Hartkopf, W. I., Douglass, G. G., and Worley, C. E. (2001). The 2001 US Naval Observatory Double Star CD-ROM. I. The Washington Double Star Catalog. *AJ*, 122:3466–3471.
- McAlister, H. A., ten Brummelaar, T. A., Gies, D. R., Huang, W., Bagnuolo, W. G., Shure, M. A., Sturmann, J., Sturmann, L., Turner, N. H., Taylor, S. F., Berger, D. H., Baines, E. K., Grundstrom, E., Ogden, C., Ridgway, S. T., and van Belle, G. (2005). First Results from the CHARA Array. I. An Interferometric and Spectroscopic Study of the Fast Rotator α Leonis (Regulus). *ApJ*, 628:439–452.
- Merand, A., Birlan, M., Lelu de Brach, R., and Coude Du Foresto, V. (2004). Remote observations with FLUOR and the CHARA Array. In *New Frontiers in Stellar Interferometry, Proceedings of SPIE Volume 5491. Edited by Wesley A. Traub. Bellingham, WA: The International Society for Optical Engineering*, page 1661.

- Mérand, A., Kervella, P., Coudé Du Foresto, V., Ridgway, S. T., Aufdenberg, J. P., Ten Brummelaar, T. A., Berger, D. H., Sturmann, J., Sturmann, L., Turner, N. H., and McAlister, H. A. (2005). The projection factor of δ Cephei. A calibration of the Baade-Wesselink method using the CHARA Array. *A&A*, 438:L9–L12.
- Michelson, A. A. (1890). On The Application of Interference Methods to Astronomical Measurements. *Phil. Mag.*, 30:1.
- Michelson, A. A. (1891). *Nature*, 45:160.
- Michelson, A. A., and Pease, F. G. (1921). Measurement of the diameter of alpha Orionis with the interferometer. *ApJ*, 53:249–259.
- Miller, D. D., and Fischer, D. (2004). Current status of the TPF formation flying interferometer concept. In *New Frontiers in Stellar Interferometry, Proceedings of SPIE Volume 5491. Edited by Wesley A. Traub. Bellingham, WA: The International Society for Optical Engineering*, page 296.
- Monnier, J. D. (2003). Optical interferometry in astronomy. *Reports of Progress in Physics*, 66:789–857.
- Monnier, J. D., Berger, J.-P., Millan-Gabet, R., and Ten Brummelaar, T. A. (2004a). The Michigan Infrared Combiner (MIRC): IR imaging with the CHARA Array. In *New Frontiers in Stellar Interferometry, Proceedings of SPIE Volume 5491. Edited by Wesley A. Traub. Bellingham, WA: The International Society for Optical Engineering*, page 1370.
- Monnier, J. D., Traub, W. A., Schloerb, F. P., Millan-Gabet, R., Berger, J.-P., Pedretti, E., Carleton, N. P., Kraus, S., Lacasse, M. G., Brewer, M., Ragland, S., Ahearn, A., Coldwell, C., Hagenauer, P., Kern, P., Labeye, P., Lagny, L., Malbet, F., Malin, D., Maymounkov, P., Morel, S., Papaliolios, C., Perraut, K., Pearlman, M., Porro, I. L., Schanen, I., Souccar, K., Torres, G., and Wallace, G. (2004b). First Results with the IOTA3 Imaging Interferometer: The Spectroscopic Binaries λ Virginis and WR 140. *ApJ*, 602:L57–L60.
- Morossi, C., and Malagnini, M. L. (1985). Observed and computed spectral flux distribution of non-supergiant O9-G8 stars. III - Determination of T(eff) for the stars in the Breger Catalogue. *A&AS*, 60:365–372.
- Mourard, D., Tallon-Bosc, I., Blazit, A., Bonneau, D., Merlin, G., Morand, F., Vakili, F., and Labeyrie, A. (1994). The GI2T interferometer on Plateau de Calern. *A&A*, 283:705–713.
- Mozurkewich, D., Armstrong, J. T., Hindsley, R. B., Quirrenbach, A., Hummel, C. A., Hutter, D. J., Johnston, K. J., Hajian, A. R., Elias, N. M., Buscher, D. F.,

- and Simon, R. S. (2003). Angular Diameters of Stars from the Mark III Optical Interferometer. *AJ*, 126:2502–2520.
- Nishikawa, J., Sato, K., Yoshizawa, M., Fukushima, T., Machida, Y., Honma, Y., Torii, Y., Matsuda, K., Kubo, K., Iwashita, H., Suzuki, S., Kubota, Y., Shimazaki, K., and Nemoto, Y. (2000). Mitaka optical and infrared array first stage (MIRA-I.1) instruments. In *Proc. SPIE Vol. 4006, Interferometry in Optical Astronomy*, Pierre J. Lena; Andreas Quirrenbach; Eds., pages 681–687.
- Nishikawa, J., Yoshizawa, M., Ohishi, N., Suzuki, S., Torii, Y., Matsuda, K., Kubo, K., Iwashita, H., Yokoi, T., Kotani, T., and Sato, K. (2004). MIRA-I.2: recent progress. In *New Frontiers in Stellar Interferometry, Proceedings of SPIE Volume 5491. Edited by Wesley A. Traub. Bellingham, WA: The International Society for Optical Engineering, 2004., p.520*, page 520.
- Ochsenbein, F., and Halbwachs, J. L. (1982). A list of stars with large expected angular diameters. *A&AS*, 47:523–531.
- Pasinetti Fracassini, L. E., Pastori, L., Covino, S., and Pozzi, A. (2001). Catalogue of Apparent Diameters and Absolute Radii of Stars (CADARS) - Third edition - Comments and statistics. *A&A*, 367:521–524.
- Pease, F. G. (1931). *Exakt. Natur.*, 10:84.
- Perrin, G., Lai, O., Lena, P. J., and Coude du Foresto, V. (2000). Fibered large interferometer on top of Mauna Kea: OHANA, the optical Hawaiian array for nanoradian astronomy. In *Proc. SPIE Vol. 4006, Interferometry in Optical Astronomy*, Pierre J. Lena; Andreas Quirrenbach; Eds., pages 708–714.
- Perrin, G. S., Lai, O., Woillez, J. M., Guerin, J., Kotani, T., Vergnole, S., Adamson, A. J., Ftacilas, C., Guyon, O., Lena, P. J., Nishikawa, J., Reynaud, F., Roth, K. C., Ridgway, S. T., Tokunaga, A. T., and Wizinowich, P. L. (2004). 'ohana. In *New Frontiers in Stellar Interferometry, Proceedings of SPIE Volume 5491. Edited by Wesley A. Traub. Bellingham, WA: The International Society for Optical Engineering*, page 391.
- Quirrenbach, A. (2000). Observing Through the Turbulent Atmosphere. In *Principles of Long Baseline Stellar Interferometry*, pages 71–86. JPL Publication 000-009 07/00, available online at olbin.jpl.nasa.gov.
- Ratcliffe, J. A. (1956). Some Aspects of Diffraction Theory and their Application to the Ionosphere. *Reports of Progress in Physics*, 19:188–267.
- Richardson, L. F. (1922). *Weather Prediction by Numerical Process*. Cambridge University Press.

- Richichi, A., Percheron, I., and Khristoforova, M. (2004). CHARM2, an updated of CHARM catalog (Richichi, 2005). *VizieR Online Data Catalog*, 343:10773.
- Roddier, F. (1981). The Effects of Atmospheric Turbulence in Optical Astronomy. *Prog. Optics*, 19:281–376.
- Schwarzschild, K. (1896). Über Messung von Doppelsternen durch Interferenzen. *Astronomische Nachrichten*, 139:353.
- Shao, M. (2004). Science overview and status of the SIM project. In *New Frontiers in Stellar Interferometry, Proceedings of SPIE Volume 5491. Edited by Wesley A. Traub. Bellingham, WA: The International Society for Optical Engineering*, page 328.
- Shao, M., Colavita, M., Staelin, D., Simon, R., and Johnston, K. (1986). Present Status and Future Plans for the Two Color Astrometric Interferometer Project. In *IAU Symp. 109: Astrometric Techniques*, page 331.
- Shao, M., Colavita, M. M., Hines, B. E., Staelin, D. H., and Hutter, D. J. (1988). The Mark III stellar interferometer. *A&A*, 193:357–371.
- Shao, M., and Staelin, D. H. (1980). First fringe measurements with a phase-tracking stellar interferometer. *Appl. Opt.*, 19:1519–1522.
- Sherman, G. C. (1967). Application of the convolution theorem to Rayleigh’s integral formulas. *Journal of the Optical Society of America*, 57:546.
- Stéphan, E. (1874). *C. R. Acad. Sci.*, 78:1008.
- Tatarski, V. I. (1961). *Wave Propagation in a Turbulent Medium*. Dover, New York.
- Taylor, S. F., Harvin, J. A., and McAlister, H. A. (2003). The CHARA Catalog of Orbital Elements of Spectroscopic Binary Stars. *PASP*, 115:609–617.
- ten Brummelaar, T. A. (1992). *Taking the Twinkle Out of the Stars: An Adaptive Wavefront Tilt Correction Servo and Preliminary Seeing Study for SUSI*. PhD thesis, University of Sydney.
- ten Brummelaar, T. A., McAlister, H. A., Ridgway, S. T., Bagnuolo, W. G., Turner, N. H., Sturmann, L., Sturmann, J., Berger, D. H., Ogden, C. E., Cadman, R., Hartkopf, W. I., Hopper, C. H., and Shure, M. A. (2005). First Results from the CHARA Array. II. A Description of the Instrument. *ApJ*, 628:453–465.
- Thureau, N. D., Boysen, R. C., Buscher, D. F., Haniff, C. A., Pedretti, E., Warner, P. J., and Young, J. S. (2003). Fringe envelope tracking at COAST. In *Interferometry for Optical Astronomy II. Edited by Wesley A. Traub. Proceedings of the SPIE, Volume 4838*, pages 956–963.

- Traub, W. A. (1988). Polarization Effects in Stellar Interferometers. In *NOAO-ESO Conference on High-Resolution Imaging by Interferometry: Ground-Based Interferometry at Visible and Infrared Wavelengths, Garching bei München, Germany, Mar. 15-18, 1988. Edited by F. Merkle, ESO Conference and Workshop Proceedings No. 29*, page 1029.
- Traub, W. A. (1998). Recent results from the IOTA interferometer. In *Proc. SPIE Vol. 3350, p. 848-855, Astronomical Interferometry, Robert D. Reasenberg; Ed.*, pages 848–855.
- Tuthill, P. G., Davis, J., Ireland, M., North, J., J., O., Robertson, J. G., and Tango, W. J. (2004). SUSI: recent technology and science. In *New Frontiers in Stellar Interferometry. Proceedings of SPIE Volume 5491. Edited by Wesley A. Traub.*, page 499.
- Underhill, A. B., Divan, L., Prevot-Burnichon, M.-L., and Doazan, V. (1979). Effective temperatures, angular diameters, distances and linear radii for 160 O and B stars. *MNRAS*, 189:601–605.
- van Belle, G. T., Ciardi, D. R., ten Brummelaar, T., McAlister, H. A., Ridgway, S. T., Berger, D. H., Goldfinger, P. J., Sturmann, J., Sturmann, L., Turner, N., Boden, A. F., Thompson, R. R., and Coyne, J. (2005). First Results from the CHARA Array. III. Oblateness, Rotational Velocity and Gravity Darkening of Alderamin. *ArXiv Astrophysics e-prints*.
- van Cittert, P. H. (1934). *Physica*, 1:201.
- Wesselink, A. J., Paranya, K., and de Vorkin, K. (1972). Catalogue of stellar dimensions. *A&AS*, 7:257.
- Wittkowski, M., Kervella, P., Arsenault, R., Paresce, F., Beckert, T., and Weigelt, G. (2004). VLTI/VINCI observations of the nucleus of NGC 1068 using the adaptive optics system MACAO. *A&A*, 418:L39–L42.
- Zernike, F. (1938). *Physica*, 5:785.