The Bolocam Software Pipeline: A User Manual

Benjamin D. Knowles

Center for Astrophysics and Space Astronomy

University of Colorado, Boulder, CO

Contents

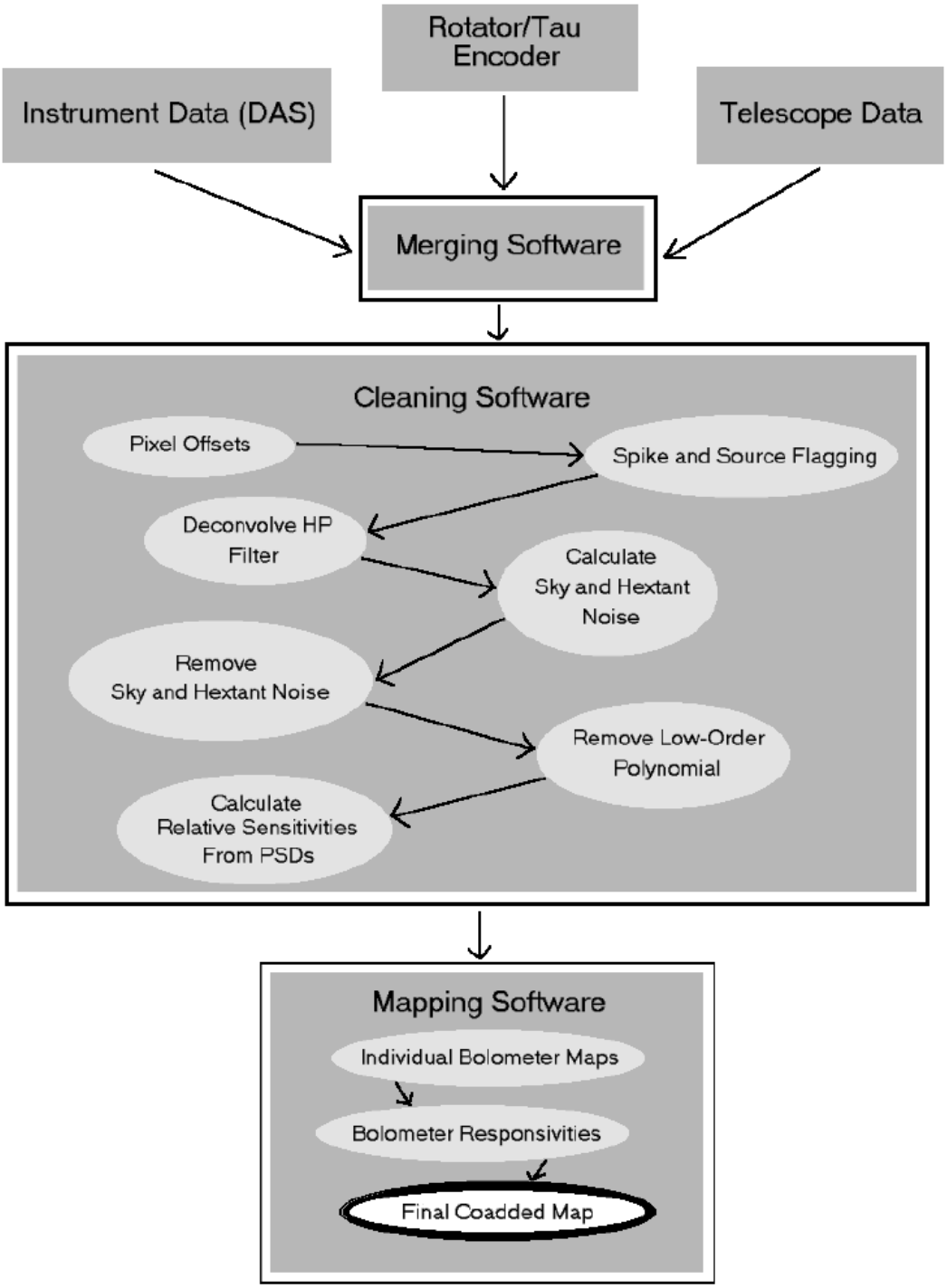

Fig. 1.— Bolocam software pipeline flowchart.

1. Introduction

As anyone who has worked on it will know, Bolocam and the data it produces are complicated. Perhaps not conceptually complicated (it's just time-sequenced data points, after all), but certainly functionally and structurally non-trivial. The data files are large (downright huge, in fact), the processing often computationally intensive, and the data format extensive, detailed, and specific. There are a great many pieces to the big picture, all important, and they all must be made to fit together if we are to produce usable science in the end. This is the challenge faced by the Bolocam software team, and this document is as much a chronical of their work as a description of the result of their efforts.

Here I will endeavor to describe clearly and in detail both how the Bolocam software pipeline works, and how it is used. The intent is that it be accessible both to those wishing only to employ the pipeline in its unaltered form, as well as to those who intend to expand and improve upon it in the future. The pipeline is *not* a finished product by any means, but every effort has been made to ensure as much functionality as possible to the "passive" user.

A basic to intermediate-level understanding of the Interactive Data Language (IDL) and C have been assumed throughout the discussion. In other words, this is not meant to be a "how-to" manual for new IDL users. Those in need of the basics should become familiar with the searchable IDL online help (type "?" at the `IDL>` prompt). Nor is this meant to be a manual for, or complete description of, the Bolocam hardware. While I will attempt to define and describe relevant terms wherever possible (notably in the "Glossary" at the end of this section), the actual use of the Bolocam hardware will generally not be discussed. Those with hardware questions are advised to consult the relevant sources listed in the bibliography.

The layout of this manual is as follows:

- Sections 2 and 3 ("Overview" and "Correlated Noise Removal") have been provided in order to present a comprehensive account *in words* of the entire software pipeline. In most cases they should be read first. For the graphically-minded, a simplified flowchart is given in Fig. 1 at the very beginning of the manual. A glossary of Bolocam-specific terminology may be found at the end of Section 2.

- Sections 4-8 comprise the main body of the manual. They contain detailed descriptions of all the software, including structure, inputs and outputs, and relevant suggestions and commentary. Section 4 contains essential information on the Bolocam data organization and the netCDF file format in general, and should therefore be read by everyone using the software. Sections 5-8, however, are intended less for the casual user and more for those who wish to alter and embellish the code.

- Section 9 contains a decidedly non-exhaustive description of how to perform statistical analysis on reduced Bolocam maps. Since as of yet there is no specific software written for this task,

this section will describe a bit of what may be accomplished from the IDL command line, as well as relevant background information for those who wish to write analysis software in the future.

- Sections 10 and 11 ("Quick Reference Guide" and "Troubleshooting Guide") should prove very useful to those casual users who just want to know how to use the code, and are less interested in altering it or learning its detailed workings. Section 10 consists primarily of specific examples of how to run the pipeline, covering all the common processing steps one may wish to perform, commonly used keyword options, and the use of IDL batch files. Sections 10.1 and 10.2 are particularly important, as they contain helpful information on directory structure and the necessary supporting files, including proper format and ordering of the cleaning module file. Finally, Section 11 addresses how to identify and correct common problems with the software, and is presented in a standard "Frequently Asked Questions" type of format.

- Appendix A contains information about the rotator, correcting for field rotation, coordinate system conventions, and other useful information about the Bolocam optics as they relate to the software (particularly the pixel offsets module). This section will only be valuable to those who intend to write software directly related to these issues, and is not intended for the casual user.

Thus far the Bolocam software pipeline comprises about 6000 lines of code. The merging software was written in C by Samantha Edgington at Caltech. With a few notable exceptions that will be pointed out, the cleaning and mapping software was written in IDL by myself, with significant portions based on various bits of "quick and dirty" code written by Phil Mauskopf at the University of Cardiff.

2. Overview

The processing of Bolocam data, from the initial raw voltages to the final calibrated maps, can be divided into three distinct stages. The first, called *merging* the data, involves combining the data recorded by the DAS with data recorded by the telescope computer, and writing the result into a NetCDF-format file. The second, called *cleaning*, is where nearly all of the signal processing occurs. This includes everything from the removal of sky noise to the calculation of RA and DEC bolometer offsets. Finally, *mapping* is where the data is transformed from a 144-channel timestream into a single 2-dimensional map.

In this and the section that follows, I will describe in words the basic structure and function of the three stages of the Bolocam software pipeline. The intent is not so much to explain "how," but rather "what" and "why," i.e. to provide the reader with the big picture of how merging, cleaning and mapping work together. An extensive discussion of the rationale behind correlated

noise removal is presented separately in Section 3. In addition, a glossary has been included in order to address the inevitable questions pertaining to instrument- and software-specific terminology.

2.1. Merging

During an observation with Bolocam, data is being written continuously to three different locations. The first is the header file, written by the telescope computer, which contains pointing, tracking, and other information documenting the telescope's behavior. The second is the data file, written by the DAS, which contains AC and DC outputs for each bolometer, bias monitor signals for each hextant of the array, and various TTL (binary on/off) signals. The third is a logfile containing whatever could not be easily recorded somewhere else: the rotator angle, for instance, and the zenith optical depth at 225 GHz recorded by the CSO "tipper." The header and data files are updated at a rate of 50 Hz, whereas the logfile is updated once per second.

The first step in the data reduction is simply to combine and organize all this information, and to write it out in a reasonable format. This is accomplished by a program called *merge_files*, written in C by Samantha Edgington at the California Institute of Technology. For the output format we recently switched from a proprietary system created by Barth Netterfield for the QMAP and BOOMERANG experiments to the more standard NetCDF database format. This decision was motivated primarily by NetCDF's straightforward and flexible layout, and its ready compatibility with IDL (Interactactive Data Language). In addition, NetCDF contains useful less-common features, such as the ability to define an infinite number of generic attributes for each variable[1].

Care has been taken to organize the data in the most intuitive way possible. For example, all AC bolometer signals are grouped together and written to one variable. Also, the logfile variables are up-sampled to match the 50 Hz outputs from the DAS and telescope. In future versions, synchronization between the DAS and telescope data may be handled in the merging process as well, but right now it is dealt with in cleaning (see below).

In retrospect, one of the most useful decisions has been to create a 1-byte channel with the same size and dimensionality as the $144 \times N_f$ bolometer signal variable, where N_f is the number of *frames*, or time samples. This is called the *flags* variable, and it serves a variety of purposes. Essentially, it allows us to mark specific frames in one of eight ways — corresponding to the eight available bits — if those frames need special consideration for whatever reason. This will be described with more specificity in Section A.2.

For the next observing run we will be using a modified version of *merge_files* which actually merges the data on the fly, as it is being written. The output will be split up into 1-hour long

[1]This was found to be a convenient tool for storing data format conversion factors, for example. The raw bolometer signals are originally stored as arbitrarily-scaled short-integers, but the inclusion of scale factor and offset attributes provides a painless way to automatically convert to floating-point voltages.

chunks, and individual scans will be differentiated by creating a new time-stream variable that will be incremented every time a new observation is initiated. We also plan to automatically record the RA and DEC of any bright sources in our field, along with their names. In this way it will be straightforward to retrieve, for example, all scans of Uranus for use in making a calibration map.

2.2. Cleaning

Cleaning encompasses all steps leading up to, and required for, the making of a map. Essentially this means that any operation that is performed in the time or frequency domains is performed in cleaning. Included in this is all removal of correlated noise, including sky noise and the so-called "hextant noise" defined in Section 5. Cleaning also includes more mundane, but equally necessary, calculations like those associated with the observation geometry.

The current form of the cleaning pipeline is a modular system based around a central "wrapper" program. Schematically, the wrapper (called *clean_ncdf*) works as follows:

1. If the data has never been cleaned before, a variety of things must first take place. One is to determine the number of subscans in the observation, and then create variables in the NetCDF file that will later hold $N_{subscans}$-sized variables. At the same time it calculates a synchronization offset between the DAS and telescope data for each subscan. And thirdly, it reads two external files, referred to as the array and bolometer parameters files, into the data. The bolometer parameters consist of three N_{bolos}-element vectors: an integer signifying the bolometer status (ie. good/bad/open) during that observation, and two numbers giving the polar coordinates of the bolometer, relative to some fiducial point on the array. Array parameters are numbers which are generally universal for a particular observing run, and are not bolometer-specific. They include the beam FWHM, fiducial array rotation angle, platescale, and the x and y offsets, if any, of the telescope boresight from the center of the array.

2. Once this has been done, the wrapper checks for a text file called the *module file*. This file contains the names of the cleaning routines (or "modules") to be run.

3. Data from one or more NetCDF files is read into a large IDL structure called *chan*. To save time and memory, only those fields that are necessary for the modules selected are read. In addition, the wrapper only reads in and processes a small number of subscans at a time, repeating the process several times until the entire observation has been cleaned.

4. Once a section of data has been read in, it is sent to each module listed in the module file, one after another.

5. The cleaned data is written out again into NetCDF format.

Once these preliminaries are finished, the bulk of the processing is done by the individual cleaning modules. These are most easily described by grouping together those that work in a similar fashion, but note that the order in which they are presented here is not necessarily the order in which they are run.

The *pixel_offsets* module calculates the RA and DEC offsets on the sky of each bolometer to the center bolometer. It is what allows the 144 bolometer signals to be combined into one map. To do this it actually simulates the way the array would look on the sky, starting with the physical posititions of the bolometers on the array, and propagating these — in the time-reversed sense — through the optics. A second-order field distortion correction is applied as well. Once the array-image has been "projected" onto the sky, the position angle recorded by the telescope computer is used to convert Altitude-Azimuth coordinates to RA and DEC.

The *desource*, *debadscan* and *despike* (the latter written by Samantha Edgington) modules are all flagging modules. That is, they are designed to look for sources, bad scans (those with particularly errant RMS values), and unwanted spikes, respectively, and switch an appropriate bit in the *flags* variable for the frames in which they are found. This flagging strategy is useful because it allows one to ignore some parts of the data sometimes, and others at other times, without having to alter the data itself. The best example of why this is important arises when calculating the sky noise for a scan that includes bright sources. In such a situation it is important *not* to include flux from these sources. By flagging them beforehand (using their known positions, if available), those regions of the subscan are excluded from the sky average. Note that it is not necessary to flag *all* sources, i.e. the faint sources which we are trying to detect. These contribute negligibly to the flux observed in a single bolometer, so including them in the sky average should not have an effect.

The *deconv* module is an altered form of a code written by Alexey Goldin at the Jet Propulsion Laboratory for optimally-filtering (or "Wiener-filtering") the timestream. The assumption of source size and shape was removed, so that its sole intention now is to deconvolve the high-pass filter transfer function in our electronics from the bolometer signals. This is necessary to remove an artifact referred to as "shadowing" (because it appears as voltage dip after a beam passes through a bright source), which is caused by the tendency of the high-pass filter to maintain a time-averaged voltage level of zero. As it stands, the *deconv* module succeeds in restoring symmetry to the wings of bright sources, but does not fully remove the shadowing. This will be investigated and improved upon in future versions.

Somewhat related to the *deconv* module is the *polysub* module. This is one of the simplest pieces of the software pipeline, since its goal is simply to fit and subtract out a low-order polynomial from each subscan after all other noise removal steps have occurred. We do this to remove the very lowest-frequency components of the remaining $1/f$ noise. Keeping the fit to a 4th-order polynomial or less (usually 3rd) assures that the flux of any beam-sized point sources is unaffected. This and the deconvolution step might eventually be eliminated altogether in favor of an all-in-one optimal filter at the end of the cleaning pipeline.

The *relsens_psd* module is run at the end of the cleaning pipeline. Its purpose is to calculate the relative sensitivity of each subscan, to be used as a weight when averaging together data from several bolometers during the mapping process. There are a variety of methods one might use to weight a subscan of data, a simple standard deviation being perhaps the most basic example. However, recognizing that we are ultimately limited by $1/f$ noise, we have chosen instead to weight a subscan by its power spectral density (PSD) at low frequencies. To do this, spikes in the data must first be replaced with straight lines[2], and a gaussian profile subtracted from known sources, before a PSD can be calculated. The PSD is then integrated between 100 and 200 mHz, the result of which is treated like an arbitrarily-scaled noise estimate, so that scans with more low-frequency power are weighted less. An important assumption must be made in order to justify this as an appropriate noise estimate, which is that any low-frequency noise due to contributions from astrophysical sources will be negligible in relation to the remaining $1/f$ noise.

(The one remaining — and most important — part of the cleaning process, the *skysub* module, is discussed at length in the "Correlated Noise Removal" section which follows.)

2.3. Mapping

Once the data has been fully processed in the time domain, it is ready to be used to generate a map. Conceptually the process of mapping is straightforward; the trick is doing all of this in a way that is both fast and easy on memory allocation. The essential idea is to use the pixel offsets calculated during cleaning to make separate RA and DEC arrays for each bolometer. Then it is only a matter of creating a grid with the desired resolution (usually 5 or $10''$ in our case) and binning the data appropriately, using the relative sensitivities already in the data stream to weight each subscan. The mapping code takes into account the effect of atmospheric optical depth on the measured flux as well. An example of a Bolcam map is shown in Fig. 2.

The basic mapping code is *map_ncdf*. It performs the steps described above, and returns a data structure containing a number of different variables. These include both 1-D and 2-D data arrays, the number of data points that went into each pixel, a 2-D map of the standard deviations of these points for each pixel, as well as RA, DEC, and resolution information.

In addition to making coadded maps using the entire array, *map_ncdf* can be instructed to make individual maps for each good bolometer. Doing so with an observation of a bright source enables the calculation of the bolometer responsivities, a process made straightforward by a separate program called *calc_responsivities*. This code reads in the structure of bolometer maps returned by *map_ncdf*, and fits a 2-dimensional gaussian to each. The peak of the gaussian, which is equivalent to the total flux of the source[3], is recorded in an IDL save file. This file is then used by the

[2]A constrained realization of high frequency noise would be better for this, but has yet to be done.

[3]This is due to the convolution with our beam imposed by the act of observation itself.

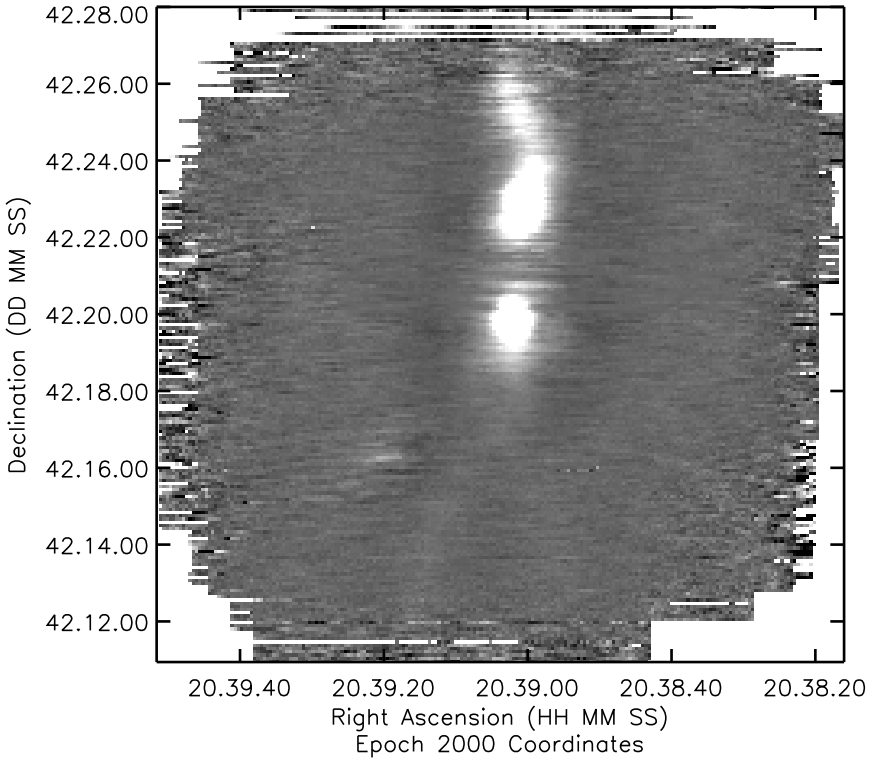

Fig. 2.— Bolocam map of the galactic star-forming region DR21 at 1.4 mm with 5″ pixels.

mapping code in all subsequent maps to scale the bolometer signals relative to the peak flux of the calibrator[4].

In cases when one searching for point sources (i.e. beam-sized sources), generally there is a final step in the mapping process, namely 2-dimensional convolution with the beam. This is very nearly the same as optimally-filtering the map, because it emphasizes any features with sizes on the order of a beamsize. As of yet there is no specific software for performing this convolution (it can be carried out quite simply from the IDL command line if need be), and it may become irrelevent once an optimal filtering module is written for the cleaning software.

[4]The primary calibrator for the May 2000 engineering run is Uranus, whose flux at millimeter wavelengths is given by Griffin & Orton (1993). This value, when observed by Bolocam at 1.4 mm passband and convolved with its beam, is ~ 27.3 Jy.

2.4. Glossary

$1/f$ Noise — pronounced "one-over-f noise," this term is generally used to refer to any noise contribution to the data that has an inverse-frequency dependence. The main source of $1/f$ noise in Bolocam is the atmosphere (see Sky Noise), but there are also significant contributions from various components in the electronics chain (see Hextant Noise).

Array — 1. The assembly, including feedhorn plate, 144 bolometer detectors, and backshort, which comprises the heart of the Bolocam instrument. The array is cryogenically cooled to 300 mK within the dewar (see Dewar). 2. Often used somewhat colloquially to refer to the beam pattern as projected on the sky. A point source may be said to drift "across the array." Note that this also defines our field of view, which is almost 8 arcminutes across at the widest. 3. The most basic and commonly-used type of data structure used in IDL.

Array Parameters File — A supporting text file containing information about the array (see entry) that in general remains constant during an observing run. It contains five floating-point numbers: platescale [arcsec/mm], beam FWHM [arcsec], fiducial array angle [degrees], and x and y offsets of the array center from the telescope boresight [inches]. See also Rotator.

Bad Data/Scan Flags — see Data Flags

Barthware — Now defunct, the data format used by Bolocam through early 2001. It was replaced by the NetCDF database format (see NetCDF).

Beam — Formally, the Fourier transform of the primary mirror illumination pattern projected onto the sky. For a single bolometer, the shape of the beam is controlled mainly by the feedhorn. One often uses a field stop (in our case a Lyot stop) to apodize (or truncate) the beam sidelobes. Typically one desires a gaussian illumination of the primary in order to produce a gaussian beam. A tophat illumination function, in contrast, produces an Airy pattern. Bolocam beams are nearly gaussian, but apodization produces low-level Airy ringing as well. The beam size is directly proportional to the wavelength, and inversely proportional to the size of the illumination pattern according to λ/D. In the software pipeline, the full-width at half-maximum (FWHM) is used as a measure of beamsize; e.g. $\theta_{FWHM} \approx 39$ arcseconds at 1.4 mm.

Bias — Each bolometer on the Bolocam array is "biased" at a specific voltage, which is picked to maximize sensitivity and ensure linearity of response to optical loading. In our case, we AC-bias each bolometer at a frequency of ~ 280 Hz, and later phase lock to return the output to near-DC. Doing so effectively modulates our signal so that we do not have to physically chop on the sky, as is often the case in sub/millimeter observing. See also Open Channel.

Bolocam — The name of our instrument; not an acronym. Bolocam consists of 144 AC-biased bolometer detectors on a single wafer of silicon. The detectors are feedhorn-coupled to the incoming radiation field, and housed within a cryogenic dewar which cools them to 300 mK.

Bolometer — By definition, a device that detects all wavelengths of light. In practice, filters need to be used to restrict the incoming radiation to the desired bandpass. Bolocam's bolometers consist of a silicon micromesh absorber onto which is bonded a thermistor, a variable resistor sensitive to temperature. Thus, when the temperature of the bolometer changes due to optical loading, so also does the output voltage.

Bolometer Flags — see Channel Flags

Bolometer Parameters File — A supporting text file which contains information about the 144 bolometers on the array. This file consists of 4 columns: bolometer name (a character string, e.g. `BOLO_E0`), channel flags (see entry), angle (in degrees) relative to a fiducial point on the array, and radial distance on the array (in units of bolometer separations). See also Rotator.

Channel Flags — An integer contained in the bolometer parameters file (see entry) which describes the status of each bolometer channel. While the notation may change, as of now it is as follows: 0 refers to dead or "railed" channels that contain essentially no useful information. Negative integers refer to "open" channels (see entry), and positive integers refer to "good" channels. In both cases, values with increasing absolute values indicate increasingly worse data quality. In general, only bolometers with channel flags of "1" are considered truly "good" in the sense that they are included in the final data product.

Cleaning — The part of the software pipeline that performs all time-domain processing, notably the removal of correlated noise (see entry). The cleaning software is modular in format, consisting of a wrapper program that handles input, output, and formatting of the data, and a number of cleaning modules which perform the specific processing tasks. Each module has the same calling sequence, accepting from the wrapper program a data structure called *chan* which contains the data to be processed. The modules to be run by the cleaning software are specified by the user in the module file (see entry).

Cleaning Module — see Cleaning

Coadding - see Mapping

Common Mode — This refers to any signal or noise component which is seen identically in every bolometer channel. Sky noise (see entry), for example, can be treated as common mode because Bolocam's beams overlap almost completely on the sky at the altitude where most of the atmospheric emission originates. There is also most certainly a common mode noise component from the hardware and electronics, although separating this from the sky noise is difficult. Note that the astronomical signal to be measured is specifically *not* common mode, as Bolocam's beams are completely separated in the far field, and our field of view is quite large (see Array).

Convolution — Mathematically, the convolution of two functions is the inverse Fourier transform of the product of their Fourier transforms. Some practical examples: 1. The observed astronomical signal recorded by a bolometer is actually the 2-D convolution of the actual far-field

pattern with the bolometer beam. Thus, a point source is seen as a gaussian (approximately) with the same FWHM as the beam. 2. An electronics filter, such as the high-pass filter in the Bolocam electronics chain, produces an output signal that is the input signal convolved with some filter function. Occasionally, such as when the beam passes over a bright source, this produces undesirable artifacts in the data. In these cases we want to *deconvolve* the filter function from the data stream.

Correlated Noise — Noise components that are correlated in *time* between more than one channel. Sky noise (see entry), for instance, is correlated across the entire array, whereas hextant noise (see entry) is correlated only across a part of the array.

Correlation Coefficient — A measure of how well-correlated two functions are. In our case, the functions are generally time-sequenced arrays. The normalized correlation coefficient is defined as:

$$Corr_{f,g} = \frac{\sum_i f_i g_i}{\sqrt{\sum_i f_i^2}\sqrt{\sum_i g_i^2}} \quad (1)$$

Data Acquisition System (DAS) — The DAS is the computer and interface which reads in data from each bolometer channel (via a 16-bit analogue-to-digital converter), and stores it in a useable format.

Data Flags — Each channel of bolometer data has a corresponding byte-format (8-bit) flag channel containing the same number of frames. Each of the 8 bits in a given data flag frame may be set (i.e. from 0 to 1) individually to indicate something of note about that particular data. For instance, frames in which a bright source happens to be passing through the beam will have their "source flag" set to 1 by the desource module. Similarly, the despike module sets the "spike flag" to 1 for frames with cosmic rays or other spikes, and the debadscan module sets the "badscan flag" to 1 for entire scans that have erroneous rms values. The last type of data flag is the "baddata flag" which is set to 1 within the sky and hextant noise removal code when, for a specific frame, there are not enough bolometers for an accurate average to be calculated. See Section 7 on the cleaning modules for more details.

Data Stream — This is somewhat arbitrarily taken to mean the entire time-sequenced Bolocam dataset, including all channels recorded by both the DAS and the telescope (see respective entries). Note that "data stream" and "time stream" (see entry) are often used interchangeably.

Deconvolution — see Convolution

Dewar — The container in which the Bolocam detector array is contained. The dewar is cryogenically cooled, and vacuum-pumped to minimize thermal conductivity by air molecules. Cooling occurs in three stages, which are thermally isolated from each other with fiberglass or Vespel

standoffs: liquid nitrogen (77 K), liquid helium (4 K), and a liquid helium refrigerator, which maintains the detectors at 300 mK.

Driftscan — see Scan, Subscan

Flags — see Channel Flags, Data Flags

Hextant — One of six symmetrical sections into which the array and electronics chain are divided. There are 6 hextants with 24 bolometers, and therefore 24 bolometer channels, each. See also Hextant Noise.

Hextant Noise — The component of electronics noise correlated in time from bolometer to bolometer across a hextant. While small in comparison to the sky noise, hextant noise is still a significant contribution to the raw data. Hextant noise is caused by the physical organization of the electronics chain. For example, the lock-in amplifiers and pre-amps for a given hextant are located on the same board, but separated physically from the other boards. Thus, differences in the thermal changes from one board to another will result in a noise template that is correlated within a hextant, but uncorrelated from hextant to hextant.

Load Curve — A plot of voltage vs. current for a bolometer detector. Used to determine various detector characteristics, as well as to set certain variable parameters, such as bias voltage. The load curve "turns over" at high currents due to ohmic heating of the detector, which reduces its resistance and therfore the measured voltage.

Mapping — The part of the software pipeline that converts the 144 time-sequenced bolometer channels into one 2-D map.

Merging — The part of the software pipeline that combines data taken by the Data Acquisition System (see entry) with that recorded by the telescope and rotator software during an observation. Merging is the first step in the data reduction.

Module File — The file read by the cleaning wrapper (see entry) which contains the names of the cleaning modules to be executed, and the order in which they should be run.

NetCDF (network Common Data Form) — The database format in which Bolocam data is stored after merging. Newer releases of IDL contain native functions for reading from and writing to netCDF files.

Open Channel — A bolometer channel in which there is negligible contribution of the bolometer to the total resistance of the cold electronics. The presence of the load resistors

Optics Box — The box containing the tertiary, quaternary (both flat), and ellipsoidal mirrors, which couple the telescope optics to Bolocam's dewar optics. One end of the optics box mounts at the Cassegrain focus of the Caltech Submillimeter Observatory telescope, while the other end mounts to the rotator (see entry) and Bolocam dewar (see entry).

PA — The position angle (usually, but erroneously, referred to as the parallactic angle) is best understood as the angle difference between the Alt-Az axes and RA/DEC axes at a given location on the sky. Knowing the PA, which is recorded by the telescope along with RA, DEC, local sidereal time, etc, thus allows relatively easy conversion between coordinate systems. Put differently, the PA is the angle defined by the zenith, polaris, and object, in that order. PA = 0 when looking due south in the northern hemisphere, and is increasingly positive toward the west/increasingly negative toward the east.

Parallactic/Position Angle — see PA

Power Spectral Density (PSD) — The power spectral density of a time-sequenced function $g(t)$ is given by

$$PSD = \frac{2}{\Delta f}|G(f)|^2, \tag{2}$$

where $G(f)$ is the fourier transform of $g(t)$, and Δf is the smallest possible frequency allowed by the Δt-sampled function, i.e. $\Delta f = (\Delta t \times n_{samples})^{-1}$. Note that the "PSD" normally quoted is actually the square-root of the PSD, e.g. in units of Volts Hertz$^{-1/2}$.

Relative Sensitivities — In this context, relative sensitivities are simply the relative weights given to each bolometer during coadding. Relative sensitivities are currently calculated by integrating the PSD of each bolometer and each subscan at low frequencies (generally between about 100 and 200 mHz). Thus, subscans with relatively greater power at low frequencies are weighted less.

Responsivity — The responsivity of a bolometer is its sensitivity to optical loading, i.e. the gain. Responsivity is determined by measuring the peak flux of a calibration source (usually a planet, such as Uranus) in each bolometer. In the mapping step, bolometers are scaled and inversely-weighted by their relative responsivities. Absolute calibration is then performed as a separate scaling on the final, coadded map.

Rotator — The rotator is essentially a large stepper motor which sits between the dewar and optics box (see entries). Because of the Alt-Az mount of the telescope (see entry), it is required in order to maintain a constant scan angle between the array and RA direction as projected on the sky.

Scan — The generic term for a complete, rastered drift-scan observation with Bolocam. "Drift-scanning" refers to our observing strategy, which involves keep the telescope stationary while the Earth's rotation causes the sky to drift past our beams in right ascension. Repeating this step at several different declinations results in a "scan." See also Subscan.

Sensitivity — see Relative Sensitivities

Skydip — A test performed at the telescope (see entry) which measures the sky brightness over a range of telescope elevations in order to simultaneously determine the atmospheric opacity

and bolometer responsivities. It can also be used to measure optical efficiencies and as a general diagnostic tool. As of this writing, skydip measurements are only applied directly to the data at one point in the software pipeline: the conversion, during the mapping process, of the optical depth measured at 225 GHz by the CSO "tipper" (see entry) to that for the appropriate Bolocam bandpass.

Sky Noise — Noise produced by fluctuations in the brightness temperature (column density) of the atmosphere. Sky noise is by far the dominate contribution to the raw data, having an amplitude 3 to 4 orders of magnitude greater than the faint galaxy sources Bolocam is trying to detect. However, the time-dependence of the sky noise is also strongly correlated across the array. That is, because our beams overlap on the sky almost completely at the altitude at which the sky fluctuations are greatest, the sky noise is seen essentially identically in each bolometer channel. This allows us to approximate sky noise by simply averaging together many bolometer signals. See section 3 on correlated noise removal.

Source Flags — see Data Flags

Spikes/Spike Flags - see Data Flags

Subscan — A drift-scan (see Scan) across a field at one declination only. A "scan" is comprised of many "subscans."

Telescope — As of this writing, the only version of Bolocam yet constructed - Bolocam I - has been specifically designed to be mounted to the Caltech Submillimeter Observatory 10-m telescope on Mauna Kea, Hawaii. It is a Cassegrain telescope with an Alt-Az mount. See also Dewar, Optics Box, and Rotator.

Time Stream/Trace — The 1-D time-sequenced array of data for a given channel in the datastream. Drift-scanning across a point source produces a gaussian in the "trace" or "time stream" of each bolometer channel whose beam has crossed the source. See also Data Stream, Scan.

Tipper — A radiometer at the Caltech Submillimeter Observatory that measures the zenith atmospheric opacity of molecular water emission at a frequency of 225 GHz. This value is recorded by the same encoder software that records the rotator position, and is referred to as the "tipper tau." The actual measurement is similar to a skydip (see entry), in that data must be taken at several elevations. The instrument must therefore tip, which is why it is called a "tipper."

Tracking — The tracking signal is a TTL signal (binary format, either on or off) recorded by the telescope computer as well as the Bolocam data acquisition system (see entry). The tracking TTL is "high" (or "on" or 1) when the telescope motor is on, and "low" (or "off" or 0) when the motor is off, i.e. when we are drift-scanning. In other words, a tracking signal of 0 corresponds to when we are acquiring data.

TTL — Short for "transistor-transistor logic," a common type of digital circuit in which the output is derived from two transistors. Some of the signals recorded by the DAS (see entry) are binary TTL signals. A "high" voltage indicates a binary 1, and a "low" voltage indicates a binary 0. The first semiconductor devices incorporating TTL were developed by Texas Instruments in the mid-1960's.

Wiener Filter — Also called an "optimal filter," a Wiener filter is used to reclaim a signal that has been "corrupted" in some known way. Provided that we are able to somehow independently estimate the power spectrum of the desired signal and that of the noise, we can construct a Wiener filter that will reclaim the desired signal. This technique can also be used to deconvolve a known filter function from a signal that has been "corrupted" by that filter. This is the technique employed for the deconvolution (see entry) of the electronics high-pass filter. See Numerical Recipes, as well as section 7.5, for details on how to construct a Wiener filter.

3. Correlated Noise Removal

The term *correlated* refers here to a time-dependent signal whose signature is seen in more than one bolometer channel. Such correlation is extemely beneficial for the software removal of $1/f$ noise contributions because, as will be described below, it allows the approximation of these contributions by simply taking averages over many bolometer channels. However, note that correlation alone does not ensure that the removal of a correlated signal will be straightforward. Here we are aided greatly by the fact that the $1/f$ noise components are correlated in *time*, whereas the astrophysical signal being measured is correlated in *space*. Put differently, the time-functionality of the astronomical signal produced by our scan strategy is, on average, completely *uncorrelated* to the time-functionality of the sky noise, which is seen nearly identically in each bolometer channel. It is this fact that is actually being exploited by the noise removal portion of the cleaning software, which will now be discussed in detail.

3.1. Formalism

For the purposes of noise subtraction, it is useful to think of one channel of one subscan of Bolocam data as a superposition of several distinct functions in time. Essentially these are just "signal" and "noise," but with the latter consisting of superposed white and $1/f$ components. The $1/f$ component, which in general dominates the raw signal amplitude, can further be broken down into the sum of three distinct noise functions: uncorrelated, array-correlated (or "common mode") and hextant-correlated. The latter refers to the fact that our array is hexagonal and divided into 6 hextants, each with its own set of components in the electronics chain; thus *hextant noise* is electronics noise which is correlated over a hextant only. There's not much to be done about the

white noise and uncorrelated $1/f$ noise except to filter out the frequencies in which we are not interested[5], and to average over as many subscans as possible. Array and hextant-correlated noise, however, can be dealt with directly. For bolometer i, the time-dependent response for a single subscan is given by

$$D_i(t) = R_i(t)S_i(t) + N_i(t) \ , \tag{3}$$

where D_i is the raw data, R_i is the responsivity of the bolometer to optical loading, S_i is the astronomical signal to be measured, and N_i is the superposition of noise components described above. Specifically,

$$N_i(t) = \alpha_i N_{hex}(t) + R_i(t)N_{sky}(t) + N_{uncorr}(t) + N_{white}(t) \ , \tag{4}$$

where α_i is a bolometer-specific scaling factor, N_{sky}, N_{hex}, and N_{uncorr} are the differently-time-correlated $1/f$ noise components, and N_{white} is the white noise (subscripts omitted for clarity). Note that the bolometer responsivity R_i is a weak function of time, and thus is treated similarly to α_i, as a scaling factor to be derived for each subscan. Correlated noise removal is thus accomplished by calculating arbitrarily-scaled time traces for N_{sky} and N_{hex} in the form of time-ordered 1-D arrays, and subtracting these templates from the raw data D_i by using a least-squares multiple linear regression fit to determine the scaling factors R_i and α_i. The calculation is performed separately for each subscan, with N_{hex} only needing to be calculated six times and N_{sky}, if treated as common mode, only once.

Two techniques have so far been developed to calculate $N_{sky}(t)$ and $N_{hex}(t)$. The first, which is the simpler and slightly more effective of the two, is to treat the sky and hextant noise simultaneously by simply averaging over all bolometers in a hextant:

$$\alpha_i N_{hex}(t) + R_i(t)N_{sky}(t) \approx \frac{\beta_i}{n_{hex}} \sum_{i=1}^{n_{hex}} D_i(t) \ . \tag{5}$$

Here, n_{hex} is the number of working bolometers in the hextant in which bolometer i resides, β is another scaling factor which acts as a surrogate for α and R, and S_i is assumed to be negligible compared to N_i in a given channel i. The average is also weighted in such a way as to ignore contributions from bright sources and spikes flagged by the flagging modules (see "Overview" section). By design, this method cannot be the best possible method for removing sky and hextant noise, because it ignores the fact that the N_{sky} and N_{hex} templates scale differently for a given bolometer; that is, that they have different gain characteristics. However, because we are taking an average, the resulting sum of hextant and sky components has them scaled by their average relative scalings to each other. Thus, assuming that the individual bolometer relative scalings do not deviate significantly from this template average, the approximation is valid. This method is the best method to use as of this writing.

---

[5]This is basically what the *polysub* module does at the low-frequency end; see Section 7.4

The second technique for calculating the sky and hextant noise components treats them each separately. The sky noise template is derived analogously to the method described above, i.e. by replacing n_{hex} with n_{arr} on the right side of equation (5), so that the average is now over all bolometers on the array:

$$N_{sky}(t) \approx \frac{1}{n_{arr}} \sum_{i=1}^{n_{arr}} D_i(t) \ . \tag{6}$$

The removal of the hextant noise N_{hex} (which, ideally, should be performed before N_{sky} is calculated) is more complicated. Initially the open[6] bolometers were used as a measure of the hextant-correlated noise. Poor correlation of this estimate to the other hextant bolometers eventually revealed that the bias does not contribute significantly to the electronics noise, so a different technique had to be developed. The new method works on the simplifying assumption that

$$N_{hex}(t) \approx \frac{1}{n_{hex}} \left[\sum_{i=1}^{n_{hex}} D_i(t) \right] - \gamma_i N_{sky}(t) \ , \tag{7}$$

where γ is a coefficient that scales the sky noise N_{sky} to the level of the summation term. Such a scaling is valid under the assumption that the hextant and sky noise templates are completely uncorrelated to one another. The hextant noise is thus approximated by taking the difference between the average of all bolometers, and the average of the bolometers in the hextant. The latter will include both hextant *and* common mode noise as shown in equation (5), whereas in the former one hopes to have averaged over the hextant noise, leaving only common mode. Again note that S_i, which is on average completely uncorrelated to N_{hex} and N_{sky}, contributes negligibly to D_i, and so is not preserved in either of the averages.

The technique described by equations (6) and (7) should be considered only a first-order approximation since we are again ignoring the different gains for N_{sky} and N_{hex}, as well as contamination of the sky template by hextant noise (which, albeit small in comparison to the level of sky noise variation, still fundamentally limits the accuracy of the approximation). The formalism is presented here so that it may be used as the basis for future improvements. A direct comparison between the first (hextant-average) and second (array-average/first-order hextant removal) methods will be made in the following subsection.

3.2. Analysis

Plotted in Fig. 3 is the time stream for a typical bolometer before and after cleaning. The subscan shown was taken from an observation of Uranus during somewhat poorer-than-usual sky conditions, so that sky noise is by far the dominant noise contribution. The effect of the cleaning

[6] "Open" refers to non-working channels where there is negligible bolometer contribution to the resistance of the cold electronics. The presence of the load resistors, however, makes them a measure of the bias signal.

algorithm is obvious (although note that Uranus is a factor of $\sim 10^4$ brighter than our target sensitivity). In this subsection the actual improvement to the data, as well as the justification for the methodology, will be quantified in a robust manner using data from the May 2000 engineering run.

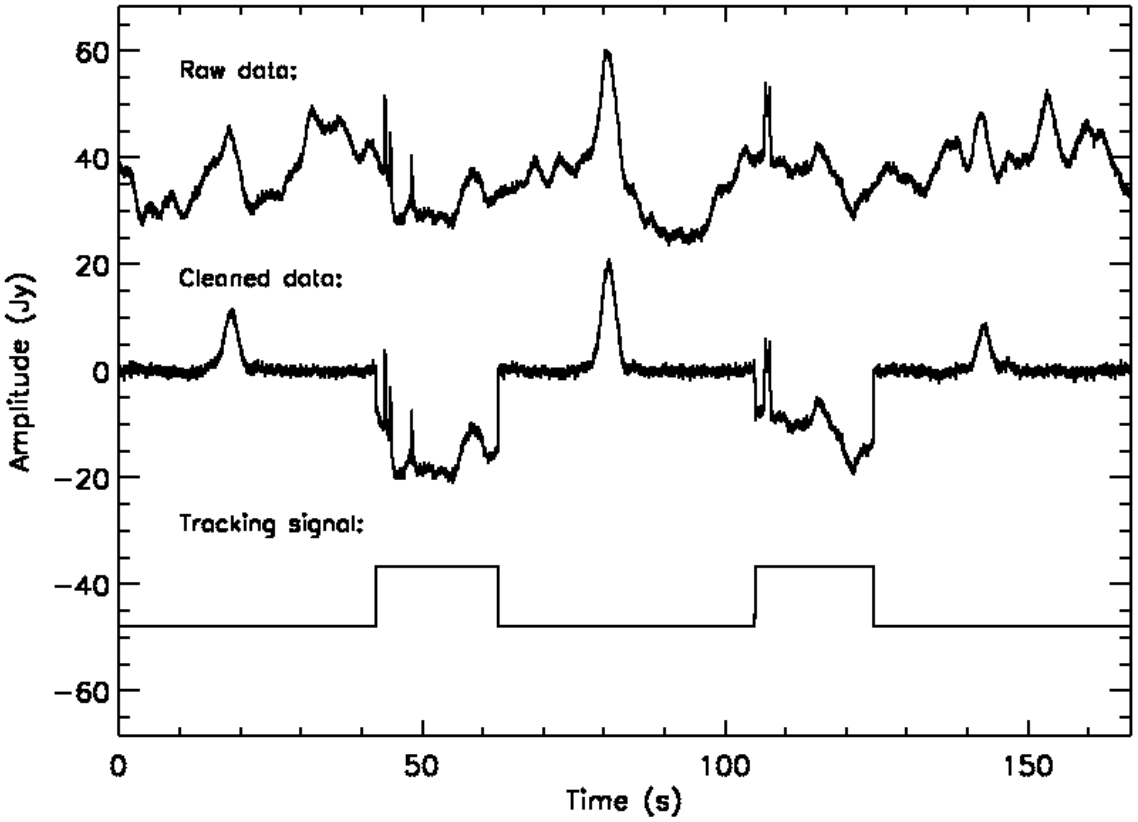

Fig. 3.— Three subscans of the Uranus scan from May 21, 2000, before and after cleaning with the hextant-only method. Arbitrary offsets have been applied for clarity. The data shown is from a slightly off-axis bolometer, so the peak of Uranus seen here is not as high as its actual peak of 27.3 Jy. The tracking signal is "low" when the telescope is drift-scanning. Raw and cleaned data were both taken from bolometer B3.

The basic premise behind the treatment of sky noise in the cleaning software is this: if our beams overlap in the region of the troposphere where most of the millimeter-wave atmospheric flux originates ($h_{av} \sim 1.3$ km, according to Lay & Halverson), then the sky noise will be correlated from channel-to-channel. If the overlap is good enough, then the correlations will be such that a simple average over the array — the common mode — will be an adequate measure of the sky noise. Note that the common mode will also contain any electronics noise that happens to be correlated over the array.

A footprint analysis of the beams on the primary mirror indicates that the beams overlap very

well in the lower troposphere. In Fig. 4 is plotted the results of a test to determine how exactly this overlap corresponds to the level of correlation of the sky noise across the array. Here the average normalized correlation coefficients of each bolometer on the array to a chosen reference bolometer have been calculated as a function of bolometer separation. The only processing that occurred was the flagging of sources and spikes, and the removal of a DC offset from each subscan.

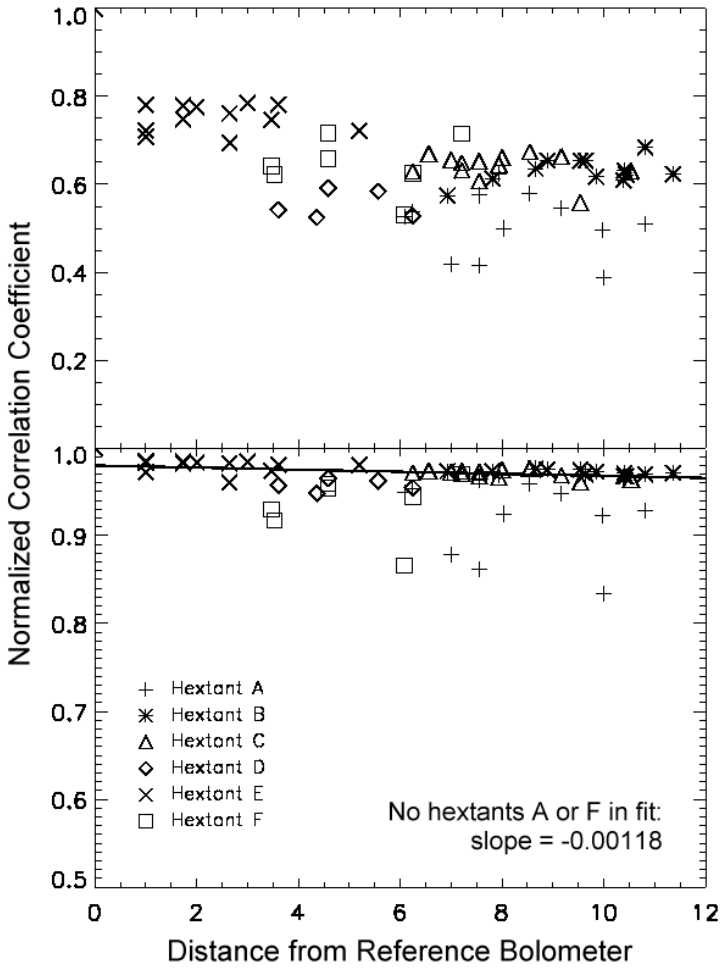

Fig. 4.— Average normalized correlation coefficients for raw data taken during observations of 3C273 (top) and Uranus.

In the plot for the Uranus observation the correlations are greater than 95% for most bolometers, suggesting that array-correlated noise is almost completely dominating the signal. Some hextants, specifically A and F, have systematically lower and more varied correlation values. This implies excesses of both hextant-correlated and uncorrelated noise in these bolometers, or alternatively, lower response to optical loading (i.e. lower responsivity). Applying a linear fit to the

correlation values, and excluding the A and F hextants, we find that correlations decrease with spatial separation between bolometers. This change is very slight, however: from 98% to 97% correlation across the *entire extent of the array*[7]. This finding indicates that nearly all sky noise can be removed by treating it as common mode.

The plot for an observation of quasar 3C273 taken on the same night exhibits significantly lower and more variable correlation values. In addition, hextant-to-hextant correlation is much more apparent. Assuming no appreciable change in the spatial scale of atmospheric fluctuations or in the properties of the detectors themselves, this suggests that the sky noise during this observation was less than that for Uranus. That is, a smaller sky noise amplitude has made the uncorrelated and hextant noise components more noticeable. This is further supported by the fact that even bolometers in the same hextant as the reference bolometer do not have correlation values as high as in the Uranus observation.

The conclusion to be drawn from this analysis has already been implied: it is necessary to remove hextant-correlated noise in addition to the array-correlated common mode noise. In truth, this hextant noise is a direct consequence of the organization of certain components in the Bolocam electronics chain. One example is the placing of all 24 preamps for a given hextant on the same board. Because they are in physical proximity with each other but separated from the other hextants, and because even slight temperature fluctuations can cause $1/f$ drifts, the resulting electronics noise is seen as correlated within a board, and uncorrelated from board-to-board. The presence of this extra correlated noise component explains why the hextant-only sky avergage method described at the end of Section 5 works as well as it does.

The power spectral density (PSD) at low frequencies provides a figure of merit for the removal of $1/f$ noise. The PSDs of a few different bolometers are plotted in Fig. 5. Each PSD has been averaged over many subscans after cleaning with the two best cleaning methods. It is evident that, in most cases, the hextant-only method is slightly better at removing low-frequency noise.

To get a better indication of how well the correlated noise removal is working overall, one should compare PSDs of the cleaned data to that of electronics noise alone. Fig. 6 shows overlaid PSD plots of raw data, data cleaned with the hextant-only method, and electronics noise recorded in the lab. The latter was taken without any cold electronics (i.e. bolometers or JFETs), so it just includes contributions from the bias, preamps and lock-in amplifiers. Its white noise level has been scaled to match that of the other two. Note that because we don't have information on which channel the warm electronics noise was was taken from, we cannot do a one-to-one comparison with the cleaned data; the electronics noise is therefore meant to be indicative only.

From Fig. 6 it seems that the current cleaning techniques are able to remove $1/f$ noise to a level approaching the electronics level alone. In effect, this means that the engineering run data are

[7]Bolometer E19 was used for these plots because its proximity to the edge of the wafer gives as large a range in separations as possible

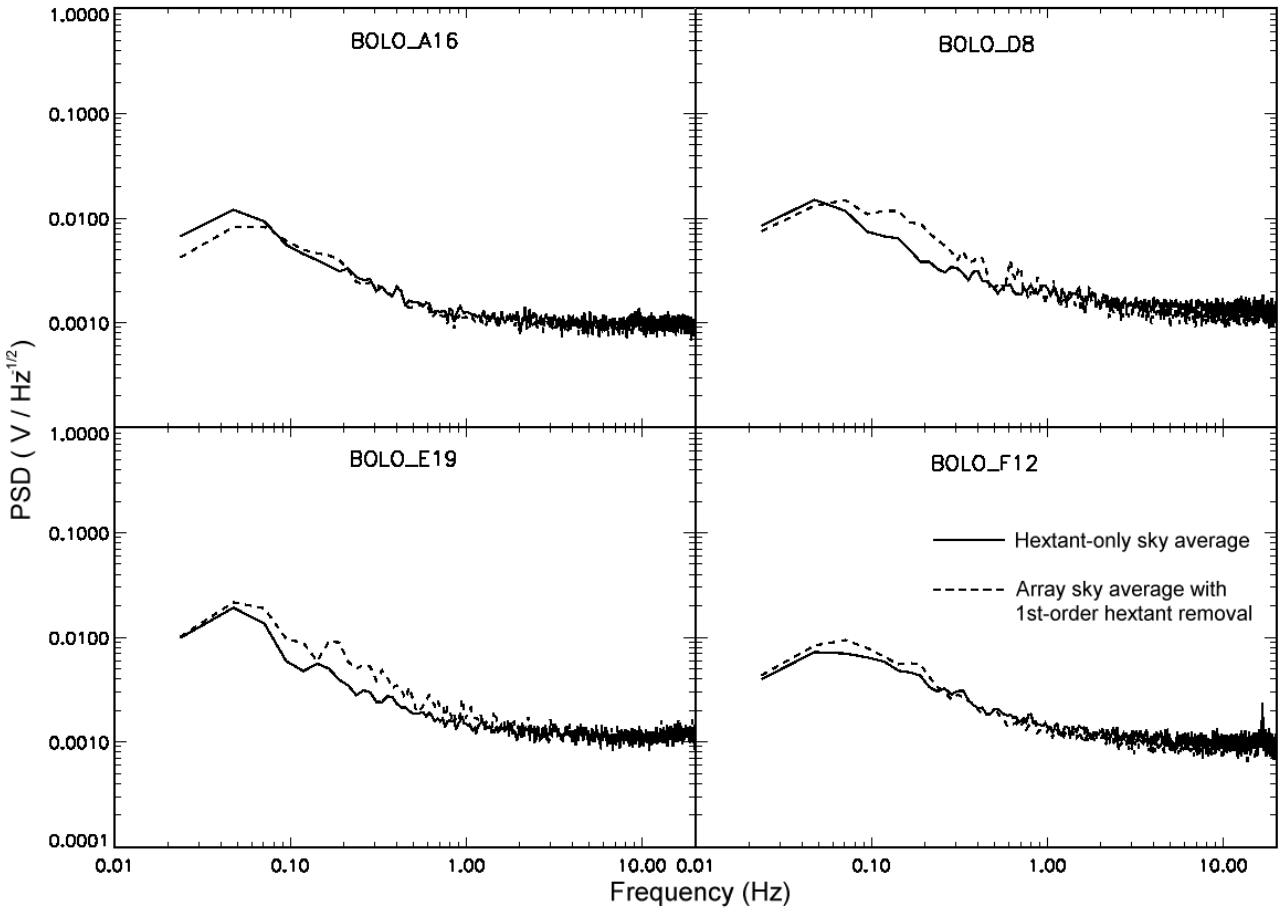

Fig. 5.— Average power spectral density of four bolometers after cleaning. Solid and dashed lines correspond to hextant-only sky-average and array sky-average with first-order hextant noise removal, respectively.

limited as much by electronics as by sky noise. Many improvements to the electronics chain have occurred since this data was taken, however, so the same may not be true for future observing runs. In future runs it may be found that sky noise is the limiting noise contribution. If this happens, further work will be put into improving the sky estimate, perhaps including a spatial, as well as temporal, fit to the sky flux.

4. The NetCDF Data Format

NetCDF (network Common Data Form) is a database format used for array-based data access and organization. It was developed at the Unidata Program Center in Boulder, Colorado, and consists of a freely-distributed collection of software libraries which provide compatibility

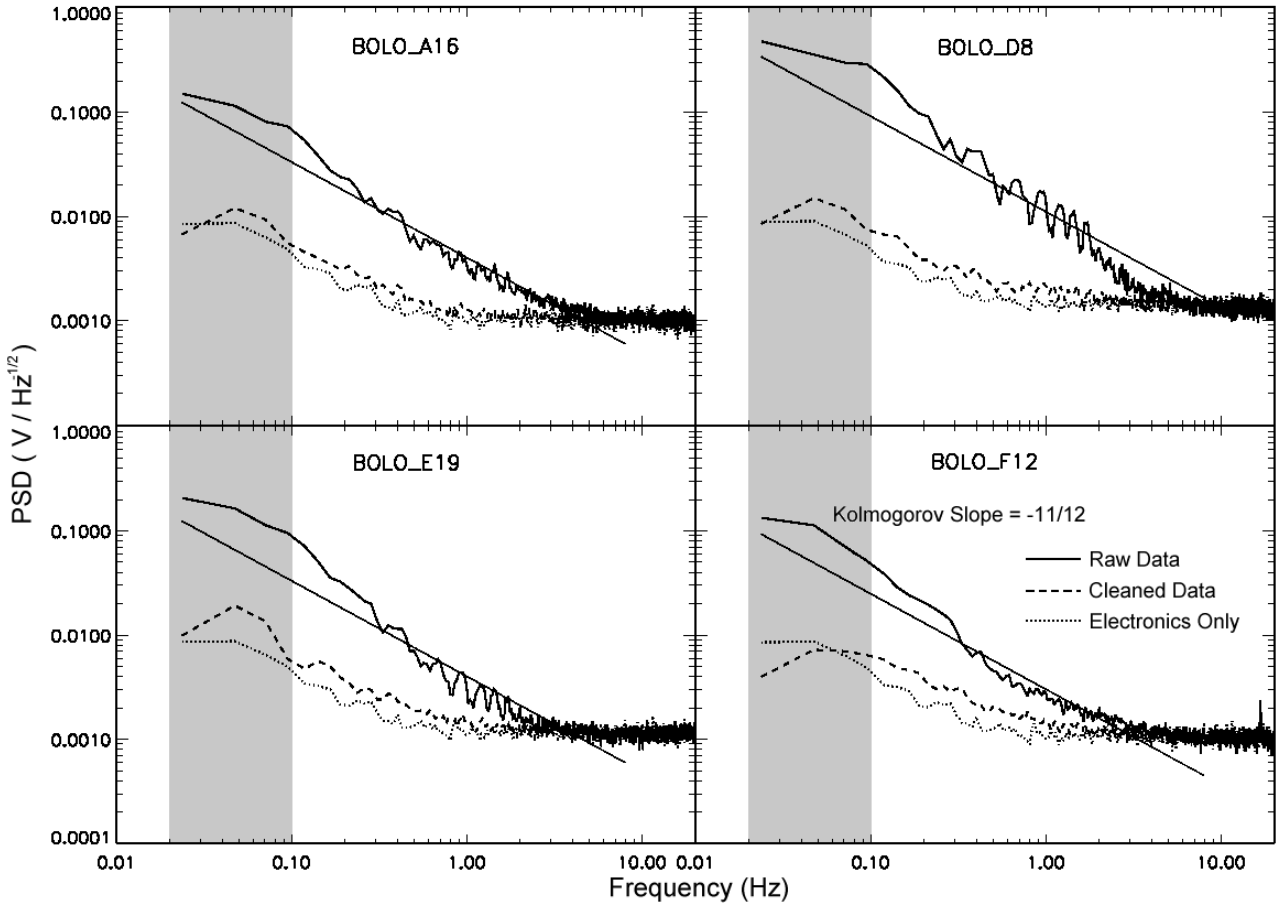

Fig. 6.— Average power spectral density of four bolometers compared to the typical noise from the warm electronics. The straight solid lines are not fits, but rather are intended as a visual indication of the Kolmogorov sky noise slope. Shaded areas correspond to the approximate frequency range in which noise is most important to our sensitivity.

with C, C++, Java, Fortran, Perl, etc. Certain data processing languages like IDL also contain native routines for the reading and writing of netCDF-format files. According to the FAQ at http://www.unidata.ucar.edu/packages/netcdf/, netCDF data is:

- Self-Describing. A netCDF file includes information about the data it contains.
- Architecture-independent. A netCDF file is represented in a form that can be accessed by computers with different ways of storing integers, characters, and floating-point numbers.
- Direct-access. A small subset of a large dataset may be accessed efficiently, without first reading through all the preceding data.
- Appendable. Data can be appended to a netCDF dataset along one dimension without

copying the dataset or redefining its structure. The structure of a netCDF dataset can be changed, though this sometimes causes the dataset to be copied.

- Sharable. One writer and multiple readers may simultaneously access the same netCDF file.

A netCDF dataset consists of three basic components: *dimensions*, *variables* and *attributes*. These can be thought of as a kind of hierarchy, in the sense that variables are declared in terms of pre-defined dimensions, and attributes correspond to pre-defined variables. Note that there can also be global attributes, which do not correspond to a specific variable, but apply to the dataset as a whole.

The declaration of a new dimension requires only a name and a value which designates its size. When a variable is declared, it must be given a name, data type and shape (dimensionality), the latter of which is given in terms of the pre-declared dimensions. Once declared, these properties cannot be changed, although new variables and attributes can always be added. Note that netCDF also allows the use of one "unlimited" dimension, which must be given a specific name just like a normal dimension, but which does not have a set length. Data can therefore be continually appended to a variable declared with this unlimited dimension. This is particularly useful in situations where data is being constantly sampled, in time for example, and the total sampling period is not known with certainty beforehand. Unfortunately, only one dimension can be declared as "unlimited," so its declaration must be chosen with care.

Often it is useful to declare a variable whose sole purpose is to describe a specific dimension. For example, consider a dimension called *column* which has a size of 3, corresponding to three separate columns of related data. One might then declare a variable "column_names" of data type *char* and size *column*$\times$20, which would simply contain the (up to 20 characters long) names of the three columns. Note that this is not the same as a "coordinate variable," by which is meant a variable which corresponds directly to a particular dimension, and contains the actual values of that dimension. (An array of minute values for the time dimension, for example.)

Global attributes are a convenient way to keep track of properties which correspond to the entirety of a dataset. These can range from simple scalar values (e.g. number of times opened, number of subscans within a given file, etc.) to large strings containing the entire processing history of the file. Examples of both of these can be found in Bolocam data. In addition, attributes corresponding to specific variables are a useful way to specify and keep track of such things as units and unit conversions. Organizationally, it is generally good practice to assign the same attributes – unit and conversion factor, for instance – to *all* the variables in a netCDF dataset.

There is one general-purpose UNIX command that is particularly useful in obtaining information about a netCDF-format file: *ncdump*. Specifically, typing

```
ncdump -c [filename]
```

at the shell prompt will print to the screen a list of all dimensions (and their sizes), variables (with dimensionalities and data types), and attributes (including global attributes).

More specific information about the organization of Bolocam data in netCDF format is given in the rest of this section, below. For further general information on netCDF, including how to obtain it, consult the URL provided near the top of this section. Programming environment-specific documentation for netCDF is readily available there as well, including comprehensive user manuals for both C and Fortran, each in a variety of viewing formats. In IDL, typing the term "ncdf" into the online help index will bring up the list of available IDL-native routines for creating, accessing and appending to netCDF files. In addition, two IDL functions have been written specifically for the reading and writing of netCDF-format Bolocam data. These are `readncdf_cal.pro` and `writencdf_cal.pro`, and will be described below.

Finally, before going into the specifics of the Bolocam data format, one other important piece of general netCDF usage should be noted: the difference between and use of the *data* and *define* control modes. When a file is first opened (using `ncdf_open` in IDL, for instance) for reading, one is automatically put into *data* mode. If one wishes to write to the file as well as read, this needs to be specified (e.g. by setting the /write keyword in `ncdf_open`), but this is still *data* mode. However, if one wants to *add* dimensions or variables, as opposed to simply writing into variables that are already defined within the file, it is necessary to switch to the *define* control mode. In IDL this can be accomplished with the `ncdf_control` routine, using the /redef keyword to put an open netCDF file into *define* mode, and the /endef keyword to return it to *data* mode. See the IDL online help for more information on control modes.

The translation of Bolocam data into netCDF format occurs during merging. As will be described in Section 5, beginning with the December 2001 observing run the merging of Bolocam data will be performed "on the fly," that is, as soon as it is recorded by the DAS. However, rather than continuously append the data to an "unlimited" time dimension, we have decided instead to specify the length of this dimension by requiring that each file contains exactly 1 hour's worth of data. The necessary result, that some observations – or even subscans – will be split between more than one file, can be dealt with in a straightforward manner by adding an "observation number" field to the data stream, and making small modifications to the cleaning wrapper software. The advantage of doing it this way is both functional, since we will always know the size of the time dimension, as well as aesthetic, since each output file will be the same size. It is also a good idea in general to limit our file sizes to something "manageable": an hour's worth of Bolocam data occupies about 150 Mb!

Perhaps the most illustrative way to present the Bolocam netCDF-format organization is to reproduce the results of the *ncdump* command on a typical newly-merged data file[8]:

[8]This particular file was *not* merged with the most recent on-the-fly merging software, so some of the variables and dimensions are different than what they will eventually be. In addition, a few variables that are not particularly

```
netcdf 000521_010_raw {
dimensions:
        time = UNLIMITED ; // (114000 currently)
        bolo = 144 ;
        hex = 6 ;
        bolo_param_names = 3 ;
        array_param_names = 5 ;
        char_len = 30 ;
variables:
        int frames_written ;
        char array_param_names(array_param_names, char_len) ;
        float array_params(array_param_names) ;
                array_params:scale_factor = 1 ;
                array_params:add_offset = 0 ;
        float bolo_params(bolo, bolo_param_names) ;
                bolo_params:scale_factor = 1 ;
                bolo_params:add_offset = 0 ;
        char bolo_param_names(bolo_param_names, char_len) ;
        char bolo_names(bolo, char_len) ;
        short ac_bolos(time, bolo) ;
                ac_bolos:scale_factor = 0.0003051758f ;
                ac_bolos:add_offset = 0.f ;
        short dc_bolos(time, bolo) ;
                dc_bolos:scale_factor = 0.0003051758f ;
                dc_bolos:add_offset = 0.f ;
        byte flags(time, bolo) ;
                flags:scale_factor = 1 ;
                flags:add_offset = 0 ;
        short ac_lbias(time, hex) ;
                ac_lbias:scale_factor = 0.0003051758f ;
                ac_lbias:add_offset = 0.f ;
        short trck_das(time) ;
                trck_das:scale_factor = 0.0003051758f ;
                trck_das:add_offset = 0.f ;
        short tran_das(time) ;
                tran_das:scale_factor = 0.0003051758f ;
                tran_das:add_offset = 0.f ;
        short rotating(time) ;
```

useful and/or never used by the software pipeline have been omitted for brevity and clarity.

```
        rotating:scale_factor = 0.0003051758f ;
        rotating:add_offset = 0.f ;
int ra(time) ;
        ra:scale_factor = 2.777778e-06f ;
        ra:add_offset = 0.f ;
int dec(time) ;
        dec:scale_factor = 2.777778e-05f ;
        dec:add_offset = 0.f ;
int lst(time) ;
        lst:scale_factor = 2.777778e-06f ;
        lst:add_offset = 0.f ;
int ut(time) ;
        ut:scale_factor = 2.777778e-06f ;
        ut:add_offset = 0.f ;
int pa(time) ;
        pa:scale_factor = 2.777778e-05f ;
        pa:add_offset = 0.f ;
int az(time) ;
        az:scale_factor = 2.777778e-05f ;
        az:add_offset = 0.f ;
int el(time) ;
        el:scale_factor = 2.777778e-05f ;
        el:add_offset = 0.f ;
int eaz(time) ;
        eaz:scale_factor = 0.1f ;
        eaz:add_offset = 0.f ;
int eel(time) ;
        eel:scale_factor = 0.1f ;
        eel:add_offset = 0.f ;
byte trck_tel(time) ;
        trck_tel:scale_factor = 1 ;
        trck_tel:add_offset = 0 ;
byte tran_tel(time) ;
        tran_tel:scale_factor = 1 ;
        tran_tel:add_offset = 0 ;
float rotangle(time) ;
        rotangle:scale_factor = 1 ;
        rotangle:add_offset = 0 ;
float tau(time) ;
        tau:scale_factor = 1 ;
```

```
                tau:add_offset = 0 ;

// global attributes:
                :num_cleaned = 0 ;
                :history = "merge_array_nc version: 1.000000 Fri Jun 29 16:24:27 2001\n",
}
```

While much of this should be relatively self-explanatory, much also is worthy of more detailed discussion. First I will make some general comments about the overall data organization, after which I will point out a few of the more important but subtle details, and define any terms which are not obvious.

The first thing to note is the filename convention: `yymmdd_xxx_raw`, where `xxx` corresponds to the file number for that night, and the `raw` will be changed to `clean1`, `clean2`, etc., on successive cleanings. In this way file organization is kept completely chronological.

Now take a look at the dimensions. Most important is the `time` dimension, which for this file is unlimited (variable in size and infinitely appendable), but in the future will have a set length of 180,000 (one hour sampled at 50 Hz). The `hex` dimension is primarily for the bias variables (and any variable that contains hextant-correlated information), while `bolo` is hopefully self-explanatory; these two dimensions allow the logical organization of all of the bolometer data channels in just one array. Finally, `char_len` was added to allow for strings (character arrays) of up to 30 characters in length, and the two `_names` dimensions are for the `bolo_params` and `array_params` arrays that will be read in from column-formatted text files of the same names (see Glossary in Section 2.4, or information on supporting files in Section 7.11).

Moving on to the variables, we first notice that nearly all of them are functions of the `time` dimension, which is of course why we refer to this as "time stream" data. With a few exceptions (notably the *rotator* and *tau* fields), the time-streams are identically sampled at 50 Hz. Note, however, that in the engineering run data there is a time offset of about 30 samples (or "frames") between fields recorded by the DAS and fields recorded by the telescope. This problem is dealt with in the cleaning wrapper (see Section 6) and will probably be eliminated altogether in future observing runs.

Another thing we notice about the time-stream variables is that they each have been given `scale_factor` and `add_offset` attributes, which are simply scalar values. These numbers allow a straightforward conversion from the arbitrary units of the computer's analogue-digital converter to volts, degrees, or whatever the desired unit is for each field. In practice, this conversion is accomplished by the `readncdf_cal` and `writencdf_cal` routines, which should always be used when when reading and writing data to and from a file in IDL. The calling sequences are as follows:

```
IDL> readncdf_cal(file,field,offset,count)

IDL> writencdf_cal,data,file,field,offset,count)
```

Here, *file* and *field* are text strings containing the path to the NetCDF file and the name of the variable to be read or written, respectively. (*file* can also be the file ID number for a NetCDF file that has already been opened with the IDL routine `ncdf_open`.) For `writencdf_cal` (which, take note, is a program rather than a function), *data* is simply the name of the idl variable containing the data to be written. Finally, *offset* and *count* are vectors which designate the range of the *field* variable to be read or written. These inputs are formatted identically to those in IDL-native NetCDF input/output routines such as `ncdf_varget` and `ncdf_varput`, so it is important for the user to understand how they work. Basically, it's straightforward: *offset* gives the starting point of the chunk of data to be dealt with (it is a vector because there must be one number corresponding to each dimension), and *count* gives the size of that chunk of data (again, in each dimension). So for example, if you wanted to read in 1000 frames of data for one bolometer, say BOLO_A3, you could do so with the following commands:

```
IDL> file_id=ncdf_open('/home/data/datafile.nc')

IDL> bolo_a3=readncdf_cal(file_id,'ac_bolos',[2,0],[1,1000])
```

The function will then return *bolo_a3* as a 1×1000-element array. *Note that the order of the dimensions specified by* `readncdf_cal` *is reversed from the declared order of the* `ac_bolos` *variable!* This is an idiosyncracy of the IDL-NetCDF interface, and necessitates a standard format for NetCDF variable declaration:

- NetCDF variables should always be declared such that their dimensions are ordered from *largest* to *smallest*. So if declaring a variable in C using a function like `nc_def_var`, this is the order to use.
- If using IDL routines, either for variable declaration or subsequent reading and/or writing, this order is *reversed*. That is, if using `ncdf_varput`, for instance, the order of the declared dimensions is from *smallest* to *largest*.
- The largest dimension (unlimited in the case of the May 2000 engineering run data) is defined to be the `time` dimension.
- After the `scan` dimension has been declared (see Section 6, "Cleaning Wrapper"), it is treated by definition as the second-largest dimension behind `time`. This is because, even though `scan` is generally not as large as `bolo`, for instance, the latter is always constant at 144, whereas `scan` has the potential to be larger.

As mentioned above, the reason one should always use `readncdf_cal` and `writencdf_cal` when reading and writing from IDL is that they automatically apply the `scale_factor` and `add_offset` attributes to the data. This is mechanically very straightforward: the code simply reads the attributes into IDL variables and then adds/multiplies (when reading) or subtracts/divides (when writing) as appropriate.

Several of the NetCDF variables listed above have yet to be defined. First, notice that some variables have the same prefix, but with two different suffixes, `_tel` and `_das`. These correspond to the same data, but recorded separately by the telescope computer and DAS computer. The reason this is important is that the telescope and DAS clocks are not necessarily perfectly synchronized (although in the future they might be), so recording the same TTL[9] signals twice allows us to sync the telescope to the DAS in software.

Specifically, `tran` is a TTL signal, short for "transit," which is 0 before an object has transitted (reached its highest point in the sky), and 1 after. More useful to the software pipeline is `track`, a TTL signal which is zero when the telescope is stationary (drive disengaged), and one when it is moving, or "tracking." In other words, the `track` TTL tells us when an observation is being taken: it is zero when the telescope is drift-scanning, and one otherwise. DAS-telescope synchronization is performed with the `scans.pro` subroutine in the cleaning wrapper (see Section 6 for details).

Finally, here is a list of the remaining Bolocam variables which have not yet been discussed, along with a bit of explanation:

ac_bolos, dc_bolos — These variables have dimensions of time×144, and contain the AC (low- and high-pass filtered) and DC (low-pass filtered only) bolometer signals. The `add_offset` and `scale_factor` attributes convert the units to volts.

ra, dec, pa — In hours, degrees and degrees, respectively.

el, eel, az, eaz — Elevation and azimuth are measured in degrees, and correspond to the *intended* telescope position; `ra` and `dec` are then derived from these values using `pa` and the observatory latitude. However, because wind and gravity can cause the telescope to move slightly from the desired position, the "errors" `eel` and `eaz` must be applied to obtain a completely accurate telescope position on the sky. Note that these have units of arcseconds.

lst, ut — In hours; these should be obvious.

flags — A variable with the same dimensions as the `ac_bolos` variable that contains the bolometer data, but in byte format, as opposed to floating-point. Usually called "data flags," these have several very important functions in the software pipeline. See Glossary entry and Section 7.2 on "Data Flagging" for more information.

rotangle, rotating — The latter of these is simply a TTL that is high (i.e. 1) whenever the dewar is rotating. The former is the current rotation angle as recorded by the rotator encoder. This angle is measured with respect to some fiducial angle, which is contained in the supplementary Array Parameters File (see Glossary entry).

---

[9]TTL stands for transistor-transistor logic, and refers in this case to a binary voltage signal that at a given point in time is either "high" or "low." See also Glossary entry.

tau — Zenith optical depth measured at 225 GHz by the CSO "tipper" (see Glossary entry).

Special note: ac_lbias — This time×6-dimension variable contains the AC bias signal as measured at the lock-in amplifiers; as such, it is essentially a measure of the bias noise. (See Glossary entry on "bias.") However, once it was discovered that the bias is a negligible contribution to the hextant-correlated noise, this variable was over-written with the hextant noise traces calculated by the correlated noise removal software. (See sections on relevant cleaning modules for details.) This practice will most likely be changed in future incarnations of the data pipeline.

If you've read this far, hopefully you have a good idea of the overall intent, organization, and structure of the Bolocam software pipeline. Those users who intend only to use the pipeline as-is, but not to alter it in any way should move on to the Quick Reference Guide (Section 10) and start reducing data. Those who wish to dig a little deeper will want to continue on through the following four sections (5 through 8), where I will detail specifically the mechanics of each piece of code.

5. Merging Software

This code was written in C. Hopefully Sam will write about it at some point!

C Program Name:

Description:

Calling sequence:

Inputs:

Outputs:

Options:

Are there any???

Required subroutines:

Required files:

Program structure:

Additional notes:

6. Cleaning Wrapper

IDL Program Name:

`clean_ncdf.pro`

Description:

This is the only procedure that need be explicitly called in order to clean a Bolocam data file. It is a wrapper program that controls data organization, input and output, and which cleaning steps in particular are to be performed. It is organized in such a way as to be both flexible and easy to use, although there is somewhat of a trade-off between these two characteristics.

Simply put, the cleaning wrapper takes as input the name of the data file to be cleaned and a list of cleaning modules to be run, reads in only the data required to run those modules, and writes out only the data that was changed. While straightforward in concept, there is a great deal of miscellaneous preparation and bookkeeping that needs to take place for this process to run correctly.

All data that is to be used and/or processed by the cleaning modules is read into an IDL structure called *chan*. Individual netCDF variables can be extracted from *chan* by using a command such as, for example:

```
IDL> ac_bolos = chan.ac_bolos
```

or

```
IDL> boloflag_a3 = chan.flags[2,*]
```

(See the IDL online help for more information on the formatting of structures.) Requiring all the data to be contained within this structure is an organizationally simple and effective way of providing the cleaning modules with access to any data that they might need. In addition, by only reading and writing to *chan* the data necessary to run the specified modules, we reduce processing time and memory requirements considerably.

A few words should be said about frame synchronization between DAS and telescope data, which has already been mentioned briefly in Sections 2 and 4. Because data is being taken simultaneously by two different computers, and because we have no good way of making sure the computers' clocks are perfectly synchronized with one another (especially since we record at a rate of 50 frames per second), the frame numbers corresponding to the data recorded by each computer will not be precisely matched. We correct for this in software by first recording the same TTL signal simultaneously with both the DAS and telescope – these are the tracking signals `trck_das` and `trck_tel`. Then, using the code `scans.pro` described below, we calculate the offset between these signals for each subscan. Note that the frame offset drifts slightly over time, which is why it must be recorded for each subscan. These offsets are recorded both to the netCDF file itself and an IDL variable, the latter of which is subsequently passed by the cleaning wrapper to each cleaning module. It is then left up to the author of each module to make sure that the synchronization offsets are properly applied. Further details can be found in Section 6.1, below.

For a more complete general description of the purpose and organization of `clean_ncdf.pro`,

see Section 2.2 before reading further.

Calling sequence:

```
clean_ncdf, [mod_file = module_file, infile = infile.nc, outdir = outdir,
 nscans_to_process = nscans_to_process, array_params_file = array_params_file,
 bolo_params_file = bolo_params_file, /nooutput, /noprompt, /newfile]
```

Inputs:

mod_file — Set this equal to a text string specifying the path to the module file. The module file is an ascii text file containing the names of the modules to be run (".pro" subscript omitted), in the order in which they are to be run. The default value of this parameter can be set at the beginning of `clean_ncdf.pro`, and is usually something like:

```
mod_file = '/home/misc_path/cleaning_input/module_file.txt'.
```

infile — Set this equal to a text string containing the path to the netCDF-format Bolocam data file to be cleaned (with ".nc" subscript included). If this parameter is not specified, the user is prompted for it. If it is not given and the /noprompt keyword is set (see below), an error message is returned.

outdir — Set this equal to a text string containing the path to the output directory. This parameter is only needed when either a) the data has not been cleaned before (i.e. the input file ends in "raw.nc"), or b) the data has been cleaned before but the /newfile keyword is set. In other words, it is needed whenever the input file is to be copied to a new file; otherwise, the old file is overwritten by default.

nscans_to_process — Set this equal to the number of subscans to be read in and cleaned at a time. The default value is 1. If this parameter is set to a number equal to or greater than the total number of subscans in a file, the entire file is cleaned at once. Note that for very large files, doing this may pose a challenge to the machine's RAM, and either cause the cleaning routine, or IDL, to crash.

array_params_file — Set this equal to a text string containing the path to the array parameters file. See the Glossary for a description of this file. For the May 2000 observing run, the array parameters file looks like:

```
platescale_("/mm)          7.40284
beamsize_(",FWHM)          39.0000
fid_arr_ang(deg)           96.
x_bore_off(inches)         0.
y_bore_off(inches)         0.
```

(Note that the underscores in the parameter names are necessary for `readcol.pro` to correctly interpret the column formatting.) Upon first cleaning, the array parameters file (as well as the bolometer parameters file; see below) are automatically appended to the output file. Therefore, this path need only be specified if the input data has not been cleaned before, or if the data has already been cleaned, but the user wants to overwrite the old array parameters with new values. If the data has *not* already been cleaned and this parameter is omitted, the user is prompted for it; if this occurs and the /noprompt keyword has been set, an error is returned.

bolo_params_file — Set this equal to a text string containing the path to the bolometer parameters file. See the Glossary for a description of this file. For the May 2000 observing run, the first few lines of the bolometer parameters file look like:

```
BOLO_A1       1    300.0000000      1.0000000
BOLO_A2       1    300.0000000      2.0000000
BOLO_A3       1    330.0000000      1.7320508
BOLO_A4       1    300.0000000      3.0000000
BOLO_A5       2    319.1065979      2.6457512
BOLO_A6       -1   340.8934021      2.6457512
BOLO_A7       0    300.0000000      4.0000000
BOLO_A8       0    313.8978882      3.6055512
BOLO_A9       1    330.0000000      3.4641016
BOLO_A10      0    346.1021118      3.6055512
                        ⋮
```

The columns correspond to the bolometer channel name, bolometer flag, bolometer angle (relative to the fiducial angle given in the array parameters file) and the bolometer distance from the center of the array, specified in units of bolometer separations. (One bolometer separation equals 5 mm in physical space on the wafer.) Like the array parameters file, this parameter need only be specified if the data has not been cleaned before, or if the user wants to overwrite the old bolometer parameters with new values. The default actions are identical to those for the array parameters file as well.

Outputs:

Excepting the one case (see `avgpsd_module.pro`) where a cleaning module outputs information which is not included in the netCDF data stream, the output of the cleaning wrapper is simply a cleaned netCDF file. Also, keep in mind that any data that has passed through the cleaning wrapper will always have bolometer and array parameters, as well as subscan information, appended to it.

Keyword options:

/nooutput — Setting this will instruct the cleaning wrapper not to write an output file. This is useful in situations where it is desirable to read in and process the data, but not save the results; when testing out new cleaning modules, for instance. Note that if the input file has not been

cleaned before and /nooutput is set, an error message will be returned. This was done to insure that the raw data file is never directly appended to or altered in any way.

/noprompt — Setting this will suppress all prompts for additional input. This is useful when cleaning large files, and the user wishes to leave the computer unattended for long periods of time. Keep in mind that setting this keyword necessitates that all required parameters be properly specified, otherwise error messages will result.

/newfile — Setting this when processing an already-cleaned file overrides the default action, which is to overwrite the cleaned file with new data. The output filename will have the suffix "clean$X$," where X is an integer which has been incremented by one from the previous filename. (Upon first cleaning, the output filename suffix is "clean1.")

Required subroutines:

`readcol.pro` — comes with the IDL astrolib, which can be found at:

http://idlastro.gsfc.nasa.gov/homepage.html

`scans.pro` — see description below

`getchan.pro` — see description below

`putchan.pro` — see description below

`update_history.pro` — see description below under "Program structure"

`readncdf_cal.pro` - see description in Section 4

`writencdf_cal.pro` - see description in Section 4

Required files:

`input netCDF file` — always required. See Inputs, above.

`module_file` — always required. See Inputs, above.

`array_params` — only required for first cleaning, optional otherwise. See Inputs, above.

`bolo_params` — only required for first cleaning, optional otherwise. See Inputs, above.

Program structure:

The first part of the code checks to see what keywords are set, and sets the defaults for parameters that are not explicitly specified.

Next the code checks the `num_cleaned` global attribute in the netCDF file to determine whether or not the file has been cleaned before. If it has been cleaned, and the /newfile keyword has *not* been set, the default action is just to use the input file as the output file, and overwrite any data that has been changed. If the input file has not already been cleaned, there is a section of code

which extracts the filename and copies the file to the specified output directory *outdir*, replacing the "raw" suffix with a "clean$X$" suffix (as described above under */newfile*).

The next couple of sections of code only deal with files which have not been cleaned before. First, `scans.pro` is run to determine the subscan information in the file (see code description, below). Once this has been done, a `scans` dimension is created, along with `scans`-sized variables and their corresponding attributes. These variables are `pixel_offsets`, `rmsnoise` and `scans_info`. The former two of these will be filled in with data from the pixel offsets and rmsnoise cleaning modules, respectively (see Section 7). The latter is created to contain the output of the `scans.pro` code (see Section 6.1, below) so that it doesn't have to be run again for future cleanings; `scans_info` has dimensions of `scans`×3, with the three "columns" containing *npstart*, *npend* and *npoff*, respectively. In addition, `num_scans` is written to the netCDF output file as a global attribute of the file. Note that the control mode must be switched from *define* to *data* using the `ncdf_control` routine before the newly-defined variables can be filled with data.

Once all new variables have been defined, the code next checks to make sure that it has the necessary paths to the `bolo_params` and `array_params` files. (User is prompted if it does not, unless /noprompt keyword is set). These files are then read in, using the IDL astrolib routine `readcol.pro`, from the user-specified paths, and written to the relevant variables (created during merging and previously empty) in the netCDF file. The `bolo_params` and `array_params` filenames are also recorded as attributes of their corresponding variables for bookkeeping purposes.

By this point in the code (line 265, for the version last updated 11/19/01) all preliminary and optional steps have been taken care of. Everything that remains (with the exception of whether or not output is written) is essentially the same no matter what the input parameters.

The first step is to read the `scans_info` parameters *back* into the IDL variables *nscans*, *npstart*, *npend* and *npoff* (since this has to be done for already-cleaned files as well as first-time-cleaned files). Next the `readcol.pro` routine is again used, this time to read the user-specified module file. The names of the modules to be run are put, in their listed order, into the IDL variable *modules*. In addition, the variable *modnames* is created to contain the "root" module names; these have had all text after "`_module`" removed. This little trick allows the user to create and keep track of different versions of the same cleaning module, without having to re-specify in the `getchan` and `putchan` codes the required variables for each module version. (See subsections below on "Reading in the data" and "Writing out the data" if this is not clear.)

Now the data is ready to be split up into chunks[10], read into the IDL *chan* structure, and fed through the cleaning modules. The wrapper loops over the total number of subscans in the file, in increments of *nscans_to_process*. For each data chunk the number of subscans, starting

[10]I will use the word "chunk" throughout the rest of the manual to refer to time-sequenced data that has been split up into *nscans_to_process*-sized pieces. While not exactly a formal expression, it is in my opinion just as functional, and more straightforward, than the alternatives.

frame, and number of frames are recorded as *sub_nscans*, *sub_startframe*, and *sub_nframes*. These are then input, along with the beginning subscan number *iscan*, module names *modnames*, and netCDF filename, into the `getchan.pro` routine. Discussion of this routine will be deferred to its own subsection, below; suffice it to say here that it is a function that returns the structure *chan*, with all data necessary for the specified cleaning modules to run properly.

After putting the scan information for the current chunk of data into the IDL variables *sub_npstart*, *sub_npend* and *sub_npoff*, the wrapper is ready to send these variables along with *chan* to the individual cleaning modules. For this, the IDL `call_procedure` routine is invoked in a loop, as follows:

```
for imod=0, n_elements(modules)-1 do begin
  print,'Running module ' + modules[imod]
  call_procedure, modules[imod], chan, sub_nscans, sub_npstart, sub_npend,$
   sub_npoff, status
endfor
```

Note that `imod` is an index which loops over the number of requested modules, and `status` is generally not used, but was left as is in case we ever find a need for it.

This loop runs uninterrupted until the current data chunk of *nscans_to_process* subscans has been run through all the cleaning modules contained in the module file. This may take seconds or hours, depending on the size of the data chunk and the modules being run. Once this step is complete, the results – contained in *chan*, as always – can be written to the output file. This is accomplished by the routine `putchan.pro`, which works in exactly the opposite manner as `getchan.pro`; see the corresponding subsections, below, for details. After `putchan` has run, the entire process repeats with the next chunk of *nscans_to_process* subscans. Of course, care is taken so that the wrapper recognizes if there are less than *nscans_to_process* subscans left in a file, and deals with the remaining data accordingly.

Once all subscans of the data have been cleaned, all that is left to do is update the `history` attribute of the netCDF output file. The `update_history.pro` routine takes as input the list of modules that were run and the output filename, and adds the module names, version numbers and system date and time to `history`. To access this attribute in IDL from an already-opened file with netCDF ID *fileid*, you can use the `ncdf_attget` command, as follows:

```
IDL> ncdf_attget,fileid,/global,'history',history
IDL> print,string(history)
merge_array_nc version: 1.000000 Fri Jun 29 14:36:01 2001
Sun Mar  5 23:34:11 2000 pixel_offsets_module version: 3.0
Sun Mar  5 23:34:11 2000 desource_module_3c273 version: 4.0
Sun Mar  5 23:34:11 2000 despike_module version: 3.2
```

```
Sun Mar  5 23:34:11 2000 deconv_module version: 1.1
Sun Mar  5 23:34:11 2000 getbias_module_flag version: 4.0
Sun Mar  5 23:34:11 2000 skybiassub_module_hex_flag version: 2.0
Sun Mar  5 23:34:11 2000 polysub_module version: 1.2
Sun Mar  5 23:34:11 2000 relsens_module_psd version: 3.0
Sun Mar  5 23:34:11 2000 rmsnoise_module version: 2.0
Sun Mar  5 23:34:11 2000 debadscan_module version: 2.0
```

Finally, note that calling a cleaning module with the /version keyword set to a named variable (no other inputs necessary) will return the version number in that variable. In `update_history` this looks like:

```
for i = 0, nmodules - 1 do begin
   call_procedure, modules[i], version = temp
   versions[i]=temp
endfor
```

where the loop is over the number of specified modules *nmodules*, and *versions* is an *nmodules*-sized array.

Additional notes:

See below for descriptions of the principal subroutines.

See beginning of Section 7 for cleaning module naming conventions and proper calling sequence.

6.1. Getting subscan information

IDL Program Name:

`scans.pro`

Description:

`Scans.pro` uses the `trck_das` and `trck_tel` variables to determine several important parameters which are necessary for the cleaning process. These include the beginning and ending frame numbers of each subscan, the total number of subscans in a file, and the offset in frames between the DAS and telescope data for each subscan. These values are calculated only once for a given file – during its first cleaning – and then written into the output file, after which point they are always directly accessible in the same netCDF format as the rest of the data stream. Also note that the number of subscans found by `scans.pro` is used during the first cleaning to create a *scans* dimension within the output file, and subsequently a number of *scans*-sized variables.

A few words about the *npoff* variable

Calling sequence:

`scans, nscans, npstart, npend, npoff, trck_das, trck_tel`

Inputs:

`trck_das` and `trck_tel` — These are extracted directly from the raw netCDF data file.

Outputs:

nscans — The number of subscans in the input file. All of the following three output variables are *nscans*-sized arrays, i.e. having one value per subscan.

npstart — The starting frame numbers of each subscan, defined by the `trck_tel` variable.

npend — The ending frame numbers of each subscan, defined by the `trck_tel` variable.

npoff — The offset, in number of frames, between the DAS and telescope data for each subscan. For the May 2000 observing run data it is always positive. Specifically:

$$\text{DAS frame \#} = \text{telescope frame \#} - \text{npoff}$$

Program structure:

`Scans.pro` uses the IDL `where` command to pick out the parts of `trck_tel` and `trck_das` which correspond to driftscans (i.e. where the tracking signals are "low"). Doing this for the telescope tracking signal, and differencing the resulting array and the same array shifted by 1 and -1 gives us the start and end frames, respectively, for each subscan. Then it is only a matter of repeating this for the DAS tracking signal, and calculating *npoff* from the difference of the telescope and DAS start frames for each scan.

6.2. Reading in the data

IDL Program Name:

`getchan.pro`

Description:

This program is a function which takes as input the name of the netCDF file to be cleaned, a string array of the names of the modules to be run, and the range of data to be processed (usually some fraction of the total). A series of `IF` statements check to see if a given module has been requested, and if so, reads the data required by that module from the netCDF file into an IDL structure called *chan*. When this process is complete, *chan* is returned to the cleaning wrapper as the function value.

Calling sequence:

```
chan = getchan(modules, filename, iscan, nscans, startframe, nframes)
```

Inputs:

modules — A string array containing the names of the requested modules found in the module file. Any module version subscripts after "_module" have been omitted in order to simplify coding; i.e. so that all versions of the same cleaning module are dealt with identically, without having to repeat commands.

filename — A string containing the path and filename of the netCDF file to be cleaned.

iscan — An integer corresponding to the beginning subscan number for the current data chunk.

nscans — An integer corresponding to the number of subscans in the current data chunk.

startframe — A long integer corresponding to the beginning frame number of the current data chunk.

nframes — A long integer corresponding to the number of frames in the current data chunk.

Outputs:

chan — A structure containing all (and only) the data required by the cleaning modules specified in the module file.

Required subroutines:

`readncdf_cal.pro` - see description in Section 4.

Program structure:

After the netCDF file given by *filename* is opened, the *chan* structure is defined. Initially, it contains only three fields: `bolo` (a string array of the bolometer names), `trck_tel` and `trck_das`. These are always present in *chan*, regardless of what cleaning modules are run.

The next section of the code defines a list of variables – keywords, really – corresponding to the names of all the modules that the user might have wanted to run. If a given module name is present in *modules*, its keyword variable (e.g. *pixo* for "pixel_offsets_module") is set to 1; otherwise, it is set to 0.

Under the "Description" heading above, I stated that a series of `IF` statements for each module determine what variables are read into *chan*. In actuality, there is an `IF` statement for each *variable*. That is, for a given variable – `array_params`, say – the code checks to see if any of the modules that requrire that variable – pixel_offsets_module or desource_module – have been called. If any of them have, the variable in question is read into *chan*, and the code moves on to the next variable. `Readncdf_cal.pro` is used for the actual reading in, so that data scaling is treated consistently

from file to file.

Having finished this process, the input (soon to be output) netCDF file is closed, and *chan* is returned.

Additional notes:

This code takes advantage of an especially useful application of the IDL `create_struct` routine; namely, concatenation. It is possible to add new variables to a pre-defined structure with a command like:

```
chan = create_struct(chan,'new_variable_name',new_variable),
```

which leaves all the previous variables in *chan* unaffected. Then to access this variable at a later time, you would use:

```
new_variable = chan.new_variable_name.
```

6.3. Writing out the data

IDL Program Name:

`putchan.pro`

Description:

This code works essentially identically to `getchan.pro`, but in reverse. Using the list of module names in *modules*, the procedure determines which variables were changed during the cleaning process, and writes them to the specified output netCDF file. This is again accomplished with a series of `IF` statements. Note that `putchan` does not return an IDL variable or structure; it only writes data from the IDL structure *chan* to an external file.

Calling sequence:

```
putchan, modules, filename, iscan, nscans, startframe, nframes
```

Inputs:

chan — A structure containing the data to be written out to the netCDF output file.

modules — A string array containing the names of the requested modules found in the module file. Any module version subscripts after "_module" have been omitted in order to simplify coding; i.e. so that all versions of the same cleaning module are dealt with identically, without having to repeat commands.

filename — A string containing the path and filename of the netCDF file to be written to.

iscan — An integer corresponding to the beginning subscan number for the current data chunk.

nscans — An integer corresponding to the number of subscans in the current data chunk.

startframe — A long integer corresponding to the beginning frame number of the current data chunk.

nframes — A long integer corresponding to the number of frames in the current data chunk.

Outputs:

Required subroutines:

`writencdf_cal.pro` - see description in Section 4.

Program structure:

The output file specified by *filename* is opened, and as with `getchan.pro` (see above), keyword variables are defined for each module that might have been run.

Next is a series of `IF` statements, one for each variable that might have been changed during cleaning. If one or more of the cleaning modules that affects the variable in question is found to be in *modules* (i.e. if their corresponding keywords have been set to 1), data from *chan* is written to the relevant variable in the netCDF output file. The `writencdf_cal.pro` routine is used for this, so that the scaling of the data is treated consistently from file to file.

Once all the possible altered variables have been written out, memory is cleared by resetting *chan* to a scalar 0, and the output file is closed.

7. Cleaning Modules

This section will cover the specifics of and inter-relations between the individual cleaning modules, the workhorses of the Bolocam software pipeline. By this point it is assumed that the reader is familiar with the organization of Bolocam data, as well as the overall structure of the software pipeline. Furthermore, the information here is generally intended for those who really want to understand the details of the Bolocam cleaning process, and not necessarily for the casual user. Also note that, because new modules are always being written and old modules improved upon, the descriptions herein should be considered a work in progress. However, after reading through this section, the user should have everything he or she needs to start adding his or her own code to the software pipeline.

A few general cleaning module characteristics need to be discussed before getting into specifics. First, the naming convention; the cleaning module IDL code files have names of the form:

XXXXX_module[_XXXXX].pro

where the first series of "X"s represents an arbitrarily-long sequence of characters that describes the essential function of the module, and the second series of "X"s (in brackets) represents an optional sequence of characters which identifies different versions of the same module. Note that cleaning modules with the same prefix (but different suffixes) must take the same input variables and alter the same output variables; otherwise they should be written as entirely different modules with different prefixes, with `getchan.pro` and `putchan.pro` modified accordingly.

Another important thing to note is that all modules are procedures (not functions) and, more crucially, *the calling sequence for all cleaning modules is identical.* This calling sequence is:

```
generic_module, chan, nscans, npstart, npend, npoff, status, version=version
```

where *chan* is the IDL data structure containing all necessary information for the modules to run (see Sections 6.2 and 6.3, above), *nscans* is the number of subscans in the current chunk of data being processed, and *npstart*, *npend* and *npoff* contain the subscan information for the current data chunk (see Section 6.1, above). The *status* variable is currently unused, but was left as is in case a good use is ever found for it.

The *version* keyword, while not often used[11], requires a bit more explanation, since it directly affects the format of the cleaning module code. Essentially, you can set *version* to a named variable that will, upon return, contain the version number of the cleaning module. Note that no other inputs are required with *version* is invoked, since the module returns immediately, without any data processing having actually taken place. In order for this to work, every module must start off with the lines:

```
version_number=0.0 ;(or whatever the version number is)

if n_params() lt 4 then begin
   version=version_number
   return
endif
```

Also note that the version number returned by the *version* keyword is not the same as what is meant by the "Version" subheading in the discussions of some of the modules below. I.e., each version of a cleaning module (to which the subheadings refer) may have its own succession of version *numbers.*

[11] This keyword is really only used by the `update_history` procedure, described under the "Program structure" heading of the cleaning wrapper discussion.

Because every module shares the same calling sequence, the "Inputs" and "Outputs" headings below do not simply correspond to the *chan* structure and subscan information; these are universally implied. Rather, "Inputs" refers only to the specific *chan* variables that are required by a given module (and explicitly specified in `getchan.pro`), and "Outputs" refers only to the *chan* variables that are permanently changed by the module (and explicitly specified in `putchan.pro`).

On a similar note, the "Required subroutines" headings are intended only as a list of *non-IDL-native* (i.e. Bolocam-specific) subroutines, and not of all subroutines in the module.

Finally, here are a few additional things to keep in mind if you decide to write your own cleaning modules:

1. Identify all the input variables that the module will need to work properly, and alter `getchan.pro` to make sure those variables are read in when your module is called. This is straightforward, and consists essentially of adding a module-specific "keyword variable" at the top of the code (see discussion in previous section), and appending this to the relevant `IF` statements further down in the code.

2. Similarly, identify all the output variables that are changed by the module, and alter `putchan.pro` accordingly. Again, this involves adding a "keyword variable" and appending it to the `IF` statements.

3. Be careful when reading and writing to the *chan* structure. It is good practice to always copy any required variables from *chan* into a new variable for the duration of the code, so that the risk of making accidental permanent changes is minimized. Then, if permanent changes to *chan* do need to be made, leave the decisive step to the end of the code.

4. Think about how your module will change the data in *chan*, and make sure that these changes do not interfere with any other modules. Similarly, make sure that, when running your module along with a list of other modules, they are ordered in a logical way. See Section 10.2 on "Proper ordering of cleaning modules" for more information.

7.1. Calculating pixel offsets

IDL Program Name:

`pixel_offsets_module.pro`

Description:

This program calculates the right ascension and declination offsets for each bolometer and subscan. Adding these offsets to the telescope boresight RA and DEC (*chan.ra* and *chan.dec* when read into IDL) gives the actual sky coordinates "seen" by each bolometer as a function of time. This step is necessary for mapping, obviously, but is also required for the source-flagging module

(see Section 7.2 which follows), or any situation where the individual beam positions on the sky need to be known precisely.

Given the parameters found in the `array_params` and `bolo_params` files, the `rotangle` and average `pa` for each scan, and coefficients describing the field distortion, the code simply applies what we know of the geometrical optics in order to simulate what the array looks like projected on the sky. The beam positions are calculated in the RA/DEC *offset* coordinate frame, such that the telescope boresight is always the origin. This means that *chan.ra* and *chan.dec* are not actually used in the calculation at all.

This code works no matter where on the sky the telescope was pointed, and whether or not the rotator was engaged during the observation, although it requires that the value of `rotangle` be the actual value. (For this reason, `rotangle` should be recorded into the data stream at all times, and not just when the rotator is on.) For more information on the rotator, correcting for PA, and various coordinate conventions, see Appendix A at the end of this manual.

Inputs:

arrang — The fiducial rotation angle of the dewar in degrees; specified in the `array_params` file. The definition of this angle is somewhat arbitrary, since it is degenerate with *boloang* (see below). For the May 2000 observing run, *arrang* was defined to be the angle (measured clockwise when standing behind the dewar) between an imaginary horizontal line on the array, and the line that divides the A and F hextants: about 96 degrees.

xoff and *yoff* — From the `array_params` file, these are the spatial offsets, in inches on the focal plane, of the center of the array from the telescope boresight. They are hopefully zero, or as close zero as possible. The positive x-axis of this coordinate system is horizontal and points to the right (when standing behind the dewar, looking down into the optics box). The positive y-axis is therefore perpendicular to this, pointing along the array toward the telescope[12].

platescale — Also from `array_params`, this is the plate scale at the focal plane, in units of arcseconds per millimeter.

bolodist and *boloang* — From the `bolo_params` file, these are 144-element arrays containing the distances (in bolometer separation units; see Section 6) and angles (in degrees, relative to some fiducial position) of each bolometer.

`pa` — This is essentially a conversion factor between the Alt/Az to RA/DEC coordinate systems as a function of time.

`rotangle` — The rotation angle of the dewar relative to *arrang*, in degrees.

Outputs:

[12]This is also sometimes called the "mirror symmetry axis." See Appendix A for more on coordinate conventions.

RA and DEC pixel[13] offsets, in units of arcseconds. These are written to *chan.pixel_offsets*, and then to the `pixel_offsets` netCDF variable created by the cleaning wrapper during the first cleaning of a file. They should be *added* to the telescope boresight `ra` and `dec` to produce the correct coordinates for each beam on the sky.

Other important parameters:

xcs and *ycs* — These are the field distortion coefficients, and are written explicitly into the module code (at least for now). They are derived by propagating a grid of points through the Bolocam optics in Zemax (or similar optical design software), and recording the focal plane positions. A second-degree transformation of the form:

$$x_{new} = a_1 + a_2 x + a_3 y + a_4 xx + a_5 yy + a_6 xy + a_7 xxy + a_8 xyy + a_9 xxyy \tag{8}$$

and

$$y_{new} = b_1 + b_2 x + b_3 y + b_4 xx + b_5 yy + b_6 xy + b_7 xxy + b_8 xyy + b_9 xxyy \tag{9}$$

can then be derived for the *inverse*, i.e. time-reversed, process, corresponding to known focal plane positions being projected onto the sky. Here, $xcs = \vec{a}$ and $ycs = \vec{b}$.

Required subroutines:

None.

Associated cleaning modules:

No modules are required to be run before `pixel_offsets_module.pro`, and only `desource_module.pro` needs to be run after.

Program structure:

After reading in all the input variables, the program picks the average PA out of the `pa` timestream for each subscan. Similarly, it finds the rotation angle for each subscan (which is easy, since the rotator is stationary while we are driftscanning).

Next the code loops over each bolometer and subscan, calculating the pixel offsets for each case. This calculation goes as follows. First, for a given bolometer, *boloang*, *bolodist*, `rotangle` and *arrang* are used to determine its x and y positions on the array. Then the (time-reversed) field distortion corrections are applied, and the new values are multiplied by the plate scale. Flipping about the x-axis yields the Alt-Az coordinates of the bolometer as projected on the sky, in arcseconds. These are then converted to RA and DEC by rotating by the average PA of the subscan, and flipping about the y-axis (since right ascension increases in the direction opposite of Az at PA = 0).

Finally, the results are written to *chan.pixel_offsets* and the module returns.

---

[13]The word "pixel" is here used to refer to a bolometer detector and, more specifically, its beam as projected on the sky.

Additional notes:

A useful feature of the IDL *atan* function was used for this calculation. Specifically, if two arguments are provided, in the form:

```
IDL> theta = atan(y,x)
```

then theta will correspond to the angle whose tangent equals y/x. This provides a straightforward way to convert from cartesian to polar coordinates while maintaining proper sign conventions.

7.2. Data flagging

The concept and usefulness of data flagging has already been covered in Sections 2 and 4. This section will first describe the subroutine functions that actually read and write the data flags, and then will go on to discuss the three primary flagging modules.

7.2.1. Setting data flags

IDL Program Name:

`set_flag.pro`

Description:

This function is used to set data flags. It takes as input byte-format data from *chan.flags*, and outputs the same data with all bits corresponding to a desired data flag set to binary "1." The data flag to be set is specified with a keyword, and no other data flags are altered by the process. The user can then check later on whether a specific data flag has been set by using the `read_flag.pro` function, described below.

Calling sequence:

```
set_flag, flag, spike = spike, source = source, badscan = badscan,$
 baddata = baddata, bit4 = bit4, bit5 = bit5, bit6 = bit6, bit7 = bit7
```

Inputs:

flag — A byte-format array of any size corresponding to an arbitrary subsection of the *chan.flags* array.

Outputs:

This code returns the same byte-format array as the input array, but with all flag bits (and *only* those bits) corresponding to the user-specified keyword set to binary "1".

Keyword options:

/spike, /source, /badscan, /baddata, /bit4, /bit5, /bit6, /bit7 — These keywords each correspond to one of the 8 bits in each byte of the input *flag* array. The last four are not currently used. No more or less than one keyword may be specified in a single call to `set_flag.pro`.

Program structure:

This is a very simple code containing only 8 statements, 1 for each of the 8 bits in each byte of the input array. The first three of these statements are:

```
if keyword_set(spike) then newflag = flag OR 1
if keyword_set(source) then newflag = flag OR 2
if keyword_set(badscan) then newflag = flag OR 4
```

OR is IDL's Boolean inclusive operator. To understand how this works, recall that the 8-bit syntax for the numbers 1, 2 and 4 are given by 00000001, 00000010, and 00000100. Because all the bits are 0s except for one, and because *OR operates in a bitwise fashion*, all the bits in *newflag* are identical to those in *flag*, except for the one corresponding to the specified keyword. This bit will always be 1, regardless of its original value in *flag*.

Once all 8 keywords have been checked, *newflag* is returned as the output function value.

Additional notes:

The OR operator is like any other IDL operator in that it works on arrays as well as scalar values. So if the input *flag* is an array (in *byte* format, of course), the output will also be an array (in *byte* format). Thus the entire *chan.flags* array, or any subsection thereof, can be fed at once into `set_flag.pro`, if desired.

Note that there is currently no way to "unset" a data flag once it has been set.

7.2.2. Reading data flags

IDL Program Name:

`read_flag.pro`

Description:

This function checks to see whether or not a specific data flag in the input array has been set, returning a non-zero value if it has. The data flag to be checked is specified by a keyword corresponding to one of the 8 bits in each byte of the input array.

Calling sequence:

```
read_flag, flag, spike = spike, source = source, badscan = badscan, $
 baddata = baddata, bit4 = bit4, bit5 = bit5, bit6 = bit6, bit7 = bit7
```

Inputs:

flag — A byte-format array of any size corresponding to an arbitrary subsection of the *chan.flags* array.

Outputs:

The returned function has a non-zero value if the flag bit corresponding to the specified keyword has been set, and 0 otherwise. (The specific value returned depends on which keyword is specified: 1 for /spike, 2 for /source, 4 for /badscan, 8 for /baddata, etc.)

Keyword options:

/spike, /source, /badscan, /baddata, /bit4, /bit5, /bit6, /bit7 — These keywords each correspond to one of the 8 bits in each byte of the input *flag* array. The last four are not currently used. No more or less than one keyword may be specified in a single call to `read_flag.pro`.

Program structure:

This code is almost exactly like `set_flag.pro`, and is every bit as simple. But instead of OR statements which always set a specified bit to 1, the code uses AND statements to check whether or not a specified bit has already been set to 1. For example:

```
if keyword_set(source) then result = flag AND 2
```

See discussion of `set_flag.pro`, above, for more information.

Once all keywords have been checked, the code converts the *result* variable to short integer type, and it is returned as the output function value.

Additional notes:

The AND operator is like any other IDL operator in that it works on arrays as well as scalar values. So if the input *flag* is an array (in *byte* format, of course), the output will also be an array (in *integer* format). Thus the entire *chan.flags* array, or any subsection thereof, can be fed at once into `read_flag.pro`, if desired.

7.2.3. Flagging known sources

IDL Program Name:

`desource_module.pro`

Description:

This module flags regions in each bolometer's time stream where known bright sources are passing through its beam. By "known bright sources" I mean sources of which we know the RA and DEC, and which are generally several orders of magnitude brighter than the faint (usually cosmological) sources we are trying to detect. Flagging these sources is necessary, primarily so that we do not include their flux in calculations of the sky and electronics noise.

In order to flag a known source for a given bolometer, we first need to know its RA and DEC offsets (calculated by `pixel_offsets_module.pro`) so that we have the exact location of its beam on the sky. Then we simply take all the frames where the beam position is within a certain threshold distance from the known source, and set the source flag in those frames using `set_flag.pro`, described above. The threshold distance can be changed in the code, but typically it is taken to be the 3σ distance of a gaussian fit to the beam (which is equivalent to the beam FWHM multiplied by 1.274).

This module was written such that an arbitrary number of sources may be flagged for a given observation. In addition, while the code was originally only intended for flagging point sources (i.e. circular objects the size of our beam), it is possible to use it to flag extended sources as well, by simply approximating them as many closely-spaced circular beams.

Inputs:

`ra, dec, el, pa, eel, eaz` — From the input netCDF file. These variables were defined in Section 4.

raoff and *decoff* — The RA and DEC offsets calculated by the pixel offset module and recorded to *chan.pixel_offsets*.

beamsize — The beam FWHM from the `array_params` file.

Outputs:

An updated `flags` array, with source flag bits set to 1 for frames in which a known point source passes within a 3σ radius of the beam for each bolometer.

Other important parameters:

sourcera and *sourcedec* — These are two separate 1-D arrays containing the RA(s) and DEC(s) of the source(s) to be flagged. Note that as of this writing, these variables are written into the code itself. Thus, the flagging of different sources is achieved by actually changing the code, or by having several versions of the same code, each with different *sourcera* and *sourcedec* variables. By the December 2001 observing run, we should be able to avoid this situation for most cases (i.e. for bright point sources) by recording directly to the data stream the object positions given by the telescope computer.

Required subroutines:

`set_flag.pro`

Associated cleaning modules:

`pixel_offsets_module.pro` — must be called before `desource_module.pro`. (See also "Additional notes," below.)

Program structure:

Once all the input variables have been read from chan, and the beamsize scaled to a 3σ threshold distance, the code starts a loop over all subscans. It then pulls out the telescope data (`ra`, `dec`, `pa`, etc.) for the current subscan, and calculates RA and DEC error offsets from EEL and EAZ using the expressions:

```
dra = -eaz_scan*cos((pi/180.)*pa_scan)*cos((pi/180.)*el_scan)+$
  eel_scan*sin((pi/180.)*pa_scan)
```

and

```
ddec = eaz_scan*sin((pi/180.)*pa_scan)*cos((pi/180.)*el_scan)+$
  eel_scan*cos((pi/180.)*pa_scan) .
```

Here, *dra* and *ddec* are the error offsets, and "_scan" refers to just the data for the current subscan.

Next the code loops over bolometers, applying the error offsets and pixel offsets (*raoff* and *decoff*) individually to each, for each frame in the subscan.

Finally, the code loops over the sources to be flagged (usually just one, but sometimes many, as in the case of an extended source). For each source, the distance between the beam position and source position is calculated for every frame, and those frames where this distance is less than the 3σ distance are flagged as sources with `set_flag.pro`.

Once these loops are finished and *chan.flags* updated, the module returns to the cleaning wrapper.

Versions:

For data in which the source RA and DEC has been recorded to the data stream:

`desource_module.pro`

In special cases, or for flagging sources in various observations from the May 2000 run:

`desource_module_uranus.pro`

`desource_module_3c273.pro`

`desource_module_dr21.pro`

Additional notes:

The pixel offsets and source flagging modules should generally be run together at the beginning of the cleaning process, since they are not dependent on any other modules, but many of the other modules are dependent on them. Specifically, sources need to have been flagged before the sky and hextant noise calculation and removal, before polynomial removal, and before PSDs are calculated.

7.2.4. Flagging spikes

IDL Program Name:

`despike_module.pro`

Description:

The purpose of this module is to locate and flag frames in the bolometer data streams which contain unwanted spikes. Such spikes can be a result of cosmic rays, microphonics, corrupted data, or any number of unexplained phenomena.

Several de-spiking algorithms have been attempted. The current version, which is the quickest yet, uses a variety of IDL shortcuts to treat the entire data stream for a bolometer at once, as opposed to going through the data on a frame-by-frame basis. In addition, rather than flagging erroneous frames based on whether their values exceed some threshold voltage, the code instead looks for large deviations in *slope* between adjacent frames.

Inputs:

`ac_bolos` — Bolometer timestream data.

chanflags — From `bolo_params` file; specifies which bolometers are "good," so that spikes are found for those bolometer channels only.

Outputs:

An updated `flags` array, with source flag bits set to 1 for frames which are found to have "spikes" or "glitches." See "Program structure" for specific information on how these are defined.

Required subroutines:

`set_flag.pro`

Associated cleaning modules:

None; however, this module should be run near the beginning of cleaning before the sky and hextant noise calculations.

Program structure:

Sam, please fill this in!

Additional notes:

???

7.2.5. Flagging bad subscans

IDL Program Name:

`debadscan_module.pro`

Description:

This module was designed to flag not individual frames, but rather an entire *subscan*, based on whether it exhibits an erroneously high or low standard deviation from the mean (usually just called "RMS"). The threshold RMS values are determined empirically; there is a fairly clearly-defined range in the RMS values of good subscans, so that subscans that do not conform to the standard are easy to identify.

An RMS value indicating a bad subscan can be the result a number of causes: a bad bolometer, a good bolometer that temporarily "went bad," or perhaps simply a subscan that did not get cleaned very well for some reason. Whatever the cause, this module is intended to be a last resort of sorts, to catch bad or poorly-cleaned subscans before they can contribute to the final 2-D map.

This module and the RMS noise module that must run before it are meant to be called *after* sky and hextant noise subtraction has been performed, so that sky noise does not contribute significantly to the calculated RMS value.

Inputs:

chanflags — From the `bolo_params` file; specifies which bolometers are "good," so that bad scans are found for those bolometer channels only.

rmsnoise — The array containing the RMS values for every bolometer and subscan calculated by `rmsnoise_module.pro`.

Outputs:

An updated `flags` array, with source flag bits set to 1 for entire subscans which are found to have erroneous RMS values.

Required subroutines:

`set_flag.pro`

Associated cleaning modules:

`rmsnoise_module.pro` — Must be called *before* this module.

Program structure:

This is a very straightforward code. The module loops through the good bolometers specified by *chanflags*, and over all subscans in *chan*. For each one it checks to see if *rmsnoise* falls outside the range $0.001 < rmsnoise < 0.1$ Volts; if it does, the entire subscan is flagged with the `set_flag.pro` routine using the */badscan* keyword. Note that this range currently applies only to data taken during the May 2000 observing run, and may not apply to future data sets.

7.3. Polynomial removal

IDL Program Name:

`polysub_module.pro`

Description:

This module fits and removes a low-order polynomial from each subscan in each bolometer channel. Such a polynomial subtraction accomplishes essentially the same thing as a high-pass filter with a very low-frequency cutoff: only the very lowest frequencies are removed, and all others are unaffected.

This calculation, while simple, is quite effective in removing the lowest-frequency components of the $1/f$ sky and electronics noise. And because the polynomial in question is of low (typically third) order, this process should have negligible impact on point sources convolved with our beam. The same can not necessarily be said for extended sources, so use of this module is discouraged in such cases if one wishes to avoid indiscriminately removing astrophysical flux.

Finally, since we are only interested in removing low-frequency components of the $1/f$ noise , care is taken so that bright sources and spikes are not included in the polynomial fit. Luckily, this is made quite simple by the use of data flags (described in Section 7.2, above) combined with the */measure_errors* keyword in the IDL *poly_fit* routine.

Inputs:

`ac_bolos` — Bolometer timestream data.

`flags` — Data flags for each bolometer. The module needs these so that bright sources, spikes and otherwise bad data frames are not included in the calculation.

chanflags — From the `bolo_params` file; specifies which bolometers are "good," so that calculation is performed for those bolometer channels only.

Outputs:

The code writes out polynomial-subtracted `ac_bolos` variable to *chan*.

Required subroutines:

`read_flag.pro`

Associated cleaning modules:

None. However, this code should not be called until *after* sky and hextant noise subtraction and deconvolution, but *before* the RMS noise and relative sensitivities calculations. In a sense, polynomial subtraction can be thought of as just a supplement to the other noise removal and filtering modules.

Program structure:

First, for those who may wish to experiment with varying degrees of polynomial fits, the *degree* variable is defined at the very top of the code. The value of this variable is the only difference between the different versions of `polysub_module.pro`.

This code is comprised of an outer loop over subscan, after which are some subscan-specific variable declarations, and then an inner loop over the good bolometers specified by *chanflags*. The bolometer data and data flags are then read into variables, the latter of which are translated by the `read_flag.pro` routine and combined into a corresponding *weights* array. This array consists of INFs for frames where source, spike, or baddata flags have been set, and 1s for frames where no flags have been set.

Next, the IDL `poly_fit` routine is called with the specified polynomial degree, and with the */measure_errors* keyword set to the *weights* array (see the IDL Online Help for more information on `poly_fit`). In this way the polynomial fit is made to completely ignore frames containing spikes or bright sources, while other frames are given full and equal weight.

Once the fit has been made, it is written to the *polyfit* variable and subtracted from the original bolometer data for the current subscan. Finally, the resulting data are written back out to *chan.ac_bolos*, and the code continues on to the next bolometer.

Versions:

There are various versions of this module which have names corresponding to the convention:

`polysub_module_N.pro`,

where N refers to the order of the polynomial to be removed. For the May 2000 data set it was found that, for relatively short subscans (less than about 5000 frames), third, fourth, and fifth-order polynomial fits produce essentially identical results.

Additional Notes:

Make sure that whenever you run the deconvolution module (see below) *after* hextant and

sky noise removal, always run this code immediately afterwards to take out slopes and DC offsets. (If deconvolution is performed before hextant and sky noise removal, which generally yields better results anyway, `polysub_module` should be run after `skyhexsub_module.pro`.)

7.4. Hextant and sky noise removal

This section addresses the practical application of the correlated noise removal formalism discussed in Section 3. Readers should refer there first before continuing with the software-specific discussion below.

There are many ways one might choose to organize the correlated noise removal software, and our cleaning module format seems to broaden, rather than limit, these possibilities. It is therefore important to keep in mind that the two modules discussed here represent only one of many possible implementations. In addition, as our understanding of the sources and nature of correlated noise in Bolocam improves, there will likely be many changes and additions to the current structure. To this end, the descriptions here are intended to be helpful for understanding, not only the current software format, but also the relevant issues one might encounter when making future changes.

The current correlated noise removal software consists essentially of four separate cleaning modules that are meant to be run in tandem in various combinations. The first, `gethexnoise_module.pro`, calculates timestream data arrays containing models of the hextant-correlated noise for each hextant, and writes them to the `ac_lbias` variable in *chan*. The second module, `skyhexsub_module.pro`, calculates the sky-correlated (or "common mode") noise and then scales and subtracts it, along with the hextant-correlated noise, from each bolometer. The resulting "cleaned" data is then written back out to *chan*. The third and fourth cleaning modules (`skysub` and `hexsub`) are similar to, and take the place of, the `skyhexsub` module, but perform the sky and hextant subtractions separately. That is, the hextant noise is removed first, so that it will not be included in the subsequent sky calculation.

Note that in practice this description is sometimes misleading, since one method does not require a separate calculation of hextant noise at all. Indeed, once it was discovered that the bias contributes negligibly to the $1/f$ noise, the `gethexnoise` module all of a sudden became largely irrelevant. Which is not to imply that there is no hextant noise; rather, we simply no longer had a good way to calculate it separately from the sky noise. (The one method developed so far that works reasonably well is described in equations (6) and (7).) So now, instead of treating sky and hextant noise separately, we have the option of treating them simultaneously (see equation (5)) by modifying the sky calculation and subtraction module so that it averages over each hextant separately, rather than over the entire array. This is the technique employed by `skyhexsub_module_hex.pro`, and `skysub_module.pro`. The difference between these is simply that the latter does not remove the hextant traces in `ac_lbias`, so it does not require the `gethexnoise` or `hexsub` modules to be run at all. Although `skyhexsub_module_hex.pro` still requires a hextant noise calculation, the resulting

traces typically comprise a negligible fraction of the the calculated sky noise.

The "Description" sections below will describe the correlated noise removal software in algorithmically generic terms; that is, without discussing the specific formalism by which the module accomplishes its task. Then, under the "Program structure" subheadings for each module, I will address in more structurally-specific terms a few of the various methods one might use to perform the actual calculations of the sky and hextant noise traces.

7.4.1. Hextant noise calculation

IDL Program Name:

`gethexnoise_module.pro`

Description:

Generically, this module uses the bolometer timestream data in `ac_bolos`, from which it derives the hextant-correlated component of the electronics noise. The results are in the form of an arbitrarily-scaled timestreams, one for each hextant, which are then writen to the $nhex \times nframes$-size variable `ac_lbias` in the output netCDF file. This variable can then be read in by either the `skyhexsub` or `hexsub` modules, whose job it is to actually remove the hextant noise from the datastream.

A secondary output of this module is that it sets a "baddata" flag in the `flags` variable for frames in which there is insufficient information to calculate a hextant noise trace. This generally only results from the rare situation in which several bolometers in a hextant happen to have been flagged with spike or source flags in the same frames. Note that these "baddata" frames are flagged for *every* bolometer, rather than for just individual bolometers. Thus, they essentially represent a region in *time* which is to be ignored during subsequent processing steps.

Because the organization of this code varies significantly depending on the method of hextant noise calculation, it is impossible to go into more detail here. See "Program structure," below, for details about the different versions of this code, their organization, and the methods they employ.

Inputs:

`ac_bolos` — Bolometer timestream data. Needed for any noise calculation method based on averaging bolometer channels together. At the time of writing, all methods require `ac_bolos`.

chanflags — From `bolo_params` file; specifies which bolometers are "good," and which are "open," etc. The latter provide a measure of the bias signal, although for the May 2000 data it was found that the bias correlates poorly with the hextant noise. Therefore, the open channels are generally no longer incorporated into the hextant noise calculation.

`flags` — Data flags for each bolometer. The module needs these so that bright sources and

spikes are not included in the calculation.

Outputs:

`ac_lbias` — Hextant noise traces for each hextant are returned to this variable in *chan*, and written to the output netCDF file. Note that this variable initially held the bias channels for each hextant measured at the lock-in amplifiers. However, since the bias was found not to be a significant contribution to the hextant-correlated noise, there shouldn't be a problem in overwriting this variable. (This may change in future versions of the software.)

`flags` — With "baddata" flag set for frames in which there is not enough information to calculate the hextant noise.

Required subroutines:

`set_flag.pro`

`read_flag.pro`

Associated cleaning modules:

`skyhexsub_module.pro` — If this is run, it should always be run in tandem with, but *after*, the hextant noise calculation.

`hexsub_module.pro` and `skysub_module.pro` — If these are run in place of `skyhexsub_module.pro`, they should always be run *after* the hextant noise calculation, with the hextant noise subtraction performed *before* the sky noise subtraction.

Program structure:

Versions:

`gethexnoise_module_bias.pro` —

`gethexnoise_module_diff.pro` —

Additional notes:

7.4.2. Sky noise calculation; hextant and sky noise subtraction

IDL Program Name:

`skyhexsub_module.pro`

Description:

This module has two primary functions, and one secondary function. The first primary function is the calculation of the common mode sky noise by averaging together good bolometer channels,

and assuming that all contributions to the output voltage are negligible compared to the sky noise. There are several versions of this module, each of which performs this calculation in a slightly different way. For instance, the average can be over the entire array, or over a hextant only. It can be completely unweighted, so that all bolometer channels in the average are treated equally, or it can be weighted by bolometer separation distance on the array. In the latter case, separate sky noise traces are calculated for each bolometer, each with a different weighting kernal centered on its location on the wafer. In addition, this kernal can be either linear or gaussian in shape, although in either case the weighting is a very weak function of distance (since the sky noise is nearly common mode), and so the results are very nearly identical.

The second primary function, which is the same for all versions of the module, is the subtraction of sky and hextant noise from the `ac_bolos` variable. The hextant noise is here taken to be the noise traces calculated by the `gethexnoise` module, which by this point have been written to the `ac_lbias` variable. Inherent in the noise subtraction is scaling; the sky and hextant noise traces are scaled simultaneously with each other, and separately for each bolometer. This is accomplished with the IDL `regress` function, which performs a multiple linear regression fit of the form:

$$D_{bolo} = a_0 + a_1 N_{sky} + a_2 N_{hex} \tag{10}$$

where D_{bolo} is the raw data for a given bolometer channel for one subscan, N_{sky} and N_{hex} are the arbitrarily-scaled sky and hextant noise traces, and the a_ns are the fitting coefficients. The presence of a_0 means that any DC offset in the data will have been removed after this module is run.

Finally, the secondary function of `skyhexsub_module.pro` is the flagging of frames where not there is not enough information to perform an accurate sky noise calculation. Like in the `gethexnoise` module (see above), such frames are flagged as "baddata" in all bolometers simultaneously, and so data taken during that period of time is simply ignored in subsequent processing and mapping steps.

An alternative to running the `skyhexsub` module after the hextant calculation is to instead run *both* the `hexsub` module and `skysub_module_all.pro`, in that order. See corresponding subsections below for details.

Inputs:

`ac_bolos` — Bolometer timestream data. Needed for any noise calculation method based on averaging bolometer channels together. At the time of writing, all methods require `ac_bolos`.

`ac_lbias` — Hextant noise traces computed by `gethexnoise_module.pro`, and written to the variable that initially held the bias channels for each hextant measured at the lock-in amplifiers. Since the bias is not a significant contribution to the hextant-correlated noise, there shouldn't be a problem in overwriting this variable. (Note: this may change in future versions.)

chanflags — From `bolo_params` file; specifies which bolometers are "good," and therefore which should be included in the bolometer-averaged sky noise calculation.

`flags` — Data flags for each bolometer. The module needs these so that bright sources, spikes and otherwise bad data frames are not included in the calculation.

Outputs:

`ac_bolos` — Sky and hextant noise-subtracted bolometer data channels. Upon output, data is considered to have been "cleaned," although high-pass filter deconvolution and low-order polynomial subtraction (both to be eventually replaced by optimal filtering) are yet to be done. Note that as a general rule, *this should always be the first module to actually alter the contents of the* `ac_bolos` *variable*. This is important, because both the sky and hextant noise calculations require the data to be in its still-raw form.

`flags` — A secondary output of this module is that it sets a "baddata" flag for frames in which there is insufficient information to calculate a sky noise trace. This generally only results from the rare situation in which several bolometers in a hextant happen to have been flagged with spike or source flags in the same frames.

Required subroutines:

`set_flag.pro`

`read_flag.pro`

Associated cleaning modules:

`gethexnoise_module.pro` — Should always be run in tandem with, but *before* hextant and sky subtraction.

Program structure:

Versions:

`skyhexsub_module_hex.pro` —

`skyhexsub_module_all.pro` —

`skyhexsub_module_locgauss.pro` —

`skyhexsub_module_loclin.pro` —

`skysub_module_all.pro` —

`skysub_module_hex.pro` —

Additional notes:

7.4.3. Hextant noise subtraction

IDL Program Name:

`hexsub_module.pro`

Description:

Though it has only one task to perform – the subtraction from each bolomter of the hextant noise traces calculated by `gethexnoise_module.pro` – this module is more complex than one might expect. This is because, in order to properly scale the hextant noise to each bolometer channel, we must also know roughly how much the sky noise contributes to the data. That is, it is necessary that we calculate and fit a sky noise trace as well as a hextant noise trace, so that the IDL `regress` function does not scale the hextant noise to the full amplitude of the bolometer data.

The sky noise estimate used at present is just a simple average over all bolometers. (Hence there is currently only one version of this module.) Once the proper hextant noise scaling coefficient has been found for a given bolometer (a_2 in equation (10), above), the hextant noise and DC offset are removed and the module returns. Then, when the sky noise is re-calculated by `skysub_module_all.pro`, we do not have to worry about including hextant noise in the common mode average. In this way, `hexsub_module.pro` and `skysub_module_all.pro` combine to form a kind of iterative sky subtraction process.

Inputs:

`ac_bolos` — Bolometer timestream data. Needed for any noise calculation method based on averaging bolometer channels together. At the time of writing, all methods require `ac_bolos`.

`ac_lbias` — Hextant noise traces computed by `gethexnoise_module.pro`, and written to the variable that initially held the bias channels for each hextant measured at the lock-in amplifiers. Since the bias is not a significant contribution to the hextant-correlated noise, there shouldn't be a problem in overwriting this variable. (Note: this may change in future versions.)

chanflags — From `bolo_params` file; specifies which bolometers are "good," and therefore which should be included in the rough sky noise calculation.

`flags` — Data flags for each bolometer. The module needs these so that bright sources, spikes and otherwise bad data frames are not included in the calculation.

Outputs:

`ac_bolos` — Hextant noise-subtracted bolometer data channels. Note that when this module is run, *it should be the first module to actually alter the contents of the* `ac_bolos` *variable.* This is important, because the hextant noise calculation require the data to be in its still-raw form.

Required subroutines:

`read_flag.pro`

Associated cleaning modules:

`gethexnoise_module.pro` — Should always be run in tandem with, but *before* hextant subtraction (obviously).

`skysub_module.pro` — Should always be run in tandem with, but *after* hextant subtraction, so that hextant noise is not included in the sky average.

Program structure:

Versions:

`hexsub_module_all.pro` —

Additional notes:

7.4.4. Sky noise calculation and subtraction

IDL Program Name:

`skysub_module.pro`

Description:

There are two versions of this module that, while identical in a number of ways, are actually used very differently in the context of the overall cleaning process. In the case of the first, `skysub_module_hex.pro`, both hextant and sky noise components are calculated and removed simultaneously, and so the hextant calculation and subtraction modules need not be run at all; it is a "standalone" module.

In contrast, `skysub_module_all.pro` is meant to be run in series with, and directly after, the `gethexnoise` and `hexsub` modules (in that order). In actuality, the whole point of the `hexsub` module is to remove hextant noise separately from the sky noise, so that when the sky noise is then re-calculated by `skysub_module_all.pro`, the resulting average will be contaminated with as little hextant-correlated noise as possible. This also explains why we want to average over *all* bolometers on the array for this method; once hextant noise has been removed, a straightforward average over the entire array is the best estimate for the sky noise, assuming as we do that sky noise is truly common mode in nature.

And finally, like with the `skyhexsub` and `gethexnoise` modules, a secondary purpose of this module is to flag as "baddata" frames in which an accurate sky noise calculation is not possible.

Inputs:

`ac_bolos` — Bolometer timestream data. Needed for any noise calculation method based on

averaging bolometer channels together. At the time of writing, all methods require `ac_bolos`.

`ac_lbias` — Hextant noise traces computed by `gethexnoise_module.pro`, and written to the variable that initially held the bias channels for each hextant measured at the lock-in amplifiers. Since the bias is not a significant contribution to the hextant-correlated noise, there shouldn't be a problem in overwriting this variable. (Note: this may change in future versions.)

chanflags — From `bolo_params` file; specifies which bolometers are "good," and therefore which should be included in the bolometer-averaged sky noise calculation.

`flags` — Data flags for each bolometer. The module needs these so that bright sources, spikes and otherwise bad data frames are not included in the calculation.

Outputs:

`ac_bolos` — Sky noise-subtracted bolometer data channels. Upon output, data is considered to have been "cleaned," although high-pass filter deconvolution and low-order polynomial subtraction (both to be eventually replaced by optimal filtering) are yet to be done.

`flags` — A secondary output of this module is that it sets a "baddata" flag for frames in which there is insufficient information to calculate a sky noise trace. This generally only results from the rare situation in which several bolometers in a hextant happen to have been flagged with spike or source flags in the same frames.

Required subroutines:

`set_flag.pro`

`read_flag.pro`

Associated cleaning modules:

`gethexnoise_module.pro` and `hexsub_module.pro` — If these are run, they should be run together and *before* sky noise subtraction. Note that in general these are only meant to be run with `skysub_module_all.pro` and *not* with `skysub_module_hex.pro`.

Program structure:

Versions:

`skysub_module_hex.pro` —

`skysub_module_all.pro` —

Additional notes:

7.5. Calculating and using Power Spectral Densities

We define the power spectral density, or PSD, of a real, time-sequenced function h(t) as:

$$PSD(f) = \frac{2}{\Delta f}|H(f)|^2, \tag{11}$$

where Δf is the smallest allowable frequency (defined as the inverse of the total sampling time), and H(f) is the (fast) fourier transform of h(t). By this definition the PSD has units of Volts^2 Hz^{-1}, but conventionally we quote the square-root of this quantity, which gives a value more appropriate for the assessment of noise. (I.e. since the bolometer channels are almost completely dominated by noise, PSD$^{1/2}$ gives us the *noise* in the signal as a function of frequency, which is more interesting to us than the power.)

The PSD of a time-sequenced array of data is the primary diagnostic tool used to determine the amount of $1/f$ noise present in the data. Thus, we can also use it to assess the quality of our sky and hextant noise removal methods. In this subsection I will first explain the code that actually calculates the PSD of a chunk (typically a subscan) of data, and then will move on to describe how the results of this code are used in two different cleaning modules.

For more information about PSDs, FFTs, or spectral analysis numerical techniques in general, see Numerical Recipes. The online version can be found at: http://www.nr.com.

7.5.1. The basic PSD code

IDL Program Name:

`psd_scan.pro`

Description:

This function takes as input a 1-D time-sequenced array of data, usually one subscan in length, and returns a 1-D frequency-sequenced array containing the square-root of the PSD, in units of Volts Hz$^{-1/2}$. Note that the returned array has half the number of elements of the input array, since the highest frequency in the FFT (and therefore PSD) of a discretely-sampled set of data is the Nyquist frequency, or *half* the sampling frequency.

The PSD is calculated with a "Hann" windowing function (defined explicitly in the code, but identical to the IDL `hanning` function with $\alpha = 0.5$), which falls off to zero at its edges. In addition, the average value (DC offset) of the input array is always subtracted out first, before the PSD is calculated.

Calling sequence:

```
result = psd_scan(data)
```

Inputs:

data — A 1-D time-sequenced array of data. Ususally this is the length of a subscan, and subscans tend to have lengths on the order of thousands of frames. Note that, due to our 50 Hz sampling frequency, the corresponding sampling interval Δt is 20 ms.

Outputs:

ps — A 2-D array containing the 1-D $\sqrt{\text{PSD}}$ of the input array, in units of Volts $\text{Hz}^{-1/2}$, as well as a corresponding 1-D frequency array. The output array therefore has the dimensions 2 $\times(N_{samples}/2)$, where $\text{N}_{samples}$ is the number of elements of the input array. In addition, the maximum frequency is 25 Hz (assuming a $(40\text{ ms})^{-1}$ Nyquist frequency), and the minimum frequency is 0 Hz (where PSD(0) = 0, since the DC offset has already been removed). The frequency interval Δf, which is equivalent to the *second* lowest frequency in the output array, depends on the total number of samples in the input array as:

$$\Delta f = \frac{1}{\Delta t \times N_{samples}} \ .$$

Required subroutines:

None.

Associated cleaning modules:

This code is used by `avgpsd_module.pro` and `relsens_module_psd.pro`, both of which are described below.

Program structure:

The very first thing this function does is to find the mean value of the input array (using the IDL `moment` routine), and subtract it, creating a new temporary data variable in the process.

Next the Hann windowing function is created with the same number of elements as the input array, along with a normalization coefficient W_{ss} given by:

$$W_{ss} = \frac{1}{N_{samples}} \sum_{i=0}^{N-1} W_i \ ,$$

where W_i is the time-dependent windowing function, i represents frame number, the sum is over all ($\text{N}_{samples}$) frames, and W_{ss} stands for "window summed and squared[14]."

[14]As per Numerical Recipes. Notice, however, that we define W_{ss} slightly differently than Numerical Recipes, since we desire a specific normalization of (and units for) the PSD, whereas their discussion assumes an arbitrary normalization.

Now the code is ready to calculate the fast fourier transform of the DC offset-removed input array multiplied by the windowing function W. This is accomplished with the IDL `fft` routine; by convention, the forward transform is returned if the "direction" input variable in `fft` is set to -1.

Finally, the $\sqrt{\text{PSD}}$ is calculated according to equation (11), with proper normalization obtained by multiplying the Δf in the denominator by the W_{ss} coefficient derived above. This step is performed as a loop over frequency bins j, with the frequency of each bin written to $ps(0, j)$, and the $\sqrt{\text{PSD}}$ of each bin written to $ps(1, j)$. Upon completion of this loop, the function returns ps as its value.

7.5.2. Computing relative sensitivities from PSDs

IDL Program Name:

`relsens_module_psd.pro`

Description:

Here, relative sensitivity refers (somewhat confusingly) to an estimate of the amount of low-frequency noise in one subscan of bolometer channel data. This noise estimate is useful as a way to compare all bolometers and subscans in the data stream to one another, in terms of how well the cleaning modules were able to remove $1/f$ noise from them. Specifically, we define the relative sensitivity of a bolometer and subscan in terms of its PSD as:

$$\text{Rel. Sens.} = \left[\int_{f_{min}}^{f_{max}} PSD \; df \right]^{1/2} ,$$

where f_{min} and f_{max} are parameters set within the code which define the frequency range in which we are most interested. Typical values when observing point sources are about 100 and 200 mHz, respectively, but f_{max} should be lowered in cases where one is observing extended objects.

This calculation should be performed after all noise removal and filtering modules have been run, since what we are interested in is the end product of the cleaning process. The computed relative sensitivities can then be treated by the mapping code (see below) as relative weights during coaddition, which is basically a weighted average over all bolometers and subscans. Subscans with high *relsens* values have a lot of low-frequency noise, so they are weighted less than subscans with relatively lower *relsens* values.

Note that the way we pick the frequency range over which to integrate is by calculating a typical width *in the frequency domain* of the astronomical objects that we are trying to observe. If these objects are point sources, then they will just be the size of our beam (which we can assume to be gaussian). If they are extended sources, however, the spatial domain width increases, so the frequency domain width *decreases*. Furthermore, this means that if any bright sources are present

in a subscan, they will contribute excess power to the low-frequency end of the corresponding PSD, and so we will derive an inflated value for *relsens*, resulting in a *lower* weight for that subscan. This problem is dealt with (albeit imperfectly) by fitting and subtracting a gaussian from subscans containing source flags. In addition, spikes are dealt with by interpolating straight lines between the frames which flank the spike-flagged frames.

Inputs:

`ac_bolos` — Bolometer timestream data from which PSDs will be calculated for each subscan in *chan*.

chanflags — From `bolo_params` file; specifies good bolometers.

`flags` — Data flags for each bolometer. The module needs these so that bright sources and spikes are not included in the calculation. See "Program structure" for more information on how these are dealt with.

Outputs:

The module returns the relative sensitivities for each good bolometer and subscan, written to *chan.relsens*, and then to the `relsens` variable in the output netCDF file. These will be used as relative weights by the mapping routine during subscan coaddition. The output relative sensitivities have units of volts, and can be treated somewhat analogously to 1σ noise values.

Required subroutines:

`read_flag.pro`

`psd_scan.pro`

`gaussfit_nan.pro` — This is a custom-modified version of the IDL `gaussfit` routine that recognizes NaN values in the data to be fit, and exclueds them from the fitting process.

Associated cleaning modules:

None, although the relative sensitivities computed by this module are necessary for the mapping process.

Program structure:

First, the code uses the *npend* and *npstart* variables to determine the maximum length of the PSDs to be returned by `psd_scan.pro`, so that a single array can be created to hold all of them. This array is created, along with another to hold the integrated PSD values, and the code enters a double loop over good bolometers and all subscans.

Once within the loops, the code defines subscan-sized variables to hold the bolometer data and corresponding flag arrays, calculates the length of the PSD that will be returned by the current subscan, and then checks (with an `IF` statement) to make sure that this length is reasonable. If

all is okay, the code proceeds to read in the bolometer and flag data, and translates the latter into separate *spike* and *source* arrays.

If spikes or source flags are found in the current subscan, a new `IF` loop is entered to deal with them. Spikes are treated as follows: first, the frames flanking the spikes are found by careful use of the IDL `where` and `shift` routines on the *spikes* array (similarly to how they are used in `scans.pro`; see Section 6.1, above). Next, the program loops through all regions of contiguously-flagged spikes in the subscan, and for each one interpolates a straight line between the flanking frames.

If the code finds that at least 25 frames have been flagged as containing a source, and the source in question is *not* located at either edge of the subscan, then it begins the following process to remove it: first it excludes from the source region frames that have also been flagged as spikes, and makes sure that there are still at least 25 frames remaining. Then it sets all spike-flagged frames to NaN (using the IDL system variable `!values.f_nan`), so that they are easily ignored by the fitting routines to come. Now the code begins calculating initial estimates of the parameters to be fit by a gaussian-fitting routine. These include the peak value of the gaussian, the x-position of the peak, a guess at the gaussian half-width, and linear coefficients determined by fitting the source region with the IDL `poly_fit` routine. All of these are then fed into `gaussfit_nan.pro`, which is a custom-modified version of the IDL `gaussfit` routine that ignores NaN frames during the fit calculation. Note that the fit is performed for the entire subscan, and not just the source region. The resulting gaussian is finally subtracted from the subscan, and the code continues.

At this point we are ready to use `psd_scan.pro` to calculate the PSD for the flag-interpolated, source-removed subscan. The resulting PSD is then integrated over the frequency range defined by f_{min} and f_{max} by summing the product of Δf and the PSD over the corresponding frequency bins. The square root of this is written to *chan.relsens*, and the code proceeds to the next subscan.

Additional notes:

Often the IDL `curvefit` routine used in the gaussian-fitting part of the code returns a "Failed to Converge" error message. In my experience, this message can just be ignored, since it generally occurs only for those subscans in which the source flux is significantly less than the RMS of the leftover $1/f$ noise. In these cases no gaussian is subtracted, and the code proceeds with the PSD calculation as normal.

7.5.3. Computing the scan-averaged PSD

IDL Program Name:

`avgpsd_module.pro`

Description:

The `avgpsd` module is a cleaning module in name only: it is not a necessary component of the

cleaning process, and it is the only module that does not return any values to *chan*. Rather, it is intended as an important diagnostic tool. Its job is to calculate the average PSD (in Volts $Hz^{-1/2}$ units) for each bolometer on the array over many subscans. Once this has been accomplished, the default action of the code is just to stop running; it is up to the user to do with the resulting data what he or she wishes.

No "Program structure" subheading has been included below, since this module is almost exactly identical to `relsens_module_psd.pro`, described above. The differences are threefold: 1) the calculated PSDs are not integrated over frequency; 2) the PSDs of all subscans for a given bolometer are averaged together (frequency bin by frequency bin) into one array representing the average bolometer PSD for the entire chunk of data in *chan*; and 3) the bolometer data arrays are truncated to the same length (that of the shortest subscan), so that the frequency bins match up exactly during averaging. Note that all data to be included in this average must be present in *chan* at once. That is, if the PSD for an entire observation is desired, the *nscans_to_process* keyword in the cleaning wrapper must be set to a number greater than or equal to the total number of subscans in the observation. Otherwise, only PSD data from a portion of the observation will be included in the average.

A few words should be said about the average performed at the end of the code. Once PSDs have been calculated for all subscans in the current bolometer (all of which are saved into one 2-D array), a loop is started over frequency bins. For each bin, the code checks to see how many non-zero PSD values are present, and any zero values are overwritten as NaN. Then, if there are 2 or more "good" PSD values, they are averaged *in quadrature*, i.e. by taking the square root of the average of their squares. This is necessary because, as mentioned before, the $\sqrt{\text{PSDs}}$ can be thought of as noise values, and noise adds in quadrature. If there is only 1 "good" PSD value, it is simply taken to be the PSD value for that frequency bin. And if there are no "good" PSD values, that frequency bin is replaced by NaN.

Inputs:

`ac_bolos` — Bolometer timestream data from which PSDs will be calculated for each subscan in *chan*.

chanflags — From `bolo_params` file; specifies good bolometers.

`flags` — Data flags for each bolometer. The module needs these so that bright sources and spikes are not included in the calculation.

Outputs:

avgpsd — The subscan-averaged $\sqrt{\text{PSD}}$ for each good bolometer on the array. Note that this is not returned in *chan*; rather, the code stops (using the IDL `stop` command), and it is left up to the user to alter this code in such a way as to fit his or her needs. One possible option that is often used is to simply write the output array of PSDs to an IDL save set (using the `save` command), which can then be easily accessed (using `restore`) later on.

Required subroutines:

`read_flag.pro`

`psd_scan.pro`

`gaussfit_nan.pro` — This is a custom-modified version of the IDL `gaussfit` routine that recognizes NaN values in the data to be fit, and exclueds them from the fitting process.

7.6. Computing subscan RMS

IDL Program Name:

`rmsnoise_module.pro`

Description:

This module calculates the standard deviation, or RMS noise, of each subscan for each good bolometer channel in the data stream. These values are written to the *chan.rmsnoise* variable as well as the output netCDF file.

In addition to being a useful diagnostic in general, the RMS noise values are used by `debadscan_module.pro` (see above) to identify bad or poorly-cleaned subscans, and flag them so that they are ignored during map-making. For data taken during the May 2000 observing run, good subscans tend to have RMS noises between 0.001 and 0.1 V.

In some situations, it may also be desirable to use the calculated RMS values as weights when averaging subscans together to make maps; usually, however, these weights are given by the power spectral densities (PSDs) of each subscan. See Section 7.6 for more information.

Inputs:

`ac_bolos` — Bolometer timestream data.

`flags` — Data flags for each bolometer. The module needs these so that bright sources, spikes and otherwise bad data frames are not included in the calculation.

chanflags — From the `bolo_params` file; specifies which bolometers are "good," so that RMS values are calculated for those bolometer channels only.

Outputs:

The RMS for each bolometer and subscan, written to *chan.rmsnoise*, and subsequently to the netCDF variable `rmsnoise` created by the cleaning wrapper during first cleaning.

Required subroutines:

`set_flag.pro`

`read_flag.pro`

Associated cleaning modules:

`debadscan_module.pro` — Must be run *after* the RMS noise module.

Program structure:

This code is structured as an outer loop over the good bolometers specified by *chanflags*, and an inner loop over each subscan. Once inside the loop, the data and data flags for the current bolometer and subscan are read into variables. Separate variables are created for spike, source and baddata flags, and the IDL *where* statement is used to isolate only the frames that do not have any of these flags set. Frames that *do* have one or more of these flags set are filled with NaN values using the IDL system variable `!values.f_nan`.

Next the IDL *moment* function is invoked with the */nan* keyword to calculate the first four moments of the data in the good frames while ignoring the NaN values in the bad frames. Note that the first moment corresponds to the variance, or square of the standard deviation. Thus, the square root of the first moment is written to *chan.rmsnoise*, and the loop proceeds to the next subscan.

7.7. Optimal Filtering

Hopefully Alexey will fill in this section...

7.7.1. High-pass filter deconvolution

IDL Program Name:

`deconv_module.pro`

Description:

This code deconvolves the high-pass filter of the lock-in amplifiers from the bolometer data channels...

Inputs:

`ac_bolos` — Bolometer timestream data.

chanflags — From the `bolo_params` file; specifies which bolometers are "good," so that only those bolometer channels are processed.

`ac_bolos_buffer` — Bolometer timestream data corresponding to the region immediately before the data in `ac_bolos`. This variable is necessary because the first frame in `ac_bolos` is

always the first frame of the first subscan in the current data chunk. That is, in order to obtain buffer data for the first subscan of the current data chunk, we have to read in this data in addition to, and separately from, `ac_bolos`.

Outputs:

`ac_bolos` — High-pass filter-deconvolved bolometer timestream data, written to *chan.ac_bolos*, and then to the output netCDF file.

Required subroutines:

`fft_mkfreq.pro`

`mk_transfer.pro`

Associated cleaning modules:

None; however, see "Additional notes," below.

Program structure:

Additional notes:

This code produces noticeably better results when run *before* the correlated noise removal modules. This is most likely because, once a subscan has been cleaned, the buffer region (i.e. the timestream data immediately before the subscan) no longer shares the same spectral characterists as the noise-subtracted data. Furthermore, there will generally be a voltage jump between the subscan and buffer regions. Together, these result in weird artifacts being introduced into the FFTs, and therefore into the deconvolved time traces as well. Future versions of the deconvolution code should try to find a solution to this problem, so that deconvolution can be performed, along with true optimal filtering, at the end of the cleaning process.

8. Mapping Software

The third and final component of the Bolocam data reduction pipeline is the mapping software. This is where we take the cleaned time-domain data and translate it into a 2-D spatial map. Most of the preparatory calculations required for this task have already been performed: relative sensitivites and pixel offsets, for instance, were calculated during the cleaning process. Thus, it is now simply a matter of putting all the pieces together in a logical, and hopefully computationally expedient, manner.

In this section I will start out by describing the primary mapping code, `map_ncdf.pro`, which is capable of making either individual maps for each bolometer, or a combined map including data from all bolometers. I will then go on to address the derivation of bolometer responsivities, which is a necessary intermediary step between the making of individual bolometer maps and a final,

all-bolometers-combined map. The final subsection will address issues associated with performing 2-D convolutions of Bolocam maps.

Note that a general overview of the mapping software was given in Section 2.3. It is recommended that the reader consult that section first before continuing on with the detailed description given below.

8.1. Main program

IDL Program Name:

`map_ncdf.pro`

Description:

The majority of this code is dedicated to data organization. That is, in order to make a map, the 144 bolometer timestreams and all supporting information must first be put into a form in which they can most easily be combined and translated from the temporal domain to a spatial one. This form is predicated on a very simple idea: the order of the data, in terms of its time-sequenced aspect, is no longer important; what is important is that every frame of bolometer data has associated with it the correct RA and DEC upon which the beam was centered when it was taken. Once this is accomplished, the remainder is little more than a matter binning.

That said, `map_ncdf.pro` is divided into the following completely independent processes: 1) reading the data, 2) preparing the data, 3) binning the data, and 4) mapping the data. Each of these will be discussed below under the "Program structure" subheading.

The output of this routine, which is actually called as a function, is an IDL data structure containing much more than just a 2-D map (although it does contain this as well). The idea was to make the final data product as general and flexible as possible so that various anomalies (such as empty pixels) and nonstandard formats (such as different RA and DEC pixel resolutions) may be accomodated. Enough information is then provided so that the user may reconstruct the map to his or her liking.

The contents of the output structure are listed below, under "Outputs." Of these, the most fundamentally important are *.signal* and *.pixels* (where the period refers to the fact that they are variables within the structure, e.g. *output_structure.signal*). Both of these are simple 1-D arrays, with the values in *.signal* corresponding to the pixel numbers in *.pixels*. To clarify by example, the following IDL code fragment shows one way in which one might go about constructing a map from these arrays, starting with the mapping prgram output structure `output_struct`:

```
data_1D = output_struct.signal        ;1-D data array
rms_1D = output_struct.serror         ;1-D within-pixel rms array
```

```
goodpix = output_struct.pixels       ;data-containing pixel #s
numx = output_struct.nra             ;x-size of map, in pixels
numy = output_struct.ndec            ;y-size of map, in pixels

data_2D=fltarr(numx,numy)            ;create 2-D array for data map
rms_2D=fltarr(numx,numy)             ;create 2-D array for rms map

baddata=!values.f_nan                ;IDL system variable for NaN

for iy=0,numy-1 do begin                        ;loop over rows
   for ix=0,numx-1 do begin                     ;loop over columns
      index=iy*numx+ix                          ;get linear pixel # index
      good=(where(goodpix eq index,ngood))[0] ;does pixel contain data?
      if ngood ne 0 then begin                  ;if so:
         data_2D(ix,iy)=data_1D(good)              ;read data into 2-D map
         rms_2D(ix,iy)=rms_1D(good)                ;read rms into 2-D map
      endif else begin                          ;if not:
         data_2D(ix,iy)=baddata                    ;set data map pixel to NaN
         rms_2D(ix,iy)=baddata                     ;set rms map pixel to NaN
      endelse
   endfor
endfor                               ;close loops
```

Note that *deltara*, *deltadec*, *ramin* and *decmin* are only needed if one wishes to make corresponding RA and DEC arrays (for plotting, etc.).

The discussion so far has applied only to the making of all-bolometer coadded maps (by setting the */all* keyword), as opposed to individual bolometer maps (by setting the */bolos* keyword). In the latter case, the returned output structure contains a number of substructures, as many as there are good bolometers on the array. These substructures do not contain the generalized information described above; rather, in this case the 2-D maps themselves are the important data product.

Why is this? If you have read ahead at all, you might have noticed that one of the optional input variables to `map_ncdf.pro` is *resp_path*, which is a string variable specifying the path to a file containing the relative bolometer responsivities for all good bolometers on the array. Here, "responsivity" essentially means the normalized gain of the bolometer; a multiplicative coefficient by which each bolometer channel is scaled so that the peak flux of a source is the same when measured in all channels. The responsivities file is created by `calc_responsivities.pro` (described in its own subsection, below) which performs the calculation by comparing the amplitude of a bright point source (usually a calibration source, like a planet) as seen by each bolometer. The input to this code, therefore, is the structure of individual bolometer maps of this source returned by `map_ncdf.pro` with the */bolos* keyword set. This is the primary function of these maps (and this

keyword), and once the relative responsivities have been calculated for a given night of observing, they usually need not be calculated again.

Once the responsivities file has been created, a final, bolometer-coadded map can be made by setting both the */all* and *resp_path* keywords. The responsivities are actually used by the mapping code *twice*: once to scale each bolometer to the level of the detector with the maximum responsivity, and again to weight each bolometer by the inverse of its responsivity. The latter ensures that less-responsive detectors, which we can assume to have lower S/N ratios, will contribute proportionally less toward the final average comprising each pixel of the output map.

It is important to stress that, even after applying the relative bolometer responsivities, unless the user has already explicitly converted the data from Volts to flux units the output map will have arbitrary scaling. Final flux calibration is usually performed by simply multiplying the map by the known flux density of the calibration source (in Janskys) and dividing by the peak value of the mapped calibration source (in Volts). The result will be a peak-normalized calibration, therefore, as opposed to integral-normalized. In other words, the actual total flux from a point source will be equal to the *peak* of the mapped source, and not, as one might hope, to the integral over the beam area covering the source.

As a final general comment, note that no correction is made by the mapping code for the fact that the signal recorded by each detector has been diluted by the far-field beam pattern. That is, we construct the map as if the signal contained in each data frame originated from one discreet point on the sky corresponding the the center of the beam. Because of this, and because the DEC sampling of a single detector beam is limited by the subscan interval of the observation, individual bolometer maps will tend to have significant gaps in DEC sandwiched between well-sampled rows in RA. (The simultaneous coverage of all bolometers together is much better, of course, so these gaps are generally only seen at the very edges of bolometer-coadded maps.) If one then wishes to add the effects of beam dilution back into the map, this can be accomplised by performing a 2-D convolution of the map with the beam pattern. This calculation and its uses will be discussed in a separate subsection, below.

Calling sequence:

```
output_structure = map_ncdf(infile, startframe, nframes, [ra_range=rarange],$
 [dec_range=decrange], [resolution=res], [/all], [/bolos], [tau=tau],$
 [resp_path=resp_path])
```

Command line inputs:

infile — Input netCDF file containing cleaned Bolocam data.

startframe and *nframes* — The first frame number and total number of frames to be read in; together these specify the range of data to go into the final map.

Inputs taken from *infile*:

`ac_bolos` — Cleaned bolometer timestream data.

`flags` — Data flags for each bolometer, needed so that frames flagged as "spike," "badscan," and/or "baddata" are excluded from the map-making process.

chanflags — From `bolo_params` file; specifies good bolometers to be included in output map.

`relsens` — Relative sensitivities calculated for each bolometer and subscan by `relsens_module_psd.pro`.

`ra` and `dec` — 1-D time-sequenced arrays of RA and DEC positions of telescope boresight.

`pixel_offsets` — Arrays of RA and DEC bolometer offset positions for each bolometer and subscan; calculated by `pixel_offsets_module.pro`.

`scans_info` — Subscan information calculated by `scans.pro` (see above). Includes the variables *nscans*, *npstart*, *npend*, and *npoff*.

Outputs:

If the */all* keyword is set, the `map_ncdf` function returns a named IDL structure containing the following variables:

.map — 2-D array containing coadded voltages as a function of RA and DEC.

.mapweights — 2-D array containing within-pixel RMS values (in Volts) as a function of RA and DEC.

.npix — Number of pixels in output map containing data.

.pixels — A 1-D array of length *npix* containing the pixel numbers of the data-containing pixels in the output map.

.signal — A 1-D array of length *npix* containing the coadded voltages corresponding to the pixel numbers in *pixels*.

.serror — A 1-D array of length *npix* containing the within-pixel RMS values (in Volts) corresponding to the pixel numbers in *pixels*.

.npoints — A 1-D array of length *npix* containing the number of data points that went into each pixel in *pixels*.

.nra and *.ndec* — The number of pixels in the RA and DEC directions, respectively.

.deltara and *.deltadec* — The pixel resolution in RA and DEC, respectively, in arcseconds. These are generally equal.

.ramin and *.decmin* — The minimum RA and DEC of the output map. Together with *deltara*, *deltadec*, *nra* and *ndec*, these specify the spatial extent of the output map.

If the */bolos* keyword is set, the `map_ncdf` function returns a named IDL structure containing the following sub-structures, each of which is identical to the corresponding variables above, except replicated by the number of good bolometers:

.map

.mapweights

.npix

.nra and *.ndec*

.deltara and *.deltadec*

.ramin and *.decmin*

Keyword options:

ra_range and *dec_range* — These are two-element arrays containing the range of RA and DEC to be mapped. They should each have the format: [(min),(max)]. If these are not specified, the default is to map the full range of RA and DEC covered by the observation.

resolution — The pixel size, in arcseconds. A typical value of 5, for instance, will yield a map with square pixels 5 arcseconds on a side.

/all and */bolos* — These specify which of the two mapping procedures are to be performed: individual bolometer maps, or one combined map containing data from all bolometers. The code requires that one, and only one, of these keywords be set.

/tau — Set this keyword to perform an airmass correction based on the zenith optical depth[15] τ_z recorded to the netCDF `tau` variable. OR, if the `tau` variable was not recorded to the datastream during the observation in question, you may also choose to set this to an approximate value for τ_z to be used for the entire data set.

resp_path — A string variable containing the path to the responsivities file created by `calc_responsivities.p`
See relevant subsection below.

Required subroutines:

`readncdf_cal.pro`

`read_flag.pro`

Required files:

[15] Measured at 225 GHz by the CSO "tipper"; see Glossary entry.

`input netCDF file` — always required; specified by the *infile* input variable. See Inputs, above.

`responsivities file` — not required, but recommended; specified by the */resp_path* keyword. Note that the responsivities are derived from the individual bolometer maps, so the mapping code needs to be run at least once before */resp_path* can be specified.

Program structure:

Step 1: Reading in the data

Step 2: Preparing the data

Step 3: Binning the data

Step 4: Mapping the data

Versions:

`map_ncdf.pro` — The standard version described here.

`map_ncdf_exptime.pro` — A modified version which simply replaces the bolometer signals with 1s. The coadded map thus gives an indication of the relative exposure time across the observed field.

`map_ncdf_correlation_correction.pro` — This is identical to the normal version, except for the within-pixel RMS calculation. Instead of simply calculating the standard deviation of all data points that go into each pixel, this code calculates the standard deviation of all *subscans* that go *through* each pixel. Doing this corrects for the fact that, because of residual $1/f$ noise, data points from a given subscan will tend to be correlated over small (pixel-sized) regions of time and space. The correction is performed by first averaging together all data points within the same subscans, and then finding the weighted standard deviation of these averages.

8.2. Calculating bolometer responsivities

IDL Program Name:

`calc_responsivities.pro`

Description:

This program uses individual bolometer maps of a bright calibration point source to calculate the relative responsivities of each good bolometer on the array. The bolometer maps are contained in an IDL structure returned by `map_ncdf.pro` with the */bolos* keyword set. These responsivities are simply taken to be the peak voltages of the calibration source in each bolometer map, thus making them "relative" since they are arbitrarily normalized to the maximum bolometer responsivity.

Because of the typically poor DEC sampling of an individual bolometer[16], the peak source voltage in a map cannot be directly measured. Rather, it must be inferred by first fitting to the map a 2-D gaussian approximating the "true" beam shape. It is the peak of this gaussian that is actually taken to be the bolometer responsivity. Note that we are aided greatly in the task of measuring and fitting to the calibration source by the fact that the individual bolometer maps returned by `map_ncdf.pro` are already aligned with each other. That is, each map covers the exact same range in RA and DEC (only a fraction of which actually contains data from any given bolometer), so that the source always appears centered on the same pixel.

The calculated responsivities are written to an IDL save file (and, optionally, to a named output variable) which is then immediately readable by the mapping code. Thus, a responsivity-corrected bolometer-coadded map is made by re-running the mapping code with the */all* keyword toggled, and the *resp_path* keyword set to a string containing the path of the responsivities file. While it is probably enough to calculate the relative bolometer responsivities once or twice over the course of a night, this has yet to be confirmed with absolute certainty.

Calling sequence:

```
calc_responsivities, instruct, datafile, filesuffix, [halfcropsize = $
 halfcropsize], [responsivities], [inmaps], [fitmaps]
```

Command line inputs:

instruct — A structure output by `map_ncdf.pro` (with */bolos* keyword set) containing 2-D maps of all good bolos.

datafile — A string containing the name of the netCDF file from which the maps were created. (Note: exclude the ".nc" suffix!)

filesuffix — A string specifying an arbitrary suffix to be appended to the output filename, so that different output files (for different map resolutions, e.g.) can be told apart.

responsivities, *inmaps*, *fitmaps* — These are optional named variables that will contain, on output, the calculated responsivities, cropped input maps, and gaussian-fitted maps, respectively, for each bolometer.

Prompted inputs:

xcen and *ycen* — x and y pixel numbers of the source centroid. These do not have to be exact; within a couple of pixels is adequate.

xsmin/max and *ysmin/max* — max and min pixel values describing approximate x and y range

[16]Observations made for calibration purposes should generally have a subscan spacing of no more than 20 arcseconds at 1.4 and 2.1 mm, and somewhat less than this at 1.1 mm.

of source. Again, these don't have to be exact. They are used to mask out source pixels when looking for bad (high RMS) subscans.

Outputs:

This code produces an IDL save file called `[datafile]_resp.sav`, which contains the calculated relative responsivities of all bolometers on the array. (Bad bolometers are given responsivities of 0.)

Keyword options:

halfcropsize — Determines the extent of the cropped region around the source (± halfcropsize); if not stated explicitly on the command line, the default value is 20 pixels.

Required subroutines:

`readncdf_cal`

Required files:

datafile — The cleaned netCDF-format data file from which the individual bolometer maps were made.

Program structure:

Additional notes:

8.3. 2-D convolution

I probably won't get to this.

9. Statistical Analysis

I probably won't get to this either.

10. Quick Reference Guide

This section should contain hopefully everything you need to know about running the Bolocam software pipeline, with minimal attention given to why, or what specific steps are being performed. By this point it is assumed that Sections 1-4 of this manual have already been read, as much of the terminology discussed here will be unfamiliar otherwise. The reader is encouraged to look up individual codes in their respective chapters if further clarification is needed.

The subsections that follow will attempt to address the basic use of the Bolocam software

pipeline in a logical progression. The first of these will cover the necessary codes, supporting files, and the general organization and setup required before the software pipeline can be used. It is divided into separate sections corresponding to the different parts of the software pipeline, since they each stand alone, more or less, from each other. Then, in the "Working with the data" subsection, I will discuss the organization and manipulation of the netCDF data files, concentrating on the cleaning and mapping software in terms of calling sequences, options, etc. (At this point it is assumed that the data is already in merged form.) Finally, in the "Proper ordering..." subsection, a number of specific, common cleaning scenarios will be presented and described, with particular attention given to which cleaning modules need to be run, and in what order.

10.1. Codes, supporting files, and software organization

Here I will list all executables, IDL programs, and supporting text files needed by the Bolocam software pipeline, and in some cases will recommend a specific directory structure for their organization. Various tips for the proper setup of the pipeline will be interspersed throughout.

10.1.1. IDL, the IDL astrolib, and netCDF

It is assumed that all users of the Bolocam software pipeline have the Interactive Data Language (IDL) Version 5.4 or later installed on their computers. Earlier versions may work for some, or even most of the software, but this cannot be guaranteed. If IDL 5.4 is unavailable to you, it should be possible, and relatively straightforward, to alter most of the cleaning and mapping codes in such a way as to be compatible with older IDL releases. Also note that, as of 12/01/01, none of the Bolocam software pipeline has been tested with any versions of C or IDL on non-UNIX/Linux operating systems.

Some parts of the software pipeline use codes found in the IDL Astronomy User's Library, or "astrolib," which is available for free download at: http://idlastro.gsfc.nasa.gov/homepage.html. Directions for proper installation can also be found there. For what it's worth, the astrolib code most commonly used by the Bolocam software is `readcol.pro`, which is a very useful tool for reading in column-formatted text files.

It is also assumed that the user has the netCDF software libraries installed on his or her computer. These libraries and directions for their installation can be freely obtained at http://www.unidata.ucar.edu/pa
Consult this URL, as well as Section 4 of this manual, for more information about netCDF.

10.1.2. Files required for merging

Merging requires the following executables to be in the user's path:

blah blah blah

For users who may want to alter the merging code, the C codes from which one may compile the executables listed above are:

blah blah blah

10.1.3. Files required for cleaning

The code:

First I will list the IDL program files, by which I mean all files ending in a ".pro" extension. The IDL codes required by the cleaning software as of 12/01/01 are:

`avgpsd_module.pro`	`gethexnoise_module_diff.pro`	`rmsnoise_module.pro`
`clean_ncdf.pro`	`hexsub_module_all.pro`	`scans.pro`
`debadscan_module.pro`	`mk_transfer.pro`	`set_flag.pro`
`deconv_module.pro`	`pixel_offsets_module.pro`	`skyhexsub_module_all.pro`
`desource_module_3c273.pro`	`polysub_module_2.pro`	`skyhexsub_module_hex.pro`
`desource_module_dr21.pro`	`polysub_module_3.pro`	`skyhexsub_module_locgauss.pro`
`desource_module_uranus.pro`	`polysub_module_4.pro`	`skyhexsub_module_loclin.pro`
`despike_module.pro`	`psd_scan.pro`	`skysub_module_all.pro`
`fft_mkfreq.pro`	`putchan.pro`	`skysub_module_hex.pro`
`gaussfit_nan.pro`	`read_flag.pro`	`update_history.pro`
`getchan.pro`	`readncdf_cal.pro`	`writencdf_cal.pro`
`gethexnoise_module_bias.pro`	`relsens_module_psd.pro`	

The suffixes on the `desource` modules are relics of the May 2000 observing run, during which object RA and DEC positions were not explicitly written to the data stream. In future runs we hope to correct this problem, so eventually there may very well be just one `desource_module.pro`. Note that the only differences between the versions listed above are the values of the *sourcera* and *sourcedec* variables in each, which are easily changed by the user.

It is recommended that these codes be kept together in a single, separate directory, which must be appended to the user's `IDL_PATH` system variable. Note that if codes are added or altered such that anything more than their filename suffixes are changed, the relevant sections of the `getchan.pro` and `putchan.pro` codes need to be altered accordingly. See Section 6 for more detail on these codes.

The supporting files:

There are only three supporting text files required by the cleaning wrapper. These are:

`bolo_params.txt`
`array_params.txt`
`module_file.txt`,

all of which are described in the Overview and Glossary sections, as well as in Section 6. Note that none of these files have specific naming conventions, and it may be useful – or perhaps necessary – to have multiple versions of each with different names. It is recommended that these files be kept together in a separate directory, in `/home/username/cleaning_inputs/` for example.

The module file is a required input whenever the cleaning wrapper is called, although one may set a default module file path in the `clean_ncdf.pro` code if the same cleaning modules are typically going to be run every time. In general, one may also find it useful to have multiple copies of the module file corresponding to various common cleaning situations. The proper ordering of the cleaning modules within the module file for a few of these situations will be discussed in the "Proper ordering" section below.

In contrast to the module file, the bolometer and array parameters files are required inputs to the cleaning wrapper only during the first cleaning of a particular netCDF-format data file. Thereafter they are optional inputs, and including them in the `clean_ncdf.pro` calling sequence will overwrite the previous bolometer and array parameters already in the data with those from the newly-specified file. Since these files are a crucial element in the software pipeline, I will now give a more thorough account of how they are constructed.

Creating bolometer and array parameters files

Although in theory the bolometer and array parameters should not change throughout the course of an observing run, in practice one will find this not to be the case, particularly for the former. More specifically, a new bolometer parameters file will be necessary whenever the status of one or more detectors on the array has changed (which unfortunately happens more often than we would like). Similarly for the array parameters: if the fiducial rotation angle of the array changes, for instance, the `array_params` file will have to be updated appropriately for the cleaning of any data taken thereafter.

So, how does one create these files in the first place? We'll start with the `array_params` file, since that's the simpler of the two. The entire contents of this file for the May 2000 observing run are:

```
platescale_("/mm)          7.40284
beamsize_(",FWHM)          39.0000
fid_arr_ang(deg)           96.
x_bore_off(inches)         0.
y_bore_off(inches)         0.
```

First, there are a few general format requirements. The first column can be anything, as long

as there are no spaces - this column is not read in by the cleaning wrapper at all; it is only concerned with the values in the second column. The gap between columns can be tab- or space-delimited. And finally, it is important that each line be ended with a carriage return, or else the astrolib `readcol` routine will not recognize it.

Now for the actual values, beginning at the top. The platescale is derived empirically, by observing and mapping a point source and comparing the on-sky separations between the source centroids in each bolometer map. The physical size and separations of the bolometers are known (adjacent detectors have a center-to-center separation of 5 mm), so all we need to know is the corresponding sky-projected separation in arcseconds. Note that, when cleaning the data for this calculation the `pixel_offsets` and `desource` modules should *not* be run; that way the RA and DEC values of the sources in the individual bolometer maps will be offset from their correct values by the amount that we want to measure. Alternatively, one might try running the `pixel_offsets` module with the value of *platescale* in the `array_params` file set to 0.0 – this should cause the calculated pixel offsets to be all 0s, which is what we want.

While this all may seem complicated, the good news is that the platescale should not change from observing run to observing run (the geometrical optics are identical) so in most cases this calculation need not be repeated. Still, for the record, the `platescale` value of 7.40284 arcsec/mm quoted here was determined by averaging together the platescales derived from the above method, using the radial separations of each beam from the central bolometer (`BOLO_E0`) beam.

The beam FWHM recorded to `beamsize` is, like `platescale`, derived empirically, from maps of point sources. The best way to do this is to run cleaning and mapping as usual, but temporarily assuming a beam FWHM given by the diffraction limit of the telescope for the bandpass in question. This will have minimal effect on the source flagging and deconvolution modules (the only codes that use this parameter), producing a resulting map that should be more than adequate for this measurement. Note that the observation used for this should have as small a subscan spacing as possible, so that sampling effects in the cross-scan direction are minimized. The final value should be taken from an bolometer-coadded map, and measurements in both RA and DEC directions should be averaged together.

The `x_bore_off` and `y_bore_off` variables are the offsets, in inches at the focal plane, between the telescope boresight and central bolometer. Here, "x" refers to the horizontal, and "y" the direction perpendicular to this in the plane of the array. As of this writing no methods have yet been formulated for robustly calculating these offsets, so they are simply assumed to be 0. They were left in the data stream mostly as fudge factors, for use in situations where the derived pixel offsets are incorrect for unknown reasons. In theory, they could be derived through some kind of amoeba-like fitting of the parameter space, but this has yet to be done. Also keep in mind that these parameters do *not* take into account any offset between the center of the array and the rotation axis of the rotator – this is currently assumed to be identically zero in all cases.

The value of the fiducial array angle is a function of the array geometry when the rotator angle

is 0. In addition, however, it is degenerate with the fiducial point from which the bolometer angles are measured, which makes it a convenient segue into the discussion of the `bolo_params` file. To make things easy, we will assume that the bolometer angles will always be measured identically to the way they are measured now: clockwise from the line separating the A and F hextants when looking *through the array from the back.* (Note that this convention produces bolometer angles that increase in reverse alphabetical order in terms of the hextant names.) The `fid_arr_ang` variable is then just the angle between this same A-F hextant line and the horizontal, measured clockwise from the left when looking through the array from the back. For example, if the rotator was set so that the A-F hextant line pointed straight up at a `rotangle` of 0, then the fiducial array angle would be exactly 90 degrees.

While a good guess of the fiducial array angle can be made by simply understanding the array geometry, its true value can only be derived by, again, looking at individual bolometer maps of a point source made with the pixel offsets set to 0. Unfortunately, however, there is as yet no code for easily backing out this parameter from these maps. Suffice it to say that such a code, once written, will be essentially the inverse of `pixel_offsets_module.pro`. It will take as input the measured pixel offsets, RA, DEC, PA, etc, and return the fiducial array angle.

We now move on to the `bolo_params` file. The first few lines of this file were already reproduced in the first portion of Section 6, to which the reader is now referred. The complete file has 144 lines, one for each bolometer, with 4 columns of information for each. Like with the `array_params` file, columns can be tab- or space-delimited, and each row should end in a carriage return.

The four columns correspond to the bolometer names, flags, angles, and distances, respectively. However, the first, third, and fourth of these need not ever be re-computed at all; the bolometer names and radial distances don't change, and we have already assumed a constant fiducial position from which the bolometer angles are always to be measured. Thus all we are really interested in knowing is how to calculate the bolometer flags.

As was explained in the Glossary section and elsewhere, the bolometer – or "channel" – flags (which are not to be confused with the data flags contained in the `flags` netCDF variable) simply describe the status of each bolometer. Zero corresponds to a bad bolometer, positive integers to "good" bolometers, and negative integers to "open" bolometers. Higher absolute values of these integers indicate poorer quality; in practice, the cleaning software only recognizes as truly "good" those channels having bolometer flags equal to 1.

At present, the assignment of bolometer flags is made completely subjectively: one simply has to endure the painstaking process of looking at each raw bolometer channel[17] separately to determine which are bad, good, and open. Fortunately, once this process has been started it begins to become obvious which channels are which:

---

[17]The channels in the *ac_bolos* variable should be used for this analysis, and not those in *dc_bolos*.

Bad bolometers typically have very small voltages; so much so that, when plotted against frame number, the limited precision of their long integer data type is quite apparent.

Good bolometers are similarly distinguishable by the fact that they are utterly dominated by sky noise. Because sky noise is common mode, that is, seen identically across the array, good bolometer channels plotted against time are going to look very similar in terms of the shape of the characteristic $1/f$ "drift." And finally...

Open bolometers are identifiable simply by virtue of being neither good nor bad. These typically have voltages intermediate between those of the good and bad bolometers, without nearly as much variation in amplitude as what is seen in the former. Furthermore, open bolometers will, like good bolometers, exhibit $1/f$ drifts which are correlated to one another.

While this categorization can be performed using the IDL `plot` routine, another option is to use `stripchartnc`, a command-line executable whose sole purpose is to plot netCDF Bolocam variables against frame number. Although it is not considered to be a part of the analysis pipeline proper, it is a very useful diagnostic tool, and so Appendix A has been included at the end of this manual to explain its use.

10.1.4. Files required for mapping

With the exception of the cleaned netCDF-format data files, there is only one supplemental file used by the mapping software. This is the responsivities file, which is an IDL save file returned by the `responsivities_calc.pro` routine. Typically this file will be in the same directory as the cleaned data files, although this is not required since its path is specified explicitly in the `map_ncdf.pro` calling sequence.

The responsivities file, as its name indicates, contains the responsivity of each good bolometer on the array. This value is used by the mapping routine as both a scaling and a weight. While it is not a necessary input for the creation of a bolometer-coadded map, it is certainly highly recommended for the making of a "final" map. Responsivities will generally need to be calculated once or twice for a given night of data, and require a relatively well-sampled (i.e. small steps in DEC) observation of a bright point source. The standard calibration sources – usually planets – work ideally for this calculation.

Specifics about the creation and use of the responsivities file will be discussed in the following subsection. See also Section 8.2 for more information.

10.2. Working with the data

Once all the required packages have been installed, the software codes and supporting files organized appropriately, and the data files merged into netCDF format, the cleaning and mapping process can begin. Here I will describe how this is done: what codes need to be run, what files need to be manipulated, and the best ways to go about it.

Cleaning

First off, you'll need to start IDL. Typically one will want to run IDL in the same directory where the data files are located, but this is a matter of personal preference. And while you may, if you like, start immediately typing "`clean_ncdf...` at the IDL prompt, it is advised that you consider using an IDL batch file instead.

A batch file is similar to a normal IDL program file in that it ends with a ".pro" extension and contains a string of IDL commands. However, it does not begin with "`pro`" or end with "`end`," and it is not compiled like a program; rather, it is executed at the `MAIN` command level, as if from the command line. Batch files are a useful way to run commonly-executed command line processes that would have been tedious to write out again and again. In the case of `clean_ncdf.pro`, which has a calling sequence that can easily take up several lines, it is almost a necessity.

The first thing to do is to create a batch file – call it `anything.pro` – in a directory that is specified within the `IDL_PATH` system variable. Assuming, for the time being, that the data we are to clean has not yet been through the cleaning wrapper, we type within our batch file the following:

```
clean_ncdf,mod_file='/home/username/cleaning_input/module_file.txt',$
infile='/home/username/merged_data/000000_000_raw.nc',outdir=$
'/home/username/cleaned_data/',nscans_to_process=5,array_params_file=$
'/home/username/cleaning_input/array_params.txt',bolo_params_file=$
'/home/username/cleaning_input/bolo_params.txt',/noprompt
```

altering the paths as appropriate. A value of between 5 and 10 for the *nscans_to_process* variable is standard for most cleaning tasks. No keywords are usually needed on first cleaning of a file, although setting */noprompt* is often a good idea, as it will cause the program to stop if you have left out any required inputs. In particular, the paths to the array and bolometer parameter files are always required for the first cleaning of a file. The module file indicated by the *mod_file* variable is assumed at this point to contain the cleaning modules you wish to run, in the appropriate order. The question of which modules to run when, and in what order, will be discussed throroughly in the next section; as for now, we are concerned only with the general case.

Make sure the file `anything.pro` is saved, and type "`@anything`" at the IDL command prompt. (The "`@`" character tells IDL that it is running a batch file – it is somewhat analogous to the "`.run`" command.) The cleaning process is now running. Printed to the screen will be something like the following:

```
IDL> @scratch
cp 000000_000_raw.nc /home/username/cleaned_data/000000_000_clean1.nc
NOTE: First cleaning of file 000000_000_raw.nc!
Writing array parameters to /home/username/cleaned_data/000000_000_clean1.nc...
Writing bolometer parameters to /home/username/cleaned_data/000000_000_clean1.nc...
Processing scans 1 through 5 of 31...
           frames 477 through 15085...
Running module first_module
Running module second_module
Running module third_module
Running module fourth_module
Running module fifth_module
Processing scans 6 through 10 of 31...
           frames 16044 through 30701...
Running module first_module
Running module second_module
[etc...]
```

where "`first_module`, `second_module`," etc. will be replaced by the names of the cleaning modules listed in the module file.

The first four lines will always be seen during the first cleaning of a data file. You will notice that one of the first things the cleaning wrapper does is to copy the file to be cleaned to a separate output file in the specified *outdir*. This way the original data file is never altered in any way by the cleaning process. The two "Writing..." lines correspond to the writing of the bolometer and array parameters file into the new output file, where they will be a part of the data stream from then on.

On subsequent cleanings of an already-cleaned file, the default action of the cleaning wrapper is to overwrite the old file with any new data calculated by the cleaning modules. Setting the */newfile* keyword will override this default, copying the input file to a new output file (with extension "`clean2`," "`clean3`," etc.) before writing. There is also a */nooutput* keyword, which will inhibit writing of new data to the output file altogether. Note that this pertains only to new data derived from the cleaning modules, and *not* to the contents of the array and bolometer parameters files, which should be thought of separately. The parameters in these files will be written to the data stream whenever the *array_params_file* or *bolo_params_file* keywords, respectively, are set. Doing so for an already-cleaned file (which is purely optional, and rarely necessary) will overwrite whatever parameters were there already with those in the newly-specified file.

If you realize after starting the procedure that you have made a mistake – by calling the wrong modules, for example – you always have the option of interrupting the cleaning process by typing `cntl-c`. Note, however, that it is best not to stop the process while it is reading from or writing to a netCDF file. Doing so will leave the file open, and IDL will become confused if it is instructed

to open the same file again. However, since no netCDF files are open during the actual processing of the data, this problem can be avoided by simply waiting until the "Running module..." message appears before interrupting.

This is essentially all there is to running the cleaning process; the rest is simply a matter of picking what modules to run, when, and in what order. In general, as long as all the required IDL codes are present in the `IDL_PATH` system variable, and all supporting files are accounted for, then all that one needs to do is to make sure that the module file and IDL batch file are correct, and that any changes to them have been recently saved.

Often it is advisable to check that the cleaning of a data file was successful by plotting one or more bolometer channels against time (or, identically, frame number). The cleaned data should be almost flat for all regions where the tracking signals (`trck_tel` and `trck_das`) are "low," i.e. 0, and dominated by large $1/f$ drifts when the tracking signals are "high." While this check may be done by reading the data into IDL manually and using the `plot` routine, a simpler and more straightforward method is to use the `stripchartnc` executable; see Appendix A for more information on this useful tool. For completeness' sake, however, I will include the following code fragment as an example of how one might use IDL to check that `000000_000_clean1.nc` from the example above was properly cleaned. These commands may be run at the IDL prompt, within a batch file, or as part of a program:

```
;open netCDF file:
   infile='/home/username/cleaned_data/000000_000_clean1.nc'
   fileid=ncdf_open(infile)

;read in bolometer names to a temporary array, then write them to a
;properly-formatted string array:
   ncdf_varget,fileid,'bolo',temp
   bolonames=strarr(n_elements(temp(0,*)))
   for i=0,n_elements(temp(0,*))-1 do $
      bolonames(i)=strtrim(string(temp(*,i)),2)

;designate data range of interest:
   startframe=0
   nframes=20000

;read in 'ac_bolos' variable, starting at bolometer and frame 0,
;creating an array with 144 bolometers x nframes frames:
   bolos=readncdf_cal(fileid,'ac_bolos',[0,startframe],[144,nframes])

;read in corresponding tracking signal:
   track=readncdf_cal(fileid,'trck_tel',[startframe],[nframes])
```

```
;plot an arbitrary segment of the timestream data of bolometer C16 (e.g.):
   c16=(where(bolonames eq 'bolo_c16'))[0]
   plot,bolos(c16,*)
   oplot,track(*)

;close netCDF file:
   ncdf_close,fileid
```

The commands used above can easily be altered to read in and plot any variable from a netCDF file. As long as the `readncdf_cal.pro` routine is used to read in the Bolocam data, the units on the returned arrays will be correct (e.g. Voltages, in the case of `ac_bolos`). This does not apply for string arrays, however, which is why `bolo` was read in using the `ncdf_varget` routine.

Finally, if one wishes to overplot the `flags` variable to see what frames have been flagged as sources or spikes, for example, the `read_flag.pro` routine must be invoked. This can be done by inserting the following at the end of the code fragment given above (but before the `ncdf_close` command):

```
;read in 'flags' variable, starting at bolometer and frame 0,
;creating an array with 144 bolometers x nframes frames:
   flags=readncdf_cal(fileid,'flags',[0,startframe],[144,nframes])

;convert to separate arrays for spikes and sources, and normalize:
   spikes=read_flag(flags,/spike)
   sources=read_flag(flags,/source)

   spikes(where(spikes gt 0))=1
   sources(where(sources gt 0))=1

;overplot:
   oplot,spikes(c16,*)
   oplot,sources(c16,*)
```

Mapping

The mapping of cleaned data is simpler in execution than cleaning. There are only a few IDL routines to be run, and the results are easier to evaluate.

We will start by going over the use of `map_ncdf.pro` to create individual bolometer maps, before proceeding to the making of bolometer-coadded maps. This is a logical order because the individual maps are required for the derivation of important preliminary information, including the platescale, beam size, and (once these are known) the bolometer responsivities.

Say the data within our newly-cleaned `000000_000_clean1.nc` file is of a bright calibration source, such as a planet. We will assume that the platescale and beam size have already been calculated by this point, and that now we wish to create aligned individual bolometer maps for the calculation of relative responsivities. At the IDL prompt, type:

```
IDL> mapstruct = map_ncdf('/home/username/cleaned_data/000000_000_clean1.nc',0,$
IDL> 100000,resolution=5.,/bolos,/tau)
```

replacing "`mapstruct`" with anything you like. Here, 0 and 100000 are the starting frame number and total number of frames, respectively. Thus, the first 100000 frames of data in the file will go into the making of our map. Note that these values do not have to correspond to the exact beginning and end of the observation, since all frames with the tracking signal "high" will be ignored anyway. As long as the beginning and end frames are not in the middle of a subscan (tracking signal "low") there shouldn't be any problems.

The *resolution* variable simply tells the code how finely to bin the data; here we have designated the pixels to be 5 arcseconds on a side. The */bolos* keyword tells the code to make individual bolometer maps, and the */tau* keyword makes sure that atmospheric extinction is taken into account. (If `tau` was not recorded to the data stream during the observation, you may also set this keyword to an approximate value of the zenith optical depth to be used for the entire observation.) If desired, it is also possible to give a specific range of RA and DEC to be mapped using the */ra_range* and */dec_range* variables. Excluding these from the calling sequence instructs the code to map the full range of RA and DEC in the data stream.

Upon execution of this command, something similar to the following will be printed to the screen:

```
% Compiled module: MAP_NCDF.
Reading data...
Preparing data...
Beginning mapping...
   Number of x pixels = 214
   Number of y pixels = 210
Now mapping Bolo #0...
Number of pixels with data = 3991
Now mapping Bolo #1...
Number of pixels with data = 3965
Now mapping Bolo #2...
Number of pixels with data = 4102
Now mapping Bolo #3...
Number of pixels with data = 4029
Now mapping Bolo #8...
```

```
Number of pixels with data = 4102
Now mapping Bolo #11...
[etc...]
```

Here we are told that the output 2-D map arrays will be 214 × 210 in size, along with information about each individual bolometer map. The gaps in bolometer numbers are a result of the fact that the code only makes maps for *good* bolometers, i.e. those with a bolometer flag of exactly 1.

The output structure contains a number of substructures, one for each good bolometer map. So, for example, typing "`help,mapstruct`" at the IDL prompt will return:

```
MAPSTRUCT  STRUCT    = -> <Anonymous> Array[62]
```

for data having 62 good bolometers. Individual maps can then be pulled out into separate 2-D variables by using commands like:

```
IDL> bolo_a1_map=mapstruct(0).map
IDL> bolo_a2_map=mapstruct(1).map
IDL> [etc...]
```

Similarly, 2-D arrays containing the corresponding within-pixel RMS errors can be extracted by replacing "`.map`," above, with "`.mapweights`."

For displaying maps, the standard IDL routines `tv` and `tvscl` can be used, but I personally recommend `atv.pro`, which was written by Aaron Barth at Caltech, and can be obtained at: http://www.astro.caltech.edu/ barth/atv/. (Note: don't forget to download both `atv.pro` and `cmps_form.pro`.) This program has many features for displaying and working with image data, including zoom, center, blink, statistics, photometry, etc., and in many ways is very similar to the SAOImage window commonly used in IRAF.

Now we are ready to calculate responsivities using `calc_responsivities.pro`. To do this, we first need a few pieces of information about the maps, which we can easily obtain by simply looking at them. One of these is the location of the source centroid in x and y pixel numbers. Assuming the pixel offsets were properly calculated during mapping, the source will be located at the same place in every map (usually in the center), so any individual bolometer map may be used for this. We also need to know the approximate x and y extent of the source, in terms of minimum and maximum pixel numbers, in both RA and DEC. Note that all of these values need only be accurate to within a couple of pixels, as they are only used by the code as initial rough estimates.

To begin the responsivities calculation for the same data file we've been using all along, type the following:

```
IDL> calc_responsivities, mapstruct,$
IDL> '/home/username/cleaned_data/000000_000_clean1','_res5'
```

The three inputs are 1) the name of the structure of maps just returned by `map_ncdf.pro` (with the */bolos* keyword set), 2) the name of the data file from which the maps were made, and 3) a suffix to append to the output responsivities file to keep it separate in case new versions are ever made. This is mostly useful when making maps of the same observation at different resolutions, which is why a suffix of "`_res5`" was used here. In other words, because the peak values of the sources in each bolometer map will be slightly different depending on the map resolution, it is advisable to make different responsivities files for each desired mapping resolution.

There are four more optional inputs to `calc_responsivities.pro`, which are generally left out of the calling sequence. The first of these is the *halfcropsize* keyword, which should be set to the half-width, in pixels, of a box to which each bolometer map will be cropped. The default value is 20 pixels (yielding a 40 × 40-pixel cropped region), which is about right for 1.4 mm maps with 5-arcsecond resolution. (A smaller number should be used for lower map resolutions and larger wavelengths.) The other optional inputs are *responsivities*, *inmaps* and *fitmaps*, all of which are named variables that will contain, on output, the calculated responsivities, cropped input maps, and fitted cropped input maps, respectively. These are intended as diagnostic aids only.

The program will now prompt the user for the source information discussed above:

```
Please give the x pixel # of the source centroid: xcen
Please give the y pixel # of the source centroid: ycen
Please give the x range ("min,max") of the source: xmin,xmax
Please give the y range ("min,max") of the source: ymin,ymax
```

Here, *xcen*, *ycen*, *xmin*, etc., should be replaced by the user with the values obtained from inspecting one (or more) of the bolometer maps. Note that these must be actual numbers, and not variables.

After the code is finished running, the responsivities file `000000_000_clean1_res5_resp.sav` will have been created in the same directory as the cleaned data file. This file contains one variable, *responsivities*, which may be pulled into IDL using the `restore` command. This is a 144-element array containing the relative responsivities for every bolometer on the array, with bad bolometers having been given responsivities of 0.

We now have everything we need to make a final, bolometer-coadded map of `000000_000_clean1.nc`, as well as maps of all data taken around the same time. At the IDL prompt, type:

```
IDL> finalmap = map_ncdf('/home/username/cleaned_data/000000_000_clean1.nc',0,$
IDL> 100000,resolution=5.,/all,/tau,$
IDL> resp_path='/home/username/cleaned_data/000000_000_clean1_res5_resp.sav')
```

The output structure `finalmap` will contain a number of variables, the two most important of which are the 2-D arrays containing the final coadded map and corresponding within-pixel RMS noise. These can be extracted from the structure by typing, for example:

```
IDL> data_map=finalmap.map
IDL> rms_map=finalmap.mapweights
```

Note that these "final" maps are still uncalibrated, that is, they have units of Volts instead of Janskys. The conversion is just a simple scaling, however, provided you know the flux of your calibration source. Some of the issues associated with flux calibration were discussed in Section 8.1, to which the reader is referred for further information.

And with that, we have reached the end of the Bolocam software pipeline proper. From this point on, it is up to the user how best to manipulate and process the data in the form output by the mapping routine.

Before we backtrack a bit to discuss the ordering of cleaning modules in various situations, it is useful to mention briefly the making of labelled plots from the coadded maps we have just produced. The code `plotnicemap.pro` has been included in the Bolocam software distribution for just this purpose. It provides one method for plotting maps in IDL with correct RA and DEC axes, a task which is much easier said than done. The plot in Fig. 2 was created with this routine. It will not be discussed any here any further, so check the comments within the code itself for information on its use.

10.3. Proper ordering of cleaning modules

General ordering guidelines

- Calculation of pixel offsets, source flagging (if desired) and spike flagging should always be performed first.
- Calculation of pixel offsets must be performed before source flagging.
- Deconvolution should be performed *before* correlated noise removal. (This only applies to the use of `deconv_module.pro`, and not to the optimal filtering module which is yet to be written.)
- Calculation of relative sensitivities, subscan RMS noise, and the flagging of bad subscans should always be performed last.
- The RMS noise calculation must be performed before the `debadscan` module is run.
- If any of the `skyhexsub` modules are run, a `gethexnoise` module must always be run before it. This does not apply for the `skysub` modules, but it does apply for the `hexsub` module.

First cleaning; noise removal method #1

Contents of module file:

```
pixel_offsets_module
desource_module
despike_module
deconv_module
gethexnoise_module_bias
skyhexsub_module_hex
polysub_module_3
relsens_module_psd
rmsnoise_module
debadscan_module
```

Notes:

- This is the noise removal method described by Equation (5).
- The calculation and removal of the bias noise has very little effect on the data, so it may be preferred to leave out the `gethexnoise_module` altogether. If this is done, `skyhexsub_module_hex` must also be changed to `skysub_module_hex`.
- The `desource_module` is optional, and should only be included if there is a known bright source in the observation to be cleaned. In addition, for data taken during the May 2000 observing run, the source RA and DEC were not written to the data stream. For this data, therefore, it is necessary to use a different `desource_module` for each bright source to be flagged. That is, there will be a `desource_module_uranus.pro` and `desource_module_3c273.pro` for flagging Uranus and 3C273, respectively. The only difference between these versions are the different values of the *sourcera* and *sourcedec* variables at the beginning of the code.

First cleaning; noise removal method #2

Contents of module file:

```
pixel_offsets_module
desource_module
despike_module
deconv_module
gethexnoise_module_diff
hexsub_module_all
skysub_module_all
polysub_module_3
relsens_module_psd
rmsnoise_module
debadscan_module
```

Notes:

- This is the noise removal method described by Equations (6) and (7).
- The `desource_module` comments above apply here as well.

Calculating subscan-averaged PSDs of already-cleaned data

Contents of module file:

```
avgpsd_module
```

Notes:

- Usually one wants to average together the PSDs of every subscan in an observation. To do so, simply set the *nscans_to_process* keyword equal to or greater than the number of subscans in the observation. Depending on the size of the observation and the amount of RAM installed on your computer, reading in this many subscans at a time may cause a memory allocation error.
- If desired, this module may be run during the first cleaning of an observation as well, by simply appending it to the end of one of the lists of module files given above.

11. Troubleshooting Guide

I get an IDL error saying that a certain variable is not in chan *when I know that it should be.*

If you change a module name, you must remember to update getchan and putchan to reflect these changes. Otherwise, when `getchan.pro` goes to read in the variables necessary for the modules listed in the module file, it will not recognize the new module name, and so won't know to read in the variables it requires.

IDL doesn't seem to recognize a particular code.

Make sure your `IDL_PATH` system variable has been altered to include the path to the directory in which the Bolocam software pipeline codes are kept.

After a cleaning process runs, I get all kinds of errors from IDL. What gives?

Absolutely nothing; this is normal. The following errors occur commonly when running data through the cleaning software, and can almost always be safely ignored:

```
% Program caused arithmetic error: Floating divide by 0
% Program caused arithmetic error: Floating underflow
```

```
% Program caused arithmetic error: Floating overflow
% Program caused arithmetic error: Floating illegal operand
```

In addition to these, the errors:

```
% CURVEFIT: Failed to converge- CHISQ increasing without bound.
% CURVEFIT: Failed to converge after 20 iterations.
```

are often returned by `avgpsd_module.pro` and `relsens_module_psd.pro`. This occurs during the processing of subscans for which a proper gaussian fit to a flagged source can not be made, and is common. Usually it just means that the source in that particular subscan is very faint, or the data is corrupted in some way (by an overabundance of spike flags, for example). The result is that no gaussian is subtracted from the data, which is just as well, and the PSD calculation continues normally.

12. Acknowledgements

I would like to acknowledge the efforts of the 2000/01 Bolocam software team, without whom this manual would be much, much shorter. In particular:

Samantha Edgington wrote all of the merging software, the despiking module, and an early version of the cleaning wrapper upon which the current wrapper is based.

Alexey Goldin wrote the deconvolution module, converted `stripchart` to a netCDF-compatible form, and provided much general computer and software support.

Phil Mauskopf provided the basic ideas, outlines, and early versions of virtually every code in the software pipeline, including its overall structure and organization. By which I really mean: it's all his fault.

Additional ideas, suggestions and a great many hours were generously contributed by Jason Glenn, Sunil Golwala, and Douglas Haig.

This manual is dedicated to Glenn Laurent, since he's probably the only person who will read it in its entirety.

REFERENCES

Blain, A. W., Ivison, R. J. and Smail, I. 1998, MNRAS, 296, L29

Glenn, J., Bock, J. J., Chattopadhyay, G., Edgington, S. F., Lange, A. E., Zmuidzinas, J., Mauskopf, P. D., Rownd, B., Yuen, L. and Ade, P. A. R. 1998, in Phillips, T. G., ed., Proc. SPIE Vol. 3357, Advanced Technology MMW, Radio and Terahertz telescopes. SPIE, Bellingham, p. 326

Griffin, M. J., & Orton, G. S. 1993, Icarus, 105, 537

Holzapfel, W. L., Wilbanks, T. M., Ade, P. A. R., Church, S. E., Fischer, M. L., Mauskopf, P. D., Osgood, D. E. and Lange, A. E. 1997, ApJ, 479, 17

Lay, O. P., & Halverson, N. W. 2000, ApJ, 543, 787

Mauskopf, P. D., Bock, J. J., Del Castillo, H., Holzapfel, W. L. and Lange, A. E. 1997, "Composite infrared bolometers with Si_3N_4 micromesh absorbers," Appl. Opt. 36, 765

Press, W. H., Teukolsky, S. A., Vetterling, W. T. and Flannery, B. P. 1997, "Numerical Recipes in Fortran, The Art of Scientific Computing, Second Edition." Cambridge University Press, Cambridge.

A. Appendix: Stripchart

Hopefully Alexey or Sam or Phil or someone will contribute a section about using stripchart!

B. Appendix: The Rotator and Rotator-Related Issues

Purpose of the Rotator:

To compensate for the change in position angle (PA) during an observation, allowing us to maintain the array orientation at a desired scan angle with respect to the sky for optimum sky sampling.

PA conventions:

PA is the angle between the Alt-Az and RA-DEC coordinate frames (specifically, measured clockwise on the sky from the positive Alt axis to the positive DEC axis), and its value is a function of position on the sky (which is recorded by the telescope and merged into the timestream). PA = 0 degrees when looking due south along the meridian, meaning that here the Alt-Az and RA-DEC frames are coincident. Still facing south, PA decreases to negative values as you turn toward the east, and increases to positive values as you turn toward the west. If you're looking due north along the meridian, PA = 180 degrees, increasing to the east and decreasing to the west.

Correcting for PA:

Rotator orientation convention: when I say "rotate the array CW" I mean clockwise from the point of view standing behind the dewar, looking down into the optics box. The fiducial zeropoint of the array should generally be set to where there is the least stress on the rotator. In general this will most likely correspond to the line between A and F hextants pointing up (toward the telescope, when mounted).

To correct for the effects of the changing position angle you simply rotate your detector *projected on the sky* CW by the PA at that position (so if PA is negative, you actually rotate CCW by |PA|). But, since there is a net flip about the x-axis due to the Bolocam optics (see "Coordinate Changes Due to the Optics," below), you actually want to *physically rotate the array CCW by PA* to compensate.

However, the rotator is only given a freedom of motion of ± 30 degrees around the fiducial zeropoint. Thus, if PA = -40 degrees so that you would normally rotate the array CW by 40 degrees, in actuality you will want to rotate CW by (40 – 60) degrees, or CCW by 20 degrees. So essentially, if |PA| > 30 degrees, you always want to either add or subtract 60 degrees such that you end up with |rotangle| < 30.

Note that the rotator only rotates in between subscans so as not to excite microphonics while data is being taken. That means that the orientation of the array on the sky is never exactly

compensating for PA, but is always fairly close. For a reasonably-sized subscan spanning 20 arcminutes on the sky, the PA should never change by more than a degree or so (on order), and a 1-degree rotation error corresponds to a 4-arcsecond position error at the edge of the array. If, when calculating pixel offsets, we use the PA at the middle of the scan (which we do), then the error would be only half of this, so 2" at the edge of the array. Such an approximation therefore seems perfectly acceptable for subscans up to about a third of a degree in length, but less so for longer subscans. Because of this, for the December 2001 observing run it will be necessary to write another pixel offsets routine that will be called from the mapping software, and which will calculate entire PA-corrected RA and DEC traces for a given bolometer, rather than just offsets.

Coordinate Changes Due to the Optics:

Here I will summarize what we know about orientation changes as light from a field on the sky passes through the Bolocam optical system and onto the image plane.

I used Zemax for this analysis, which was both a curse and a blessing, since its coordinate conventions take a little getting used to. Specifically, Zemax defines field angles such that a positive value means a positive slope in that direction. A field angle in x of +1 degrees means an incoming positive slope in the x direction, corresponding to a negative object position coordinate. This means that your object in Zemax is really rotated by 180 degrees from what you think it should look like. This is something to keep in mind for anyone who wants to play around with this in the future.

The following conventions were used for this analysis, and are generally considered most appropriate for the treatment of this problem:

- The proper reference frame is defined as that tracing the light path from the focal plane to the sky. If you think of all the optical elements as lenses, and the optical axis as the z-axis (with light traveling along +z), then imagine that you are looking along the −z direction. At the focal plane you are thus seeing what the array sees (not what the photon sees).
- The y-axis is the mirror-symmetry axis of the optics box. Projected onto the sky it is the Alt-axis of the telescope.
- The x-axis is perpendicular to both the light path and mirror-symmetry axis. If you are standing behind the dewar mounted onto the optics box, then it is always oriented left-right. Projected onto the sky it is the Az-axis of the telescope.
- A "flip" means that x maps to −x or y maps to −y, but not both.
- A "180-degree rotation" means that you map both x to −x and y to −y. 1 rotation = 2 perpendicular flips.

And now, here is a list of the orientation changes produced by the various elements of the Bolocam optics:

1. *Primary and secondary mirrors* — A Cassegrain telescope is an axisymmetric optical system, so results in a full 180-degree rotation (2 perpendicular flips) of the image from the original object orientation.

2. *Flat mirrors* — Each of these flips the image once about the y-axis, resulting in no net change.

3. *Ellipsoidal mirror* — This produces one flip about the x-axis.

4. *Lens* — The lens inside the dewar does not affect image orientation.

5. *Optics box mounting* — Due to the way in which the optics box is mounted on the telescope (the dewar is effectively "backwards" in relation to the telescope), there is one more transformation to be taken into account: a 180-degree about the z-axis. This one is hard to visualize without actually looking at the 3-D Zemax layout, so you'll just have to trust me.

NET RESULT: 2 180-degree rotations + 2 flips about the y-axis + 1 flip about the x-axis = 1 flip about the x-axis. Thus, when standing behind the array and looking down into the optics box, the "top" of the array actually sees lower Alt on the sky (lower DEC for PA = 0), but the left side of the array sees the left side of the field on the sky (ie. higher RA at PA = 0).