Development of a Data Reduction Pipeline for the ROVOR Observatory

Thayne A. McCombs

A senior thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Bachelor of Science

J. Ward Moody, Advisor

Department of Physics and Astronomy

Brigham Young University

August 2013

ABSTRACT

Development of a Data Reduction Pipeline for the ROVOR Observatory

Thayne A. McCombs
Department of Physics and Astronomy
Bachelor of Science

The Remote Observatory for Variable Object Research (ROVOR) has a backlog of three years of observations that have not been reduced or analyzed. In this Thesis we discuss software we developed to improve the data reduction pipeline and automate many of the necessary steps in reducing astronomical data. Specifically, we developed the RedROVOR python package to perform the tasks necessary for reducing data from ROVOR, as well as an online web interface (RovorWeb) which provides an easy to use interface to the RedROVOR toolset, as well as an online observation log management system to keep track of observations made with the ROVOR observatory.

Keywords: Remote Observing, Computing, Data reduction, Astronomy, ROVOR

ACKNOWLEDGMENTS

I would like to acknowledge J. Ward Moody for advising me with this research. I would also like to acknowledge Nathaly Young and Eric Hintz for teaching me how to use IRAF. I would like to acknowledge all the members of the ROVOR research group. In particular Benjamin Boizelle, Kimberly Bates, and Marcus Holden. I would also like to thank the National Merit Foundation and Brigham Young University for providing scholarhips for my tuition, as well as the Department of Physics and Astronomy and Office of Research and Creative Activities (ORCA) for funding my research.

Contents

Chapter 1

Introduction

1.1 Motivation

The Remote Observatory for Variable Object Research (hereafter referred to as ROVOR) is a remotely controlled observatory located at Delta, UT with a 16” RC Optics Ritchey-Crétien telescope mounted on a Paramount ME mount, all controlled using the Software Bisque suite of observatory control software including TheSky, CCDSoft, and Orchestrate.

Because of the remote location of ROVOR it is desirable to automate as much of the data collection process as possible. In particular much of the data reduction process is trivial enough that automation is feasible, and where it is not, the amount of human interaction can be minimized. Traditional methods of performing data reduction have not taken full advantage of this potential for automation.

For the specific case of the ROVOR project, automating the data reduction process has the added benefit that the reduction can be done remotely, on site, with minimal human input, and therefore we only need to transfer the final photometric data from the observatory, rather than the gigabytes of raw frames and calibration frames needed otherwise. This is especially important given the slow internet connection and remote location, and would be even more important for observatories in more remote locations.

Automation also has the obvious benefit of reducing the amount of time scientists must spend on the mundane task of removing systematic errors from the data, and extracting the desired information, and therefore increasing the time they can spend on actually analyzing and understanding the data.

However, astronomical observation has many variables, and the automated process may not always produce the desired results. For some observations, a more customized approach to the data reduction process may be desired. Furthermore the needs of the observatory, and possibly other observatories which utilize this software in the future may change over time. Therefore it is important that our automation process be flexible enough that it allows the user to override default behaviors, and to go back and check that the process was done correctly, as well as having a modular design so that individual components of the process can be changed without requiring the entire project to be modified.

1.2 Background

Astronomical data is primarily taken in the form of images taken with Charged Coupled Devices (CCDs). Since astronomical objects are very faint, noise from the CCD and other sources is very non-trivial. As a result the first step in analyzing astronomical data is to remove as much measurable error from the images as possible. This is done primarily with a number of calibration frames, specifically zeros, darks, and flats.

Zero frames, or bias frames are images taken with a zero length exposure. This image measures the effective zero-point for each pixel. Ideally the zero-point would be the same for every pixel and would be constant with time. Unfortunately in practice neither of these conditions are true. To compensate we take a number of zero exposure frames every night, average them together to get a master zero and subtract the resulting frame from all other frames, including other calibration frames. By doing this we minimize the variability in the bias between pixels and between nights in addition to subtracting the overall bias. One thing to note is that when performing the averaging it is good to remove a few of the extreme points for each pixel to account for cosmic rays or other outlyers. We remove the same number of maximal and minimal points for statistical balance.

In addition to the underlying bias, there is also noise introduced from thermal activity. Each pixel in the CCD is similar to a small capacitor, when a photon strikes the pixel it transfers its energy to an electron which can then jump to the other side of the capacitor and therefore cause a change in the voltage, signifying that a photon has been detected. Unfortunately, thermal energy can also excite an electron in the same way. The best way to deal with this problem is to cool the camera enough that the thermal noise is negligible, and in fact this is often done with astronomical instruments. However it is not always economical to provide such cooling, especially for smaller telescopes, and another method must be used to account for it.

This method is to take dark frames. The amount of thermal noise can vary from pixel to pixel, but unless the exposure is very short, or very long, the contribution to the image from thermal noise is linear with time. Therefore by taking an image with the shutter closed we can measure the amount of thermal noise per unit of time. It is best to expose for at least as long as the actual data frames. Like zeros, darks are taken every night, then after subtracting the master zero from them, they are averaged, rejecting extreme values for every pixel, to create a master dark. To apply the master dark to the flats and object frames, the dark is scaled to the same exposure time as the recipient frame and then subtracted from the recipient frame.

Unlike zero and dark frames, flat frames correct multiplicative errors rather than additive errors. Flat frames correct differences in the transmission function across the chip of the CCD. The differences could be caused by a number of both intrinsic an extrinsic causes. The primary intrinsic cause of variable transmission is different sensitivity between pixels. Each individual pixel has a slightly different sensitivity to light. Extrinsic causes include vignetting, imperfections in the optics of the telescope, and dust. These effects remain somewhat constant with time, so flats can be taken every few nights rather than every night. However since the transmission function is color-dependent, and the filter itself might contribute to non-uniformity, flats must be taken separately for each filter.

A flat frame is an image taken of a flat background, and by assuming that the background is flat, any variation in the image must be created by systematic errors, and those errors can be removed by dividing the object image by the normalized flat frame. Although there are different ways of taking flats, the most common, and the method used by ROVOR, is to take flats during twilight, once it is dark enough not to saturate the detector, but before stars are visible. Once the flats are obtained for a filter, they must be both zero and dark corrected. They are then normalized to 1 by dividing the entire frame by the mean of a square at the center of the detector. Finally they too must be averaged with a min max reject. When applying the flats the object frame is divided by the master flat for the corresponding filter.

Once the frames have been calibrated it is possible to perform photometry. Photometry is the measurement of the amount of light radiated by the source. With ROVOR we are primarily interested in variable objects, which change in brightness over time. To measure how the brightness changes there are two methods of photometry. Differential photometry compares the flux of the target to the average flux of a collection of comparison stars in the same field. All-sky photometry on the other hand uses a standardized field contains non-variable stars of known magnitudes and colors, which is calibrated on the same night at similar airmasses to get the actual apparent magnitude of the target.

Traditionally, photometry is done using the technique of aperture photometry. In this technique, a circular aperture is made around the center of the target and the total number of counts within the aperture is measured. An annulus is also made further out from the aperture, beyond the extent of the target's point spread function (PSF). This annulus is used to get a value for the sky background, which is generally computed using a mode. The instrumental magnitude of the target is then computed by

$$m_{inst} = -2.5\log\left(F_{aper} - nB\right) + Z,$$

where F_{aper} is the total number of counts in the aperture, n is the number of pixels in the aperture, B is the mode of the annulus, i.e. the background value, and Z is the zero point, which is the instrumental magnitude assigned to the sky background.

Another method of photometry, brought about by digital imaging and computers is PSF fitting photometry. This is the method which is used by *RedROVOR* project via the *daophot* IRAF package (Davis 1994). PSF fitting is more sophisticated than aperture photometry. Rather than simply counting the number of pixels in an aperture it will fit a model PSF to the star and use that, along with a determination of the background using a similar method to aperture photometry, to determine the instrumental magnitude. The primary advantage of this technique, is that it can cope with multiple stars with overlapping PSFs. It can also provide slightly more accurate results in general.

Chapter 2

Design

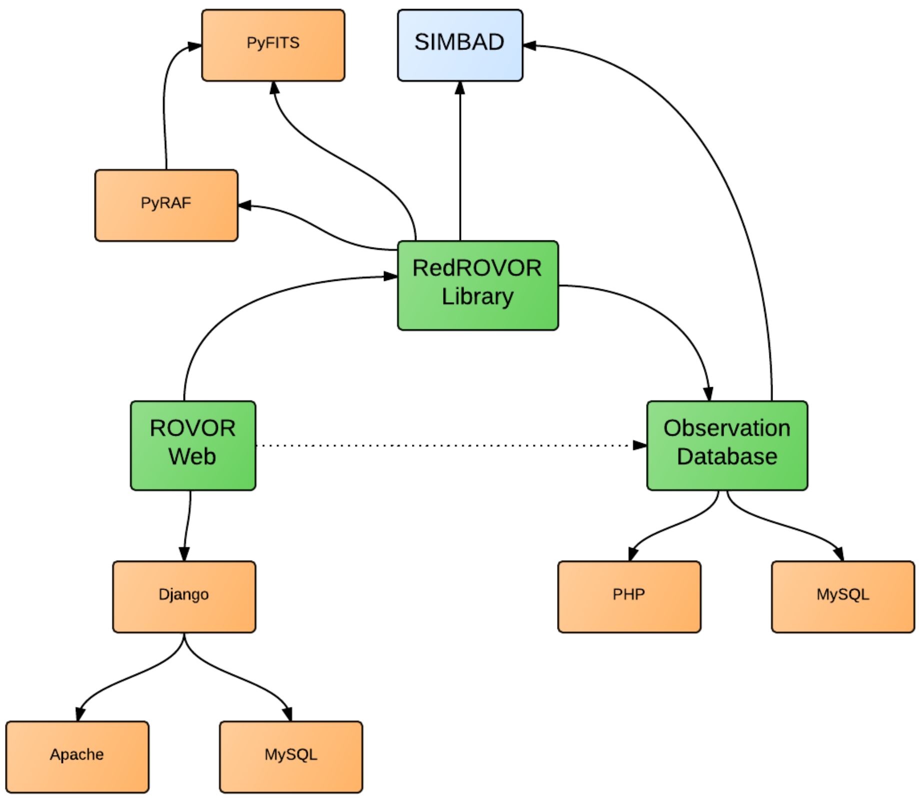
2.1 Overall Design

Our intention was to design a system which would be easy to maintain and update as our needs and existing technology changed. In addition, we wanted a system which, as much as possible, could be used by other research groups and different observational systems. With this in mind we split the project into sub projects and those sub-projects into modules. In addition the project of automating the reduction of data had to fit in with the more broad design of the entire observatory system. In particular at some future time we expect to combine the data reduction process with the observation process, once we have completed a more robust method of automating observation.

The data reduction project is split into three sub-projects, as can be seen in Figure 2.1. These projects are *RedROVOR* (Reduction for ROVOR), *RovorWeb*, and *ObsDB* (Observation Database).

RedROVOR is the collection of all the back-end functions and classes used to perform the various stages of reduction. Although it contains a couple of executable scripts, it is intended primarily as a library for front-end user interfaces such as *RovorWeb* to use. The *RedROVOR* library exposes a relatively abstract public API which allows for changes in the implementation without needing

Figure 2.1 *The Structure of our software design.* The green boxes are the components of our system, the beige boxes are external libraries or programs, and the blue box represents the SIMBAD website. Arrows represent dependencies.



to change client code. The advantage of having a seperate project for the back-end code is that multiple clients can use the same code to perform the actual computations, while giving different interfaces to the user. In addition to *RovorWeb* useful front-ends could include an automatic observation process invoking the reduction process after a night of observation, or a server-client system that uses local applications rather than a web browser as an interface. *RedROVOR* is a normal python package and can easily be installed like any other python package.

RovorWeb is an online interface for the reduction process. It uses normal web standards including HTTP, HTML, CSS, javascript, and JSON (Crockford 2006). As mentioned above it does not actually contain the code for performing the reduction, but uses the *RedROVOR* library. The interface allows the user not only to perform all the reduction stages either individually or several at a time, but also to browse the portion of the filesystem containing the data including downloading FITS files at any stage, and maintaining a database of target objects. It is also designed in such a way that another client can make use of the web-services exposed and use a different interface.

ObsDB is as its name implies a database of observations that have been taken and an online interface to access it. It is designed so that observations can be added either manually or automatically from the *RedROVOR* library or other client.

2.2 RedROVOR

Since several astronomical libraries already exist in Python, in particular *PyFITS*, *PyRAF*, and *pywcs*, it makes sense to write our *RedROVOR* library in python. Thus the *RedROVOR* library is a python library which must perform a number of tasks. These tasks include zero, dark and flat calibration, astrometric correction, and photometry. We have also chosen this library as the location for code which interacts with the SIMBAD API and *ObsDB* which is located at the rovor website (http:\\rovor.byu.edu).

2.2.1 Passes

To facilitate streamlining the pipeline as much as possible we have divided the reduction process into three “passes”. Each pass performs its processing steps on all of the images in a directory and prepares the directory for the next pass. The passes are divided by the amount of work that can be done at one time. Practically this means that the breaks between passes roughly correspond to the points when we need user input. Each also consists of several steps, which must be performed in consecutive order. Below we describe each pass.

First Pass The first pass performs the zero and dark calibration, and creates master flats, if any. The master calibration frames and processed object frames are stored in a seperate directory from the raw data. The steps are:

1. **Examine Directory**: Examine the headers in all FITS files and determine the frame type for each frame (i.e. whether it is a zero, dark, flat, or object).
2. **Create Master Zero**: Average together zeros with min/max reject to create master zero.
3. **Create Master Dark**: Subtract zero, and average scaled darks with min/max reject to create master dark.
4. **Create Master Flats**: If there are any flat frames in the directory, then for each filter, subtract the master zero, and scaled master dark, then average the normalized results with min/max reject.
5. **Subtract Zero and Dark**: For all the object frames in the directory subtract the master zero and scaled master dark. We don’t apply the flats yet, because we need the user to tell us which flat to use for each filter.

Second Pass The second pass applies flats to the object frames and performs an astrometric correction to apply a World Coordinate System to each object frame. The second pass is done in the processed directory where results from the first pass are placed. The steps are:

1. **Examine Directory**: Examine headers of object files and determine which filters for each object are present.

2. **Determine Flats to Use**: This step requires user interaction. The user must supply a flat frame for each filter that is present in the object frames. These flats were most likely created during the first pass, either for this directory, or a different one.

3. **Divide by Flats**: Each object frame is divided by the flat that corresponds to its filter.

4. **Apply WCS**: A World Coordinate System is applied to all object frames. The resulting images, along with WCS information is stored in a subdirectory named "WCS".

Third Pass The third pass is responsible for performing photometry. For greatest accuracy we use PSF fitting photometry with *IRAF*'s `daophot` package. The steps are:

1. **Examine Frames**: Examine the headers of the frames to determine which objects need to be photed.

2. **Retrive Coordinate Lists**: For each target in the directory a file containing celestial coordinates for the target and comparison stars is needed. This file must come from an outside source. The file could either be user supplied or automatically generated, as is the case for *RovorWeb* .

3. **Determine Paramaters and Perform Photometry**: Using the image frame and coordinate list for each frame, determine optimal paramaters for performing photometry, and then perform the various `daophot` tasks to perform the photometry.

4. **Extract Light Curves**: For each object-filter combination extract the HJD and instrumental magnitude from the photometry output files, and combine into a single file.

2.2.2 Modules

The *RedROVOR* library is a standard Python package consisting of several modules, and sub-packages. It can easily be installed using an installation script using Python's `distutils` package, which will properly place the package in the system's site-packages directory. The rest of this section will describe the purpose of important modules in the library.

coords The `coords` module is a relatively small module which is responsible for defining data structures for celestial coordinates. These data structures are also capable of converting between various representations of celestial coordinates.

simbad The `simbad` module, as its name suggests, provides functions that interact with the SIMBAD database. These functions include functions to get the look up the celestial coordinates of an object, get a normalized name for an object, and look up the name of an object from its celestial coordinates.

obsDB The `obsDB` module contains functions for interacting with the *ObsDB* API . These functions allow *RedROVOR* and incidentally *RovorWeb* to retrieve and store information to the observation database.

process The `process` module is the module which contains code to perform image calibration with zero, dark, and flat frames. It also supports a few other arithmetic procedures on images, such as addition, subtraction, division, averaging, etc., most of which are used as part of the calibration process.

wcs The `wcs` module is the module responsible for applying a World Coordinate System (WCS) to FITS files. It uses the astrometry.net (Lang et al. 2010) application to automatically perform the astrometric solution. It is in essence a python function that runs astrometry.net as a sub-process.

obsRecord The `obsRecord` module contains functions to automatically record observations either for a single FITS image or for all FITS images in a directory.

photometry Due to the complexity of performing photometry, an entire package is needed implementing photometry, although from the users perspective, the `photometry` package can be treated as a module. It contains the code to perform photometry and generate the light curve files .

firstpass, secondpass, and thirdpass These three modules encapsulate the three “passes” performed during the reduction process as described in 2.2.1. Each module contains a function which performs all of the steps in a pass, and a Processor class which implements each of the steps of the pass as methods.

2.3 RovorWeb

Since our backend is written in Python, it is advantageous for us to write our server logic for our web interface in python as well. Although there are multiple ways of doing this, the *Django* (Django Accessed July 22, 2013) framework is a stable and well-respected web framework for python and takes care of much of the necessary work to set up a dynamic web application. Thus we used the *Django* framework to develop *RovorWeb*.

The *Django* framework, like most web frameworks, uses a Model, View, Controller (MVC) architecture, although *Django* changes the terminology calling Controllers views and Views templates. Using the *Django* terminology, the model takes care of the data stored on the server and serializing it to a database. The template is, as its name suggests, an HTML template which is used to generate the HTML page sent to the client, and the view (or controller in MVC terminology) contains the logic to handle form input and instantiate the templates into actual HTML pages.

In the *Django* framework, a project consists of one or more "apps" which is implemented as a python package. Thus it is possible to split up a project into logical sub-projects. We take advantage of this in *RovorWeb* to separate the project into the following apps: `accounts`, `dirmanage`, `reduction`, `targets`, `obs_database`, and `root`. The rest of this section will describe these apps.

2.3.1 accounts

Since we want our web interface to be available to us remotely, our server must be available on the public internet, however we do not want our web interface to be vulnerable to corruption or other malicious intent from exterior parties. Therefore, we protect the interface using an authentication process. *Django* takes care of much of the authentication process for use using the built in `authentication` app, but we need to provide templates and views for the login page, and logging out. These are placed in the `accounts` app.

2.3.2 dirmanage

The user must have a means of selecting folders on the server in which to process images. It would also be nice for the user to be able to see which files are in the folder and to download files from the folder. All of these tasks require access to the server's filesystem and therefore it makes sense to combine them into the same app.

While we want the user to have access to some parts of the filesystem, we do not want them to have access to the entire filesystem. Therefore the `dirmanage` app contains a model which keeps track of which directories the user can access. If the user tries to access a file or folder that is not a descendant of one of those directories, the access is forbidden. The model also produces a sort of virtual filesystem in which the directories in the database are top-level directories. The model is responsible for translating paths between this virtual filesystem and the actual filesystem on the server.

RovorWeb and *RedROVOR* are intended to have at least two accessible directories: `Raw` and `Processed`. `Raw` is the directory where the raw images are stored after an observing session, and `Processed` is where all of the processed frames are stored.

The `dirmanage` app contains views which allow the user to browse the accessible folders, and download any files within those folders. It also contains some widgets which the other apps use to provide dialogs to the user which allow the user to select a file or folder. It also provides views which produce JSON (Crockford 2006) descriptions of directory contents. These views are used extensively with AJAX both in `dirmanage` and other apps, and could easily be used by other clients wishing to browse the virtual filesystem.

2.3.3 reduction

The `reduction` app is the pièce de résistance of *RovorWeb*. It is the app that actually performs all of the image processing. However, the code in the app does not actually do any image processing directly, rather it processes HTTP requests which may contain form data, and then calls the applicable code in *RedROVOR*.

The `reduction` app roughly follows the design of *RedROVOR*. Specifically it is broken down into three passes as described in 2.2.1, and the index page shows a link for each pass. The page for each pass then prompts the user for the directory to process, and any other user-required information, then presents the user with one or more buttons which allow the user to perform the pass in its entirety, or to perform a single step of the pass.

2.3.4 targets

When we perform photometry we need to know the coordinates of the target and its associated comparison stars. This information is stored in a database, which is accessed using the models in the `targets` app.

For each target the database stores the name, the celestial coordinates, and a normalized name, i.e. the first name in the name list on SIMBAD. It also has a related table which stores the coordinates for all the comparison stars (and the target). The model can generate a coordinates file for a target field from these database tables. This is preferable to a static coordinate file, because it is more dynamic and the comparison star list can be easily changed.

In addition to the models, which are used for photometry, the `targets` app contains views which allow the user to edit the list of targets, and their associated coordinate lists, see Figures B.1 and B.2. This includes the capability to upload a coordinate file, such as one produced by ds9, or to synchronize targets with the targets on *ObsDB*.

2.3.5 obs_database

The `obs_database` app, is not the actual observation database, although at a later time it may be much more closely integrated with it. The main purpose of the app is to allow users to upload automatically record observations based on the header information in the headers of all images in a folder. Since this only has one task, the interface is relatively simple. The user simply selects a folder to record observations for and push a button. See Figure B.3.

2.3.6 root

The `root` app is not in itself an independent web app. Rather it is the location of resources that are common to all the apps. In particular it contains javascript and CSS files, and a base template that most of the other templates extend.

2.4 ObsDB

The ROVOR observatory generates a large number of images each night of various objects and in different filters. Most of this has not yet been analyzed. When we later wish to go back and analyze the data it is time consuming to look through hundreds of nights to find data for the object of interest. Therefore it is advantageous to create a method to efficiently search previous nights for objects of interest. Thus we produced a searchable database system of all observations and a script which can crawl through a filesystem, automatically uploading observations into the database.

For the database we took advantage of the resources we already had in place for the ROVOR website. Specifically we had web hosting provided by the department with a MySQL database and php.

Initially the design simply used dynamically generated html. However it was quickly apparent that some aspects of the web interface lent themselves readily to AJAX (Asynchronous Javascript And XML). In addition, by using AJAX we could use the same web services to provide data and upload information from both the web interface and the automatic script, as well as potentially other clients. Although we started with using XML as the data interchange format, parsing the XML was giving us slow performance, so we switched to JSON (JavaScript Object Notation) (Crockford 2006) which was much faster.

The database stores information about each of our targets including the name, celestial coordinates, and optionally the type of the object. It also stores the following information about observations: target object, filter, date and number of frames.

The web interface has pages to manually add observations or target objects to the database, upload lists of observations or objects to the database, edit existing observations, and browse existing observations. When browsing it is possible to filter observations by date or target object.

Chapter 3

Implementation

3.1 RedROVOR

Initially we intended to use the Image Reduction and Analysis Facility (*IRAF*) for the data reduction, later we decide to use Python bindings for *IRAF* called *Pyraf*. However dealing with the finicky and temperamental requirements of *IRAF* led us to decide that it would be easier to simply rewrite the reduction routines using the lower-level *PyFits* library and the *numpy* numerical library, especially since for our observations we do not need the full capabilities of *IRAF*. This turned out to be a good decision since the hand-coded python routines were more stable and simple than using *PyRAF*, and it also simplified the code base quite a bit, since we can manipulate the image data directly in memory rather than having to store intermediate results to file. However, due to the inherent complexity of photometry, especially since we desired to perform PSF fitting photometry, we elected to use IRAF for the photometry portion of our process.

In the rest of this section we discuss the implementations of each of the modules in the *redrovor* python package.

3.1.1 coords

The `coords` module (Listing A.1) contains two classes and a named tuple.

The *RA_coord* and *Dec_coord* classes are symmetric in many ways. Both are representations of a celestial coordinate in sexigesimal form. Internally the coordinate as stored as three number, the hour/degree and minute are stored as integers, and the second is stored as a *Decimal* object, which has arbitrary precision and doesn't suffer from the rounding errors that the binary representation of a float would introduce.

The coordinate classes each have constructors that allow them to be constructed from their three components, but they also have factory class methods which construct a coordinate from a properly formatted string, or from a number in fractional degrees (and for *RA_coord* hours). The string can either be a space or colon delimited string in sexigesimal format, or a fractional degree value.

The classes also contain methods which return a *Decimal* value containing the fractional decimal representation in degrees (and for *RA_coord* hours), and arc-seconds. The formulas used to convert from the three value representation to degrees are $\delta^\circ = d + m/60 + s/3600$ for declination and $\alpha^\circ = 15(h + m/60 + s/3600)$ for right ascension. Going the other way requires repeated truncation, division and modulo operations.

The `coords` module also contains a *Coords* class which is simply a named tuple containing an *RA_coord* and *Dec_coord*. It has a convenience method *withinRadius* which determines if another *Coords* object is within a certain radius of it. The module also has a convenience function which will parse a Coords object from a coordinates file in the format output by ds9 in xy mode.

3.1.2 simbad

The SIMBAD website has a public API which allows clients to make requests with "scripts" that can be sent with the standard HTTP GET method. The URL for the service is http://simbad.

u-strasbg.fr/simbad/sim-script. We use this api to retrieve information from SIMBAD.

The heart of the `simbad` module (Listing A.2) is the `script_request` function which uses the Python `urllib` library to send a GET request to SIMBAD, then split the result into an array of lines. It automatically adds a line to the script to supress some header information that needlessly complicates the output. The rest of the functions in the database simply construct a script to send to SIMBAD, and then parse the results to create the appropriate python object.

The two most important function are `getMainName` and `getRADec`. `getMainName` gets a normalized name for an object by retrieving the first name in SIMBAD's list of names for an object. Since this procedure is performed quite frequently the function implements a cache which it uses to store previous look-ups. If the function is called with the same input name it will return the cached value. This significantly reduces the overhead of network traffic. `getRADec` will retrieve the celestial coordinates of an object, then parses them into a *Coords* object . However, SIMBAD occasionally returns the declination in a format that contains only the degrees and minutes, not the seconds, so `getRADec` must handle that special case by parses the string itself.

3.1.3 obsDB

Like the `simbad` module, `obsDB` (Listing A.3) interacts with a remote server. However, it is further complicated because *ObsDB* uses cookies to keep track of state and requires the user to be logged in for some operations.

To handle these complications we use the `urllib2` and `cookielib` libraries to create a URL opener with an associated cookie jar, and then use that opener to make the necessary login POST request. This starts the session and maintains the cookies for the session until the `logout` function is called or the module is unloaded.

Since all of the web services that *redrovor* uses return a JSON (Crockford 2006) object in the HTTP response, we have a helper function `_sendRequest` which will construct the correct URL, properly encode any request data, and once it receives a response, decode the JSON object into a python object. The rest of the functions in the module simply supply the page to use in *ObsDB* and a *dict* containing the request data, and return the result of `_sendRequest`

3.1.4 process

To simplify the logic of performing operations on several images at once we implemented an *ImageList* class in Python. This class can is in the `process` module (Listing A.4). This class was inspired by the way that *Mira Pro* handles image reduction. It represents a list of several related images on which operations can be performed simultaneously. These operations can be classified into two categories: transformations (or maps), and aggregations (or folds).

Transformations such as addition, subtraction, division, etc. are operations that are performed on each of the images in the list individually and produce a new *ImageList* or alter the existing *ImageList*. Essentially, these operations iterate over each frame in the *ImageList* and perform a simpler operation on that frame. That operation may involve a second operand, which may be a scalar value — i.e. a number — or another image frame.The operations on the individual frames are often performed by functions from the *numpy* library which operate with both scalars and arrays.

Aggregations are a little more complicated since they involve all of the images in a single computation which produces a single result. The most important of these operations are the sum and average operations (although the average is simply the sum divided by the number of frames). Although it would not be that difficult to write code to perform repeated applications of addition for the sum operation, the *numpy* library again helps us out with its "universal functions" and the *reduce* method which give us a shorthand for performing repeated computations over a collection. Since the *numpy* library is written in native C, using the *numpy* methods also gives us a performance benefit. However, we would like to reject the minimal and maximal values to account for the possibility of cosmic rays. This produces a bit of a difficulty since we don't know which points to throw out until we have finished iterating over all the frames. The way we handle this is by constructing a three dimensional array in which the first axis is the index of the images. We then sort the array along this axis using the *numpy* sort function. Finally we fold addition over the subarray excluding the top and bottom points as seen in the code:

```
all = numpy.array(map(lambda im: im[0].data, self._list))
all.sort(0) #sort along frame axis , so we can reject the min and maxes
total = numpy.add.reduce(all[minmax:-minmax])
```

The operations supported by the *ImageList* are as follows:

averageAll and avCombine These methods compute the average of all the frames in the *ImageList* by adding all the frames together by the method described above, and dividing by the number of frames minus the number of values rejected by min-max rejection. `averageAll` simply returns a numpy array. On the other hand `avCombine` wraps that array in a *PrimaryHDU* object from the *pyfits* library. It copies over the header information from the first frame in the *ImageList* as well as updating a couple of fields (see Listing A.4).

subtraction Subtraction is important for correcting both zeros and darks. Therefore we implement subtraction. If the second operand is a *PrimaryHDU* we extract the numpy array from it and subtract that from each frame, otherwise if it is a scalar or numpy array, we just loop over the frames and subtract it from each frame.

division Division is used both for scaling darks by exposure time, and doing flat correction. It is implemented similarly to subtraction.

normalization The `normalize` method computes the average value of a central block of the frame (100×100 by default) and then divides each frame by that average.This is used to normalize flat frames.

subZero, subDark, and divFlat These methods perform the calibration for zero, dark, and flat frames respectively. Each takes an argument which is a string with the path to the calibration frame which should be applied. The methods open the appropriate calibration frame, and then perform the appropriate correction to each frame in the *ImageList*. `subZero` simply subtracts the zero frame; `subDark` multiplies the dark frame by the exposure time of the image frame, then subtracts the result from the image frame; and `divFlat` divides the image frame by the flat frame, which is assumed to already be normalized.

saving The *ImageList* supports three methods of saving:

1. Each frame can be saved in place, that is, each frame is saved in the same location it was read from. This is implemented as the `saveInPlace` method.
2. Each frame is saved with the same filename, but in a different folder. This is implemented as the `saveToPath` method.
3. Each frame is saved with the same path and prefix, but with different numerical suffixes. This is implemented as the `saveIndexed` method.

In addition to the *ImageList* class, the `process` module contains functions which will create and apply all three kinds of calibration. These they accept strings containing paths to images, and take care of opening them, and making the appropriate calls to *ImageList*. They can optionally also save the results, or return the results as an *ImageList* or *PrimaryHDU*.

3.1.5 wcs

The `wcs` module (Listing A.5) is a relatively simple module. It has a single public method `astrometrySolve` which takes a variable length arguments list (varargs) containing filenames to perform photometry on and optional keyword arguments which will be explained momentarily.

We use a local installation of the *astrometry.net* (Lang et al. 2010) astrometric system to find the plate solutions for our frames, so the `astrometrySolve` function creates a child process that executes the *astrometry.net solve-field* program with command line options that were generated from the arguments passed to `astrometrySolve`. Normally the child process would execute asynchronously, however, since the number of frames to process is generally very large, spawning all of the processes at once exhausts system resources and the system freezes. To solve this problem we wait for the current child process to terminate before continuing with the program. It would also be possible to create a finite pool of threads, each one processing a single frame. This would allow the program to process multiple frames simultaneously without exhausting resources. However, due to time constraints and only marginal expected gain we did not implement it as a thread pool.

To create the command line arguments we look at the keyword arguments supplied and translate them into the corresponding command line arguments. The following options are supported:

guess A *Coords* object or tuple containing the approximate right ascension and declination of the center of the frame.

radius The radius of error for the coordinates supllied in *guess*. The solution will only be tried with coordinates within a distance of *radius* degrees of the guessed coordinates. Defaults to one.

lowscale The lowest plate scale to try.

highscale The highest plate scale to try.

outdir The directory to store the resulting files in.

isfits A boolean indicating whether or not the frames are in FITS format. Defaults to true.

options A list of strings containing any command line options to include directly.

3.1.6 obsRecord

The `obsRecord` module (Listing A.6) contains code to record observations for all the frames in a directory. It contains two functions, `recordObservation` and `recordDir`, which record individual observations, and all observations for a directory respectively.

The `recordObservation` function will get the name of the object from the headers of the frame, using “unknown” if the the TITLE field isn’t set and it is unable to determine the title from the celestial coordinates. It then gets the information about the target from the observation database, creating a new target entry if the name does not already exist in the database. After that it retrieves information from the header and uploads observation information to the remote database using the `obsDB` module. .

The `recordDir` function simply walks through all files in the directory recursively, and for every file that is a FITS file it checks the frame type. If the frame type is “object” then it will call `recordObservation` on the frame.

At the moment it sends an individual request for each frame in the directory. To conserve time and space it would be better to send all of the recorded observations together in a single HTTP request, and to group frames of the same target in the same filter together in the same record. However, due to lack of time we have not yet made these optimizations.

3.1.7 photometry

The `photometry` package is probably the most complex part of the *redrovor* package. Unlike the rest of the modules discussed in this section it is complicated enough to warrant splitting it into multiple modules and making it a package rather than putting it all in one package.

Although we opted to create our own routines for calibration, photometry is significantly more complicated, so for our first attempt at least we decided to use PyRAF, an interface to IRAF to perform the actual photometry.

Although the photometry package consists of multiple modules, the `__init__.py` file (Listing A.7) imports all the functions that the client code would use, so that those functions can be used by simply importing `redrovor.photometry`. Specifically `__init__` imports `init`, `phot`, and `makeLightCurves`.

The `irafmod` module (Listing A.8) is intended to properly initialize PyRAF. The `init` function imports pyraf and loads the necessary iraf packages. It temporarily changes directory to the directory supplied, or a default value, since when importing PyRAF, the working directory needs to contain the `login.cl` file. `check_init` simply verifies that the `irafmod` module has been properly initialized.

Since we decided to use PSF fitting photometry, specifically using *daophot* in IRAF, we need to determine a number of paramaters, some of which vary from frame to frame. In particular we need to determine the Full Width at Half Maximum (FWHM) of the PSF for the frame, as well as the average background, and the standard deviation of the background. These computations are performed in the `calc_params` module.

`getBox` takes an image frame, the celestial coordinates of the target, and optionally a size of box to get, and returns a subarray containing the pixels of the image inside a box centered at the celestial coordinates supplied. It does this by converting the right ascension and declination to degrees, then using the *pywcs* library to convert the celestial coordinates to pixel coordinates from the WCS in the headers of the image. `background_data` retrieves the average background average by taking the central box determined by `getBox` and retrieving a histogram of the values with a bin size of 1. It then sets the average to the value that corresponds to the bin with the highest frequency. The standard deviation is estimated by taking a standard deviation of the box, with any value greater than twice the average background trimmed off. This is done to ignore the values due to stars or other objects in the central frame. These measurements are intended to measure the background in the area around the target object, however they assume that most of that area is in fact background, and that most of the light from the stars is at least twice as bright as the background. If either of these assumption is false, the algorithms will not give good estimates.

The `getAverageFWHM` function uses the IRAF task *psfmeasure* to measure the average FWHM of the PSFs for a frame. It simply sets a number of paramaters for *psfmeasure* then calls *psfmeasure* and parses the average value from the result. This assumes that the supplied coordinates are good stars to measure the FWHM and does not reject outliers.

Related to the `calc_params` module is the `params` module. This contains a *Params* class which extends *dict* and keeps track of the plethora of parameters needed for photometry. *Params* is again extended by *DAO_params* which adds specific parameters for *daophot*. Both classes also have reasonable defaults for many of the parameters, and some methods which compute parameters from other parameters. The *DAO_params* class also has a method called `applyParams` which sets the appropriate IRAF parameters in the *daophot* package according to the values in the *dict* underlying the *Params* object. The module also contains the convenience function `getDAOParams` which takes an observatory object (described in Section 3.1.15) an image path, a coordinate file, and optional keyword arguments, and then uses `calc_params` to compute the FWHM, background and background standard deviation, then returns the resulting *DAO_Params* object.

If another photometry method was added to *RedROVOR* it would be straightforward to implement a seperate subclass of *Params* which specialized the parameters for that method of photometry.

Finally, the `daophot` module provides the `phot` function which performs the actual PSF fitting photometry with *daophot*. If a *Params* object is not passed to it, it will create one with `getDAOParams`. It then calls `applyParams` on the *Params* object to set the appropriate IRAF parameters. It will then temporarily change into the output directory, since IRAF works best by being in the directory in which output should be stored. While in this directory it calls the *daophot* tasks *phot*, *pstselect*, *psf*, *group* and *nstar* in that order. Each step uses output from the task preceding it so the order is imperative.

Once the `daophot` module produces the photometry files, we would like to have the photometric data in a more usable format since the output from IRAF is rather unwieldy. Therefore, the `lightcurves` module contains the `makeLightCurves` function which parses the phot files and creates two column files with the heliocentric julian date and instrumental magnitude. Rather than creating a temporary file, we use *StringIO* to manipulate the output, so that we can manipulate the data in memory. We first use the IRAF *pdump* task to dump the data from the phot file into a more conventional column-based text file (which in our case is stored in memory). We then use the *csv* library to parse that output and split it up into multiple files based on the star ID and filter. To keep track of all the different combinations of star ID and filter we use a map from tuples of the ID and filter to open file objects. While this works fine for moderate numbers of comparison stars, for very large numbers this will exhaust the number of file descriptors available to a single process.

3.1.8 firstpass

The `firstpass` module performs all of the logic necessary to complete the tasks in the first pass. Since there are multiple steps which most be performed in a specific order, with the state stored between each step we created a class, *FirstPassProcessor* which has each step of the pass as a method, and stores the state of the operation in member variables. The `firstpass` method of the *FirstPassProcessor* calls the other main methods in the correct order, to perform the first pass, and the `doFirstPass` function simply creates a *FirstPassProcessor* for the path and calls `FirstPassProcessor.firstpass`.

The first step is to go through the headers of the images and split them up according to frame type, and split the object frames according to the target object. The constructor of *FirstPassProcessor* finds all of the FITS files in the supplied folder and store them in a list. It also creates a new sub-directory under "/data/Processed" based on the date of the first frame. `buildLists` then uses functions from the `frameTypes` module to construct data structures storing lists of the different frame types, and lists of each object for object frames. Since this computation takes a non-trivial amount of time, and different parts of the first pass could be done at different times we persist these data structures by saving them in JSON (Crockford 2006) format as a file. If the class is later instantiated with the same folder, it can load the JSON file and use the data structures without having to recompute them.

The `makeZero`, `makeDark`, `makeFlats`, and `zero_and_dark_subtract` methods are simply wrappers around respective functions in the `process` module. Each step stores the result of the operation, to be used by the next step, and each step checks to see if the result of a previous operation has been created in the processed folder. Each result is stored in the processed folder.

3.1.9 secondpass

The `secondpass` module is very similar to the `firstpass` module. Like `firstpass` it uses a processor class, called *SecondPassProcessor* to contain the logic and hold the state of the procedure. Unlike `firstpass` it only cares about the object frames, but it needs to organize them by filter. Like `firstpass` it uses the `frameTypes` module to accomplish this, and stores the resulting data structures in a JSON file which can be read later to preven recomputing the structures.

The `neededFilters` method uses the computed mapping of filters to frames to return a list of the filters that are needed to flat reduce the object frames. `applyFlats` then takes a *dict* mapping filters to the correct flat frame to use, and applies said flats to the object frames in the correct filter.

The `applyWCW` method simply extracts some information from the header for every object frame, and then uses *astrometry.net* (Lang et al. 2010) to perform astrometry on it, and saves the result in a sub-directory called "WCS."

3.1.10 thirdpass

The `thirdpass` module follows the same pattern as the `firstpass` and `secondpass` modules. Like them, it also stores its appropriate data structures as JSON files.

The *ThirdPassProcessor* class has a `objectNames` method which retrieves a list of all the target objects in the folder. `phot` simply wraps the `phot` function in the `photometry` module, and stores the result in a sub-directory called "photometry," and `makeLightCurves` simply wraps the similarly named function in the `photometry` module.

3.1.11 utils

The `utils` module is a catch-all module for useful functions that don't really belong anywhere else. The `ensure_dir` function looks to see if a directory already exists, and if it doesn't it creates it. The `getTimeString` function gets a formatted string of the current time, by simply calling `strftime` on the date object for now. `writeListToFile` and `writeListtoFileName` take a list, convert every element to a string and join by newlines, then writes the result to the supplied file. `findFrames` uses normal extensions for FITS files, and uses a glob to detect all the FITS files in a folder. The `workingDirectory` object is a context manager which changes to a different working directory on entry, and returns to the original working directory on exit. This is useful for temporarily changing directory with a `with` statement. The `contextmanager` decorator turns the function into a Context Manager where the code before the yield is executed on entry, and the code after the yield is executed on exit.

3.1.12 renamer

CCDSoft outputs files with the extension "FIT" for reasons unknown. IRAF on the other hand prefers files with a lower case extension, "fit." Thus, we wrote the `renamer` module (Listing A.17) which renames all FITS files in a folder to have a “fit” extension. The renameFITS function simply strips off the old extension and appends the new extension, then renames the file, and renameAll iterates through all files in the directory and calls renamer on any files which end with the old extension (which defaults to “FIT”).

3.1.13 fitsHeader

The fitsHeader module (Listing A.18) is a utility module which abstracts some of the logic for extracting information for the header fields of FITS files. The isFits function checks the extension of a file to determine whether or not it is a FITS file, and fitsCheckMagic looks at the first few bytes of the file to determine if it is a FITS file.

The getFrameType function looks at the **IMAGETYP** header to determine the type of the frame. It uses regular expressions to compare the values, since there are multiple possible values for each frame type. It also looks at the **EXPTIME** header, and if it is zero then it returns “zero” since a zero length exposure is by definition a zero frame. The getFilter function simply extracts the value of the **FILTER** header.

getObjectName will look at the **OBJECT** and **TITLE** headers to extract the name of the target. If neither of these are present it will get the celestial coordinates from the header and look up the name from *ObsDB*. If that fails it will create a name from the right ascension and declination using makeRADecName which simply concatenates the RA and dec with underscores, ignoring the least significant components. The related normalizedName function calls getObjectName and then uses simbad to get a normalized name, i.e. a name that will be the same, even if the frames use different names for the same object.

The getRA and getDec functions, as their name imply, get the right ascension and declination of the frame from the headers. It can either use **OBJCTRA** and **OBJCTDEC**, or **RA** and **DEC**. They return tuples of strings for the sexigesimal components.

The `splitByHeader` function iterates through all the files in a list of files, and creates a dict, where the keys are values of a FITS header, and the values are lists of the frames which have that value for that FITS header.

3.1.14 frameTypes

This module (Listing A.19) deals with organizing frames according to the type of frame and the target object. `getFrameLists` will iterate over a list of files, and call the `getFrameType` function from `fitsHeader` to determine the type, and create lists for zeros, darks, flats, and object frames, and put them into a dict. There is also an "unknown" category for any files which don't match any of the categories, or are not FITS files. `saveFrameLists` will write the result of `getFrameTypes` to text files, one for each type, with a name corresponding to the type.

The `makeObjectMap` function is similar to `getFrameLists` but organizes the frames according to the target object name instead. `makObjectList` gets a list of all the objects in the directory by calling `makeObjectMap` and returning the key set. `printObjectList` saves a list of the objects used to file, and `printObjectMaps` prints a list of frames in a file for each object in the list.

3.1.15 observatories

There are quite a few parameters that we use in *RedROVOR* which are dependent on the observatory. Although we only have a single observatory at the moment, we hope that in the future this system could be used for other observatories, including other ROVOR observatories. To facilitate migrating the software to other systems we created the *Observatory* class which encapsulates many parameters which may depend on the observatory system. Many of these havae reasonable defaults, but they can be overridden if necessary. This module also contains a definition of the *Observatory* object for the ROVOR observatory.

3.2 RovorWeb

RovorWeb was developed using the *Django* framework, as mentioned above. The implementation of the web interface is not directly relevant to the reduction of the data, and we will therefore not discuss said implementation further in this text. To see the source code and associated documentation for *RovorWeb*, please refer to the *RovorWeb* project on Github (https://github.com/rovor/RedROVOR).

3.3 Observation Database

The MySQL database for the observations consists of two tables, `objects` and `observations`. The `objects` table contains all the astronomical targets that we have hit, and `observations` contains the actual observations and has a many-to-one relationship with the `objects` table.

The `objects` table contains seven columns:

obj_id A unique identifier for the object

name The User Specified Name of the Object, this is the value associated with the value in all user applications

ra The Right Ascension of the Object

declination The declination of the object

types A comma delimited list of types for the object. For example Markarian 501 may contain types of AGN, BL Lac, Galaxy, etc.

otherNames Other names which the object may be referred to with, this is less important since other names can be looked up using Simbad.

simbadName This is the "main" name of the object in the Simbad database. This makes it easy to ensure that we don't have multiple rows referring to the same object with different names.

Note that all the information in the `objects` table is available from SIMBAD, however querying SIMBAD is somewhat slow, and if we ever observe an object which is not on SIMBAD, or used a name not recognized by SIMBAD the system would break.

The `observations` table contains ten columns:

obs_id The identifiier number for the observation

object_id The identifier number for the object that was observed

utdate The date in Universal Time that the observation was taken on

filter The filter the observation was taken in

temp The temperature (in Celsius) of the CCD during observation

notes Any notes about the observation (such as weather conditions, problems, etc)

filename The location of the file(s) for the observation on the hard drive

numFrames The number of frames for this particular observation (defaults to one)

Listing 3.1 The SQL code to create the tables used by ObsDB

```
CREATE TABLE objects (
  obj_id int(11) NOT NULL AUTO_INCREMENT,
  name varchar(100) NOT NULL,
  ra double NOT NULL,
  declination double NOT NULL,
  types tinytext ,
  otherNames varchar(200) DEFAULT NULL,
  simbadName varchar(100) NOT NULL,
  PRIMARY KEY (obj_id),
  UNIQUE KEY name (name),
  UNIQUE KEY simbadName (simbadName)
)

CREATE TABLE observations (
  obs_id int(11) NOT NULL AUTO_INCREMENT,
```

```
  object_id int(11) NOT NULL,
  utdate date NOT NULL,
  filter varchar(10) DEFAULT NULL,
  exptime int(11) DEFAULT NULL,
  temp int(11) DEFAULT NULL,
  notes text,
  filename varchar(200) DEFAULT NULL,
  numFrames int(11) DEFAULT NULL,
  PRIMARY KEY (obs_id),
  KEY object_id (object_id),
  CONSTRAINT observations_ibfk_1 FOREIGN KEY (object_id)
  REFERENCES objects (obj_id)
)
```

Chapter 4

Conclusion and Future Work

Although a considerable amount of work has been done, there is still a lot more that can be done to further improve this project. While testing, we have encountered a number of bugs, and while those encountered so far have been fixed, there are likely many more that we have not yet encountered. There are also some things which could be done to improve performance or usability which, for the sake of time, we did not implement. The current state can be seen on the Github project under the issues section.

The biggest area for future work, however, is integrating this data analysis system into the telescope control system. The ROVOR project will soon begin revamping the telescope control system. This telescope control system will be able to easily interact with *RedROVOR* and feed it data as soon as the data has been taken by the telescope. *RedROVOR* was designed with this in mind from the beginning.

Currently, *RedROVOR* uses twilight flats. However, this introduces added complexity to the system and requires human interaction to choose the proper flats. We have contemplated using a mathematical flattening algorithm rather than twilight flats, but we have not yet worked out all the details or determined if such a method would provide adequate results. If such a method were used, the entire process could be automated from taking the image to producing the light curves.

According to Holden (2013), the *RedROVOR* system is easy to use and works well in most cases. It takes about four hours to complete a single night, but very little of this time requires human interaction. From the user's perspective this is preferable to the more time-consuming task of reducing the data with IRAF. The ease of use would be even further enhanced if the framework was tied directly into an automated telescope control system.

Finally, we have strived to develop a system which could be used on other observatory systems with a minimal amount of modification. We hope that this system can be ported to other observatories and benefit other projects besides our own.

Appendix A

Source Code

This appendix contains much of the source code for the RedROVOR project. The full source code is publicly available in the RedROVOR project on GitHub. At the time of writing this project is located at https://github.com/rovor/RedROVOR.

A.1 RedROVOR

Listing A.1 coords.py

```
'''classes for holding coordinate objects'''

from decimal import Decimal
from collections import namedtuple
from math import copysign
import re

decimal_re = re.compile(r'\d+(\.\d+)?') #somewhat restrictive re for decimal
    numbers

class RA_coord(object):
    '''A coordinate in RA'''

    ra_expr = re.compile(r'(\d{1,2})[:␣](\d{1,2})[:␣](\d{1,2}(\.\d+)?)')
    def __init__(self,h,m,s):
        '''intialize with hours, minutes, and seconds
        note that the signs are ignored since RA is always positive '''
        if isinstance(s,float):
            s = "{0:.2}".format(s)
        self.h = abs(int(h))
        self.m = abs(int(m))
        self.s = abs(Decimal(s))
```

```
    def toHours(self):
        '''return the RA as a Decimal approximation'''
        return self.h+Decimal(self.m)/60 +Decimal(self.s)/3600
    def toDegrees(self):
        '''convert the RA to degrees and return the result as a Decimal'''
        return  self.toHours()*15
    def toASeconds(self):
        '''compute the RA in arcseconds (more accurate representation for
           numeric computation'''
        return 15*(self.h*3600 + self.m*60 + self.s)
    def __str__(self):
        '''convert to a string of numbers seperated by colons'''
        return "{0:02}:{1:02}:{2:05.2f}".format(self.h, self.m, self.s)
    def __repr__(self):
        '''representation of RA_coord'''
        return "RA_coord({0},{1},{2})".format(self.h, self.m, self.s)
    @classmethod
    def fromStr(cls,s):
        '''retrieve the RA from a string in the fromat hh:mm:ss.ss
        if invalid format return None'''
        match = RA_coord.ra_expr.match(s)
        if match:
            h,m,s = match.group(1,2,3)
            return cls(h,m,s)
        elif decimal_re.match(s):
            return cls.fromDegrees(Decimal(s))
        else:
            return None
    @classmethod
    def fromHours(cls,hrs):
        '''convert to RA_coord from decimal representation of RA in hours'''
        tmp = hrs
        h = int(tmp)
        tmp *= 60
        m = int(tmp % 60)
        tmp *= 60
        s = tmp % 60
        return cls(h,m,s)
    @classmethod
    def fromDegrees(cls, deg):
        '''create RA_coord from decimal representation in degrees'''
        if isinstance(deg,float):
            deg = "{0:.2}".format(deg)
        return cls.fromHours(Decimal(deg)/15)

    def __sub__(self, other):
        '''compute the difference between two RA measures
        in arcseconds'''
        return self.toASeconds() - other.toASeconds()

    @property
    def hours(self):
        return self.h
    @property
    def minutes(self):
        return self.m
    @property
    def seconds(self):
        return self.s


class Dec_coord(object):
```

```
    '''A coordinate in declination'''

    dec_expr = re.compile(r'([+-]?\d{1,2})[:␣](\d{1,2})[:␣](\d{1,2}(\.\d+)?)')
    def __init__(self,d,m,s):
        '''intialize with degrees, minutes, and seconds
        note that the signs are ignored for m and s, and the sign
        of the declination is determined by the sign of d'''
        if isinstance(s,float):
            s = "{0:.2}".format(s)
        self.d = int(d)
        self.m = abs(int(m))
        self.s = abs(Decimal(s))
    def toDegrees(self):
        '''return the dec as a Decimal approximation'''
        return copysign(abs(self.d)+Decimal(self.m)/60 +Decimal(self.s)/3600,
            self.d)
    def toASeconds(self):
        '''compute the declination in arcseconds'''
        return self.d*3600 + self.m*60 + self.s

    def __str__(self):
        '''convert to a string of numbers seperated by colons'''
        return "{0:+03}:{1:02}:{2:05.2f}".format(self.d,self.m,self.s)
    def __repr__(self):
        '''representation of Dec_coord'''
        return "Dec_coord({0},{1},{2})".format(self.d,self.m,self.s)
    @classmethod
    def fromStr(cls,s):
        '''retrieve the dec from a string in the fromat hh:mm:ss.ss
        if invalid format return none'''
        match = Dec_coord.dec_expr.match(s)
        if match:
            d,m,s = match.group(1,2,3)
            return cls(d,m,s)
        elif decimal_re.match(s):
            return cls.fromDegrees(Decimal(s))
        else:
            return None
    @classmethod
    def fromDegrees(cls,deg):
        '''convert to Dec_coord from decimal representation of dec in degrees
            '''
        tmp = abs(deg)
        d = copysign(int(tmp),deg)   #keep sign in the degrees part
        tmp *= 60
        m = int(tmp % 60)
        tmp *= 60
        s = tmp % 60
        return cls(d,m,s)
    def __sub__(self,other):
        '''compute the difference between two declinations in arcseconds'''
        return self.toASeconds() - other.toASeconds()

    @property
    def degrees(self):
        return self.d
    @property
    def minutes(self):
        return self.m
    @property
    def seconds(self):
        return self.s
```

```
Coords = namedtuple('Coords',['ra','dec']) #type for tuple of ra and dec

def __cwithinradius(self, other,radius):
    '''compute whether or not the other Coords is within radius arcseconds of
    self, this assumes rectangular coordinates so it is only accurate if the
    two objects are close to each other'''
    return (self.ra-other.ra)**2 + (self.dec-other.dec)**2 < radius**2

Coords.withinRadius = __cwithinradius

def parseCoords(f):
    '''parse a list of Coords from a file like object
    @param f a file-like object which has at least two columns, the first of
    which is the RA and the second is dec, they can either be in sexigesimal
    or decimal degree format (not that if decimal RA is assumed to be degrees,
        not hours
    @returns a generator which iterates over the coordinates in a file
    and returns Coords objects'''
    for line in f.readlines():
        if not (line.isspace() or line.startswith("#") ):
            #only deal with lines that have content and don't start with '#'
            rastr, decstr = line.split()[0:2]
            yield Coords(ra=RA_coord.fromStr(rastr), dec=Dec_coord.fromStr(
                decstr))
    return
```

Listing A.2 simbad.py

```
from urllib import urlopen,urlencode
from coords import RA_coord, Dec_coord,Coords
from decimal import Decimal
import re

SIMBAD_URL = "http://simbad.u-strasbg.fr/simbad/"
SIMBAD_SCRIPT_URL = SIMBAD_URL + "/sim-script"


def script_request(script):
    '''run a simbad script and return the result as an array
    of strings (each item is a single line of the output),
    this uses caching to improve performance'''
    #first prepend a line to quiet the console and script echo
    script = "output␣console=off␣script=off\n" + script
    resource = urlopen(SIMBAD_SCRIPT_URL, urlencode({'script':script}))
    result = [x.strip() for x in resource if x.strip()]
    resource.close()
    return result


def getAllNamesFromName(name):
    '''return an array of all names for an object in simbad'''
    script = r'''format object "%IDLIST[%*(S)\n]"
    query id {0}'''.format(name)
    return script_request(script)


def getNamesFromRADec(ra,dec,radius='5m'):
    '''get the names of objects with radius of ra and dec.
    we expect ra and dec to be RA_coord and Dec_coord objects
    or at least to be convertable by string to the normal
    colon delimited sexigesimal format'''
    script = r'''format object "%IDLIST[%*(S)\n]"
    query coo {0:s} {1:s} radius={2:s}'''.format(ra,dec,radius)
```

```
    return script_request(script)

def getRADec(name):
    '''get a Coords object for the given object'''
    script = r'''format object "%COO(:s;A | D)"
    query id {0}'''.format(name)
    result = script_request(script)
    if ':error:' in result[0]:
        return None
    ra,dec = result[0].split('|')
    ra = RA_coord.fromStr(ra.strip())
    match = getRADec._min_re.match(dec.strip())
    if match:
        #we need to take care of the special case when we get fractional
            minutes instead of seconds
        d = int(match.group(1))
        mins = Decimal(match.group(2))
        m = int(mins)
        s = (mins-m)*60
        dec = Dec_coord(d,m,s)
    else:
        dec = Dec_coord.fromStr(dec.strip())
    return Coords(ra,dec)

getRADec._min_re = re.compile(r'^([+-]?\d+):(\d+\.?\d*)$')

def getMainName(name):
    '''get the "main" name for the given object in simbad,
    useful for uniquely identifying an object
    if the name wasn't found return the name that was passed in'''
    if name in getMainName.cache:
        return getMainName.cache[name]
    script = r'''format object "%IDLIST(1)[%*(S)]"
    query id {0}'''.format(name)
    response = script_request(script)
    if ':error:' in response[0]:
        #error, so just return the name that was given to us
        result = name
    else:
        result = response[0]
    getMainName.cache[name] = result
    return result
getMainName.cache = {}
```

Listing A.3 obsDB.py

```
from urllib import urlopen,urlencode
from coords import RA_coord, Dec_coord,Coords
from decimal import Decimal
import re

SIMBAD_URL = "http://simbad.u-strasbg.fr/simbad/"
SIMBAD_SCRIPT_URL = SIMBAD_URL + "/sim-script"


def script_request(script):
    '''run a simbad script and return the result as an array
    of strings (each item is a single line of the output),
    this uses caching to improve performance'''
    #first prepend a line to quiet the console and script echo
    script = "output␣console=off␣script=off\n" + script
```

```
    resource = urlopen(SIMBAD_SCRIPT_URL, urlencode({'script':script}))
    result = [x.strip() for x in resource if x.strip()]
    resource.close()
    return result

def getAllNamesFromName(name):
    '''return an array of all names for an object in simbad'''
    script = r'''format object "%IDLIST[%*(S)\n]"
    query id {0}'''.format(name)
    return script_request(script)

def getNamesFromRADec(ra,dec,radius='5m'):
    '''get the names of objects with radius of ra and dec.
    we expect ra and dec to be RA_coord and Dec_coord objects
    or at least to be convertable by string to the normal
    colon delimited sexigesimal format'''
    script = r'''format object "%IDLIST[%*(S)\n]"
    query coo {0:s} {1:s} radius={2:s}'''.format(ra,dec,radius)
    return script_request(script)

def getRADec(name):
    '''get a Coords object for the given object'''
    script = r'''format object "%COO(:s;A | D)"
    query id {0}'''.format(name)
    result = script_request(script)
    if ':error:' in result[0]:
        return None
    ra,dec = result[0].split('|')
    ra = RA_coord.fromStr(ra.strip())
    match = getRADec._min_re.match(dec.strip())
    if match:
        #we need to take care of the special case when we get fractional
            minutes instead of seconds
        d = int(match.group(1))
        mins = Decimal(match.group(2))
        m = int(mins)
        s = (mins-m)*60
        dec = Dec_coord(d,m,s)
    else:
        dec = Dec_coord.fromStr(dec.strip())
    return Coords(ra,dec)

getRADec._min_re = re.compile(r'^([+-]?\d+):(\d+\.?\d*)$')

def getMainName(name):
    '''get the "main" name for the given object in simbad,
    useful for uniquely identifying an object
    if the name wasn't found return the name that was passed in'''
    if name in getMainName.cache:
        return getMainName.cache[name]
    script = r'''format object "%IDLIST(1)[%*(S)]"
    query id {0}'''.format(name)
    response = script_request(script)
    if ':error:' in response[0]:
        #error, so just return the name that was given to us
        result = name
    else:
        result = response[0]
    getMainName.cache[name] = result
    return result
getMainName.cache = {}
```

Listing A.4 process.py

```
import pyfits
import numpy

# TODO make sure everything is closed properly if there is an exception
# fine for short scripts, but could be a big problem on a continously running
# server

from itertools import imap, chain

import os
from utils import ensure_dir, getTimeString

class ImageList:
    '''A list of images on which to perform operations such as combining,
       subtracting, dividing etc'''

    def __init__(self, *args):
        '''create an image list from supplied filenames
            to create from a collection call like ImageList(*coll)
            '''
        self._list = [pyfits.open(fname) for fname in args] # initialize the list of images


    def averageAll(self, minmax=2):
        '''compute the average image of all images in the ImageList

        defaults to removing the two maximal and minimal values, changing
        the paramater minmax changes how many to remove on each end must be nonnegative
        there must also be at least 2*minmax + 1 frames in the ImageList

        returns a numpy array with the resulting data, up to suer to pack in FITS file'''

        n = len(self._list) - 2*minmax   # the number of frames involved in the average
        return self.sumAll(minmax) / n   # sum up the frames with minmax reject, and divide by the number of effective frames


    def sumAll(self, minmax=2):
        '''compute the sum of all images in the ImageList
        minmax is the number of values to leave off at the minimum and maximum ends
        set to 0 for no minmax reject

        Returns a numpy array with the resulting data, up to user to pack in a FITS file'''
        if minmax < 0:
            raise ValueError('minmax must be non-negative')
        if len(self._list) <= 2*minmax:
            raise ValueError('must have at least {0} items in ImageList with minmax={1}, only has {2}'.format(2*minmax+1,minmax,len(self._list)))
        #create a 3-d array with the first axis along the frames
        all = numpy.array(map(lambda im: im[0].data, self._list))
        all.sort(0) #sort along frame axis , so we can reject the min and maxes

        #now add them together to get the sum, leaving off the mins and maxes
        total = numpy.add.reduce(all[minmax:-minmax]) #reduce the add operation along the z-axis to get the sum of the images
```

```
        return total

    def updateHeaders(self,newHeads={}, **kwargs):
        '''update headers in all images with the key-value pairs supplied'''
        #this would be a lot easier to do with pyfits 3.1, but 2.3 is the
            version supplied
        #with Red Hat, so we are going to use that, and it should still work
            on future versions
        for header in self.headers():
            #since the old version of update with pyfits 2.3 only handles one
                at a time we need another loop
            for (key, value) in chain(newHeads.items(), kwargs.items()):
                header.update(key,value)

    def avCombine(self,minmax=2):
        '''
        Combine all frames in the ImageList into a single frame by using an
            arithmetic mean with optional minmax rejection
        minmax defaults to 2, set to 0 for no minmax rejection, there must be
            at least 2*minmax+1 frames in the ImageList.

        At the moment this simply copies the header from the first frame, but
            we add more sophisticated manipulation of the header later.
        Returns a pyfits.PrimaryHDU
        '''
        result = pyfits.PrimaryHDU( self.averageAll(minmax), self._list[0][0].
            header)
        #NOTE: NAXIS, NAXIS1, NAXIS2, BITPIX, etc. should be updated to match
            the data portion
        result.header.update('NCOMBINE', len(self._list)) #store the number of
             images combined
        result.header.update('IRAF-TLM', getTimeString()) #store the time of
            last modification
        result.header.update('DATE',getTimeString(),'Date␣FITS␣file␣was␣
            generated')
        #TODO add code to modify header, at least mark the average time of
            observations, possibly the total exposure time
        # etc.
        return result

    def normalize(self,block_size=100):
        '''
        normalize all images in the list to have a mean of 1 within the center
             block of sixe block_size x block_size,
        block_size defaults to 100
        '''
        for im in self._list:
            normData(im[0].data,block_size)
        return self

    def closeAll(self):
        '''close all open files'''
        for f in self._list:
            if f:
                f.close()

    #guard code
    def __enter__(self):
        '''simply return self, to bind self to variable'''
        return self
    def __exit__(self, exc_type, exc_value, traceback):
        '''close all open files'''
        self.closeAll()
```

```
    #iterators and accessors
    def hdulists(self):
        '''return an iterator over the HDULists in the list'''
        return iter(self._list)
    def hdus(self):
        '''return an iterator over the Primary HDUs'''
        return imap(lambda im: im[0], self._list)
    def headers(self):
        '''return an iterator over the headers of the images'''
        return imap(lambda im: im[0].header,self._list)
    def __iter__(self):
        '''alias for hdus, so default iterator is over hdus'''
        return self.hdus()
    def __reversed__(self):
        '''reversed iterator'''
        return imap(lambda im: im[0],reversed(self._list))
    def __len__(self):
        '''length of the list'''
        return len(self._list)
    def __getitem__(self, idx):
        '''get the PrimaryHDU at the given index'''
        return self._list[idx][0]
    #note that we don't have __setitem__, that is intentional
    def __delitem__(self,idx):
        del self._list[idx] #delete the given item
    def append(self, fname):
        '''add another file to the list, pass in a filename'''
        self._list.append(pyfits.open(fname))

    #arithmetic operations on other images or contants

    def isubImage(self,other):
        '''subtract another image from all images in the ImageList inplace,
            other should be either
        a PrimaryHDU, or ImageHDU'''
        for im in self._list:
            im[0].data -= other.data
    def isubVal(self, other):
        '''subtract a constant value, or array from all images in the
            ImageList inplace, other should be int, or float, ndarray, etc.'''
        for im in self._list:
            im[0].data -= other
    def __isub__(self,other):
        '''subtract an image or a constant from all frames in ImagList'''
        if isinstance(other,pyfits.PrimaryHDU) or isinstance(other, pyfits.
            ImageHDU):
            self.isubImage(other)
        else:
            self.isubVal(other)
        return self

    def idivVal(self, other):
        '''divide each image by a constant value inplace, other should be
            something that a numpy array can be divided by'''
        for im in self._list:
            im[0].data /= other
    def idivImage(self, other):
        '''divide by another HDU'''
        for im in self._list:
            im[0].data /= other.data
    def __idiv__(self,other):
```

```
        '''divide all images by something, either a number, numpy array, or HDU'''
        if isinstance(other, pyfits.PrimaryHDU) or isinstance(other, pyfits.ImageHDU):
            self.idivImage(other)
        else:
            self.idivVal(other)
        return self

    def saveInPlace(self):
        '''save all of the images in the imagelist back to their original locations'''
        for frame in self._list:
            frame.writeto(frame.filename(),clobber=True)
    def saveToPath(self,path):
        ensure_dir(path) #make sure path is directory, or make it if it doesn't exist
        for frame in self._list:
            frame.writeto(os.path.join(path,os.path.basename(frame.filename()))) #same image


    def saveIndexed(self, baseName):
        '''save the images as the basename appended by an index starting with zero
        this is useful for renaming the files when saving them, each file is saved as baseName+i where i is the index with enough leading zeros that all images use the same number of digits'''
        digitsNeeded = len( str( len(self._list) ) ) #get the lenght of the string of the length of the list
        count = 0
        for frame in self._list:
            frame.writeto(baseName + str(count).zfill(digitsNeeded)+".fit")
            count += 1
    #convenience methods for calibration:

    def subZero(self,zero):
        '''subtract a zero from all of the images in place and return self
        zero should be the path to a zero frame
        NOTE: also zerocor header headers'''
        datestr = getTimeString("%B %d %H:%M") #get string of current date
        with pyfits.open(zero) as zeroFrame:
            for frame in self:
                frame.data -= zeroFrame[0].data
                frame.header.update('ZEROCOR','{0} Zero Image is {1}'.format(datestr,zero))
        return self

    def subDark(self, dark):
        '''subtract dark from all the images in place and return self

        dark shoulb be the path to a dark frame
        NOTE: also updates headers'''
        datestr = getTimeString("%B %d %H:%M") #get string of current date
        with pyfits.open(dark) as darkFrame:
            for frame in self:
                #scale dark to the exposure time
                #and subtract from frame for all frames
                frame.data -= darkFrame[0].data * float(frame.header['EXPTIME'])
                frame.header.update('DARKCOR','{0} with Dark frame {1}'.format(datestr,dark))
        return self
```

```
    def divFlat(self, flat):
        '''divide flat from all the images in place and return self

        flat should be the path to a flat frame
        NOTE: also update FLATCOR header'''
        datestr = getTimeString("%B␣%d␣%H:%M")
        with pyfits.open(flat) as flatFrame:
            for frame in self:
                frame.data /= flatFrame[0].data
                frame.header.update('FLATCOR','{0}␣with␣Flat␣frame␣{1}'.format
                    (datestr,flat))
        return self



def makeZero(*fnames,**kwargs):
    '''
    Take the input frames,(which we assume to be zero or bias frames) as
        strings containging filnames
    and combine them into a master Zero. The exact behaviour depends on the
        following optional keyword arguments

    output -- if provided this is the path to write the resulting zero to, if
        absent makeZero will returning the resulting PrimaryHDU
    minmax -- if provided will set how many data points to remove from the top
         and bottom of the distribution, defaults to 2
    '''
    minmax = kwargs.get('minmax',2)  #get minmax with default of 2
    with ImageList(*fnames) as imList:
        Zero = imList.avCombine(minmax=minmax)
    Zero.header.update('imagetyp','zero') #make sure imagetyp is zero
    if 'output' in kwargs:
        Zero.writeto(kwargs['output'],clobber=True)
    else:
        return Zero #otherwise return the result for the client to deal with

def applyZero(zero_path, *fnames,**kwargs):
    '''apply a zero to one or more frames, zero_path and fnames should both be
         filenames

    if save_path is supplied and not None then all the frames are saved into
        the folder save_path with the same
    basename they had before. If save_inplace is supplied and not false, then
        the images are saved in place with the zero correction'''
    imlist = ImageList(*fnames)
    imlist.subZero(zero_path)
    #mark what we have done in the headers
    #TODO write to logger, we need to figure out the best way to configure
    #a logger for redrovor
    # add ccdproc header:
    imlist.updateHeaders({ 'CCDPROC':'{0}␣CCD␣processing␣done'.format(datestr)
        , })

    #if a path was given, then write the processed files with the same name
        into that path, otherwise
    # save in place
    if 'save_path' in kwargs and kwargs['save_path']:
        imlist.saveToPath(kwargs['save_path'])
        imlist.closeAll()  #clean up
    elif 'save_inplace' in kwargs and kwargs['save_inplace']:
        #save the files in place
        imlist.saveInPlace()
```

```
        imlist.closeAll() #clean up
    else:
        return imlist #let the client do something with it

def makeDark(*fnames, **kwargs):
    '''
    Take the input frames,(which we assume to be dark frames) as strings
        containging filnames
    and combine them into a master Dark. The exact behaviour depends on the
        following optional keyword arguments

    output -- if provided this is the path to write the resulting dark to, if
        absent makeDark will returning the resulting PrimaryHDU
    minmax -- if provided will set how many data points to remove from the top
        and bottom of the distribution, defaults to 2
    zero -- if provided, the filename of the zero frame to apply first,
        otherwise assumes that zero correction has already been done
    '''
    minmax = kwargs.get('minmax',2)
    with ImageList(*fnames) as imlist:
        if 'zero' in kwargs:
            #subtract zeroFrame if supplied
            imlist.subZero(kwargs['zero'])
        #now divide all images by their exposure time for scaling
        for frame in imlist:
            frame.data /= float(frame.header['EXPTIME'])
        Dark = imlist.avCombine(minmax=minmax)
    #now update the heaers
    Dark.header.update('imagetyp','dark')
    if 'zero' in kwargs:
        #add header for zero
        Dark.header.update('ZEROCOR','{0} Zero Images is {1}'.format(
            getTimeString('%x %X'), kwargs['zero']))
    if 'output' in kwargs:
        Dark.writeto(kwargs['output'],clobber=True)
    else:
        return Dark

def applyDark(dark_path,*fnames, **kwargs):
    '''
    apply a dark to one or more frames, dark_path and fnames should both be
    filenames if save_path is supplied and not None then all the frames are
    saved into the folder save_path with the same basename they had before.
    If save_inplace is supplied and not false, then the images are saved in
    place with the zero correction
    '''
    imlist = ImageList(*fnames)
    imlist.subDark(dark_path)
    datestr = getTimeString('%x %X')
    imlist.updateHeaders(ccdproc='{0} CCD Processing done'.format(datestr))
    if 'save_path' in kwargs and kwargs['save_path']:
        imlist.saveToPath(kwargs['save_path'])
        imlist.closeAll()  #clean up
    elif 'save_inplace' in kwargs and kwargs['save_inplace']:
        #save the files in place
        imlist.saveInPlace()
        imlist.closeAll() #clean up
    else:
        return imlist #let the client do something with it


def makeFlat(*fnames, **kwargs):
    '''
```

```
    Take the input frames,(which we assume to be flat frames of the same
        filter) as strings containging filnames
    and combine them into a master Flat. The exact behaviour depends on the
        following optional keyword arguments

    output -- if provided this is the path to write the resulting flat to, if
        absent makeFlat will returning the resulting PrimaryHDU
    minmax -- if provided will set how many data points to remove from the top
         and bottom of the distribution, defaults to 2
    zero -- if provided, the filename of the zero frame to apply first,
        otherwise assumes that zero correction has already been done
    dark -- if provide, the filename of the dark frame to apply first,
        otherwise assumes that dark correction has already been done
    '''
    minmax = kwargs.get('minmax',2)
    with ImageList(*fnames) as imlist:
        if 'zero' in kwargs:
            imlist.subZero(kwargs['zero'])
        if 'dark' in kwargs:
            imlist.subDark(kwargs['dark'])
        imlist.normalize()  #normalize the flats
        Flat = imlist.avCombine(minmax=minmax)
    Flat.header.update('imagetyp','flat')
    if 'zero' in kwargs:
        Flat.header.update('ZEROCOR','{0}␣Zero␣Image␣is␣{1}'.format(
            getTimeString('%x␣%X'),kwargs['zero']))
    if 'dark' in kwargs:
        Flat.header.update('DARKCOR','{0}␣Dark␣Image␣is␣{1}'.format(
            getTimeString('%x␣%X'),kwargs['dark']))
    if 'output' in kwargs:
        Flat.writeto(kwargs['output'],clobber=True)
    else:
        return Flat

def applyFlat(flat_path,*fnames, **kwargs):
    '''
    apply a flat to one or more frames, flat_path and fnames should both be
    filenames if save_path is supplied and not None then all the frames are
    saved into the folder save_path with the same basename they had before.
    If save_inplace is supplied and not false, then the images are saved in
    place with the zero correction
    '''
    imlist = ImageList(*fnames)
    imlist.divFlat(flat_path)
    datestr = getTimeString("%x␣%X")
    imlist.updateHeaders(ccdproc='{0}␣CCD␣Processing␣done'.format(datestr))
    if 'save_path' in kwargs and kwargs['save_path']:
        imlist.saveToPath(kwargs['save_path'])
        imlist.closeAll() #clean up
    elif 'save_inplace' in kwargs and kwargs['save_inplace']:
        imlist.saveInPlace()
        imlist.closeAll()
    else:
        return imlist


def normData(imageData, block_size=100):
    '''normalize the data in the imageData to the mean in the block_size x
        block_size square
    note: this modifies imageData inplace'''
    originy = int((imageData.shape[0] - block_size)/2)
    originx = int((imageData.shape[1] - block_size)/2)
```

```
    avg = numpy.average(imageData[originy:originy+block_size,originx:originx+
        block_size])
    imageData /= avg #divide by the average of the center to normalize
    return imageData
```

Listing A.5 wcs.py

```
#!/usr/bin/python

import os
from subprocess import Popen
from collections import namedtuple


SOLVE_PATH = "/usr/local/astrometry/bin/solve-field"

Coords = namedtuple('Coords',['ra','dec'])

def astrometrySolve(*fnames,**kwargs):
    '''use the framework from astrometry.net to apply
    world coordinate systems to the files located at fnames'''
    with open(os.devnull, 'w') as dnull:
        proc = Popen(buildArgList(fnames,kwargs),stdout=dnull,stderr=dnull )
    return proc.wait() #wait until it completes, we may want to do some
        threading so that we can do more than one at at a time, but not all of
        them at once
    #which would fill up memory really fast

def buildArgList(fnames,args):
    '''build a string for the options to the solve-field command, this should
        not be
    used directly, but as a helper for astrometrySolve'''
    result = [SOLVE_PATH]
    if 'options' in args:
        result.extend(args['options'])
    if 'guess' in args:
        ra,dec = args['guess']
        radius = args.get('radius',1) # in degrees
        result.extend(['--ra',str(ra),'--dec',str(dec),'--radius',str(radius)
            ])
    if 'lowscale' in args:
        result.extend(['--scale-low',str(args['lowscale'])])
    if 'highscale' in args:
        result.extend(['--scale-high',str(args['highscale'])])
    if 'outdir' in args:
        result.extend(['--dir',args['outdir']])
    if args.get('isfits',True):
        result.append('--fits-image')
    result.extend(['--no-plots','--no-fits2fits']) #disable making plots and
        sanitizing fits files
    result.extend(fnames)
    return result
```

Listing A.6 obsRecord.py

```
#!/usr/bin/python

import obsDB
import pyfits
import frameTypes
```

```
import re

from fitsHeader import isFits, getObjectName

import os

import logging
import traceback

logger = logging.getLogger("Rovor.recordobs")

dateRegex = re.compile(r'(\d{4}-\d{2}-\d{2})T.*')

def recordObservation(fitsHeader,fname=''):
    '''record the information contained in the header to the online database
    and optionally the filename passed as fname'''
    objName = getObjectName(fitsHeader)
    obj = obsDB.obj_get_or_add(objName)
    logger.info(obj)
    objid = obj['obj_id']
    ffilter = fitsHeader['FILTER']
    exptime = fitsHeader['EXPTIME']
    temp = fitsHeader['CCD-TEMP']
    utdate = dateRegex.match(fitsHeader['DATE-OBS']).group(1)

    obsDB.newObservation(objid,utdate,ffilter,exptime,temp,fname=os.path.
        realpath(fname))
    return

def recordDir(dir):
    '''record information for all fits files of images in
    the given directory and subdirectories'''
    logger.info("Recording observations in "+dir)
    obsDB.login()
    for root, dirs, files in os.walk(dir):
        logger.info("root = "+root)
        for f in files:
            fullPath = os.path.join(root,f)
            if isFits(fullPath):
                #logger.info("Attempting to record observation for "+f)
                try:
                    header = pyfits.getheader(fullPath)
                    if frameTypes.getFrameType(header) != 'object':
                        continue
                    recordObservation(header,fullPath)
                except Exception as e:
                    logger.error(traceback.format_exc())
                    break #keep going and record everything else
```

A.1.1 Photometry

Listing A.7 photometry/__init__.py

```
from daophot import phot
from irafmod import init
from lightcurves import makeLightCurves

__all__ = ['daophot','params','irafmod']
```

Listing A.8 photometry/irafmod.py

```
import os
import tempfile
from decimal import Decimal
from redrovor.util import workingDirectory

DEFAULT_IRAF_DIR = '/home/iraf'  #default directory to start iraf in

_initialized = False

class InitializationError(Exception):
    '''Error raised when a function of this module
    is called before init() has been called'''
    def __init__(self,value=""):
        self.value = value
    def __str__(self):
        return 'Call attempted before init() was called: '+repr(self.value)

def init(iraf_dir=DEFAULT_IRAF_DIR):
    '''Initialize IRAF for use in photometry
    @param iraf_dir The home directory for iraf, i.e. where login.cl
    and uparm are located, slightly annoying that we need this, but not
    much we can do about it'''
    global _initialized
    if _initialized:
        #already initialized no need to run again
        return
    global iraf
    global yes
    global no
    with workingDirectory(iraf_dir):
        from pyraf import iraf
        yes = iraf.yes
        no = iraf.no
        #now load the packages we need
        iraf.noao()
        iraf.digiphot()
        iraf.daophot()
        iraf.obsutil()

        _initialized = True
        return

def check_init(error_msg="Not initialized"):
    '''check that the irafmod module has been initialized'''
    if not _initialized:
        raise InitializationError(error_msg)
```

Listing A.9 photometry/params.py

```
'''this module takes care of setting up paramaters
for iraf tasks'''

from calc_params import getAverageFWHM, background_data

from redrovor.coords import Coords, RA_coord, Dec_coord

import irafmod


class Params(dict):
    '''class to take care of holding paramaters, this is more abstract
```

```
    and is intended as a superclass for classes that can actually set the
    IRAF paramaters, it is sort of a wrapper around a dictionary'''
    def __init__(self,observ,**kwargs):
        #start with default options
        #TODO some of these are dependent on the system
        # we should make a way to abstract those part out into
        # a seperate object for system-dependent permanent settings
        defaults = {
            'aperture_ratio':1.2,
            'annulus_ratio':4,
            'dannulus_ratio':3,
            'zmag': 25,
            'datamax': observ.datamax, #point where CCD saturates
            #header keywords
            'obsdate':observ.date_key,
            'obstime':observ.time_key,
            'exposure':observ.exp_key,
            'airmass':observ.air_key,
            'filter':observ.filt_key,
            'epoch': observ.epoch_key,
            'ra_key': observ.ra_key,
            'dec_key':observ.dec_key,
            'observat':observ.name, #this needs to be set up for the right
                telescope
            'otime': 'hjd', #header keyword for the time to output in phot
                files
        }
        super(Params,self).__init__(defaults)
        self.update(kwargs)

    def __call__(self,*args,**kwargs):
        '''calling the method simply forwards the call to
        applyParams with the supplied arguments, it is expected
        that the subclass will implement applyParams, it is not
        implemented in this class'''
        self.applyParams(*args,**kwargs)

    @property
    def aperture(self):
        '''return the aperture in scale units'''
        return self['aperture_ratio']*self['fwhm']
    @property
    def annulus(self):
        '''return a tuple of the inner and outer annuli in scale units'''
        return self['annulus_ratio']*self['fwhm']
    @property
    def dannulus(self):
        '''return a tuple of the inner and outer annuli in scale units'''
        return self['dannulus_ratio']*self['fwhm']

    @property
    def datamax(self):
        '''return the maximum good data value'''
        return self.get('datamax','INDEF')
    @property
    def datamin(self):
        '''return the minimum good data value'''
        if 'datamin' in self:
            return self['datamin']
        elif 'background' in self and 'sigma' in self:
            #minimum good data is 6 sigma below background
            return self['background'] - 6.0*self['sigma']
```

```
        else:
            return 'INDEF'
    @property
    def cbox(self):
        '''return size for center box, if not explicetly set use
        maximum of 5 and 2*fwhm'''
        return self.get('cbox',max(5.0, 2.0*self['fwhm']))

class DAO_params(Params):
    '''class to take care of setting up paramaters for dao photting'''
    def __init__(self,observat,**kwargs):
        super(DAO_params,self).__init__(observat,fitfunction='gauss',
            readnoise=observat.readnoise,gain=observat.gain)
        self.update(kwargs)

    def applyParams(self):
        '''apply paramaters for daophot'''
        irafmod.check_init("DAO_params.applyParams")
        iraf = irafmod.iraf

        #photpars
        iraf.photpars.aperture = self.aperture
        iraf.photpars.zmag = self['zmag']
        #set world coordinates as input for phot
        iraf.phot.wcsin="world"
        #datapars
        iraf.datapars.fwhmpsf = self['fwhm']
        iraf.datapars.sigma = self.get('sigma',0)
        iraf.datapars.datamax = self.datamax
        iraf.datapars.datamin = self.datamin
        iraf.datapars.obstime = self['otime']
        iraf.datapars.exposure = self['exposure']
        iraf.datapars.airmass = self['airmass']
        iraf.datapars.filter = self['filter']
        #centerpars
        iraf.centerpars.cbox = self.cbox
        iraf.centerpars.calgorithm = self.get('calgorithm','centroid')
        #fitskypars
        iraf.fitskypars.annulus = self.annulus
        iraf.fitskypars.dannulus = self.dannulus
        iraf.fitskypars.salgorithm = self.get('salgorithm','mode')
        #daopars
        iraf.daopars.psfrad = 4.0*self['fwhm']+1.0
        iraf.daopars.fitrad = self.aperture
        #psfpars
        iraf.psf.function = self['fitfunction']
        #make sure we are using default logical coordinate system
        #for everything except the phot command
        iraf.daophot.wcsin="logical"
        iraf.daophot.wcsout="logical"
        iraf.daophot.verify=iraf.no
        # setjd paramaters
        iraf.observatory.observatory = self['observat']
        iraf.setjd.date = self['obsdate']
        iraf.setjd.time = self['obstime']
        iraf.setjd.observatory = self['observat']
        iraf.setjd.exposur = self['exposure']
        iraf.setjd.epoch = self['epoch']
        iraf.setjd.ra = self['ra_key']
        iraf.setjd.dec = self['dec_key']
```

```
def getDAOParams(observ,imageName, coord_file, target_coords=None, size=100,
    **kwargs):
    '''calculate the paramaters for performing daophot for an image
    using the coordinate file and possibly the coordinate of the target,
    if target_coords is None or not supplied, we assume that the target is the
    first set of coordinates in the coordinate file.

    The target coordinates are used to estimate the background and background
        sigma.
    size is the size of the sampling box for getting sigma and background'''
    if target_coords is None:
        target_coords = parse_first_coords(coord_file)
    params = DAO_params(observ,**kwargs)
    params['fwhm'] = getAverageFWHM(imageName, coord_file)
    params['background'], params['sigma'] = background_data(imageName,
        target_coords,size)
    return params

def parse_first_coords(coord_file):
    '''parse the coordinates of the first object
    in the coordinate file and return a Coords object'''
    with open(coord_file) as cf:
        line = cf.readline()
        while line and (line.isspace() or line.startswith('#')):
            #skip over blank lines and comments
            line = cf.readline()
    if not line:
        raise Exception("Unable to parse coordinates") #TODO use better
            exception type
    rastr, decstr = line.split()[0:2] #assume seperationg by whitespace and no
         internal whitespace
    ra = RA_coord.fromStr(rastr)
    dec = Dec_coord.fromStr(decstr)
    return Coords(ra,dec)
```

Listing A.10 photometry/calc_params.py

```
'''module to calculate paramaters for photometry, this
should only be used internally by the phot package'''

import pywcs
import numpy
import pyfits
from scipy.interpolate import UnivariateSpline as uniSpline
from scipy.stats import tstd


import irafmod
import re

def getBox(image, center, size=100):
    ''' get a box centered at \p center with a size of \p size
    pixels.

    @param image the pyfits HDU object of the image to find the coordinates
        for
    @param center a coords.Coords object containing the WCS
         coordinates for the center of the box
    @param size the length of one side of the box in pixels
    '''
```

```
    #TODO figure out what to do about the order of coordinates (ra,dec) or (
        dec,ra) we don't
    #seem to have the right paramaters set

    mywcs = pywcs.WCS(image.header) #get WCS object
    ra,dec = center
    #we need to convert to decimal degrees before doing transformation
    r = float(ra.toDegrees())
    d = float(dec.toDegrees())
    x,y = mywcs.wcs_sky2pix([(r,d)],0)[0]  #perform conversion
    #get the bottom and left coordinates
    bottom = int(y-size/2)
    left = int(x-size/2)

    #numpy arrays are column major, so we give y vallues first
    return image.data[bottom:bottom+size, left:left+size]


def im_histogram(box, bins=1000):
    '''get the histogram of a numpy array, return (x,y) where
    x is the midpoints of the bins, and y is the number of pixels in each bin
    @param bins the number of bins to use'''

    y,bins = numpy.histogram(box,bins=bins)
    x = (bins[1:]+bins[:-1])/2  #compute the average of each consecutive pair
        of elements
    return (x,y)


class MultiplePeakError(Exception):
    '''more than one peak in the distribution'''
    pass

class NoPeakFoundError(Exception):
    '''no peak in distribution'''
    pass

def center_and_fwhm(x,y,bins=1000):
    '''calculate the full width half max of the function y(x),
    and get the center of the thing
    @returns (center, fwhm) where center is the x value of the center of the
        peak and fwhm is
    the full width half max of the peak

    we won't actually use this for now, but we will keep it case we want it
        later'''
    midx = numpy.argmax(y) #get index of maximum
    half_max = y[midx]/2 #get half the maximum
    center = x[midx] #get the value of the center

    s = uniSpline(x,y-half_max)  #create a spline of the data
    roots = s.roots()  #get roots of the spline, i.e. place of half-max
    if len(roots) > 2:
        #too many roots
        raise MultiplePeakError("There␣appears␣to␣be␣more␣than␣one␣peak")
    elif len(roots) < 2:
        raise NoPeakFoundError("There␣doesn't␣appear␣to␣be␣a␣proper␣peak")
    else:
        return (center, abs(roots[1]-roots[0]))

def background_data(imageName, center_coords, size=100):
    '''calculate the background value and standard deviations
    and return as a tuple (background, sigma)
    @param imageName the path to the image to get the data for
```

```
    @param center_coords the coordinates to center the sampling box around,
        probably the coordinates of the target object
    @param size the size of the sampling box in pixels
    @returns a modal value with bins of size 1 count and a trimmed standard
        deviation reject values more than twice the background value'''
    with pyfits.open(imageName) as im:
        box = getBox(im[0], center_coords,size)
    bins = numpy.arange(box.min(), box.max(),1) #use bins of size 1 ranging
        from the minimum to maximum values of the sample box
    x,y = im_histogram(box, bins=bins)
    #compute the location of the peak of the histogram
    midx = numpy.argmax(y)
    center = x[midx]
    sigma = tstd(box, [0,2*center]) #trim to twice the the peak value
    return (center, sigma)

def getAverageFWHM(image,coord_file):
    '''calculate the average Full Width Half Max for the objects in image
    at the coords specified in coord_file
    the coordinates in coord_file should be in the same world coordiantes
    as the WCS applied to the image'''

    if not irafmod._initialized:
        raise irafmod.InitializationError()
    psfmeasure = irafmod.iraf.psfmeasure
    #set up all paramaters
    psfmeasure.coords = "mark1"
    psfmeasure.wcs = "world"
    psfmeasure.display = irafmod.no
    psfmeasure.size = "FWHM"
    psfmeasure.imagecur = coord_file
    psfmeasure.graphcur = '/dev/null' #file that is empty by definition
    res = psfmeasure(image,Stdout=1)[-1]  #get last line of output
    match = getAverageFWHM.numMatch.search(res)
    return float(match.group(1))

getAverageFWHM.numMatch = re.compile(r'(\d+(\.\d+)?)')
```

Listing A.11 photometry/daophot.py

```
'''module for actually performing daophot'''

import irafmod
from params import getDAOParams
from redrovor.utils import workingDirectory
from redrovor.observatories import ROVOR


def phot(imageName, output_dir,coordFile, target_coords=None,
    sample_size=100,params=None,observat=ROVOR,**kwargs):
    '''perform daophot on imageName with the supplied
    coordinate file, and optionally the target coordinates, which
    defaults to the first coordinates in coordFile

    sample_size is the size of the sample box used for background measurement
    kwargs are additional args to pass to constructor for params (or update
        params)'''

    #first ensure that irafmod has been initialized
    irafmod.check_init("unable␣to␣phot")
    daophot = irafmod.iraf.daophot
```

```
    #first get the paramaters we need
    if not params:
        params = getDAOParams(observat,imageName, coordFile,
            target_coords, size=sample_size)
    params.update(kwargs)
    params.applyParams()

    with workingDirectory(output_dir):
        #temporarily change working directory

        irafmod.iraf.setjd(imageName)
        daophot.phot(imageName,coordFile,"default")
        #params: image, photfile, pstfile, maxnpsf
        daophot.pstselect(imageName,"default","default",25)
        #params: imagename photfile pstfile psfimage opstfile groupfile
        daophot.psf(imageName,"default","default","default","default","default
            ",
            interactive=irafmod.no)
        #TODO should we be more sophisticated and do multiple runs of psf
        #along with using nstar and substar to try and get best fit?

        #use nstar for now, but we will make it a seperate function
        #so it is easy to switch out with peak or allstar
        #if we desire later
        do_nstar(imageName)

def do_nstar(imageName):
    '''perform nstar stuff, should not be called by user code
    only a helper for phot'''
    daophot = irafmod.iraf.daophot
    daophot.group(imageName,"default","default","default")
    #we will assume that we don't have any big groups for now,
    #so we don't have to use grpselect
    daophot.nstar(imageName,"default","default","default","default")

def do_allstar(imageName):
    '''perform allstar stuuf, should only be called by phot'''
    irafmod.iraf.daophot.allstar(imageName,"default","default","default","
        default","default")
```

Listing A.12 photometry/lightcurves.py

```
'''module to perform various operations to manipulate
the output from photting specifically related to producing
lightcurves'''

import irafmod
import os
import csv

from cStringIO import StringIO

import logging
logger = logging.getLogger('Rovor.photometry')

def sortphotfiles(folder, suffix=".nst.1"):
    '''sort all of the phot files in \p folder
    by id. They must end in \p suffix, which defaults to
    .nst.1'''
    irafmod.check_init("can't␣sort")
    irafmod.iraf.psort(os.path.join(folder,"*"+suffix),'id')
```

```
    return

#constant holding the list of fields to dump from phot files
FIELD_STR = "id,ifilter,otime,mag,airmass"

def photdump(files, output):
    '''dump photometric information in the given list of
    photometry files into output

    output can be either a file-like object open for writing, or a string,
    if it is a string it is the path to a file, which is then opened in 'w'
    mode.

    photdump returns the (still open) file object when done.

    it dumpst the following fields in the given order:

    id
    ifilter
    otime (observation time)
    magnitude
    airmass
    '''
    irafmod.check_init("can't␣dump")
    iraf = irafmod.iraf
    if isinstance(output, str):
        output = open(output,'w')
    for pfile in files:
        iraf.pdump(pfile, FIELD_STR, iraf.yes, Stdout=output)
    return output

def photdump_all(globber,output):
    '''similar to photdump, except that instead
    of a list of files, it takes a string, which is a
    glob expression for the files to use,
    ex. *.nst.1 '''
    irafmod.check_init("can't␣dump")
    iraf = irafmod.iraf
    if isinstance(output,str):
        output = open(output,'w')
    iraf.pdump(globber,FIELD_STR,iraf.yes,Stdout=output)
    return output

def splitdump(dumpfile,prefix,delim='␣'):
    '''split a phot dump into seperate files for each id and filter
    combination.

    dumpfile is a file like object open for reading in text mode,
    unfortunately it would be rather difficult to also support opening
    the file for you. Sorry '''
    fdict = {}
    reader = csv.reader(dumpfile,delimiter=delim,skipinitialspace=True)
    try:
        for line in reader:
            #in python three we could write it like this:
            #starid, filt, *rest = line
            logger.debug(str(line))
            starid, filt = line[0:2]
            rest = line[2:]
            if (starid, filt) not in fdict:
                fdict[(starid,filt)] = open(prefix+"_"+filt+"_"+starid+".lc",'
                    w')
            fdict[(starid,filt)].write(delim.join(rest)+"\n")
    finally:
```

```
        for f in fdict.values():
            #close all the open files
            f.close()
def makeLightCurves(photFiles, prefix):
    '''Create light curves for an object.

    photFiles is a list of photometry files, such as nst files
    which will be dumped to create the light curves.

    prefix is the prefix to save the light curves to. This should be the
    full path to the folder to save it in, and probably the name of the target
        or field.
    The prefix will be appended with the filter and the object id and the
        suffix .lc.'''
    #how hard would it be to parallelize this and pipe the result of photdump
        to the input of
    # splitdump
    buffer = StringIO()
    photdump(photFiles, buffer)
    buffer.reset() #reset to beginning of 'file' for reading
    splitdump(buffer, prefix)
    buffer.close()
    return True
```

A.1.2 Passes

Listing A.13 firstpass.py

```
#!/usr/bin/python

import sys
import datetime
import os
import os.path as path
import shutil
import tempfile

import json

import logging

from glob import glob
from collections import defaultdict

import pyfits

import updateHeaders
import frameTypes
import process
from fitsHeader import splitByHeader

from utils import writeListToFileName, getTimeString

#TODO allow more flexibility in where the output files are stored
ProcessedFolderBase = '/data/Processed'
MasterCalFolder = '/data/Calibration'


def genDate(date):
    return date.strftime('/%Y/%b/%d%b%Y').lower()
```

```
def createResultFolder(date):
    '''given a datetime.date create a new folder to hold the resulting
processed images.'''
    folderName = ProcessedFolderBase+genDate(date)
    if not path.isdir(folderName):
        os.makedirs(folderName)  #create folder in format ddmonyyyy inside
            folder for month inside folder for year
    return folderName

def relocateFiles(fileList, destFolder):
    '''takes a list of filenames (fileList) and adds the destFolder to the
        beginning of them, returning a new list'''
    return [ path.join(destFolder, path.basename(fname)) for fname in fileList
        ]

class FirstPassProcessor:
    '''Class to handle image processing for a folder, we use a class to
make it easier to keep track of the state'''

    def __init__(self, rawFolder, processedFolder = None):
        '''initialize the processor in the folder containing the raw data'''

        # set up logger
        self.logger = logging.getLogger('Rovor.FirstPassProcessor_{0}'.format(
            id(self)))

        self.rawFolder = rawFolder
        #TODO figure out a robust way to determine the date of observation
        #for now we will simply look at the date of observation for the first
        #fits file

        self._findFrames()  #find frames
        header = pyfits.getheader(self.frames[0])
        self.obsDate = datetime.datetime.strptime(header['date-obs'], '%Y-%m-%
            dT%H:%M:%S.%f').date()
        #create the folder to store results in
        self.processedFolder = processedFolder or createResultFolder(self.
            obsDate)


        self.zeroFrame = None
        self.darkFrame = None
        self.flatBase = None
        self.flatFrames = {}
        self.frameTypes = None
        self.objects = None
        return
    def _findFrames(self):
        '''find  all fits files in the folder (anything ending with .fits, .
            fit, .FIT, or .fts'''
        self.logger.info('Looking for frames...')
        validExtensions = ['.fits','.fit','.FIT','.fts']
        self.frames=list()
        for ext in validExtensions:
            self.frames.extend( glob(self.rawFolder+'/*'+ext) )
        return self
    def updateHeaders(self,inplace=True):
        '''update the headers for all the frames in the folder
always works in place'''
        self.logger.info('Updating Headers...')
        for frame in self.frames:
            updateHeaders.updateFrame(frame)
        return self
```

```
def buildLists(self):
    self.logger.info("Building␣Lists")
    self.frameTypes = frameTypes.getFrameLists( self.frames ) #get frame
        types
    self.objects = frameTypes.makeObjectMap( self.frameTypes['object'] )
    #save a cache of the frame info to speed up future uses of the
        FirstPassProcessor
    with open(path.join(self.rawFolder,'frameInfo.json'),'w') as f:
        json.dump([self.frameTypes,self.objects],f)
    return self
def ensure_frameTypes(self):
    '''ensure that frameTypes is set'''
    if not self.frameTypes:
        frameInfoPath = path.join(self.rawFolder,'frameInfo.json')
        #if we have a previously made file load that
        if path.isfile(frameInfoPath):
            with open(frameInfoPath,'r') as f:
                self.frameTypes, self.objects = json.load(f)
        else: #otherwise build the lists
            self.logger.warning('Type␣lists␣were␣not␣previously␣made,␣
                making␣them␣now')
            self.buildLists()
def makeZero(self):
    #insure that we have the frame types already
    self.ensure_frameTypes()
    self.logger.info('Making␣Zero')
    self.zeroFrame = path.join(self.processedFolder, 'Zero.fits')
    self.logger.info("Set␣zeroFrame␣to␣"+ self.zeroFrame)
    process.makeZero(*self.frameTypes['zero'], output=self.zeroFrame)
    return self
def ensure_zero(self):
    '''check to see if we already have a zeroFrame location,
    if not look for a zero in the processed folder, if we still can't find
        one
    then run makeZero'''
    self.zeroFrame = self.zeroFrame or path.join(self.processedFolder, '
        Zero.fits')
    if not path.isfile(self.zeroFrame):
        #the zero hasn't been made yet
        self.makeZero()
def makeDark(self):
    self.logger.info('Making␣Dark')
    self.ensure_frameTypes()
    self.ensure_zero()  #make sure we have a zero to use
    self.darkFrame = path.join(self.processedFolder,'Dark.fits')
    #apply zeros to darks
    process.makeDark(*self.frameTypes['dark'],zero=self.zeroFrame,output=
        self.darkFrame)
    return self
def ensure_dark(self):
    '''check to see if we already have a zeroFrame location,
    if not look for a zero in the processed folder, if we still can't find
        one
    then run makeZero'''
    self.darkFrame = self.darkFrame or path.join(self.processedFolder, '
        Dark.fits')
    if not path.isfile(self.darkFrame):
        #the dark hasn't been made yet
        self.makeDark()
def makeFlats(self):
    self.logger.info('Making␣Flats')
    self.ensure_frameTypes()
```

```
        self.ensure_zero()
        self.ensure_dark()
        flatBase = path.join(self.processedFolder, 'Flat')
        flats = splitByHeader(self.frameTypes['flat'], 'filter')
        for filter in flats:
            outName = "{0}_{1}.fits".format(flatBase, filter)  #name is the
                base flat name plus the filter type
            process.makeFlat(*flats[filter], zero=self.zeroFrame, dark=self.
                darkFrame, output=outName)
            self.flatFrames[filter] = outName
        #TODO should we copy the flats to calibration folder?
        return self

    def zero_and_dark_subtract(self):
        '''subtract zeros and darks from image files
        and save in the processed folder'''
        self.logger.info("Subtracting␣zeros␣and␣darks")
        #ensure we have everything we need
        self.ensure_frameTypes()
        self.ensure_zero()
        self.ensure_dark()
        for (obj, flist) in self.objects.items():
            #iterate over each object
            for (filt, frames) in splitByHeader(flist, 'filter').items():
                baseName = "{0}/{1}{2}-".format(self.processedFolder, obj.
                    replace('␣', '_'), filt)
                with process.ImageList(*frames) as imlist:
                    imlist.subZero(self.zeroFrame)
                    imlist.subDark(self.darkFrame)
                    imlist.updateHeaders(ccdproc='{0}␣CCD␣Processing␣done'.
                        format(getTimeString("%x␣%X")))
                    imlist.saveIndexed(baseName) #save the processed images


    def firstPass(self):
        '''
        perform the first pass of reduction on the folder. This does the
            following
            - Create Master zero and place in Processed folder
            - Create Master dark and place in Processed folder
            - Create Master flats if any and place in Processed folder
            - Apply zero and darks to all object frames and store the
                processed images in Processed folder
        Note that the processed images have not been flat reduced yet, this is
            done in the second pass
        '''
        self.logger.info('Processing␣Directory␣{0}...'.format(self.rawFolder))
        #self.updateHeaders()
        self.buildLists()
        self.makeZero()
        self.makeDark()
        self.makeFlats()
        self.zero_and_dark_subtract()

def doFirstPass(path):
    '''convenience wrapper function for the FirstPassProcessor.firstPass,
        which takes the path
    opens a FirstPassProcessor and calls firstPass'''
    improc = FirstPassProcessor(path)
    improc.firstPass()
```

```
if __name__ == '__main__':
    #optparse is deprecated after python 2.7, but
    #argparse isn't available on python 2.6, which is what
    #we are currently using, if we upgrade to a newer version
    #of python then we should change to argparser

    from optparse import OptionParser
    parser = OptionParser()

    #parser.add_option('-F', '--use-flats', action='store_true', dest='
        useFlats', default=False, help='apply flats when doing the calibration
         (not done by default because flats can be tricky)')

    (options,args)=parser.parse_args()
    if len(args) != 1:
        parser.error("incorrect␣number␣of␣arguments")
    rawDirectory = args[0]
    doFirstPass(rawDirectory) #for now we will just do the first pass
```

Listing A.14 secondpass.py

```
'''module for code to perform the second pass, which consists of:
    * Apply Flats
    * Apply WCS to images

NOTE: we may change the name of this module at a later time'''

import logging
import json
import pyfits
from os import path

import observatories

from utils import findFrames
from frameTypes import getFrameLists
from fitsHeader import getRA, getDec, splitByHeader
from wcs import astrometrySolve

from process import applyFlat

import renamer


class SecondPassProcessor:
    '''Class for taking care of all the processing needed for
    the second pass. Again the name may change, but I couldn't
    think of a good name since Flats and WCS coordinates don't
    have a lot in common except that we are going to do them at
    the same time'''

    def __init__(self,folder):
        '''initalize the class with the path of the folder that we are going
           to process.
        This path will mostly be a subdirectory of the Processed folder'''
        self.folder = folder
        self.logger =  logging.getLogger("Rovor.secondpass")
        self.objects = None
    def buildObjectList(self):
        self.logger.info("Building␣Object␣List")
        frames = findFrames(self.folder)
        frameTypes = getFrameLists( frames ) #get frame types
        self.objects = splitByHeader(frameTypes['object'],'filter')
```

```
        #save a cache of the frame info to speed up future uses of the
            ZeroDarkProcessor
        with open(path.join(self.folder,'filterLists.json'),'w') as f:
            json.dump(self.objects,f)
        return self
    def ensure_objectList(self):
        '''ensure that frameTypes is set'''
        if not self.objects:
            objectPath = path.join(self.folder,'filterLists.json')
            #if we have a previously made file load that
            if path.isfile(objectPath):
                with open(objectPath,'r') as f:
                    self.objects = json.load(f)
            else: #otherwise build the lists
                self.logger.warning('Object␣lists␣were␣not␣previously␣made,␣
                    making␣them␣now')
                self.buildObjectList()
    def neededFilters(self):
        '''get the filters that the object frames are in so that we know which
            filters we need to use for
        flat processing, returns a set'''
        self.ensure_objectList()
        return list(self.objects.keys())
    def applyFlats(self,flatDict):
        '''@brief Apply Flats in \p flatdict to object images in the folder

        For each flat in the dictionary flatDict apply the flats to all object
        images in that filter

        @param[in] flatDict a dictionary mapping the names of filters to paths
        of the flat to use for that filter
        @returns self
        '''
        self.logger.info("Applying␣Flats␣to␣"+self.folder)
        self.ensure_objectList()
        for filt,flat in flatDict.items():
            self.logger.debug("filt={0}".format(filt))
            self.logger.debug("flat={0}".format(flat))
            if filt and filt in self.objects:
                applyFlat(flat,*self.objects[filt],save_inplace=True)
        return self

    def applyWCS(self,observatory = observatories.ROVOR):
        '''apply world coordinate systems to the images using data from
        the observatory information to set paramaters to astrometry.net'''
        self.ensure_objectList()
        for frames in self.objects.values():
            for frame in frames:
                header = pyfits.getheader(frame)
                ra = ':'.join(getRA(header))
                dec = ':'.join(getDec(header))
                astrometrySolve(frame,
                    guess=(ra,dec),
                    lowscale=observatory.lowscale,
                    highscale=observatory.highscale,
                    outdir=path.join(self.folder,'WCS')
                )
        #astrometry.net names the new files with .new extension, rename them
        renamer.renameAll(path.join(self.folder,'WCS'),oldExt=".new")

def doSecondPass(path,flatDict):
    '''perform the second pass on images in the given folder'''
```

```
    improc = SecondPassProcessor(path)
    improc.applyFlats(flatDict)
    improc.applyWCS()
```

Listing A.15 thirdpass.py

```
from collections import defaultdict
import os
from os import path
import pyfits
import json

import photometry
from photometry import phot, makeLightCurves
from utils import findFrames
from fitsHeader import normalizedName

import logging
logger = logging.getLogger("Rovor.thirdpass")

#go ahead and initialize the photometry package
photometry.init()

class ThirdPassProcessor:
    '''Class for taking care of all the processing
    for the third, pass.

    For photometry this is kind of overkill,
    but it will make things easier when we add more to this phase
    '''

    def __init__(self, folder):
        '''initialize the class with the path of the folder that we are
        going to process, this is most likley the WCS folder produced by
        the second pass.'''
        self.folder = folder
        self.objects = defaultdict(list)
    def buildObjectList(self):
        '''discover object frames in the folder, and what the targets are'''
        for im in findFrames(self.folder):
            header = pyfits.getheader(im)
            self.objects[normalizedName(header)].append(im)
        #now save the object lists
        with open(path.join(self.folder, 'objectLists.json'),'w') as f:
            json.dump(self.objects, f)
        return self
    def ensure_objectLists(self):
        '''ensure that self.objects is set'''
        if not self.objects:
            objectPath = path.join(self.folder, 'objectLists.json')
            #if we already made the file load it
            if path.isfile(objectPath):
                with open(objectPath,'r') as f:
                    self.objects = json.load(f)
            else:
                logger.info("Creating␣object␣lists")
                self.buildObjectList()
        return self
    def objectNames(self):
        '''Get the normalized names of the objects we
        are dealing with in this folder.'''
        self.ensure_objectLists()
```

```
        return list(self.objects.keys())
    def phot(self,obj_mapping,**kwargs):
        '''
        Phot the frames in the folder

        obj_mapping must be a dict mapping the normalized name of objects
        to a tuple containing the coordinate file and optionally a
        coords.Coords object  containing the coordinates of the target.
        output_dir is the directory to save the output folders in,
        it defaults to a subdirectory of self.folder named "photometry"
        and will be created if it does not already exist.

        all kwargs are passed through to the phot method
        '''
        logger.info("Photting folder: "+self.folder)
        self.ensure_objectLists()
        output_dir = path.join(self.folder,'photometry')
        if not path.isdir(output_dir):
            #only create directory if it does not already exist
            os.makedirs(output_dir)
        for objName, (coordfile,targetCoords) in obj_mapping.items():
            for im in self.objects[objName]:
                try:
                    logger.info("Photting image: "+im)
                    phot(im,output_dir,coordfile,targetCoords,**kwargs)
                except Exception as ex:
                    logger.warning(ex)
                    #continue photting the rest

        return self

    def makeLightCurves(self):
        '''create light curves from the output of the
        photting process.'''
        #this will just wrap a function in redrovor.photometry
        for targ,flist in self.objects.items():
            prefix = path.join(self.folder,'photometry',targ)
            try:
                makeLightCurves(map(self._getNstName,flist),prefix)
            except Exception as ex:
                logger.warning(ex)
                #continue making the rest of the light curves

    def _getNstName(self, fitsPath):
        base = path.basename(fitsPath)
        return path.join(self.folder,'photometry',base+".nst.1")


def doThirdPass(path, obj_mapping,**kwargs):
    '''perform the third pass, for now this just does the photometry,
    although at some later point we may add other processing such as
    combining data for the same object-filter combinations, not that this
    uses the photometry package so changing the photometry package is
    sufficient to change how photometry is done.
    also, additional options to control the phot process can be passed in
    as kwargs
    '''
    proc = ThirdPassProcessor(path)
    proc.phot(obj_mapping,**kwargs)
    proc.makeLightCurves()
```

A.1.3 Utilities

These are modules of *RedROVOR* which are used internally, but are probably not necessary for user code.

Listing A.16 utils.py

```
import os
from datetime import datetime
from glob import glob
from contextlib import contextmanager
import sys

def ensure_dir(path):
    #first see if it is exists
    if os.path.exists(path):
        #now if it is a dir return successfully
        if os.path.isdir(path):
            return
        else:
            #a non-directory file, throw an error
            raise ValueError('Path to non-directory')
    else:
        #attempt to create the path
        os.makedirs(path)

def getTimeString(frmt='%Y-%m-%dT%H:%M:%S'):
    '''get a formatted timeString'''
    return datetime.now().strftime(frmt)

def writeListToFile(ll, ff=sys.stdout, delimeter='\n'):
        '''Write the supplied list to the given file, one element per line,
            with no other delimeters
ll -- The list of items to write
ff -- The file to write to (as in file object)
delimeter -- the delimeter between items in the list'''
        ff.write('\n'.join( str(item) for item in ll))
        return

def writeListToFileName(ll, fname, delimeter='\n'):
        '''write the list to the file given by fname (opens a file object for
            writing) '''
        with open(fname, 'w') as ff:
                writeListToFile(ll,ff,delimeter)
        return

def findFrames(folder):
    '''find  all fits files in the folder (anything ending with .fits, .fit, .
       FIT, or .fts'''
    validExtensions = ['.fits','.fit','.FIT','.fts']
    frames=list()
    for ext in validExtensions:
        frames.extend( glob(folder+'/*'+ext) )
    return frames

def shell_quote(s):
    'quote string to be safe in shell'
    return "'" + s.replace("'",r"'\''") + "'"

@contextmanager
```

```python
def workingDirectory(path):
    '''Create a context manager that temporarily changes into a
    working directory, and gaurantees to return to the original workign
        directory'''
    oldDir = os.getcwd()
    try:
        os.chdir(path)
        yield
    finally:
        os.chdir(oldDir)
    return
```

Listing A.17 renamer.py

```python
#!/usr/bin/python

'''A Module to take care of renaming fits files to a more friendly extension (
    i.e. fit instead of FIT).'''

import os
import fitsHeader

def renameFITS(origFile,newExt=".fit"):
    '''Rename a single fits file to have the extension provided (defaults to .
        fit)
    origFile is a string with the correct path to the file to rename (absolute
        or relative to working directory)
    return true if the rename was successful, false otherwise'''
    if os.path.isfile(origFile):
        newName = os.path.splitext(origFile)[0] + newExt
        try:
            os.rename(origFile,newName)
            return True
        except OSError:
            return False

    else:
        return False

def renameAll(path,newExt=".fit", oldExt=".FIT"):
    '''Rename all FITS files with extension oldExt to extension newExt (
        defaults
    to .fit) which are in directory path. If path is empty or not a directory
    renameAll will silently return without doing anything.
    '''
    if not os.path.isdir(path):
        return
    for f in os.listdir(path):
        if f.endswith(oldExt):
            renameFITS(os.path.join(path,f),newExt)
```

Listing A.18 fitsHeader.py

```python
import os
import os.path
import re

import obsDB
import simbad
```

```
from collections import defaultdict
import pyfits

zeroRE = re.compile(r'([zZ]ero)|([Bb]ias)')
darkRE = re.compile(r'[dD]ark')
flatRE = re.compile(r'[Ff]lat')
objectRE = re.compile(r'([iI]mage)|([Ll]ight)|([oO]bject)')

fitsSuffixes = set(['.fits','.fts','.FIT','.FITS','.fit'])

def isFits(fname):
    '''Determine if the filename is right for a fits file or not'''
    return os.path.isfile(fname) and os.path.splitext(fname)[1] in
        fitsSuffixes

def fitsCheckMagic(fname):
    '''check the magic number to make sure it actually is a fits file'''
    with open(fname, 'rb') as fl:
        return fl.read(len('SIMPLE')) == 'SIMPLE'

def getFrameType(header):
    '''Given the header for a frame determine if it is a
       zero, dark, flat, or image frame using the imagetyp header
       and possibly the exposure time'''
    imtype = header['imagetyp']
    exptime = header['exptime']
    if zeroRE.search(imtype) or exptime == 0:
            return 'zero'
    elif darkRE.search(imtype):
            return 'dark'
    elif flatRE.search(imtype):
            return 'flat'
    elif objectRE.search(imtype):
            return 'object'
    else:
            return None

def getObjectName(header):
    '''get the name of the object in the frame'''
    if 'object' in header:
            return header['object'].replace('_',' ')
    elif 'title' in header:
            return header['title'].replace('_',' ')
    else:
        ra= header.get('objctra','0:0:0').replace(' ',':')
        dec = header.get('objctdec','0:0:0').replace(' ',':')
        try:
            name = obsDB.lookup_name(ra,dec)
            return str(name) #convert from unicode to string
        except obsDB.ObsDBError as e:
            #return 'unknown' #if there was a problem retrieving it is unknown
            #if we don't know the name create a name from RA and dec
            return makeRADecName(header)

def normalizedName(header):
    '''get a normalized name from simbad from the header,

    this is just a convenience method which calls getObjectName and
    then uses simbad to get the "main name" wich is the primary name
    from simbad'''
    name = getObjectName(header)
    return simbad.getMainName(name)
```

```
def getFilter(header):
    if 'filter' in header:
        return header['filter']
    else:
        return 'unknown'

def getRA(header):
    '''get the Right Ascension and return a tuple containing the
    hour, minute and second'''
    if 'objctra' in header:
        return tuple(header['objctra'].split())
    elif 'ra' in header:
        return tuple(header['ra'].split())
    else:
        return None # ra isn't there

def getDec(header):
    ''' get the declination and return a tuple containing the degree,
    arcminute, and arcsecond, or None if no dec is there'''
    if 'objctdec' in header:
        return tuple(header['objctdec'].split())
    elif 'dec' in header:
        return tuple(header['dec'].split())
    else:
        return None

def makeRADecName(header):
    '''If we don't have the name of the object, build a name from the RA and
    dec, but only use hours/degrees and minutes/arcminutes, so that we get a
        single
    name for each object. We may still get collisions, but it is better than
    just using unknown which would give us a lot more collisions'''
    ra = getRA(header)
    dec = getDec(header)
    if not ra or not dec:
        #either ra or dec was None so just return 'unknown'
        return 'unknown'
    return 'R{0}_{1}D{2}_{3}'.format(ra[0],ra[1],dec[0],dec[1])

def splitByHeader(imlist,keyword):
    '''split a list of filenames by a header keyword, throw out any non-fits
    files.
    @returns a dict where the keys are the values of the supplied header and
    the values are lists of images which have that value in the header.

    if the keyword isn't in the header, then the empty string is used as
    the value'''
    result = defaultdict(list)
    for im in iter(imlist):
        if isFits(im):
            #put each image into a list identified by the filter
            #if the filter keyword isn't supplied default to empty string
            result[pyfits.getheader(im).get(keyword,'')].append(im)
    return result
```

Listing A.19 frameTypes.py

```
#!/usr/bin/python

import pyfits

import sys
import os
```

```
import os.path
from collections import defaultdict

import obsDB
from fitsHeader import isFits, getFrameType, getObjectName, splitByHeader

def getFrameLists(fileList):
        '''given an iterator of filenames, go through each one,
        get the the type of the frame and add it to the appropriate list
        return a dictionary containing lists of files for 'zero', 'dark',
        'object', and 'none'. The 'none' category will contain fits files
        that we can't determine the type for and files that we are unable to
        open'''
        results = {'zero':[], 'dark':[], 'flat':[], 'object':[], 'unknown':[]}
        for f in iter(fileList):
                try:
                        imType = getFrameType(pyfits.getheader(f))
                except:
                        imType = 'unknown'
                if imType is None:
                        imType = 'unknown'
                results[imType].append(f)
        return results

def saveFrameLists(frameLists, zeroFile='zeros.lst',darkFile='darks.lst',
        flatFile='flats.lst',objectFile='objects.lst',unknownFile='unknown.lst
            '):
        '''Take the output from getFrameLists, and save them to files'''
        with open(zeroFile,'w') as zf:
                for frame in frameLists['zero']:
                        zf.write('{0}\n'.format(frame))
        with open(darkFile,'w') as df:
                for frame in frameLists['dark']:
                        df.write('{0}\n'.format(frame))
        with open(flatFile,'w') as ff:
                for frame in frameLists['flat']:
                        ff.write('{0}\n'.format(frame))
        with open(objectFile,'w') as of:
                for frame in frameLists['object']:
                        of.write('{0}\n'.format(frame))
        with open(unknownFile,'w') as uf:
                for frame in frameLists['unknown']:
                        uf.write('{0}\n'.format(frame))

def makeObjectMap(files):
        '''create a dictionary with keys of the objects, and
           the values are lists of all the frames of that object
        '''
        result = defaultdict(list)
        for frame in iter(files):
                result[getObjectName(pyfits.getheader(frame))].append(frame)
        return result


def makeObjectList(files):
        '''create a list of all the objects observed'''
        return makeObjectMap(files).keys()

def printObjectList(objectlist,objectFile='objectList.lst'):
        '''create a file containing a list of all the objects in objectlist'''
        with open(objectFile,'w') as of:
                for obj in iter(objectlist):
                        of.write(obj)
```

```
                        of.write('\n')
        return

def printObjectMaps(objectMap, fileBase='obj_',ext='.lst'):
        '''for each object create file named fileBase+objName+ext
        which contains a single line header in the formate #(objname)
        followed a list of the frames of that object, one per line'''
        for obj, frames in objectMap.items():
                fname = fileBase + obj + ext #build name for the file
                with open(fname,'w') as olist:
                        olist.write('#({0})\n'.format(obj)) #write header with
                            object name
                        for frame in frames:
                                olist.write(frame)
                                olist.write('\n')
        return

#main function
def main(fileList=None):
        if fileList is None:
                #default to everything in the folder
                fileList = os.listdir('.')
        #look at the frame types
        frameTypes = getFrameLists(fileList)
        #get object names
        objNames = makeObjectMap(frameTypes['object'])

        #now print out the files
        printObjectMaps(objNames)
        printObjectList(objNames.keys())
        saveFrameLists(frameTypes)



#run main if the script is directly executed
if __name__ == '__main__':
        main(sys.argv[1:])
```

Listing A.20 observatories.py

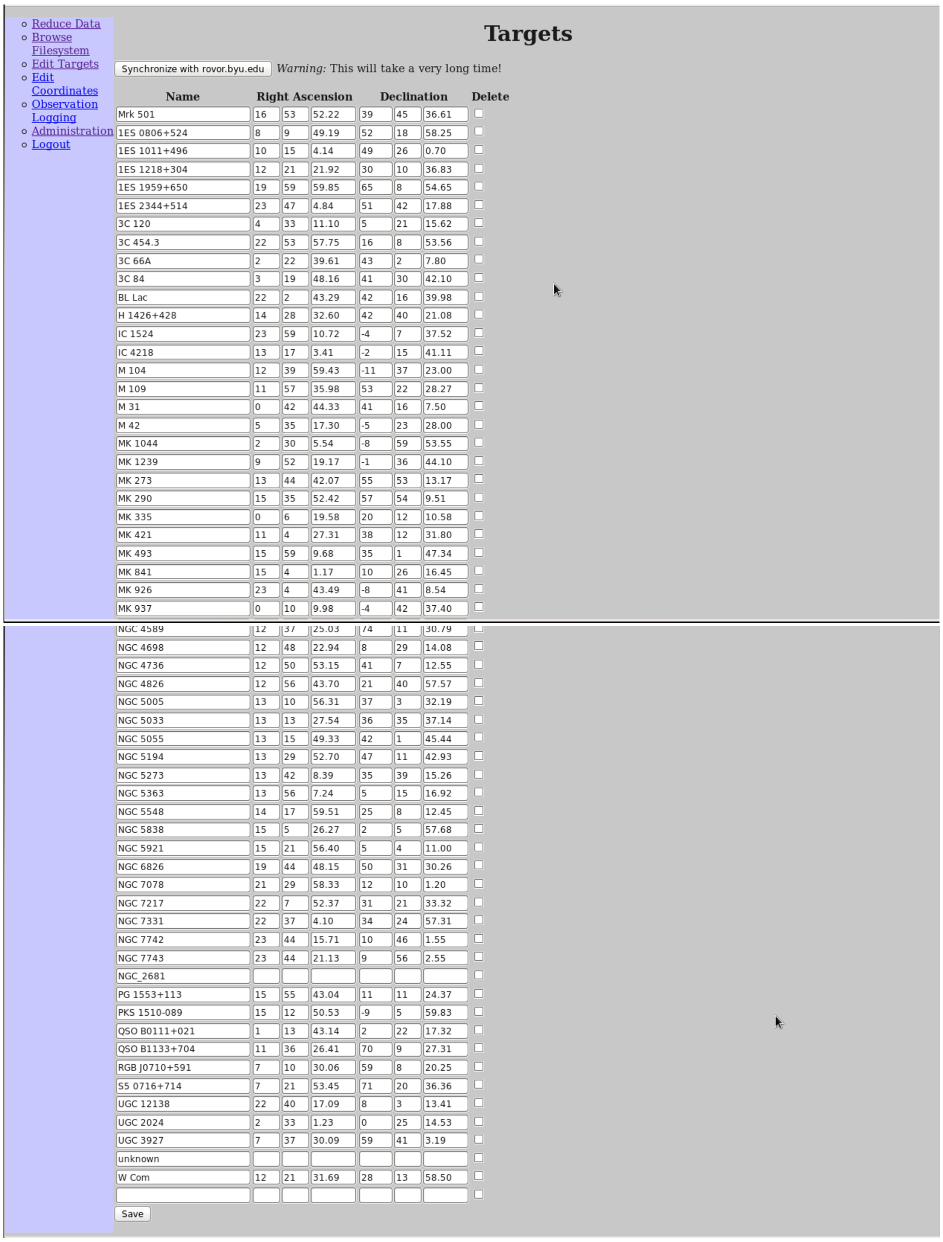
```
from decimal import Decimal

class Observatory:
    '''class to hold constants for a specific observatory,
    this should be set for any observatory you use.

    The name should be the name of the observatory and should
    match the name of the observatory in the IRAF observatory
    database. This is necessary for the HJD to be set when photting.

    For accurate photometry the readnoise and gain of the detector should
    be set to good values for the system.'''
    def __init__(self,
        name, #name of the observatory, should match IRAF observatory database
        width=Decimal(1),
        height=Decimal(1),
        lowscale=Decimal('0.1'),
        highscale=Decimal('2'),
        ra_key = 'objctra',
        dec_key = 'objctdec',
        exp_key = 'exptime',
```

```
        date_key = 'date-obs',
        time_key = 'date-obs',
        epoch_key = 'equinox',
        air_key = 'airmass',
        filt_key = 'filter',
        datamax = 50000,
        #these should be properly set for accurate photometry
        readnoise = 0,
        gain = 1,
        **kwargs):
        self.__dict__['_dict'] = kwargs
        self._dict['name'] = name
        self._dict['units'] = 'degrees' #default units are degrees
        self._dict['width'] = Decimal(width)
        self._dict['height'] = Decimal(height)
        self._dict['lowscale'] = Decimal(lowscale)  #value for low scale when
            using astrometry.net
        self._dict['highscale'] = Decimal(highscale) #value for high scale
            when using astrometry.net
        self._dict['ra_key'] = ra_key
        self._dict['dec_key'] = dec_key
        self._dict['exp_key'] = exp_key
        self._dict['date_key'] = date_key
        self._dict['time_key'] = time_key
        self._dict['epoch_key'] = epoch_key
        self._dict['air_key'] = air_key
        self._dict['filt_key'] = filt_key
        self._dict['datamax'] = datamax
        self._dict['readnoise'] = readnoise
        self._dict['gain'] = gain
    def __getitem__(self,key):
        return self._dict[key]
    def __setitem__(self,key,value):
        self._dict[key] = value
    def __getattr__(self,name):
        if name == '_dict':
            return self.__dict__[name]
        else:
            return self._dict[name]
    def __setattr__(self,name,value):
        self._dict[name] = value



#constants for the ROVOR observatory
ROVOR = Observatory('rovor',width=Decimal(23)/60,height=Decimal(23)/60,
    lowscale='0.3',highscale='0.4')
```

Appendix B

Figures

Figure B.1 Screenshot of Target List page



Name	Right Ascension			Declination			Delete
Mrk 501	16	53	52.22	39	45	36.61	
1ES 0806+524	8	9	49.19	52	18	58.25	
1ES 1011+496	10	15	4.14	49	26	0.70	
1ES 1218+304	12	21	21.92	30	10	36.83	
1ES 1959+650	19	59	59.85	65	8	54.65	
1ES 2344+514	23	47	4.84	51	42	17.88	
3C 120	4	33	11.10	5	21	15.62	
3C 454.3	22	53	57.75	16	8	53.56	
3C 66A	2	22	39.61	43	2	7.80	
3C 84	3	19	48.16	41	30	42.10	
BL Lac	22	2	43.29	42	16	39.98	
H 1426+428	14	28	32.60	42	40	21.08	
IC 1524	23	59	10.72	-4	7	37.52	
IC 4218	13	17	3.41	-2	15	41.11	
M 104	12	39	59.43	-11	37	23.00	
M 109	11	57	35.98	53	22	28.27	
M 31	0	42	44.33	41	16	7.50	
M 42	5	35	17.30	-5	23	28.00	
MK 1044	2	30	5.54	-8	59	53.55	
MK 1239	9	52	19.17	-1	36	44.10	
MK 273	13	44	42.07	55	53	13.17	
MK 290	15	35	52.42	57	54	9.51	
MK 335	0	6	19.58	20	12	10.58	
MK 421	11	4	27.31	38	12	31.80	
MK 493	15	59	9.68	35	1	47.34	
MK 841	15	4	1.17	10	26	16.45	
MK 926	23	4	43.49	-8	41	8.54	
MK 937	0	10	9.98	-4	42	37.40	
NGC 4589	12	37	25.03	74	11	30.79	
NGC 4698	12	48	22.94	8	29	14.08	
NGC 4736	12	50	53.15	41	7	12.55	
NGC 4826	12	56	43.70	21	40	57.57	
NGC 5005	13	10	56.31	37	3	32.19	
NGC 5033	13	13	27.54	36	35	37.14	
NGC 5055	13	15	49.33	42	1	45.44	
NGC 5194	13	29	52.70	47	11	42.93	
NGC 5273	13	42	8.39	35	39	15.26	
NGC 5363	13	56	7.24	5	15	16.92	
NGC 5548	14	17	59.51	25	8	12.45	
NGC 5838	15	5	26.27	2	5	57.68	
NGC 5921	15	21	56.40	5	4	11.00	
NGC 6826	19	44	48.15	50	31	30.26	
NGC 7078	21	29	58.33	12	10	1.20	
NGC 7217	22	7	52.37	31	21	33.32	
NGC 7331	22	37	4.10	34	24	57.31	
NGC 7742	23	44	15.71	10	46	1.55	
NGC 7743	23	44	21.13	9	56	2.55	
NGC_2681							
PG 1553+113	15	55	43.04	11	11	24.37	
PKS 1510-089	15	12	50.53	-9	5	59.83	
QSO B0111+021	1	13	43.14	2	22	17.32	
QSO B1133+704	11	36	26.41	70	9	27.31	
RGB J0710+591	7	10	30.06	59	8	20.25	
S5 0716+714	7	21	53.45	71	20	36.36	
UGC 12138	22	40	17.09	8	3	13.41	
UGC 2024	2	33	1.23	0	25	14.53	
UGC 3927	7	37	30.09	59	41	3.19	
unknown							
W Com	12	21	31.69	28	13	58.50	

Figure B.2 Screenshots of page to edit coordinate lists

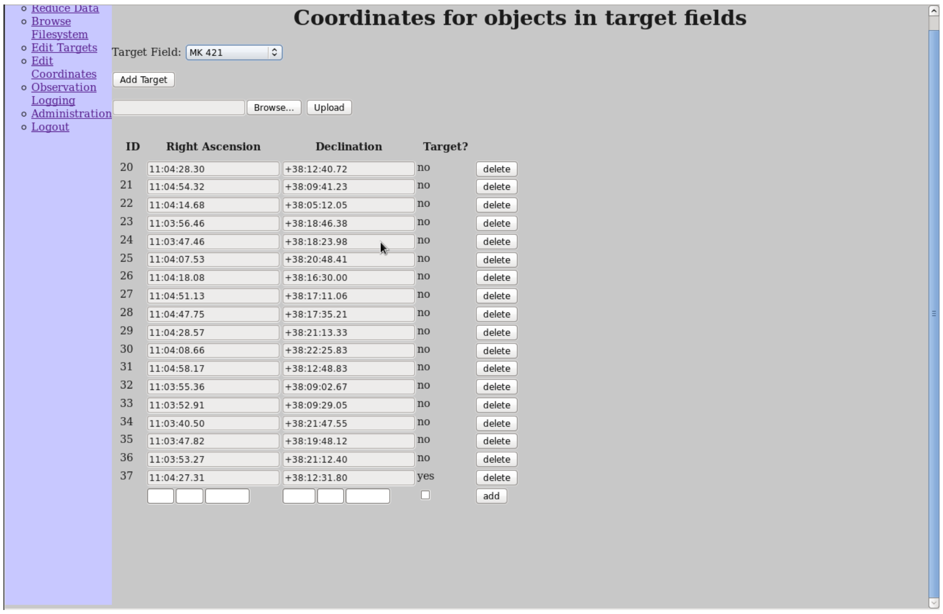


Figure B.3 Screenshot of Observation recording page

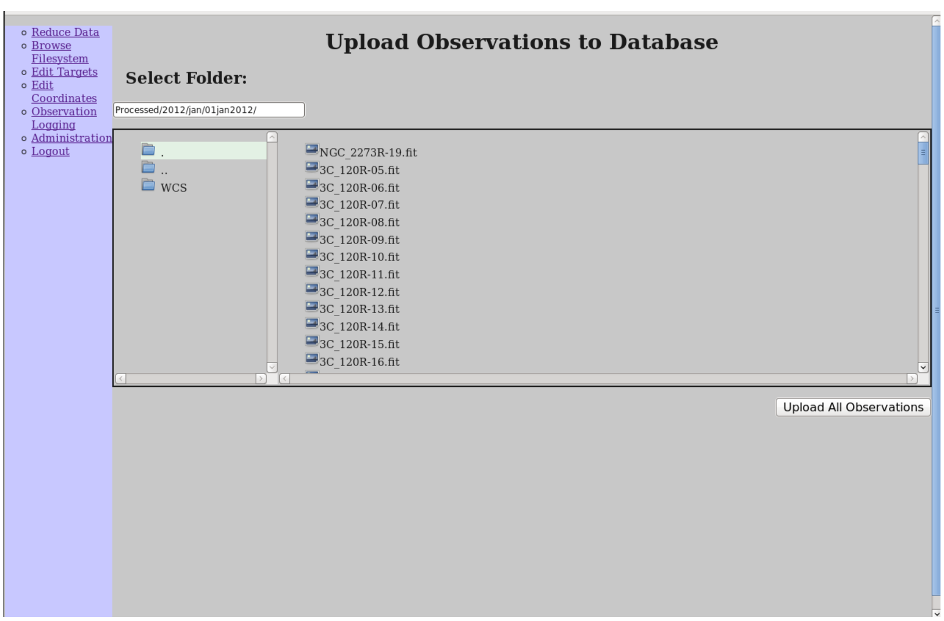


Figure B.4 First Pass page

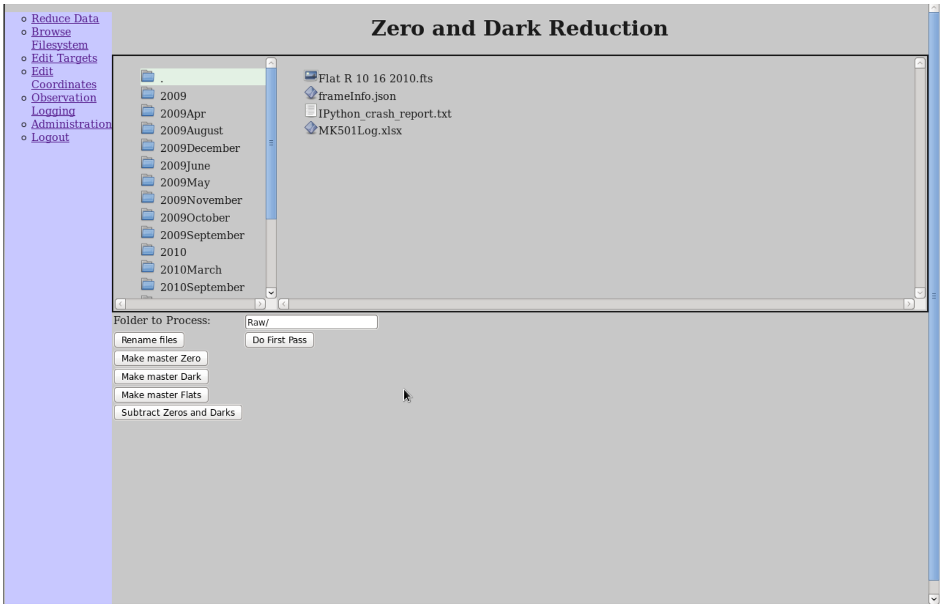
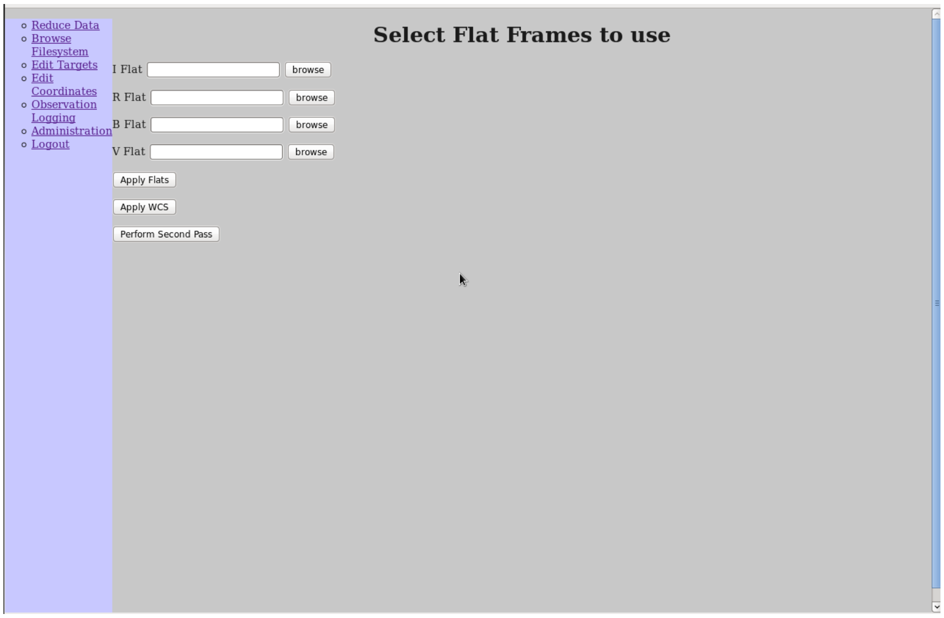


Figure B.5 Flat Selection Page



Bibliography

Crockford, D. 2006, RFC 4627, http://www.ietf.org/rfc/rfc4627.txt?number=4627

Davis, L. E. 1994, IRAF Programming Group, NOAO, Tucson

Django. Accessed July 22, 2013, https://www.djangoproject.com/

Holden, M. 2013, Private Communication

Lang, D., Hogg, D. W., Mierle, K., Blanton, M., & Roweis, S. 2010, *The Astronomical Journal*, 137, 1782, arXiv:0910.2233

Index