reduce Users Manual

Release X1

Kennneth Anderson

September 03, 2014

CONTENTS

CHAPTER

ONE

INTRODUCTION

This document is version 1.0 of the `reduce` Users Manual. This manual will describe the usage of `reduce` as an application provided by the Gemini Observatory Astrodata package suite. `reduce` is an application that allows users to invoke the Gemini Recipe System to perform data processing and reduction on one or more astronomical datasets.

This document presents details on applying `reduce` to astronomical datasets, currently defined as multi-extension FITS (MEF) files, both through the application's command line interface and the application programming interface (API). Details and information about the `astrodata` package, the Recipe System, and/or the data processing involved in data reduction are beyond the scope of this document and will only be engaged when directly pertinent to the operations of `reduce`.

1.1 Applicable Documents

- *GMOS Imaging – Quality Assessment Pipeline Processing Walkthrough*, K. Labrie, GDPSG, v2.0, 12-06-2011.
- *Image Quality Assessment and the QAP*, P. Hirst, GS/GN Science Staff, 16-05-2012
- *Gemini IRAF Imaging Tutorial*, E. Hogan, GDPSG, Observational Techniques Wrokshop, 1-4, Apr. 2014.
- Install AstroData

1.2 Reference Documents

- *Astrodata Package Programmer's Manual*, C. Allen, GDPSG, 07-05-2013
- *Gemini AstroData Type Reference*, C. Allen, GDPSG, 09-03-2012
- *Class AstroData Application Programming Interface (API)*, K. Anderson, M. Simpson, 14-06-2014.
- *The Gemini Recipe System: a dynamic workflow for automated data reduction*, K. Labrie *et al*, SPIE, 2010.
- *Developing for Gemini's extensible pipeline environment*, K. Labrie, C. Allen, P. Hirst, ADASS, 2011
- *Gemini's Recipe System; A publicly available instrument-agnostic pipeline infrastructure*, K. Labrie et al, ADASS 2013.

1.3 Overview

As an application, `reduce` provides interfaces to configure and launch the Gemini Recipe System, a framework for developing and running configurable data processing pipelines and which can accommodate processing pipelines for

arbitrary dataset types. In conjunction with the development of `astrodata`, Gemini Observatory has also developed the compatible `astrodata_Gemini` package, the code base currently providing abstraction of, and processing for, Gemini Observatory astronomical observations.

In Gemini Observatory's operational environment "on summit," `reduce`, `astrodata`, and the `astrodata_Gemini` packages provide a currently defined, near-realtime, quality assurance pipeline, the so-called QAP. `reduce` is used to launch this pipeline on newly acquired data and provide image quality metrics to observers, who then assess the metrics and apply observational decisions on telescope operations.

Users unfamiliar with terms and concepts heretofore presented should consult documentation cited in the previous sections, or consult your nearest pipeline physician.

1.4 Glossary

adcc – Automatated Data Communication Center. Provides XML-RPC and HTTP services for pipeline operations. Can be run externally to `reduce`. Users need not know about or invoke the `adcc` for `reduce` operations. `reduce` will launch an `adcc` instance if one is not available. See Sec. *The adcc* for further discussion on `adcc`.

astrodata (or Astrodata) – part of the **gemini_python** package suite that defines the dataset abstraction layer for the Recipe System.

AstroData – not to be confused with **astrodata**, this is the main class of the `astrodata` package, and the one most users and developers will interact with at a programmatic level.

AstroDataType – Represents a data classification. A dataset will be classified by a number of types that describe both the data and its processing state. The AstroDataTypes are hierarchical, from generic to specific. For example, a typical GMOS image might have a set of types like

'GMOS_S', 'GMOS_IMAGE', 'GEMINI', 'SIDEREAL', 'IMAGE', 'GMOS', 'GEMINI_SOUTH', 'GMOS_RAW', 'UNPREPARED', 'RAW' (see **types** below).

astrodata_Gemini – the **gemini_python** package that provides all observatory specific definitions of data types, **recipes**, and associated **primitives** for Gemini Observatory data.

astrodata_X – conceivably a data reduction package that could reduce other observatory and telescope data. Under the Astrodata system, it is entirely possible for the Recipe System to process HST or Keck data, given the development of an associated package, astrodata_HST or astrodata_Keck. Pipelines and processing functions are defined for the particulars of each telescope and its various instruments.

descriptor – Represents a high-level metadata name. Descriptors allow access to essential information about the data through a uniform, instrument-agnostic interface to the FITS headers.

GDPSG – Gemini Data Processing Software Group

gemini_python – A suite of packages comprising **astrodata**, **astrodata_Gemini**, and **gempy** all of which provide the full functionality needed to run **Recipe System** pipelines on observational datasets.

gempy – a **gemini_python** package comprising functional utilities to the **astrodata_Gemini** package.

MEF – Multiple Extension FITS, the standard data format not only for Gemini Observatory but many observatories.

primitive – A function defined within an **astrodata_[X]** package that performs actual work on the passed dataset. Primitives observe tightly controlled interfaces in support of re-use of primitives and recipes for different types of data, when possible. For example, all primitives called `flatCorrect` must apply the flat field correction appropriate for the data's current AstroDataType, and must have the same set of input parameters.

recipe – Represents the sequence of transformations. A recipe is a simple text file that enumerates the set and order of **primitives** that will process the passed dataset. A **recipe** is the high-level pipeline definition. Users can pass recipe names directly to reduce. Essentially, a recipe is a pipeline.

Recipe System – The gemin_python framework that accommodates an arbitrary number of defined recipes and the primitives

reduce – The user/caller interface to the Recipe System and its associated recipes/pipelines.

type or **typeset** – Not to be confused with language primitive or programmatic data types, these are data types defined within an **astrodata_[X]** package used to describe the kind of observational data that has been passed to the Recipe System., Eg., GMOS_IMAGE, NIRI. In this document, these terms are synonymous with **AstroDataType** unless otherwise indicated.

CHAPTER

TWO

USER ENVIRONMENT

Once astrodata has been installed, only minor adjustments need to be made to the user environment in order to make astrodata importable and allow `reduce` to work properly. Presumably, the user has retrieved the gemini_python package from the SVN repository describe in the link above.

2.1 Configuration

Let us start with checkout of the gemini_python trunk:

```
$ svn co http://chara.hi.gemini.edu/svn/DRSoftware/gemini_python/trunk
```

Users need not leave the checkout name *trunk* in place and obviously can rename it to be whatever they like. Whatever that path, add this path to the PYTHONPATH environment variable.

For example, `gemini_python/trunk` is checked out as described above. In '.' the directory *trunk* is now present and populated:

```
$ ls -l
drwxr-xr-x  19 user  group     646 Aug 20 14:44 trunk/
```

Rename trunk to some friendlier name:

```
$ mv trunk gemsoft
$ ls -l
drwxr-xr-x  19 user  group     646 Aug 20 14:44 gemsoft/
```

Set the environment to make astrodata *et al* importable:

```
$ export PYTHONPATH=${PYTHONPATH}:/user/path/to/gemsoft
```

Add the path *astrodata/scripts* to the PATH environment variable:

```
$ export PATH=${PATH}:/user/path/to/gemsoft/astrodata/scripts
```

reduce is made available on the command line from here. It should already have the executable bit set from the repository, but if not, chmod `reduce` to an appropriate mask.

2.2 Test the installation

Start up the python interpreter and import astrodata:

```
$ python
>>> import astrodata
```

Next, return to the command line and test that `reduce` is reachable and runs:

```
$ reduce -h [--help]
```

This will print the reduce help to the screen.

CHAPTER

THREE

INTERFACES

3.1 Introduction

The `reduce` application provides both a command line interface and an API, both of which can configure and launch a Recipe System processing pipeline (a 'recipe') on the passed dataset. Control of `reduce` and and the Recipe System is provided by a variety of options and switches. Of course, all options and switches can be accessed and controlled through the API.

`reduce` itself is essentially a skeleton script. After parsing the command line, the script then passes the parsed arguments to the defined function, main(), which in turn calls the Reduce() class constructor with "args". Class Reduce() is implemented in the module coreReduce.py, found under `astrodata/adutils/reduceutils`.

3.2 Command line interface

We begin with the command line help provide by `reduce -h, --help`, followed by further description and discussion of certain non-trivial options that require detailed explanation.

```
usage: reduce [options] fitsfile [fitsfile ...]
```

positional arguments:

```
fitsfile [fitsfile ...]
```

The [options] are described in the following sections.

3.2.1 Informational switches

-h, –help show the help message and exit

-v, –version show program's version number and exit

-d, –displayflags Display all parsed option flags and exit.

> When specified, this switch will present the user with a table of all parsed arguments and then exit without running. This allows the user to check that the configuration is as intended. The table provides a convenient view of all passed and default values. Unless a user has specified a recipe (-r, –recipe), 'recipename' indicates 'None' because at this point, the Recipe System has not yet been engaged and a default recipe not yet determined.
>
> Eg.,:

```
$ reduce -d --logmode console fitsfile.fits

--------------------   switches, vars, vals  --------------------

Literals                  var 'dest'              Value
------------------------------------------------------------------
['--invoked']             :: invoked              :: False
['--addprimset']          :: primsetname          :: None
['-d', '--displayflags'] :: displayflags         :: True
['-p', '--param']         :: userparam            :: None
['--logmode']             :: logmode              :: ['console']
['-r', '--recipe']        :: recipename           :: None
['--throw_descriptor_exceptions'] :: throwDescriptorExceptions   :: False
['--logfile']             :: logfile              :: reduce.log
['-t', '--astrotype']     :: astrotype            :: None
['--override_cal']        :: user_cals            :: None
['--context']             :: running_contexts     :: None
['--calmgr']              :: cal_mgr              :: None
['--suffix']              :: suffix               :: None
['--loglevel']            :: loglevel             :: stdinfo
------------------------------------------------------------------

Input fits file(s):       fitsfile.fits
```

3.2.2 Configuration Switches, Options

–addprimset <PRIMSETNAME> Add this path to user supplied primitives to reduction. I.e. path to a primitives module.

–calmgr <CAL_MGR> This is a URL specifying a calibration manager service. A calibration manager overides Recipe System table.

–context <RUNNING_CONTEXTS> Use <RUNNING_CONTEXTS> for primitives sensitive to context. Eg., –context QA When not specified, the context defaults to 'QA'.

–invoked Boolean indicating that reduce was invoked by adcc.

–logmode <LOGMODE> Set logging mode. One of 'standard', 'console', 'quiet', 'debug', or 'null', where 'console' writes only to screen and 'quiet' writes only to the log file. Default is 'standard'.

–logfile <LOGFILE> Set the log file name. Default is 'reduce.log' in '.'

–loglevel <LOGLEVEL> Set the verbose level for console logging. One of 'critical', 'error', 'warning', 'status', 'stdinfo', 'fullinfo', 'debug'. Default is 'stdinfo'.

–override_cal <USER_CALS [USER_CALS ...]> Add calibration to User Calibration Service. '–override_cal CALTYPE:CAL_PATH' Eg.,:

> –override_cal processed_arc:wcal/gsTest_arc.fits

-p <USERPARAM [USERPARAM ...]>, –param <USERPARAM [USERPARAM ...]> Set a primitive parameter from the command line. The form '-p par=val' sets the parameter in the reduction context such that all primitives will 'see' it. The form:

> -p ASTROTYPE:primitivename:par=val

sets the parameter such that it applies only when the current reduction type (type of current reference image) is 'ASTROTYPE' and the primitive is 'primitivename'. Separate parameter-value pairs by whitespace: (eg. '-p par1=val1 par2=val2')

See Sec. *Overriding Primitive Parameters*, for more information on these values.

-r <RECIPENAME>, –recipe <RECIPENAME> Specify an explicit recipe to be used rather than internally determined by a dataset's <ASTROTYPE>. Default is None and later determined by the Recipe System based on the determined AstroDataType.

-t <ASTROTYPE>, –astrotype <ASTROTYPE> Run a recipe based on this AstroDataType, which overrides default type or begins without initial input. Eg., recipes that begin with primitives that acquire data. `reduce` default is None and determined internally.

–suffix <SUFFIX> Add 'suffix' to output filenames at end of reduction.

–throw_descriptor_exceptions Boolean indicating descriptor exceptions are to be raised.

3.2.3 Nominal Usage

In most use cases, users will likely call `reduce` very simply with one or more positional arguments and nothing more. Oftentimes `reduce` will be run on a single FITS files:

```
$ reduce <fitsfile.fits>
```

Such a simple command for complex processing of data is possible because AstroData and the Recipe System do all the necessary work in determining how the data are to be processed, which is critcially based upon the determination of the *typeset* that applies to that data.

Without any user-specified recipe (-r –recipe), the default recipe is `qaReduce`, which is defined for various AstroDataTypes. For example, the `qaReduce` recipe for a GMOS_IMAGE specifies that the following primitives are called on the data:

- prepare
- addDQ
- addVAR
- detectSources
- measureIQ
- measureBG
- measureCCAndAstrometry
- overscanCorrect
- biasCorrect
- ADUToElectrons
- addVAR
- flatCorrect
- mosaicDetectors
- makeFringe
- fringeCorrect
- detectSources
- measureIQ
- measureBG
- measureCCAndAstrometry

- addToList

The point here is not to overwhelm readers with a stack of primitive names, but to present both the default pipeline processing that the above simple `reduce` command invokes and to demonstrate how much the `reduce` interface abstracts away the complexity of the processing that is engaged with the simplist of commands.

3.2.4 Overriding Primitive Parameters

In some cases, users may wish to change the functional behaviour of certain processing steps, i.e. change default Recipe System behaviour of primitive functions.

Primitives defined within the Recipe System each have a set of pre-defined parameters, which are used to control functional behaviour of the primitive. Each defined parameter has a "user override" token, which indicates that a particular parameter may be overridden by the user. Users can adjust parameter values from the reduce command line with the option,

-p, –param

If permitted by the "user override" token, parameters and values specified through the **-p, –param** option will *override* the defined parameter default value and may alter default behaviour of the primitive accessing this parameter from the reduction context. A user may pass several parameter-value pairs with this option.

Eg.:

```
$ reduce -p par1=val1 par2=val2 [par3=val3 ... ] <fitsfile1.fits>
```

For example, some photometry primitives perform source detection on an image. The 'detection threshold' has a defined default, but a user may alter this parameter default to change the source detection behaviour:

```
$ reduce -p threshold=4.5 <fitsfile.fits>
```

This overrides the defined default value.

3.2.5 The @file facility

The reduce command line interface is implemented under the `argparse` module's `ArgumentParser` class. ArgumentParser provides what might be called the '@file' facility (users and readers familiar with IRAF will recognize this facility). The `argparse` documentation actually calls this a 'from file', but it is the same thing.

By passing an @file to `reduce` on the command line, users can encapsulate all the options and positional arguments they might wish to specify in a single @file. It is possible to use multiple @files and even to embed one or more @files in another. The parser opens all files sequentially and parses all arguments in the same manner as if they were specified on the command line. Essentially, an @file is some or all of the command line and parsed identically. While ArgumentParser allows this facility to be configured to recognize one or more characters other than '@' as indicating an "at-file", the `reduce` parser is configured thusly. Using other characters is fraught, as the shell will interpret many such characters before the parser ever sees them.

To illustrate the convenience provided by an '@file', let us begin with an example *reduce* command line that has a number of arguments:

```
$ reduce -p GMOS_IMAGE:contextReport:tpar=100 GMOS_IMAGE:contextReport:report_inputs=True
  -r recipe.ArgsTest --context qa S20130616S0019.fits N20100311S0090.fits
```

Ungainly, to be sure. Here, two (2) *user parameters* are being specified with **-p**, a *recipe* with **-r**, and a *context* argument is specified to be **qa** . This can be wrapped in a plain text @file called *reduce_args.par*:

```
S20130616S0019.fits
N20100311S0090.fits
--param
GMOS_IMAGE:contextReport:tpar=100
GMOS_IMAGE:contextReport:report_inputs=True
-r recipe.ArgsTests
--context qa
```

This then turns the previous reduce command line into something a little more *keyboard friendly*:

```
$ reduce @reduce_args.par
```

The order of these arguments is irrelevant. The parser will figure out what is what. The above file could be thus written like:

```
-r recipe.ArgsTests
--param
GMOS_IMAGE:contextReport:tpar=100
GMOS_IMAGE:contextReport:report_inputs=True
--context qa
S20130616S0019.fits
N20100311S0090.fits
```

Note: Comments are accommodated, both line and in-line. '=' signs *may* be used but this has meaning only for arguments that expect unitary values. The '=' is really quite unnecessary.

White space is the only significant separator of arguments: spaces, tabs, newlines are all equivalent when argument parsing. This means the user can 'arrange' their @file for clarity.

Eg., a more readable version of the above file might be written as:

```
# reduce parameter file
# yyyy-mm-dd
# GDPSG

# Spec the recipe
-r
    recipe.ArgsTests  # test recipe

# primitive parameters here
# These are 'untyped', i.e. global
--param
    tpar=100
    report_inputs=True

--context
    qa                # QA context

S20130616S0019.fits
N20100311S0090.fits
```

All the above examples of `reduce_args.par` are equivalently parsed. Which, of course, users may check by adding the **-d** flag:

```
$ reduce -d @redpars.par

--------------------   switches, vars, vals  --------------------
```

```
Literals                       var 'dest'              Value
-----------------------------------------------------------------
['--invoked']                  :: invoked              :: False
['--addprimset']               :: primsetname          :: None
['-d', '--displayflags']       :: displayflags         :: True
['-p', '--param']              :: userparam            :: ['tpar=100', 'report_inputs=True']
['--logmode']                  :: logmode              :: standard
['-r', '--recipe']             :: recipename           :: ['recipe.ArgTests']
['--throw_descriptor_exceptions'] :: throwDescriptorExceptions        :: False
['--logfile']                  :: logfile              :: reduce.log
['-t', '--astrotype']          :: astrotype            :: None
['--override_cal']             :: user_cals            :: None
['--context']                  :: running_contexts     :: ['QA']
['--calmgr']                   :: cal_mgr              :: None
['--suffix']                   :: suffix               :: None
['--loglevel']                 :: loglevel             :: stdinfo
-----------------------------------------------------------------

Input fits file(s):   S20130616S0019.fits
Input fits file(s):   N20100311S0090.fits
```

3.2.6 Recursive @file processing

As implemented, the @file facility will recursively handle, and process correctly, other @file specifications that appear in a passed @file or on the command line. For example, we may have another file containing a list of fits files, separating the command line flags from the positional arguments.

We have a plain text 'fitsfiles' containing the line:

```
test_data/S20130616S0019.fits
```

We can indicate that this file is to be consumed with the prefix character "@" as well. In this case, the 'reduce_args.par' file could thus appear:

```
# reduce test parameter file

@fitsfiles       # file with fits files

# AstroDataType
-t GMOS_IMAGE

# primitive parameters.
--param
    report_inputs=True
    tpar=99
    FOO=BAR

# Spec the recipe
-r recipe.ArgTests
```

The parser will open and read the @fitsfiles, consuming those lines in the same way as any other command line arguments. Indeed, such a file need not only contain fits files (positional arguments), but other arguments as well. This is recursive. That is, the @fitsfiles can contain other at-files", which can contain other "at-files", which can contain ..., *ad infinitum*. These will be processed serially.

As stipulated earlier, because the @file facility provides arguments equivalent to those that appear on the command line, employment of this facility means that a reduce command line could assume the form:

```
$ reduce @parfile @fitsfiles
```

or equally:

```
$ reduce @fitsfiles @parfile
```

where 'parfile' could contain the flags and user parameters, and 'fitsfiles' could contain a list of datasets.

Eg., fitsfiles comprises the one line:

```
test_data/N20100311S0090.fits
```

while parfile holds all other specifications:

```
# reduce test parameter file
# GDPSG

# AstroDataType
-t GMOS_IMAGE

# primitive parameters.
--param
    report_inputs=True
    tpar=99            # This is a test parameter
    FOO=BAR            # This is a test parameter

# Spec the recipe
-r recipe.ArgTests
```

3.2.7 Overriding @file values

The GDPSG have specified customized parser actions such that a command line **-p, –param** option will accumulate to the set of primitive parameters or override a particular parameter, if that is specified. For other arguments where singular values are passed, the command line value *overrides* the @file value.

It is further specified that if one or more datasets (i.e. positional arguments) are passed on the command line, *all fits files appearing as positional arguments in the parameter file will be replaced by the command line file(s).*

Using the parfile above,

Eg. 1) Accumulate a new parameter:

```
$ reduce @parfile --param FOO=BARSOOM

parsed options:
--------------------
AstroDataType: GMOS_IMAGE
FITS files:    ['S20130616S0019.fits', 'N20100311S0090.fits']
Parameters:    tpar=100, report_inputs=True, FOO=BARSOOM
RECIPE:        recipe.ArgsTest
```

Eg. 2) Override a parameter in the @file:

```
$ reduce @parfile --param tpar=99

parsed options:
--------------------
AstroDataType: GMOS_IMAGE
FITS files:    ['S20130616S0019.fits', 'N20100311S0090.fits']
```

```
Parameters:    tpar=99, report_inputs=True
RECIPE:        recipe.ArgsTest
```

Eg. 3) Override the recipe:

```
$ reduce @parfile -r=recipe.FOO

parsed options:
--------------------
AstroDataType:    GMOS_IMAGE
FITS files:       ['S20130616S0019.fits', 'N20100311S0090.fits']
Parameters:       tpar=100, report_inputs=True
RECIPE:           recipe.FOO
```

Eg. 4) Override a recipe and specify another fits file

```
$ reduce @parfile -r=recipe.FOO test_data/N20100311S0090_1.fits

parsed options:
--------------------
AstroDataType:    GMOS_IMAGE
FITS files:       ['test_data/N20100311S0090_1.fits']
Parameters:       tpar=100, report_inputs=True
RECIPE:           recipe.FOO
```

3.3 Application Programming Interface (API)

The `reduce` application is essentially a skeleton script providing the described command line interface. After parsing the command line, the script then passes the parsed arguments to its main() function, which in turn calls the Reduce() class constructor with "args". Class Reduce() is defined in the module `coreReduce.py`. `reduce` and class Reduce are both scriptable, as the following discussion will illustrate.

3.3.1 reduce.main()

The main() function of reduce receives one (1) parameter that is a Namespace object as returned by a call on ArgumentParser.parse_args(). Specific to reduce, the caller can supply this object by a call on the parseUtils.buildParser() function, which returns a fully defined reduce parser. As usual, the parser object should then be called with the parse_args() method to return a valid reduce parser Namespace. Since there is no interaction with sys.argv, as in a command line call, all Namespace attributes have only their defined default values. It is for the caller to set these values as needed.

As the example below demonstrates, once the "args" Namespace object is instantiated, a caller can set any arguments as needed. Bu they must be set to the correct type. The caller should examine the various "args" types to determine how to set values. For example, args.files is type list, whereas args.recipename is type string.

Eg.,

```
>>> from astrodata.adutils.reduceutils import reduce
>>> from astrodata.adutils.reduceutils import parseUtils
>>> args = parseUtils.buildParser("Reduce,v2.0").parse_args()
>>> args.files
[]
>>> args.files.append('S20130616S0019.fits')
>>> args.recipename = "recipe.FOO"
>>> reduce.main(args)
```

```
--- reduce, v2.0 ---
Starting Reduction on set #1 of 1
Processing dataset(s):
S20130616S0019.fits
...
```

Processing will proceed as usual.

3.3.2 Class Reduce and the runr() method

Class Reduce is defined in `astrodata.adutils.reduceutils` module, `coreReduce.py`.

The reduce.main() function serves mainly as a callable for the command line interface. While main() is callable by users supplying the correct "args" parameter (See *reduce.main()*), the Reduce() class is also callable and can be used directly, and more appropriately. Callers need not supply an "args" parameter to the class constructor. The instance of Reduce will have all the same arguments as in a command line scenario, available as attributes on the instance. Once an instance of Reduce() is instantiated and instance attributes set as needed, there is one (1) method to call, **runr**(). This is the only public method on the class.

Note: When using Reduce() directly, callers must configure their own logger. Reduce() does not configure logutils prior to using a logger as returned by logutils.get_logger(). The following example will illustrate how this is easily done. It is *highly recommended* that callers configure the logger.

Eg.,

```
>>> from astrodata.adutils.reduceutils.coreReduce import Reduce
>>> reduce = Reduce()
>>> reduce.files
[]
>>> reduce.files.append('S20130616S0019.fits')
>>> reduce.files
['S20130616S0019.fits']
```

Once an instance of Reduce has been made, callers can then configure logutils with the appropriate settings supplied on the instance. This is precisely what `reduce` does when it configures logutils.

```
>>> from astrodata.adutils import logutils
>>> logutils.config(file_name=reduce.logfile, mode=reduce.logmode,
                    console_lvl=reduce.loglevel)
```

At this point, the caller is able to call the runr() method on the "reduce" instance. Processing will then proceed in the usual manner.

```
>>> reduce.runr()
All submitted files appear valid
Starting Reduction on set #1 of 1
Processing dataset(s):
S20130616S0019.fits
...
```

CHAPTER

FOUR

SUPPLEMENTAL TOOLS

The astrodata package provides a number of command line driven tools, two of which users may find helpful in executing reduce on their data. These tools can present primitive names available, their parameters and defaults, as well as perform data and type discovery in a directory tree.

If the user environment has been correctly configured (Sec. *Configuration*), these applications will work directly.

4.1 listprimitives

In a correct environment, the `listprimitives.py` module (linked as `listprimitives`) will be available as a command line executable. This tool displays available primitives for all AstroDataTypes, their parameters, and defaults. These are the parameters discussed in Sec. *Overriding Primitive Parameters* that can be changed by the user with the **-p, –param** option on reduce. under the AstroDataTypes. The help describes more options:

```
$ listprimitives -h

Usage: listprimitives [options]

Gemini Observatory Primitive Inspection Tool, v1.0 2011

Options:
-h, --help            show this help message and exit
-c, --use-color       apply color output scheme
-e, --engineering     show engineering recipes
-i, --info            show more information
-p, --parameters      show parameters
-r, --recipes         list top recipes
-s, --primitive-set   show primitive sets (Astrodata types)
-v, --verbose         set verbose mode
--view-recipe=VIEW_RECIPE
                      display the recipe
```

Admittedly, this tool is in need of refinement and the GDPSG is working on building a tool that will present primitives and parameters in a more focused way. I.e. report only those primitives and parameters relevant to a given dataset.

4.2 typewalk

The application `typewalk` can help users find data that match type criteria. As with `listprimitives`, if users have reduce available on the command line, `typewalk` will be available, too. See the help, **-h, –help**, for all available options on `typewalk`.

More generally, `typewalk` examines files in a directory or directory tree and reports the types and status values through the AstroDataType classification scheme. Files are selected and reported through a regular expression mask which, by default, finds all ".fits" and ".FITS" files. Users can change this mask with the **-f, –filemask** option.

By default, `typewalk` will recurse all subdirectories under the current directory. Users may specify an explicit directory with the **-d, –dir** option.

A user may request that an output file is written when AstroDataType qualifiers are passed by the **–types** option. An output file is specified through the **-o, –out** option. Output files are formatted so they may be passed *directly to the reduce command line* via that applications 'at-file' (@file) facility. See *The @file facility* or the reduce help for more on 'at-files'.

Users may select type matching logic with the **–or** switch. By default, qualifying logic is AND. I.e. the logic specifies that *all* types must be present (x AND y); **–or** specifies that ANY types, enumerated with **–types**, may be present (x OR y). **–or** is only effective when **–types** is used.

For example, find all gmos images from Cerro Pachon in the top level directory and write out the matching files, then run reduce on them (**-n** is 'norecurse'):

```
$ typewalk -n --types GEMINI_SOUTH GMOS_IMAGE --out gmos_images_south
$ reduce @gmos_images_south
```

This will also report match results to stdout, colourized if requested (**-c**).

CHAPTER

FIVE

DISCUSSION

5.1 The adcc

As a matter of operations, `reduce` and the Recipe System depend upon the services of what is called the `adcc`, the Automated Data Communication Center. The `adcc` provides services to pipeline operations through two proxy servers, an XML-RPC server and an HTTP server. The XML_RPC server serves calibration requests made on it, and retrieves calibrations that satisfiy those requests from the Gemini FITS Store, a service that provides automated calibration lookup and retrieval.

The `adcc` can be run externally and will run continuously until it is shutdown. Any instances of `reduce` (and the Recipe System) will employ this external instance of the `adcc` to service a pipeline's calibration requests. However, a user of `reduce` need not start an instance of the `adcc` nor, indeed, know anytihng about the `adcc` *per se*. If one is not available, an instance of the `adcc` will be started by `reduce` itself, and will serve that particular `reduce` process and then terminate.

This note is provided should users notice an `adcc` process and wonder what it is.

5.2 Future Enhancements

5.2.1 Intelligence

One enhancement long imagined is what has been generally termed 'intelligence'. That is, an ability for either `reduce` or some utility to automatically do AstroDataType classification of a set of data, group them appropriately, and then pass these grouped data to the Recipe System.

As things stand now, it is up to the user to pass commonly typed data to `reduce`. As shown in the previous section, *typewalk*, `typewalk` can help a user perform this task and create a 'ready-to-run' @file that can be passed directly to `reduce`. Properly implemented 'intelligence' will *not* require the user to determine the AstroDataTypes of datasets.

CHAPTER

SIX

6. ACKNOWLEDGMENTS

*The Gemini Observatory is operated by the Association of Universities for Research in Astronomy, Inc., under a cooperative agreement with the NSF on behalf of the Gemini partnership: the National Science Foundation (United States), the Science and Technology Facilities Council (United Kingdom), the National Research Council (Canada), CONICYT (Chile), the Australian Research Council (Australia), Ministerio da Ciencia e Tecnologia (Brazil), and Ministerio de Ciencia, Tecnologia e Innovacion Productiva (Argentina).