Appendix A

User's Guide to Arcreduce

Arcreduce is a Python module that provides methods for reducing data from the 40 m data archives. Its interface is provided primarily through two classes: `CalManager`, a high-level data reduction interface, and `ArchiveReader`, a lower-level engine for directly accessing the archives to decode procedure data. In addition, a number of methods and classes are included in the module to support these interfaces. Arcreduce is built atop the readarc module provided by Martin C. Shepherd. The readarc module is a Python wrapper around a C library that provides raw access to the data in the archive. Arcreduce was written in order to simplify access to the archive and to provide a more "Pythonic" set of interfaces.

In addition to readarc, Arcreduce depends on several other Python modules. These include the public libraries NumPy,[1] SciPy,[2] and PyEphem.[3] The py40m module, a library of useful routines developed alongside the 40 m reduction pipeline, is also required.

A.1 High-Level Data Reduction: `CalManager`

`CalManager` is the main class intended for high-level analysis and reduction of data recorded with the MCS control system. It is built upon `ArchiveReader`, and uses that class to read data from the archive and convert it into reduced procedure result data. `CalManager` provides tools to simplify the operations needed to examine and calibrate those data.

A.1.1 `CalManager` Concepts

The work flow within `CalManager` is modeled after a small subset of CMBPROG (Leitch 1998). The results of all executions of a particular procedure, are stored in an instance of the `Procedure` class. Within the procedure are a set of vector-like `Member` objects, each of which represents a time series of values from the results of that procedure.

[1] `http://numpy.scipy.org/`
[2] `http://scipy.org/`
[3] `http://rhodesmill.org/pyephem/`

Table A.1. List of `CalManager` procedures

Attribute	Radiometry Procedure	Members
`flux`	FLUX	a, b, c, d, atp, btp, ctp, dtp, swd, swp
`cal`	CAL	diode, a, b, c, d, atp, btp, ctp, dtp, swd, swp
`point`	POINT	snr, hpbw, failed
—	common to all	source, flux, flags, time, mjd, azo, zao, az, za, taf, tlr, focus, pa, samples

Note: Data for each procedure are contained in the members `Procedure` accessed as an attribute of the `CalManager` object. Each procedure type contains the common members listed, as well as members unique to its function. Table A.2 defines the various members.

The reader should be aware that the word "procedure" is frequently used in this section to refer either to a specific execution of a radiometry procedure, to the `Procedure` class, or to an instance of that class. It should be clear from context which meaning is intended, but careful reading is warranted.

A.1.1.1 `Procedure` Class

When data are loaded from the archive, the radiometry procedures are processed and decoded into a stream of results. The `Procedure` class represents the time-ordered series of results of one of these procedures. Table A.1 lists the procedures that are available. These procedures are accessible as attributes of the `CalManager` instance, so results from, say, FLUX procedures are stored in `cm.flux`.

The `CalManager` class defines a number of methods needed to manage and calibrate the data stored in the `Procedure` objects. Generally, methods defined in the `CalManager` class affect the calibration process as a whole or operate on more than one `Procedure` object. Methods that affect a single `Procedure` or `Member` are normally defined within those classes. For historical reasons, a number of relevant methods are defined at the top level of the `arcreduce` module.

A.1.1.2 `Member` Class

The `Member` class represents a member of a procedure—a series of time-ordered data. All members within a procedure share a common set of time stamps. The time stamps themselves are represented by the `time` member of the procedure. The `Member` class supports data masking to allow the parent `Procedure` to restrict operations to a subset of the values in the time series. This is used, for example, when working with FLUX results for a single source or CAL results for only the NOISE diode. Table A.2 lists and briefly describes the members that belong to the various procedures.

Vector arithmetic operations are supported for `Member` objects that contain numerical values. To avoid wastefully copying data, unary operations (e.g., `x+=y`) operate in place on the data within the `Member`. When a copy is desired, binary operations (e.g., `z=x+y`) allocate a new `Member` to store the result. This behavior is illustrated in the examples in section A.1.2.3.

Table A.2. Descriptions of the `CalManager` procedure members

Member	Used by	Description
`source`	all	Name of source being observed
`time`	all	UTC time stamp
`mjd`	all	MJD time stamp (derived from `time`)
`flags`	all	Reduction flags
`az`	all	Telescope azimuth
`za`	all	Telescope zenith angle
`taf`	all	Aft-forward tilt meter reading
`tlr`	all	Left-right tilt meter reading
`focus`	all	Telescope z-axis focus setting
`pa`	all	Parallactic angle of source
`samples`	all	Number of samples in procedure (definition varies by procedure)
`diode`	`cal`	Name of diode measured ("CAL" or "NOISE")
`a`, `b`, `c`, `d`	`flux`, `cal`	Measured switched power in each segment of the procedure
`atp`, `btp`, `ctp`, `dtp`	`flux`, `cal`	Measured total power in each segment of the procedure
`swd`, `swp`	`flux`, `cal`	"Switched difference" and "switched power" diagnostic signals, as described in section 2.2.2.2.
`hpbw`	`point`	Half-power beam width used for pointing measurement
`snr`	`point`	Signal-to-noise ratio of the pointing measurement
`failed`	`point`	True if a pointing measurement failed or gave an unreliable result
`azo`	all	Azimuthal offset from the pointing model (contains the result of a pointing measurement)
`zao`	all	Zenith angle offset from the pointing model (contains the result of a pointing measurement)

When relevant, a `Member` represents both a value and its uncertainty. Arithmetic operations propagate uncertainties between members assuming they are random and uncorrelated. Care must be taken if this assumption is not true. Constant values are assumed to have no uncertainty.

If access to the data values or uncertainties is needed, the `get_val()` or `get_err()` methods should be used. These return a NumPy `ndarray` with the requested values, after applying any active masks. Data stored in the member can be directly updated using the `set_val()` or `set_err()` method.

A.1.1.3 Masking and Flagging

When working with data, it is frequently useful to restrict the working data set to one or a few sources of interest. For the CAL procedure, normally only one diode (CAL or NOISE) is of interest at a time. To support this, `CalManager` provides a mechanism for masking procedures based on source or diode names. Initially upon load, all data are active. If a mask is subsequently applied, only those procedures that satisfy the mask are operated on until the mask is changed. The source mask is specified by passing a source name

or list of source names to the `set_active_sources()` method. The diode mask is specified via the `set_active_diodes()` method.

During reduction, data that are unreliable must be identified and discarded. `CalManager` allows data points to be flagged (stored as a bit field in the `flags` member) to identify the reason the data point was discarded. Flagged data are not used during later operations nor retrieved via the `get_val()` or `get_err()` methods. The `remove_flag()` method can be used to remove all instances of a particular flag from a `Procedure`.

A.1.1.4 Plotting and Advanced Processing

Python supports powerful numerical processing (e.g., the NumPy and SciPy packages) and plotting (e.g., Matplotlib). Routines of this nature are not implemented in `CalManager`. Instead, only basic, common operations are directly implemented for the `Procedure` and `Member` classes. When necessary, the data contained within a `Member` can be easily extracted to a NumPy `ndarray` and plotted or processed using a suitable external routine.

A.1.2 `CalManager` Tutorial

This simple tutorial demonstrates the essential functions of `CalManager` and its related classes. Before starting, the `arcreduce` module must be loaded and a `CalManager` object must be instantiated.

```
>>> import arcreduce as ar
>>> cm = ar.CalManager()
```

A.1.2.1 Loading Data

To load procedure data from the archive, the `load_data` method of the `CalManager` object is used. The start and end dates to be loaded can be specified as calendar dates or MJD days.

```
>>> cm.load_data('2011-05-01 00:00:00', '2011-05-01 12:00:00')
```

This example loads and decodes the procedures for 12 hours of data, beginning at midnight UTC on May 1, 2011.

A.1.2.2 Examining a `Procedure`

We can find the names of the members within a `Procedure` object as follows.

```
>>> print cm.flux.members
['source', 'flags', 'time', 'mjd', 'flux', 'az', 'za', 'taf', 'tlr',
 'focus', 'samples', 'azo', 'zao', 'pa', 'a', 'b', 'c', 'd',
 'atp', 'btp', 'ctp', 'dtp']
```

Each member of the procedure is represented by a `Member` object, which can be accessed as an attribute of the `Procedure` object. `Member` objects provide a convenient pretty-printing interface, illustrated here.

```
>>> print len(cm.flux)
212
>>> print cm.flux.flux
[   485.656 +/- 0.363,
   148.228 +/- 0.524,
   45.524 +/- 0.394,
   ...]
```

We see that 212 FLUX procedures were decoded. When the `flux` member of `cm.flux` is printed, the first few values are shown, along with their errors. Not all members have errors defined—for those without errors, these are shown as zero.

A.1.2.3 Working with a `Member`

`Member` objects that contain numerical data support vector arithmetic operations. In-place operations are performed using the unary Python arithmetic operators.

```
>>> cm.flux.flux *= 2
>>> print cm.flux.flux
[   971.313 +/- 0.726,
   296.456 +/- 1.047,
   91.048 +/- 0.787,
   ...]
```

Note that the errors were propagated assuming that the constant value in the multiplication was exact.

If two `Member` objects have the same length, they can be combined arithmetically. Within a single `Procedure`, all `Member` objects are guaranteed to be the same length, so this is always possible.

```
>>> cm.flux.flux += cm.flux.flux
>>> print cm.flux.flux
[   1942.625 +/- 1.026,
   592.912 +/- 1.481,
   182.097 +/- 1.113,
   ...]
```

The errors of the two inputs (here both `cm.flux.flux`) are combined to produce the error in the output. Note that the errors are propagated assuming they are uncorrelated (which is incorrect in this case).

If a binary Python operator is used, a new `Member` is created to contain the result. This new `Member` can (and normally would) be assigned to a `Procedure` as we do here.

```
>>> cm.flux.flux_copy = cm.flux.flux / 4.0
>>> print cm.flux.flux_copy
[   485.656 +/- 0.257,
   148.228 +/- 0.370,
   45.524 +/- 0.278,
   ...]
```

We now have a new `Member` whose values are equal to the initial values we loaded. The errors are smaller than they were at the start because they were assumed to be uncorrelated in the previous example. The `cm.flux.flux` object is unchanged because we created a copy.

A.1.2.4 Accessing Data from a `Member`

While basic arithmetic operations on `Member` data are supported directly, it is frequently useful to use ordinary Python methods to work with the data. This is possible using the `get_val()` and `get_err()` methods of the `Member`.

```
>>> val = cm.flux.flux.get_val()
>>> err = cm.flux.flux.get_err()
>>> val[0:3]
array([ 1942.62528571,   592.91225   ,   182.09658333])
>>> err[0:3]
array([ 1.02642661,  1.48117483,  1.11319623])
```

Python data may be inserted into a `Member` using the `set_val()` and `set_err()` methods. Here, we'll correct the uncertainty that was incorrectly propagated when we added `cm.flux.flux` to itself.

```
>>> import numpy as np
>>> cm.flux.flux.set_err(np.sqrt(2)*err)
>>> cm.flux.flux.err[0:3]
array([ 1.45158643, 2.09469753, 1.57429721])
```

When masks are applied to a procedure, such as when a specific source or diode is selected, these get and set methods act only on the actively selected elements of the procedure.

A.1.2.5 Selecting a Source or Diode

To restrict a procedure to a particular source, we call the `CalManager` object's `set_active_sources()` method. A similar method, `set_active_diodes()`, allows CAL procedures to be restricted to either the CAL or the NOISE diode.

```
>>> sources=cm.flux.source.get_val()
>>> print sources[0:3]
['j0750+1231' 'j0811+0146' 'j0805-0111']
>>> cm.set_active_sources('j0750+1231')
>>> print cm.flux.flux.get_val()
[ 1942.62528571]
```

To select multiple sources or diodes, simply pass these as a list of source names. To reenable all available sources or diodes, call this procedure with an empty list.

```
>>> cm.set_active_sources([])
```

A.1.2.6 Applying a Calibration Factor

Calibration by dividing by a filtered member or by a polynomial function of a member are supported. For example, we can normalize the flux density by the interpolated value of the CAL diode to remove the effect of gain fluctuations. As an example, here we simply calibrate the CAL diode flux member by itself. The deviation from 1.0 results from the averaging of nearby values.

```
>>> cm.set_active_diodes('CAL')
>>> print cm.cal.flux.get_val()[0:3]
[ 471.95144286  472.43159286  471.22714286]
>>> cm.apply_flux_cal(cm.cal, cm.cal.flux, 1.0)
>>> print cm.cal.flux.get_val()[0:3]
[ 0.99949928  1.00051614  0.99920581]
```

A.1.2.7 Flagging Data

Unreliable data points can be flagged to remove them from further processing and to indicate the nature of the problem with the data point. Table A.3 lists the supported flags. Several processing routines will implicitly flag data points when problems are encountered. Additionally, explicit flagging based on data or uncertainty values can be performed. In this example, we flag CAL values between 0 and 1, leaving only values greater than 1 unflagged.

```
>>> print cm.cal.flux.get_val()[0:3]
```

Table A.3. Flag values supported by `CalManager`

Name	Value	Description
FLAG_OUTLIER	1	Outlying data point flagged by `arcreduce.flag_outliers()`.
FLAG_SWEEP	2	Outlying data point flagged by `arcreduce.flag_sweep_outliers()`.
FLAG_VALUE_CLIP	4	Value was explicitly flagged by the user via `arcreduce.clip_values()`.
FLAG_ERROR_CLIP	8	Uncertainty was explicitly clipped by the user via `arcreduce.clip_errors()`.
FLAG_FAILED_POINT	16	Procedure was preceded by a failed (or no) point.
FLAG_BAD_CAL	32	Problem interpolating calibration diode to the data time stamp.

```
[ 0.99949928  1.00051614  0.99920581]
>>> ar.clip_values(cm.cal.flux, 0, 1.0, inclusive=True)
>>> print cm.cal.flux.get_val()[0:3]
[ 1.00051614  1.00238618  1.00077693]
```

A.1.2.8 Example Reduction Script

Listings A.1 and A.2 are excerpts of the Python reduction pipeline script and demonstrate actual use of the `CalManager` class.

A.1.3 Module Reference

- `class arcreduce.`**`CalManager`**: Interface class to manage high-level reduction and calibration.

  - **`load_data`**(*`start_date, end_date`*)

    Loads data from the archive.

  - **`apply_flux_cal`**(*`target_proc, source_mem, cal_value, inverse=False`*)

    Scales appropriate flux density-like members of *`target_proc`* by dividing by the value of *`source_mem`* at each sample. This is normally used to calibrate flux densities by dividing by an interpolated CAL member. *Note: this method implicitly applies a 7200 s boxcar interpolation to the source member to reconcile its time base with that of the target procedure.*

  - **`apply_polynomial_cal`**(*`target_mem, source_mem, poly, inverse=False`*)

    Divide flux density-like members of *`target_proc`* by a polynomial evaluated at the value of *`source_mem`* at each sample. This is normally used to apply the gain-versus-elevation curve using the `flux.za` member as *`source_mem`*.

  - **`set_active_diodes`**(*`diodes`*)

```
def reduce_data(start_date, stop_date):
    """Reduce data between given dates."""
    cm = arcreduce.CalManager()
    cm.load_data(start_date, stop_date)

    # clip low-SNR POINTs
    arcreduce.clip_values(cm.point.snr, -np.inf, 2, inclusive=True)

    # flag FLUXes with bad pointings
    arcreduce.flag_failed_point(cm.flux, cm.point, 4800)

    # clip total powers to plausible levels
    arcreduce.clip_values(cm.flux.atp, 10000, 50000, inclusive=False)
    arcreduce.clip_values(cm.flux.btp, 10000, 50000, inclusive=False)
    arcreduce.clip_values(cm.flux.ctp, 10000, 50000, inclusive=False)
    arcreduce.clip_values(cm.flux.dtp, 10000, 50000, inclusive=False)

    # clean the cal procedure
    cm.set_active_diodes(['CAL']) # use small cal diode
    clean_cal(cm.cal)

    # apply the CAL diode
    cm.apply_flux_cal(cm.flux, cm.cal.flux, 8.33/2.0)

    # All done.
    return cm
```

Listing A.1. The CalManager class is the foundation for calibration of data in the reduction pipeline.

Limits all CAL procedures to those using the specified diodes (CAL or NOISE). To activate all diodes, call with *diodes*=None. By default, all diodes are active.

- **set_active_sources**()

  Limits all procedures to those with the specified source name. To activate all sources, call with *sources*=None. By default, all sources are active.

- **boxcar_interpolate**(*source_time, source_y, target_time, dt_max, n_min, dy_max*)

  Creates a new Member containing the $\pm$*dt_max* seconds boxcar interpolation of the *source_y* member with sample times from *source_time*. The new samples are evaluated at times from *target_time*. If any such interpolation has fewer than *n_min* input samples or if

  $$|\max(y) - \min(y)| \,/\, |\min(y)| \geq \texttt{dy\_max}$$

  within that bin, that sample is flagged with the BAD_CAL flag.

- class arcreduce.**Procedure**: Container for the results from a particular radiometer procedure. These results are stored in Member objects that are accessible as attributes of the Procedure.

```
def clean_cal(cal):
    """Apply the cal cleaning process to the procedure."""
    # first flag data that's equal to zero (with a little wiggle-room
    # since they're float values)
    arcreduce.clip_values(cal.a, -1e-6, 1e-6, inclusive=True)
    arcreduce.clip_values(cal.b, -1e-6, 1e-6, inclusive=True)
    arcreduce.clip_values(cal.c, -1e-6, 1e-6, inclusive=True)
    arcreduce.clip_values(cal.d, -1e-6, 1e-6, inclusive=True)

    # Now clip values less than 5 which are completely unbelievable.
    arcreduce.clip_values(cal.flux, -np.inf, 5, inclusive=True)

    # Clip values with implausible measured uncertainties.  The range
    # 0 to 6 is acceptable; flag OUTSIDE that range.
    arcreduce.clip_errors(cal.flux, 0, 6, inclusive=False)

    # Apply iterative outlier filters.  We apply to swp twice because
    # the swd filter sometimes triggers a few new swp outliers.
    arcreduce.flag_outliers(cal.swp, 4)
    arcreduce.flag_outliers(cal.swd, 4)
    arcreduce.flag_outliers(cal.swp, 4)

    # Now sweep along in day-long buffers and flag outliers
    arcreduce.flag_sweep_outliers(cal.mjd, cal.flux, 1.0, 3.5)
```

Listing A.2. Helper function used by the reduction routine in Listing A.1.

`Member` objects can be added to a `Procedure` using simple Python assignment, e.g., `proc.new_mem=Member(...)`.

  - **`remove_flag`**(*flag*)

    Remove the specified flag from all data points in each `Member` of the `Procedure`.

- `class arcreduce.`**`Member`**: Representation of one parameter or result of the execution of a procedure.

  - **`get_err`**()

    Return a NumPy `ndarray` containing the uncertainties of the active elements of the `Member`.

  - **`set_err`**(*err*)

    Set the uncertainties of the active elements of the `Member` to the values given.

  - **`get_val`**()

    Return a NumPy `ndarray` containing the data values of the active elements of the `Member`.

  - **`set_val`**(*val*)

    Set the data values of the active elements of the `Member` to the values given.

- `arcreduce.`**`clip_errors`**(*mem, emin, emax, inclusive=False*)

Apply the ERROR_CLIP flag to data when the uncertainty in member *mem* is outside (inside if *inclusive* is True) the specified range.

- arcreduce.**clip_values**(*mem, xmin, xmax, inclusive=False*)

  Apply the VALUE_CLIP flag to data when the value in member *mem* is outside (inside if *inclusive* is True) the specified range.

- arcreduce.**flag_failed_point**(*target_proc, point_proc, dt_max*)

  Apply the FAILED_POINT flag to data in *target_proc* if the procedure in *point_proc* that immediately precedes it failed, or if there is no procedure within *dt_max* seconds prior.

- arcreduce.**flag_outliers**(*mem, nsigma*)

  Iteratively apply the OUTLIER flag to data that lie more than *nsigma* standard deviations from the mean of the *mem* member until no further data are flagged.

- arcreduce.**flag_sweep_outliers**(*xmem, ymem, dx, nsigma*)

  Apply the SWEEP flag to data in a sliding window spanning $\pm dx/2$ along *xmem* when the value in member *ymem* exceeds *nsigma* standard deviations from the mean in the window.

A.2 Low-Level Data Processing: `ArchiveReader`

The ArchiveReader class is the low-level Python interface to the data archive. It relies on the readarc module to extract data from the binary archive files recorded by the MCS control system. Details about the architecture and use of this class are described in section 3.1.2.2. In this section, we provide a detailed module reference for the classes and methods relevant to the ArchiveReader class.

A.2.1 `ArchiveReader` Class

- class arcreduce.**ArchiveReader**: Pythonic object-oriented wrapper around the readarc library. Instances of this class manage a data reduction session.

  - **add_frame_handler**(*self, fh*)

    Add a decoder that should be notified when data from a new frame are read. The object must implement the decoder interface and should normally be a subclass of the GenericDecoder class.

  - **add_output_handler**(*self, oh*)

    Add an output handler to be notified when a completed procedure result is available from any of the attached decoders. The object must implement the object handler interface and should normally be a subclass of the OutputHandler class.

– **add_register**(*self, reg_name, reg_type, index=0, length=None, our_name=None*)

Add a register to the set that will be extracted from each frame.

* *reg_name*: name of the register in the archive
* *type*: one of the RT_* types indicating the type of the register
* *index*: index of the register element to read (default: 0)
* *length*: max length of the array to be returned, only used for arrays where it must be set
* *our_name*: name by which to refer to the register (see below)

If a register is added with an existing name, an `ArchiveError` will be raised if the parameters for the new register do not exactly match the existing entry. Otherwise the second addition of the register will have no effect.

If a local name is needed (e.g., to provide a shorthand name for one index in a multi-index register), specify it with `our_name`. By default, registers are known by their official names. For example, to extract `mount.tracker.horiz_actual[1]` and refer to it by the name `mount.tracker.horiz_actual_el`, one would call

```
add_register('mount.tracker.horiz_actual', RT_FLOAT, index=1,
    our_name='mount.tracker.horiz_actual_el')
```

– **dispatch_frame**(*self*)

Call all registered frame handlers to notify of new frame.

Might be able to allow handlers to notify this class which frame labels are of interest, but would have to add a protocol to let them detect the end of a sequence of frames. For now easier to just let them implement that as needed.

Handler should return a `ProcedureData` object when a complete procedure has been processed. It will then be output using the current output method. If a procedure is not complete, return `None`.

– **handle_completed_procedure**(*self, p*)

Manage output of a completed procedure by calling output handlers.

– **handle_frame**(*self*)

Process the next frame. Returns `True` until all frames are consumed. After each call, the label of the frame just processed will be available from the `self.last_frame_label` member. (This will be `None` before the first or after the last frame).

– **read_register**(*self, our_name*)

Read a register, properly managing the data type.

- **update_registers**(*self*)

  Read next frame and all the configured registers in an unspecified order. Returns `True` on success, `False` if no more frames were available.

A.2.2 Decoders

- class arcreduce.**GenericDecoder**: Generic base class for decoder implementations.

  A decoder is an object that can be called by the `ArchiveReader` to process frames for a particular type of procedure (or, perhaps, for other reasons). The `GenericDecoder` base class provides generally useful features, such as a dynamically sized buffer for storing samples.

  Buffers are stored as NumPy `ndarray` objects with an array size that is generally larger than the valid data stored. The `buffer_index` member stores the index of the next element to be written, which equals the number of valid entries in the array. Buffers added through `add_buffer()` will be managed (via `reset_buffers()` and `grow_buffers()`). These can be accessed as member variables (i.e., `self.x`). There is some risk of namespace collisions, but this is unlikely.

  Procedure decoders should be subclasses of `GenericDecoder`. To implement the decoder interface, they must provide `handle_frame(`*`self, timestamp, frame_label, registers`*`)` to process data from each frame and a `install_registers(`*`self, ar`*`)` method to notify their parent `ArchiveReader` instance of the registers from which they require data. These methods are not explicitly listed in the references below for subclasses of this class.

  - **add_buffer**(*self, buffer_name, dtype, register_name=None*)

    Add a buffer to be managed.

  - **finalize_buffers**(*self*)

    Clip the buffers to their actual length so that they can be accessed without considering the `buffer_index` member. Do not load any additional data after doing this.

  - **grow_buffers**(*self*)

    Increase the size of the managed buffers by a factor of 2.

  - **reset_buffers**(*self*)

    Resets the data buffers and associated data to prepare for a fresh decoding attempt.

  - **update_buffers**(*self, registers, data_len*)

    Pull data out of the register associated with each buffer. Pulls `data_len` samples out of each register. Grows buffers as needed.

- class arcreduce.**MillisecondSampleDumpDecoder**(GenericDecoder): Decoder for dumping millisecond samples. Implements the decoder interface.

- class arcreduce.**Point2dDecoder**(GenericDecoder): Class to decode POINT2D procedures. The FWHM beam width must be specified when instantiating this object. Implements the decoder interface.

  - **decode**(*self*)

    Decode the results and compute flux / bg levels and some statistics.

    We do not fit the beam ourselves, we just take the results from the on-line procedure. We compute the signal-to-noise ratio (SNR) in a method roughly analogous to the old control system's SNR method, which was the minimum of the peak or half-power SNRs. Here, we take the minimum among any points measured at half-power or greater.

    Raises PointError if there is a problem decoding the data.

  - **find_result_frame_index**(*self*)

    Find the frame that indicates the completion of the procedure.

  - **find_valid_data**(*self, start, end, acq_delay=6*)

    Find valid data for integrating.

    Looks only between start and end indexes. Currently just identifies the last run of source-acquired data according to tracker_state. Returns indexes of valid data.

    Note: indexes are relative to the same zero-point as start/end, i.e., the absolute origin of the buffers.

    Waits acq_delay frames (seconds) after acquisition before it considers the source really acquired.

    Raises:

    * AcquisitionLostError: lost acquisition during the segment.
    * NoDataError: no valid data available.

  - **finish_proc**(*self, timestamp*)

    Called when the end of a procedure has been found. Processes the data and fills in a PointProcedureData object with the result.

  - **fit_gauss_amp**(*self, x, y, y_err, fwhm, amp0, bg0*)

    Helper function. Fits amplitude and background level of Gaussian.

    * fwhm: FWHM of the Gaussian
    * amp0: initial guess for amplitude
    * bg0: initial guess for background level

  - **identify_segments**(*self*)

    Identify pointing segments in the data.

This only identifies the start and end index of the period during which the telescope was intended to track a particular offset. It does not check that the telescope was acquired, that is done elsewhere.

- **integrate_segment**(*self, start, end*)

  Integrate a single segment.

- **integrate_segments**(*self, segs*)

  Integrate valid data within all the segments.

- `class arcreduce.`**PointDecoder**`(GenericDecoder)`: Class to decode POINT procedures. Implements the decoder interface.

  - **find_segments**(*self*)

    Locate the start and and indexes of each of the POINT segments.

    Since the details of when tracker acquisitions occur is a bit fragile, we look at the commanded tracker offsets as a more reliable indicator of what is going on. We count the final acquisition within each segment as the one intended for integration.

  - **finish_proc**(*self, timestamp*)

    Called when the end of a procedure has been found. Processes the data and fills in a `PointProcedureData` object with the result.

- `class arcreduce.`**RegisterDumpDecoder**`(GenericDecoder)`: Decoder class for dumping samples. Only works for one-sample-per-frame registers (will read only the first element from each frame for array registers). Implements the decoder interface.

- `class arcreduce.`**SimpleAbcdDecoder**`(GenericDecoder)`: Class for decoding ABCD procedures. This includes FLUX and CAL procedures, and eventually perhaps others. Implements the decoder interface.

  To create a decoder, subclass this and call its `__init__()` routine. Pass a four-element list with the names of the labels that correspond to the four states of the procedure. For a FLUX, these are 'flux:A' through 'flux:D'.

  - **finish_proc**(*self*)

    An entire procedure has been received, process it.

  - **flux_a_handle_frame**(*self, timestamp, frame_label, registers*)

    Handle 'flux:A' state and keep accepting 'flux:A' frames until 'flux:B' comes along.

  - **flux_b_handle_frame**(*self, timestamp, frame_label, registers*)

    Handle 'flux:B' state and keep accepting 'flux:B' frames until 'flux:C' comes along.

- **flux_c_handle_frame**(*self, timestamp, frame_label, registers*)

  Handle 'flux:C' state and keep accepting 'flux:C' frames until 'flux:D' comes along.

- **flux_d_handle_frame**(*self, timestamp, frame_label, registers*)

  Handle 'flux:D' state and keep accepting 'flux:D' frames, then wrap up.

- **handle_frame**(*self, timestamp, frame_label, registers*)

  An entire procedure has been received, process it.

- **identify_segment**(*self, segment_num, i_start, i_end*)

  Pick out a valid segment and return the valid indexes for the segment. Override this to change the behavior of identifying segments.

- **init_abcd**(*self*)

  Set or reset data to prepare for a new procedure.

- **store_samples**(*self, registers*)

  Store samples to the appropriate location in `self.switched_samples`.

  Updates auxiliary variables as necessary. Data to copy should be passed in the `new_samples` argument. Extra samples will be ignored.

- **waiting_handle_frame**(*self, timestamp, frame_label, registers*)

  Method to implement the "waiting" state: goes from anything to 'flux:A'.

- class arcreduce.**SimpleCalDecoder**(SimpleAbcdDecoder): Class for decoding CAL procedures. This uses only the mean value from each frame rather than the low-level millisecond samples. Implements the decoder interface.

  - **finish_proc**(*self*)

    Finish the procedure. Overrides the parent class. Uses the `SimpleAbcdDecoder` to do the real work, but need to do a few CAL-specific checks also.

  - **identify_segment**(*self, segment_num, i_start, i_end*)

    Identify segment indices for a CAL procedure.

- class arcreduce.**SimpleFluxDecoder**(SimpleAbcdDecoder): Class for decoding FLUX procedures. This uses only the mean value from each frame rather than the low-level samples. Implements the decoder interface.

  - **finish_proc**(*self*)

    Finish the procedure. Overrides the parent class. Uses the `SimpleAbcdDecoder` to do the real work, but need to do a few FLUX-specific checks also.

A.2.3 Output Handlers

- class arcreduce.**OutputHandler**: Generic output handler class, just defines the interface. Subclasses must provide a handle_procedure(*self, p*) to process ProcedureData objects containing processed results.

- class arcreduce.**CallbackOutputHandler**(OutputHandler): Output class that passes the output procedure to a callback. The callback should accept two parameters. First is the procedure just completed, second is the argument passed to the __init__() routine. Implements the output handler interface. This is the output handler used for actual data reduction in the pipeline.

- class arcreduce.**FileOutputHandler**: Output class that dumps to a file handle. Default output file is stdout.

  - **handle_cal_procedure**(*self, p*)

    Output details of a CAL procedure.

  - **handle_flux_procedure**(*self, p*)

    Output details of a FLUX procedure.

  - **handle_point_procedure**(*self, p*)

    Output details of a POINT procedure.

A.2.4 Procedure Data Structures

- class arcreduce.**ProcedureData**: Generic class to contain procedure output data.

- class arcreduce.**AbcdProcedureData**(ProcedureData): Class to contain output data relevant to ABCD procedures. Not normally used directly.

- class arcreduce.**CalProcedureData**(AbcdProcedureData: CAL procedure data output class.

- class arcreduce.**FluxProcedureData**(AbcdProcedureData): FLUX procedure data output class.

- class arcreduce.**PointProcedureData**: Point procedure data output class.

A.2.5 Exceptions

- class arcreduce.**ArchiveReduceError**(Exception): Superclass for all the exceptions defined in this module.

- `class arcreduce.`**`ArchiveError`**`(ArchiveReduceError)`: Generic exception for errors during archive data processing.
- `class arcreduce.`**`AcquisitionLostError`**: Irrecoverably lost acquisition of a source during a procedure.
- `class arcreduce.`**`FluxError`**: Error encountered while processing a procedure with a `SimpleAbcdDecoder` (FLUX or CAL).
- `class arcreduce.`**`NoDataError`**: Encountered a procedure segment with no valid data.
- `class arcreduce.`**`OutputHandlerError`**: Error encountered during output handling.
- `class arcreduce.`**`PointError`**: Error occurred during handling of a POINT procedure.