The James Webb Space Telescope Data Calibration Pipeline

Howard Bushouse[‡*], Michael Droettboom[‡], Perry Greenfield[‡]

https://www.youtube.com/watch?v=o-D4TpRFza4

Abstract—The James Webb Space Telescope (JWST) is the successor to the Hubble Space Telescope (HST) and is currently expected to be launched in late 2018. The Space Telescope Science Institute (STScI) is developing the software systems that will be used to provide routine calibration of the science data received from JWST. The calibration operations use a processing environment provided by a Python module called `stpipe` that provides many common services to each calibration step, relieving step developers from having to implement such functionality. The `stpipe` module provides common configuration handling, parameter validation and persistence, and I/O management.

Individual steps are written as Python classes that can be invoked individually from within Python or from the `stpipe` command line. Any set of step classes can be configured into a pipeline, with `stpipe` handling the flow of data between steps. The `stpipe` environment includes the use of standard data models. The data models, defined using json schema, provide a means of validating the correct format of the data files presented to the pipeline, as well as presenting an abstract interface to isolate the calibration steps from details of how the data are stored on disk.

Index Terms—pipelines, astronomy

Introduction

Data coming from the electronic detectors in scientific instruments attached to telescopes (both on the ground and in space) look nothing like the end product on which astronomers do their analysis or the pictures that show up in the media. Raw images and spectra contain artifacts and extra signals that are intrinsic to the instrumentation itself, rather than the source being observed. These artifacts include things like dead detector pixels, pixel-to-pixel variations in sensitivity, background signal from the detector and instrument, non-linear detector response, anomalous signals due to impacts of cosmic-rays, and spatial distortions due to the optics. All anomalies must be removed or corrected before the data are suitable for scientific analysis. In addition, processing such as combining the data from multiple exposures and extracting one-dimensional spectra from the two-dimensional detector format in which they were recorded must also be performed. This is the job of astronomical data reduction and calibration pipelines.

The Space Telescope Science Institute (STScI), which is the science operations center for the Hubble Space Telescope (HST), has developed and maintained data calibration pipelines for all of the HST scientific instruments and is now in the process of developing the pipelines that will be used for the James Webb Space Telescope (JWST) after it is launched in late 2018. The HST pipelines for the different scientific instruments on the telescope were developed over a span of more than 20 years and hence show an evolution in both software languages and design. The pipelines for each instrument, which now number 11 over the 25 year history of HST, were all written independently of one another and used an assortment of programming languages, including the Subset Preprocessor (SPP) language [Tody83], which is unique to the astronomical community, Fortran, C, and Python. This assortment of languages made maintenance and enhancement rather difficult, and precluded any code sharing between instruments. The HST calibration pipelines also used monolithic, procedural designs, with very little modularity. This approach worked as long as data were allowed to flow uninterrupted from beginning to end, but made it very difficult, if not impossible, to start or stop processing midstream, skip one or more steps, or insert additional steps. Customizing the processing in this way is often necessary for an astronomer to get the most out of their particular observations.

The JWST calibration pipelines are being developed from scratch using a completely new design approach and using almost nothing but Python. There is a common framework for all 4 of the scientific instruments, with extensive sharing of routines and a common code base. The new design allows for flexibility in swapping in and out specific processing steps, easily changing the ordering of steps within pipelines, and the ability for astronomers to plug-in custom processing. This flexibility is necessary due to the fact that the knowledge of the science instruments and the intricacies of the data they produce is constantly evolving, often over the entire lifetime of the mission. The calibration pipelines will be used not only in the production environment at STScI, which will apply an initial round of processing to all data coming from JWST and archiving the results, but will also be distributed to astronmers to run at their home institutions. This gives the users the ability to rerun and refine the processing applied to their observations. The highly modular and flexible nature of the design will allow them to even add in their own custom processing steps, either as part of the pipeline itself or as standalone routines that are run on the data and then reinserted back into the pipeline flow.

Before continuing, a clarification of exactly what we mean by the term "pipeline" is in order. A high-level workflow management system is used to guide the entire flow of data processing. This end-to-end process includes the receipt of telemetry downlinks from the telescope, reformatting the raw telemetry packets into

* *Corresponding author: bushouse@stsci.edu*
‡ *Space Telescope Science Institute*

useful data file formats, integrating meta data from various database systems, reducing and calibrating the raw data read out from the detectors in order to remove instrumental artifacts, storing the fully reduced data into an archive, and automaticlly notifying the astronomers who obtained the observations that the data are available. The calibration pipelines reported on here concern only the middle step of reducing and calibrating the raw images and spectra so that they are ready for scientific analysis. As such, the calibration pipelines do *not* provide any kind of high-level process management functions, interfaces to databases, and so on. The calibration pipelines are strictly devoted to applying a series of operations to the pixel values that comprise an image in order to remove instrumental artifacts and place the data values onto scales involving physical units. The particular series of such steps varies according to the observation modes used by the different instruments on the telescope. The calibration pipelines define and control the data flow within these different series of processing steps. The calibration pipelines, therefore, don't require a large, high-level task scheduling and workflow management system (e.g. Luigi [BF12]). A separate high-level process management system is used to control the execution of all the pieces involved in the end-to-end system described above, of which the calibration pipelines are one small part.

A primary goal for the JWST calibration pipelines is to have the system distributable to astronomers to execute on their own systems at their home institutions. It's often necessary for an astronomer to tailor or modify the details of the processing that's applied to their particular observations in order to get the greatest scientific return. The calibration pipeline package has therefore been designed to be as light-weight and self-contained as possible in order to make it as easy as possible for users to install and run themselves. The only external interface required is to our Calibration Reference Data System (CRDS), which is used to supply reference data needed by some of the calibration steps. The CRDS server at STScI will accept requests for reference files from the client on an astronomer's home system and automatically download the requested files to their systems for use locally.

stpipe

The heart - or perhaps more appropriately, the nervous system - of the JWST calibration pipeline environment is a Python module called `stpipe`. `stpipe` manages individual processing steps that can be combined into pipelines. The `stpipe` environment provides functionality that is common to all steps and pipelines so that they behave in a consistent manner. It provides:

- running steps and pipelines from the command line
- parsing of configuration settings
- composing steps into pipelines
- file management and data I/O between pipeline steps
- interface to the Calibration Reference Data System (CRDS)
- logging

Each pipeline step is embodied as a Python class, with a pipeline being composed of multiple steps. Pipelines can in turn be strung together, just like steps, to compose an even higher-order flow. Steps and pipelines can be executed from the command-line using `stpipe`, which is the normal mode of operations in the production environment that processes data in real-time as it is downlinked from the telescope. The step and pipeline classes can also be instantiated and executed from within a Python shell, which provides a lot of flexibility for developers when testing the code and to astronomers who may need to occasionally tweak or otherwise customize the processing of their particular data sets.

When run from the command line, `stpipe` handles the parsing of configuration parameters that can be provided either as arguments on the command line or within configuration files. Configuration files use the well-known ini-file format and `stpipe` uses the ConfigObj library to parse them. `stpipe` handles all of the file I/O for each step and the passing of data between pipeline steps, as well as providing access within each step to a common logging facility. It also provides a common interface for all steps to reference data files that are stored in the STScI Calibration Reference Data System (CRDS). Having all of these functions handled by the `stpipe` environment relieves developers from having to include these features in each step or pipeline and provides a consistent interface to users as well.

Command-line Execution

`stpipe` can be used from the command line to execute a step or pipeline by providing either the class name of the desired step/pipeline or a configuration file that references the step/pipeline class and provides optional argument values. An example that directly calls a class is:

```
> strun jwst_pipeline.SloperPipeline input.fits
  --output_file="myimage.fits"
```

The same thing can be accomplished by specifying a config file, e.g.:

```
> strun sloper.cfg input.fits
```

where sloper.cfg contains:

```
name = "SloperPipeline"
class = "jwst_pipeline.SloperPipeline"
output_file = "myimage.fits"
save_calibrated_ramp = True
```

Note that in the absence of the user explicitly specifying an output file name for saving the results, `stpipe` includes a mechanism for constructing an output file name that is composed of the input root file name and the name of the pipeline or step class that has been applied to produce the output.

Python Execution

Steps and pipelines can also be called from within Python using the class "call" method:

```
>>> from jwst_pipeline import SloperPipeline
>>> SloperPipeline.call('input.fits',
                        config_file='sloper.cfg')
```

Logging

The `stpipe` logging mechanism is based on the standard Python logging framework. The framework has certain built-in things that it automatically logs, such as the step and pipeline start/stop times, as well as platform information. Steps can log their own specific items and every log entry is time-stamped. Every log message that's posted has an associated level of severity, including DEBUG, INFO, WARN, ERROR, and CRITICAL (the same levels provided in the Python stdlib). The user can control how verbose the logging is via arguments in the config file or on the command line.

Steps and Pipelines

Steps define the parameters that are available, their data types (specified in "configspec" format), and their default values. As mentioned earlier, users can override the default parameter values by supplying values in configuration files or on the command-line. Steps can be combined into pipelines, and pipelines are themselves steps, allowing for arbitrary levels of nesting.

Simple linear pipelines can be constructed as a straight sequence of steps, where the output of each step feeds into the input of the next. These linear pipelines can be started and stopped at arbitrary points, via arguments supplied by the user, with all of the status saved to disk and then resumed later if desired. More complex (non-linear) pipelines can be defined using a Python function, so that the flow between steps is completely flexible. This is useful, for example, when the output of a step is multiple products that need to be looped over by subsequent steps. Because of their non-linear nature, these more complex types of pipeline can not be started or stopped mid-stream. Both types of pipelines, however, allow the user to skip certain steps by supplying configuration overrides.

Step configuration files can also specify pre- and post-hooks, to introduce custom processing into the pipeline. The hooks can be Python functions or shell commands. This allows astronomers to examine or modify data, or insert a custom correction, at any point along the pipeline without needing to write their own Python code.

A hypothetical pipeline is shown below. In this example, the input data is modified in-place by each processing step and the results passed along from one step to the next. The final result is saved to disk by the `stpipe` environment. Each pipeline subclass inherits from the *Pipeline* class. The subclass defines the Steps that will be used so that the framework can configure parameters for the individual Steps. This is done with the `step_defs` member, which is a dictionary that maps step names to step classes. This dictionary defines what the Steps are, but says nothing about their order or how data flows from one Step to the next. That is defined in Python code in the Pipeline's `process` method. By the time the Pipeline's `process` method is called, the Steps in `step_defs` will be instantiated as member variables.

```
from jwst_lib.stpipe import Pipeline

# pipeline step imports
from jwst_pipeline.dq import dq_step
from jwst_pipeline.ipc import ipc_step
from jwst_pipeline.bias import bias_step
from jwst_pipeline.reset import reset_step
from jwst_pipeline.frame import frame_step
from jwst_pipeline.jump import jump_step
from jwst_pipeline.ramp import ramp_step

# setup logging
import logging
log = logging.getLogger()
log.setLevel(logging.DEBUG)

# the pipeline class
class SloperPipeline(Pipeline)

    spec = """
        save_cal = boolean(default=False)
    """

    # step definitions
    step_defs = {"dq" : dq_step.DQInitStep,
                 "ipc" : ipc_step.IPCStep,
                 "bias" : bias_step.SuperBiasStep,
                 "reset" : reset_step.ResetStep,
                 "frame" : frame_step.LastFrameStep,
                 "jump" : jump_step.JumpStep,
                 "ramp_fit" : ramp_step.RampFitStep,
                }

    # the pipeline process
    def process(self, input):
        log.info("Starting calwebb_sloper ...")

        input = self.dq(input)
        input = self.ipc(input)

        # don't apply superbias to MIRI data
        if input.meta.instrument.name != "MIRI":
            input = self.bias(input)

        # only apply reset and lastframe to MIRI data
        if input.meta.instrument.name == "MIRI":
            input = self.reset(input)
            input = self.frame(input)

        input = self.jump(input)

        # save the results so far
        if save_cal:
            input.save(product_name(self, "cal"))

        input = self.ramp_fit(input)

        log.info("... ending calwebb_sloper")
        return input
```

Another example listed below shows how a pipeline can be included within a pipeline, just like a step, using all the same means to declare the pipeline and receiving all the same configuration handling from `stpipe`. In this example an existing pipeline is first applied to the input, followed by two more individual steps.

```
from jwst_lib.stpipe import Pipeline

# pipeline and step imports
from jwst_pipeline.pipeline import sloper_pipe
from jwst_pipeline.wcs import wcs_step
from jwst_pipeline.flat import flat_step

# setup logging
import logging
log = logging.getLogger()
log.setLevel(logging.DEBUG)

# the pipeline class
class MyPipeline(Pipeline)

    # step definitions
    step_defs = {"sloper": sloper_pipe.SloperPipe,
                 "wcs" : wcs_step.WcsStep,
                 "flat" : flat_step.FlatStep,
                }

    # the pipeline process
    def process(self, input):

        slope_model = self.sloper(input)
        slope_model = self.wcs(slope_model)
        result = self.flat(slope_model)

        return result
```

Data Models

For nearly 35 years most astronomers, observatories, and astronomical data processing packages have used a common data file format known as the Flexible Image Transport System (FITS).

While a common file format has made it very easy to share data across groups of people and software, the format is used in many different ways to store the unique aspects of different types of observational data (e.g. images versus spectra). The burden of loading, parsing, and interpreting the contents of any particular FITS file has always fallen to the processing code that's trying to do something to the data. For the JWST calibration pipelines, the `stpipe` environment takes care of all the file I/O, leaving the developers of steps and pipelines to concentrate on processing the data itself.

This has been implemented through the use of software data models in `stpipe`, through which it performs all the necessary I/O between files on disk and the data models. The data models allow the on-disk representation of the data to be abstracted from the pipeline steps via the I/O mechanisms built into `stpipe`. The use of software data models in the processing steps also has the benefit of eliminating or at least being able to manage dependencies between the various steps. Because all of the actual science data and its associated meta data are completely self-contained within a model, each step has all of the information it needs to do its work. For example, if one of the final steps in a particular pipeline gets modified in some way, there's no need to restart the processing for a particular data set from the beginning. The results from the step immediately preceding the change can be reloaded and the modified step executed from that point. If a particular processing step changes the overall format or content of the data set in some way, the result is saved in a different type of data model. Each step can perform a check to ensure that the input it's been given conforms to the type of data model expected in that step. Any inconsistencies will be detected immediately and the process will shutdown with a warning to the user, rather than the undesirable behavior of having a step crash because the input data were not compatible with that step.

The `stpipe` models interface currently reads and writes FITS files, but will soon also support the Advanced Scientific Data Format (ASDF) file format being developed by STScI [DB15]. The interface provides the same methods of access within the pipeline steps whether the data is on disk or already in memory. Futhermore, the `stpipe` interface can decide the best way to manage memory, rather than leaving it up to the code in individual steps. The use of the data models isolates the processing code from future changes in file formats or keywords.

Each model is a bundle of array or tabular data, along with metadata. The structure of the data and metadata for any model is defined using JSON Schema [Dro14]. JSON Schema works with any structured data, such as YAML and XML. The data model schemas are modular, such that a core schema that contains elements common to all models can also include any number of additional sub-schema that are unique to one or more particular models.

An example is the simple "ImageModel", shown below, which contains a total of three 2-dimensional data arrays. The schema defines the name of each model attribute, its data type, array dimensions (in the case of data arrays), and default values. Attributes can also be designated as required or optional. The "core.schema.json" and "sens.schema.json" files contain additional definitions of metadata attributes.

```
{ "allOf": [
      {"$ref": "core.schema.json"},
      {"type" : "object",
          "properties" : {
                "data" :
                {"type" : "data",
                 "title" : "The science data",
                 "fits_hdu" : "SCI",
                 "default" : 0.0,
                 "ndim" : 2,
                 "dtype" : "float32"
                },

                "dq" :
                {"type" : "data",
                 "title" : "Data quality array",
                 "fits_hdu" : "DQ",
                 "default" : 0,
                 "dtype" : "uint32"
                },

                "err" :
                {"type" : "data",
                 "title" : "Error array",
                 "fits_hdu" : "ERR",
                 "default" : 0.0,
                 "dtype" : "float32"
                },
                "sens" : {"$ref": "sens.schema.json"}
           }
      }
  ]
}
```

Within the pipeline or step code the developer loads a data model using simple statements like:

```
from jwst_lib.stpipe import Step, cmdline
from jwst_lib import models

class FlatFieldStep(Step):

    def process(self, input):

        with models.ImageModel(input) as im:
            result = flat_field.correct(im)

        return result
```

In a case like this, `stpipe` takes care of determining whether "input" is a model already loaded into memory or a file on disk. If the latter, it opens and loads the file contents into an ImageModel. The step code then has direct access to all the attributes of the ImageModel, such as the data, dq, and err arrays defined in the ImageModel schema above. If this is the only step being executed, `stpipe` will save the returned data model to disk. If this step is part of a pipeline, on the other hand, `stpipe` will pass the returned data model in memory to the next step. At the end of the pipeline the final model will be saved to disk.

Conclusions

We are in the process of building the data calibration pipelines that will be used to remove instrumental artificats from images and spectra obtained by the James Webb Space Telescope. The calibration pipelines rely on the `stpipe` environment developed at STScI, which handles all data I/O and configuration hanlding for the individual calibration steps. The entire package is designed to be relatively light-weight and self-contained so that it can be easily distributed to and run by individual astronomers at their home institutions. Calibration steps and pipelines can be executed from the command line, or their classes can be instantiated and called from with an interactive Python environment. This latter feature in particular allows for great flexibility to tweak or enhance the processing that's applied to a given data set. A user can, for

example, invoke a standard pipeline or a set of individual steps from within Python and at any point during the processing apply their own custom processing to the resulting data model in an interactive way. The ability to interact in real time with the data as it proceeds through the processing is new to the JWST calibration environment and did not exist at all for users of Hubble Space Telescope data.

REFERENCES

[BF12] E. Bernhardsson and E. Freider. *The Luigi Python module*, https://github.com/spotify/luigi

[Dro14] M. Droettboom. *JSON Schema*, http://json-schema.org

[DB15] M. Droettboom and E. Bray. *The ASDF Standard*, http://asdf-standard.readthedocs.org/en/latest/

[Tody83] D. Tody. *A Reference Manual for the IRAF Subset Preprocessor Language*, 1983