The JWST Data Calibration Pipeline

Howard Bushouse, Jonathan Eisenhamer, and James Davies

Space Telescope Science Institute, Baltimore, MD, USA; `bushouse@stsci.edu`

Abstract. STScI is developing the software systems that will provide routine calibration of the science data received from the James Webb Space Telescope (JWST). The processing uses an 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 yaml 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.

1. Introduction

STScI 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). The HST pipelines 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 – a total of 11 over the history of HST — were written mostly independently of one another and used an assortment of languages, ranging from IRAF SPP to Fortran, C, and Python. This made maintenance and enhancement rather difficult, and precluded much code sharing between instruments. The HST 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.

The JWST calibration pipelines are being developed using a completely new design approach using mostly Python. There is a common framework for all 5 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 insert custom routines. The calibration pipelines will be distributed to astronmers, giving them the ability to rerun and refine the processing of their observations. The highly modular and flexible nature of the design will allow them to add 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. The calibration pipeline package

has been designed to be as light-weight and self-contained as possible in order to make it easy for users to install and run. The only external interface required is to our Calibration Reference Data System (CRDS), which is used to supply reference data needed by 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.

2. `stpipe`

The central 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 for 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, an interface to the CRDS, and logging.

Each 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. 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 tweak or otherwise customize the processing.

When run from the command line, `stpipe` handles the parsing of configuration parameters, which 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 CRDS. Having all of these functions handled by the `stpipe` environment relieves developers from having to include these features in each step and provides a consistent interface to users as well.

`stpipe` is used 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
```

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

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

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 user can control how verbose the logging is via arguments in the config file or on the command line.

3. Steps and Pipelines

Steps define parameters, their data types (in "configspec" format), and 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. Because of their non-linear nature, these types of pipeline can not be started or stopped mid-stream. Both types of pipelines, however, allow the user to skip 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.

Excerpts (for brevity) of a pipeline are 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.stpipe import Pipeline
from jwst.dq import dq_step
from jwst.ramp import ramp_step

# the pipeline class
class SloperPipeline(Pipeline)
    # step definitions
    step_defs = {"dq" : dq_step.DQInitStep,
                 "ramp_fit" : ramp_step.RampFitStep}

    # the pipeline process
    def process(self, input):
        log.info("Starting calwebb_sloper ...")
        input = self.dq(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 self.save_cal:
    self.save_model(input, "ramp"))
input = self.ramp_fit(input)
log.info("... ending calwebb_sloper")
return input
```

4. Data Models

The burden of loading, parsing, and interpreting the contents of FITS data files usually falls to the processing code that's trying to do something with the data. For the JWST calibration pipelines, the `stpipe` environment takes care of all the file I/O, leaving the developers to concentrate on processing the data. This is accomplished through the use of software 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 data models 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 meta data are completely self-contained within a model, each step has all of the information it needs to do its work. If a particular processing step changes the overall format or content of the data 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 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.

The models interface currently reads and writes FITS files, as well as the Advanced Scientific Data Format (ASDF) file format developed by STScI. The interface provides the same methods of access within the pipeline steps whether the data are on disk or already in memory. Furthermore, the interface can decide the best way to manage memory, rather than leaving it up to the step code. The use of data models also isolates the processing code from future changes in file formats or keywords. Each model is a bundle of array or tabular data, and meta data, with the model structure defined using schemas in YAML format. The model schemas are modular, such that a core schema that contains elements common to all models can include any number of additional sub-schema that are unique to one or more particular models.

Step code loads a data model using a simple statement like:

```
im = datamodels.ImageModel(input)
```

`stpipe` determins whether "input" is a model already in memory or a file on disk. If the latter, it loads the file into an ImageModel. The step code then has direct access to all attributes of the ImageModel, such as the data, dq, and error arrays defined in the ImageModel schema. If only a single step is executed, `stpipe` saves the returned data model to disk. If part of a pipeline, `stpipe` passes the returned data model in memory to the next step and saves the final result at the end of the pipeline.