PFS 2D Pipeline User Documentation Documentation

The PFS 2D Pipeline Team

Jan 25, 2019

CONTENTS:

CHAPTER

ONE

INTRODUCTION

The Prime Focus Spectrograph consists of approximately 2400 science fibers, distributed over a 1.3 deg^2 field, feeding four spectrographs, each comprised of blue, red and infrared arms which together cover wavelengths of 0.38 - 1.26 microns. It is the responsibility of the 2D Data Reduction Pipeline (DRP) to process the raw data to produce wavelength-calibrated, flux-calibrated coadded spectra suitable for science investigations.

This document describes how to use the pipeline software to deliver science spectra. First we outline how to install the software and how to use our package management software (EUPS). Next we outline how to ingest raw data into a data repository, and give an overview of common command-line arguments for the pipeline scripts. Then we describe how to use the two main components to the DRP to process that data: the calib construction pipeline and the science pipeline. Next, we give a survey of some of the building blocks for the pipeline. Finally, we give suggestions on where you can get help.

CHAPTER

TWO

INSTALLATION

The PFS 2D DRP uses components from the LSST Data Management stack. We currently use `v16_0` of the LSST stack, so installing the PFS 2D DRP requires also installing this version of the LSST stack. There are multiple ways of installing these.

1. *Using Docker* : use a pre-built Docker image containing both the LSST stack and the PFS 2D DRP modules.
2. *Install with a script* : use our script to both install the LSST stack and the PFS 2D DRP modules.
3. *Install manually* : install the LSST stack and then the PFS 2D DRP modules.

For most work, we recommend *Using Docker*, as this provides the exact same environment we use in development, thus completely eliminating installation problems; however, this may not be possible in all cases, and hence we provide the other installation mechanisms. Before choosing one, we encourage you to read through each of these, understanding the benefits and limitations of each.

No matter how you choose to install the software, be sure to read the section on *EUPS*, which gives instructions for loading the software into the environment. After you've installed the software, try *Testing your installation*.

2.1 Using Docker

Docker allows you to run a binary distribution in a pre-defined sandbox environment (an "image") on your machine. The great advantage of this is that this is the exact same image produced and used by the development team, which completely eliminates installation problems such as due to missing dependencies, incompatible dependencies, or unexpected environments. However, because the environment is carefully controlled, you need to explicitly specify any use of your machine's physical characteristics.

The use of Docker is straightforward on a user-administered machine, but since it runs as root, admins of shared systems may be reluctant to allow use of Docker. However, there are alternatives available which work around this problem, including Shifter and Singularity, allowing the use of Docker images without root.

Your favourite package manager should already have a build of Docker, so installation is simple, e.g.:

- Homebrew on Mac OSX: `brew cask install docker`
- MacPorts on Mac OSX: `sudo port install docker`
- Redhat/CentOS on Linux: `sudo yum -y install docker`
- Ubuntu on Linux: `sudo apt-get install docker`

The Docker web site has much more detailed information on installing Docker.

With Docker installed, you can download and start the image:

```
docker run -ti paprice/pfs_pipe2d:latest
```

That runs the image in its own area (a "container"), with none of your machine's resources (e.g., disks, network) shared. If you want to access files on your machine (e.g., data to be reduced, or code to be used) or persist anything you create beyond the lifetime of the container, you will need to mount some directories (with the argument `-v localDir:containerDir`) so they are accessible under Docker. If you want to be able to display X windows from the container (e.g., for matplotlib plots, or ds9), additional arguments are required. We recommend `bash` definitions like the following:

```
pfsDocker () {
    docker run -ti -v ~/docker:/home/pfs -v ~/pfs:/home/pfs/pfs $DOCKER_OPTIONS --cap-
↪add=SYS_PTRACE --security-opt seccomp=unconfined ${1-paprice/pfs_pipe2d:latest}
}
export DOCKER_OPTIONS="-e DISPLAY=docker.for.mac.localhost:0"  # For Mac OSX only
```

This mounts the `~/docker` directory on your machine to the container's home directory (which allows for persistence of anything you do under the home directory, including shell configuration files that make an environment comfortable), and mounts the `~/pfs` directory on your machine to the same directory in the container (useful for developing the code that you've put in that directory), and allows network communication between your machine and the container (you may have to do `xhost +localhost` on your machine to allow X windows from the container). You can choose a different image by specifying it on the command-line; e.g., the following runs an image that includes additional debugging facilities (i.e., gdb, valgrind, igprof and a library required by ds9):

```
pfsDocker paprice/pfs_pipe2d_debug:latest
```

Docker images are light on operating system features so as to minimize the size of the image. If you find that you need an additional feature in the operating system (e.g., a library or tool you want is missing), you have two choices. Firstly, the temporary solution is to log into the container as `root` and install what you want:

```
docker exec -ti --user=root <containerId> /bin/bash
```

You can find the `containerId` from the `hostname` of the container, or by running `docker ps` on your machine. Note that this modifies the container (not the image), so the effect will be lost when you exit the container. For a more permanent solution, you should build your own Docker image on top of the `pfs_pipe2d` image. To do this, you need to write a `Dockerfile`, which is a prescription for generating the image; then run:

```
docker build -t <tagname> -f /path/to/Dockerfile
```

Here, `<tagname>` is a symbolic name for the image; this is usually composed of a DockerHub username (e.g., `paprice`), a repository name (e.g., `pfs_pipe2d`), and a version tag (e.g., `latest`) all put together: `<username>/<repositoryName>:<versionTag>`. Once it's built, you can run your image in the same way as above. For an example `Dockerfile` of an image layered on top of the `pfs_pipe2d` image, see `Dockerfile.pipe2d_debug` and the accompanying `Makefile`.

If you're having problems in the container, try increasing the memory, since Docker defaults to a mere 2GB of memory for its containers. We regularly run with 4 cores and 8GB of memory, which generally works fine.

2.2 Install with a script

We provide scripts for building and installing the LSST stack and the PFS 2D DRP software on your machine. However, these require that several *Dependencies* are installed first, and they may be fragile under idiosyncratic environments. If you have problems with these scripts, see *Getting Help* or try an alternative installation method[1].

The scripts are:

- `install_lsst.sh`: installs the LSST stack.

[1] We recommend *Using Docker*, as that completely eliminates build problems.

- `build_pfs.sh`: builds the PFS 2D DRP software on top of an existing LSST stack installation.
- `install_pfs.sh`: installs the LSST stack and builds the PFS 2D DRP software on top.

These scripts can be obtained either by downloading them directly from GitHub, e.g.:

```
wget https://raw.githubusercontent.com/Subaru-PFS/pfs_pipe2d/master/bin/install_pfs.sh
```

or by cloning the entire pfs_pipe2d repository with `git` and then looking in the `bin` subdirectory:

```
git clone http://github.com/Subaru-PFS/pfs_pipe2d
cd pfs_pipe2d/bin
```

2.2.1 Dependencies

The LSST stack, on which the PFS software is built, requires the following Redhat/CentOS packages:

```
epel-release
bison curl blas bzip2-devel bzip2 flex fontconfig
freetype-devel git libuuid-devel
libXext libXrender libXt-devel make openssl-devel patch perl
readline-devel tar zlib-devel ncurses-devel cmake glib2-devel
java-1.8.0-openjdk gettext perl-ExtUtils-MakeMaker
which
```

If you're not running Redhat/CentOS, check the list of prerequisites for the LSST stack and install the packages you need for your system.

In addition to the above, `git-lfs` should be installed.

2.2.2 Install LSST+PFS

The last of the above-listed scripts, `install_pfs.sh`, combines the first two; it is the preferred choice for installing the software if you do not have an existing installation of the LSST stack. If you encounter a problem running this script, try running the first two scripts in succession, which will hopefully give more information on where the problem lies. Running the script with the `--help` or `-h` command-line arguments gives the usage information:

```
foo@bar:~/pfs/pfs_pipe2d/bin $ install_pfs.sh -h
Install the PFS 2D pipeline.

Usage: /home/foo/pfs/pfs_pipe2d/bin/install_pfs.sh [-b <BRANCH>] [-e] [-l] [-L
↪<VERSION>] <PREFIX>

    -b <BRANCH> : name of branch on PFS to install
    -e : install bleeding-edge LSST
    -l : limited install (w/o drp_stella, pfs_pipe2d)
    -L <VERSION> : version of LSST to install
    -t : tag name to apply
    <PREFIX> : directory in which to install
```

`-e`, `-l` and `-L` are black-belt options: do not use them unless you know what you are doing. The `-b` option allows you to specify a particular version of the PFS pipeline to install (e.g., a ticket branch, or an official release). The `-t` option allows you to apply a *EUPS* tag (often `current`). An example usage, which will install the master branch under `~/pfs/stack` and tag it as `current` is:

```
foo@bar:~/pfs/pfs_pipe2d/bin $ install_pfs.sh -t current ~/pfs/stack
[...]
All done.

To use the PFS software, do:

    source /home/foo/pfs/stack/loadLSST.bash
    setup pfs_pipe2d -t current
```

Follow the instructions to configure your environment[2].

2.2.3 Install PFS on existing LSST stack

The `build_pfs.sh` script builds the PFS 2D DRP software on top of an existing installation of the LSST stack. It is useful if you have already used `install_pfs.sh` and want to upgrade the PFS software version, or if you have independently installed the LSST stack (perhaps with the `install_lsst.sh` script, or manually).

Running the script with the `--help` or `-h` command-line arguments gives the usage information:

```
foo@bar:~/pfs/pfs_pipe2d/bin $ build_pfs.sh -h
Install the PFS 2D pipeline.

Requires that the LSST pipeline has already been installed and setup.

Usage: /home/foo/pfs/pfs_pipe2d/bin/build_pfs.sh [-b <BRANCH>] [-l] [-t TAG]

    -b <BRANCH> : name of branch on PFS to install
    -l : limited install (w/o drp_stella, pfs_pipe2d)
    -t : tag name to apply
```

`-l` is a black-belt option: do not use it unless you know what you are doing. The `-b` option allows you to specify a particular version of the PFS pipeline to install (e.g., a ticket branch, or an official release). The `-t` option allows you to apply a *EUPS* tag (often `current`)

Before running this script, make sure you have configured your environment so it is aware of the LSST stack (often by `source`ing a `loadLSST.bash` script; however you did it, `EUPS_PATH` should be set), and `setup pipe_drivers`. An example usage, which will install the master branch and tag it as `current` is:

```
foo@bar:~/pfs/pfs_pipe2d/bin $ build_pfs.sh -t current
```

2.3 Install manually

Manual installation is the least-recommended method of installing the PFS 2D DRP pipeline, because it is labor intensive and can be done in different ways, making installation problems more difficult to debug. However, it may provide a successful installation when the scripts fail (this is essentially what the scripts attempt to do). If you have problems, see *Getting Help* or try an alternative installation method[3].

Manual installation is achieved by first installing the LSST stack and then installing the PFS packages on top.

[2] The use of `-t current` in the `setup` command is not strictly necessary: `eups` defaults to looking for packages tagged `current`.
[3] We recommend *Using Docker*, as that completely eliminates build problems.

2.3.1 Install LSST

Follow the LSST install instructions. Make sure you install the correct version of the LSST stack (currently, we use `v16_0`). Instead of installing the `lsst_distrib` product, you can install just `pipe_drivers` for a faster install[4]. Follow their instructions for configuring your environment, and `setup pipe_drivers`.

2.3.2 Install PFS packages

Install the following PFS packages, in this order:

- datamodel
- obs_pfs
- drp_stella
- pfs_pipe2d[5]

Installation of each package involves:

1. Download the package. You can either use `git`:

   ```
   git clone http://github.com/Subaru-PFS/<packageName>
   ```

   or you can download the package directly:

   ```
   curl -Lfk https://api.github.com/repos/Subaru-PFS/<packageName>/tarball/master |␣
   ↪tar xvz
   ```

2. Change into the package directory.
3. Put the package into your environment:

   ```
   setup -k -r .
   ```

   Note the use of the `-k` flag, which tells *EUPS* to *keep* the current versions of any dependencies you've configured (so versions won't change underneath you).

4. Build and install the package:

   ```
   scons install declare --tag=current
   ```

   (The use of `--tag=current` is optional, but it makes it easier to select later.)

5. Put the installed version of the package into your environment:

   ```
   setup <packageName>
   ```

   You may also need to specify a version or tag name to select the correct version.

2.4 Testing your installation

The `pfs_pipe2d` package includes an integration test, which should run all the way through if your installation is working.

[4] You may also want to install the `display_ds9` and/or `display_matplotlib` products, if you intend to use the `lsst.afw.display` functionality.

[5] The `pfs_pipe2d` package is not strictly necessary for running the PFS 2D DRP, but it contains the integration test, which is useful for validating the installation.

First, be sure you've loaded the pipeline software into your environment:

```
eups list -s pfs_pipe2d
```

If that generates an error (`eups list: Unable to find product pfs_pipe2d tagged "setup"`) then you need to load the pipeline software into your environment:

```
setup pfs_pipe2d
```

Now, you should be able to be able to access `pfs_integration_test.sh`. The usage information is:

```
Exercise the PFS 2D pipeline code

Usage: /home/pfs/pfs/pfs_pipe2d/bin/pfs_integration_test.sh [-b <BRANCH>] [-r <RERUN>
↪] [-d DIRNAME] [-c CORES] [-n] <PREFIX>

    -b <BRANCH> : branch of drp_stella_data to use
    -r <RERUN> : rerun name to use (default: 'integration')
    -d <DIRNAME> : directory name to give data repo (default: 'INTEGRATION')
    -c <CORES> : number of cores to use (default: 1)
    -G : don't clone or update from git
    -n : don't cleanup temporary products
    <PREFIX> : directory under which to operate
```

The main options you should care about are `-c` (more cores makes it go a bit faster; but you won't see much gain beyond about 4 cores) and the `PREFIX` positional argument (where to do the test). The `-b` option is for developers testing new features. The `-r` and `-d` allow different runs of the integration test in the same directory. Don't use the `-G` option unless you know what you're doing. The `-n` option keeps some temporary products around, at the cost of more disk usage.

We recommend running the integration test something like this:

```
mkdir -p /path/to/integrationTest
cd /path/to/integrationTest
pfs_integration_test.sh -c 4 .
```

CHAPTER

THREE

EUPS

EUPS stands for "Extended[1] Unix Products System". It is a system for managing software products with multiple versions, along with their dependencies. The great strength of EUPS compared to a regular system package manager (like Homebrew or yum) is that it allows different versions of software products to be used, so you can substitute one version for another[2] without the need to recompile (it is similar to the `module` command commonly used on clusters). It works by configuring certain environment variables that Unix-ish systems recognise, including `PATH` and `LD_LIBRARY_PATH`[3]. The code is available on GitHub.

EUPS has a great many features, but many are not relevant for simply using the PFS 2D DRP. Here we introduce a few useful commands that will allow you to operate EUPS for use with the pipeline.

3.1 setup

`setup` is the main interface of EUPS. It loads a particular version of a particular package (and, by default, its dependencies) into the environment. For example:

```
setup pfs_pipe2d
```

will load the `pfs_pipe2d` product tagged `current`, and its dependencies.

To load a particular version (and its dependencies), simply specify the version along with the product:

```
setup pfs_pipe2d 5.0
```

To use the definition of the `pfs_pipe2d` that you have locally (e.g., cloned from GitHub and being edited) but not modify any dependencies that have already been loaded:

```
setup -j -r /path/to/pfs_pipe2d
```

Here, the `-j` flag means to load *just* this package (i.e., no dependencies). Its cousin, the `-k` flag means to load dependencies, but *keep* the version of any dependency that has already been loaded.

3.2 unsetup

`unsetup` is the opposite of `setup`: it removes a particular package (and, by default, its dependencies) from the environment. The `-j` flag means the same as for `setup` (i.e., don't `unsetup` the dependencies), and is often useful.

[1] Or "Evil", depending on who you ask.
[2] Modulo concerns about ABI compatibility.
[3] `DYLD_LIBRARY_PATH` on OSX; and to work around SIP, `LSST_LIBRARY_PATH` as well.

3.3 eups list

`eups list` lists the available packages and versions, along with tags applied to them. If a particular package and version has been loaded, `setup` is among the tags.

For a list of packages that have been loaded, use:

```
eups list -s
```

You can list the available versions of a particular package by specifying the package name, e.g.,:

```
eups list pfs_pipe2d
```

Or, to discover what version of a package you're using, use the `-s` flag with the package name, e.g.:

```
eups list -s pfs_pipe2d
```

3.4 Tags

Packages can be tagged with a symbolic name (e.g., the ticket name during development). By default, the list of supported symbolic names is limited to `current` and the user name[4]. To expand the list of supported symbolic names, you need to edit your `~/.eups/startup.py` file to include, e.g.:

```
hooks.config.Eups.userTags += ["myTag1", "myTag2"]
```

Then you can tag package versions:

```
eups declare pfs_pipe2d 5.0 -t myTag1
```

[4] This limitation is deliberate, to prevent typos.

CHAPTER

FOUR

INGESTION OF RAW DATA

All data, both raw and the pipeline products, live in a "data repository". Ideally, from the point of view of the user, this "data repository" is a black box that contains the data: the user should never assume a particular implementation for the data repository (including directory structure and filenames), but instead access it through the *Data Butler*.

Here, we create a data repository and stuff raw data into it, for use by the pipeline.

4.1 Creating the data repository

Create the data repository by making an empty directory, and inserting a `_mapper` file with the correct contents:

```
mkdir -p /path/to/dataRepo
echo lsst.obs.pfs.PfsMapper > /path/to/dataRepo/_mapper
```

4.2 Ingesting the data

Use `ingestPfsImages.py` to ingest data into the data repository:

```
ingestPfsImages.py /path/to/dataRepo /path/to/rawData/PFFA*.fits
```

Note: The `ingestPfsImages.py` script assumes that the `pfsConfig` files are in the same location as the raw images. The `pfsConfig` files are required in order to reduce the data, and will be ingested along with the raw images.

The default behaviour of `ingestPfsImages.py` is to link the files into the data repository. This behaviour can be changed with the `--mode` command-line argument (e.g. `--mode move` or `--mode copy`).

Run `ingestPfsImages.py` with the `--help` command-line argument for an extensive list of command-line arguments that are supported.

Note: `ingestPfsImages.py` will not work well with the `-j` argument (multiple processes) as it is designed to run with a single process.

4.3 Data Butler

Once data have been ingested into the data repository, you can access it through the data butler. The data butler finds data of the requested dataset type on the basis of keyword-value pairs that identify the data, e.g., in python:

```
>>> from lsst.daf.persistence import Butler
>>> butler = Butler("/path/to/dataRepo")
>>> image = butler.get("raw", visit=12, spectrograph=1, arm="r")
>>> image
<lsst.afw.image.exposure.exposure.ExposureU object at 0x7ff0b4cff490>
```

The keywords `visit`, `spectrograph` and `arm` are sufficient to identify a single image, but they are not all that is available. You can search the repository for keyword values using `Butler.queryMetadata`:

```
>>> keywords = ["visit", "spectrograph", "arm"]
>>> for values in butler.queryMetadata("raw", keywords, field="OBJECT"):
...     dataId = dict(zip(keywords, values))
...     print(dataId, butler.get("raw", dataId))
...
{'visit': 32, 'spectrograph': 1, 'arm': 'r'} <lsst.afw.image.exposure.exposure.
↪ExposureU object at 0x7f1fd22e4880>
{'visit': 33, 'spectrograph': 1, 'arm': 'r'} <lsst.afw.image.exposure.exposure.
↪ExposureU object at 0x7f1fd22e4dc0>
```

Note: The data butler has several shortcomings, but LSST is working on a redesigned version.

CHAPTER

FIVE

COMMON COMMAND-LINE ARGUMENTS

Many (all?) of the scripts use a common front-end argument parser, which means many of the possible arguments are common across many (all?) of the scripts. If you run a script with the `--help` or `-h` argument, it shows what arguments are possible:

```
positional arguments:
input                 path to input data repository, relative to
                        $PIPE_INPUT_ROOT

optional arguments:
-h, --help            show this help message and exit
--calib RAWCALIB      path to input calibration repository, relative to
                        $PIPE_CALIB_ROOT
--output RAWOUTPUT    path to output data repository (need not exist),
                        relative to $PIPE_OUTPUT_ROOT
--rerun [INPUT:]OUTPUT
                        rerun name: sets OUTPUT to ROOT/rerun/OUTPUT;
                        optionally sets ROOT to ROOT/rerun/INPUT
-c [NAME=VALUE [NAME=VALUE ...]], --config [NAME=VALUE [NAME=VALUE ...]]
                        config override(s), e.g. -c foo=newfoo bar.baz=3
-C [CONFIGFILE [CONFIGFILE ...]], --configfile [CONFIGFILE [CONFIGFILE ...]]
                        config override file(s)
-L [LEVEL|COMPONENT=LEVEL [LEVEL|COMPONENT=LEVEL ...]], --loglevel␣
↪[LEVEL|COMPONENT=LEVEL [LEVEL|COMPONENT=LEVEL ...]]
                        logging level; supported levels are
                        [trace|debug|info|warn|error|fatal]
--longlog             use a more verbose format for the logging
--debug               enable debugging output?
--doraise             raise an exception on error (else log a message and
                        continue)?
--noExit              Do not exit even upon failure (i.e. return a struct to
                        the calling script)
--profile PROFILE     Dump cProfile statistics to filename
--show SHOW [SHOW ...]
                        display the specified information to stdout and quit
                        (unless run is specified).
-j PROCESSES, --processes PROCESSES
                        Number of processes to use
-t TIMEOUT, --timeout TIMEOUT
                        Timeout for multiprocessing; maximum wall time (sec)
--clobber-output      remove and re-create the output directory if it
                        already exists (safe with -j, but not all other forms
                        of parallel execution)
--clobber-config      backup and then overwrite existing config files
                        instead of checking them (safe with -j, but not all
```

(continues on next page)

(continued from previous page)

```
                        other forms of parallel execution)
--no-backup-config    Don't copy config to file~N backup.
--clobber-versions    backup and then overwrite existing package versions
                        instead of checkingthem (safe with -j, but not all
                        other forms of parallel execution)
--no-versions         don't check package versions; useful for development
--id [KEY=VALUE1[^VALUE2[^VALUE3...] [KEY=VALUE1[^VALUE2[^VALUE3...] ...]]
                        data IDs, e.g. --id visit=12345 ccd=1,2^0,3

Notes:
            * --config, --configfile, --id, --loglevel and @file may appear multiple␣
↪times;
                all values are used, in order left to right
            * @file reads command-line options from the specified file:
                * data may be distributed among multiple lines (e.g. one option per␣
↪line)
                * data after # is treated as a comment and ignored
                * blank lines and lines starting with # are ignored
            * To specify multiple values for an option, do not use = after the option␣
↪name:
                * right: --configfile foo bar
                * wrong: --configfile=foo bar
```

5.1 input

This is the path to the data repository. As a positional argument, it does not have any `--whatever` string preceding it.

5.2 --calib

This is the path to the calib repository. It *should* default to a `CALIB` subdirectory of the data repository, but we've seen instances where that is broken, so we encourage you to list it explicitly.

5.3 --output

This specifies a directory which will become a data repository holding the outputs of the script. This is useful for writing the outputs to a different destination than the inputs, but in general we recommend using the `--rerun` option instead.

5.4 --rerun

This specifies a symbolic name for the processing you're undertaking. The output data will then be written to `<input>/rerun/<rerunName>` in the data repository. This defaults to your username, which is helpful for avoiding mixups in output products between different users, but in general we recommend using something like `--rerun <username>/<projectName>` (e.g., `--rerun price/pipe2d-310`), as that gives the output products an informative name that can be used later to identify what it is.

You can also specify separate input and output rerun names, separated by a colon. This is useful for when you want to build on the results from one rerun, but want to write to a separate directory, e.g., `--rerun price/pipe2d-310:price/pipe2d-310-tweak`.

5.5 `--config` and `-c`

These allow you to dynamically modify the configuration that will be used in the processing, directly on the command-line, by specifying (potentially) multiple configuration `keyword=value` pairs. This is very convenient for testing how configuration tweaks modify the processing. The trick, of course, is identifying the appropriate configuration keyword names; for that, try `--show config`.

The values must be plain-old-data (string/integer/float); for more complicated configuration settings, see `--configfile` and `-C`.

5.6 `--configfile` and `-C`

These allow you to dynamically modify the configuration that will be used in the processing, by specifying a configuration override file. The configuration override file is written in python, modifying an implicitly-declared value called `config`, which is the configuration tree. This allows you to change the values in a list, or set values programmatically using code, e.g.:

```
config.foo.masks = ["NO_DATA", "SAT", "BAD"]
for thingy in (config.foo.thingy, config.bar.thingy):
    thingy.whatsit = "gizmo"
```

It also allows for the substitution of modules using the `retarget` method, e.g.:

```
from pfs.drp.stella.alternativeImpl import AlternativeTask
config.foo.retarget(AlternativeTask)
```

As with `--config`/`-c`, the trick is identifying the appropriate configuration keyword names; for that, try `--show config`.

5.7 `--loglevel` and `-L`

These change the verbosity level of the logging. You can set the logging level globally (for all logging) by specifying the new logging level; or just override the logging level for a single component by `<component>=<level>`. The logging levels, in order of decreasing verbosity are: `trace`, `debug`, `info`, `warn`, `error`, `fatal`.

The default logging level is `info`. If you're having trouble figuring out why something is happening, try `-L debug`. If you're getting too much output to the screen, try `-L warn`.

If you want to change just a single component, the logging name should be present in the log messages; it should also be the same as for corresponding entry in the configuration tree (without the leading `config.`, of course).

5.8 `--longlog`

This enables a longer format for the logging messages.

5.9 --debug

This enables debugging output from the pipeline, which might include plots or image displays. This requires putting a file named `debug.py` somewhere on your `PYTHONPATH`, with contents similar to the following:

```
try:
    import lsstDebug
    print("Importing debug settings...")
    def DebugInfo(name):
        di = lsstDebug.getInfo(name)  # Note: lsstDebug.Info(name) would call us␣
↪recursively
        if name in (
#                    "pfs.drp.stella.fitContinuum",
                    ):
            di.display = True
            di.plot = True
        elif name in (
                    "pfs.drp.stella.calibrateWavelengthsTask",
                    ):
            di.display = True
            di.showArcLines = True
            di.showFibers = range(3000)
            di.plotWavelengthResiduals = True
            di.plotArcLinesLambda = True
            di.arc_frame = 1
    lsstDebug.Info = DebugInfo
    lsstDebug.frame = 1
except ImportError:
    import sys
    print("Unable to import lsstDebug;  not setting display intelligently", file=sys.
↪stderr)
```

Unfortunately, there's not a good catalog of what settings are possible, so setting this up usually requires knowledge of the code. But if you're using this then you're probably fixing some code, so you'll be able to figure that out.

5.10 --doraise

This has the script raise an exception in the event of an error, rather than swallowing the exception and attempting to push on. This is useful in combination with the `pdb` debugger:

```
python -m pdb $(which reduceExposure.py) ... --doraise
```

will run `reduceExposure.py` and put drop you into the debugger when it hits the exception (it does start you off in the debugger; tell it `c` to continue, and then it will run the program until it hits the exception).

5.11 --noExit

You can generally ignore this one.

5.12 --profile

This runs the main operation under `cProfile`, and dumps the profile statistics to the provided filename. You can then get a basic profile in python:

```
>>> from pstats import Stats
>>> stats = Stats("myStats.dat")  # Or whatever your filename was
>>> stats.sort_stats("cumulative")
>>> stats.print_stats(30)  # Print top 30
```

This probably won't work well with the MPI-based scripts, but they have their own way of dumping profile statistics.

5.13 --show

This provides information about what the script is going to operate on, and how it's configured. This argument takes one or more keywords:

- `config[=PATTERN]`: Show the configuration tree, or just the entries that match the glob pattern.
- `history=PATTERN`: Show where the configuration entries that match the glob pattern were set.
- `tasks`: Show the task hierarchy.
- `data`: Show the data that will be processed.
- `run`: Continue running after showing whatever was requested; if this is not specified, the script will exit.

5.14 -j

This provides for parallelisation of the processing. Specify the number of processes to use.

This does not work with MPI-based scripts, but they have their own way of doing parallelisation (e.g., `--cores`).

5.15 --timeout and -t

This specifies the maximum wall time in seconds for the processing when running in parallel. This should be set to a very large number so that it never needs to be modified.

5.16 --clobber-output

This removes and recreates the output directory if it already exists. I've never used it, so I'm not sure it works.

5.17 --clobber-config

The scripts save their configuration tree, and if a configuration tree has already been saved then they check to see if there's any difference. This helps prevent accidental mixing of multiple configuration settings in a single output. If this argument is specified, then instead of checking for configuration differences, the script will clobber the existing saved configuration tree.

Danger: In general, this argument should **never** be used during production. It is acceptable to use this when developing or debugging, using a separate output directory or rerun.

5.18 --no-backup-config

When `--clobber-config` is used, the script will make a backup copy of the old (clobbered) configuration. This argument disables that behaviour.

5.19 --clobber-versions

The scripts save information on the software versions being used, and if this information has already been saved then they check to see if there's any difference. This helps prevent accidental mixing of multiple versions in a single output. If this argument is specified, then instead of checking for configuration differences, the script will clobber the existing saved software version information.

Danger: In general, this argument should **never** be used during production. It is acceptable to use this when developing or debugging, using a separate output directory or rerun.

5.20 --no-versions

If this argument is specified, no check will be made for software version differences.

Danger: In general, this argument should **never** be used during production. It is acceptable to use this when developing or debugging, using a separate output directory or rerun.

5.21 --id

This argument specifies the data to be processed. Whenever a pipeline command takes an `--id` argument, *any* set of keyword-value pairs can be used that makes sense. For example, one can specify an explicit list of visits (either joining individual values with a caret, `^`, or specifying a range with `..`):

```
--id visit=1^2^3^4^5
--id visit=1..5
```

or specifying every second one:

```
--id visit=2^4^6^8^10
--id visit=2..10:2
```

Alternatively, other keywords can be used if that is sufficient to uniquely identify the data:

```
--id field=BIAS dateObs=2019-01-11
```

would select all bias exposures from 2019-01-11.

The following keywords are currently supported:

- `site` (string): where the data was obtained:
  - `J`: JHU
  - `L`: LAM
  - `X`: Subaru offline
  - `I`: IPMU
  - `A`: ASIAA
  - `S`: Summit
  - `P`: Princeton
  - `F`: Simulation ("fake")
- `category` (string): category of data:
  - `A`: science
  - `B`: NTR
  - `C`: Meterology
  - `D`: HG
- `field` (string): the field target (usually recorded by the observer).
- `expId` (integer): the exposure identifier (an alias for `visit`)[1].
- `visit` (integer): the exposure identifier.
- `ccd` (integer): the CCD identifier (unique combination of `spectrograph` and `arm`).
- `filter` (string): an alias for `arm`[2].
- `arm` (string): the arm of the spectrograph:
  - `b`: Blue
  - `r`: Red
  - `m`: Medium-resolution, red.
  - `n`: Near infrared.
- `spectrograph` (integer): the spectrograph module number (1-4).
- `dateObs` (string): the date of observation (YYYY-MM-DD).
- `expTime` (float): the exposure time (sec).
- `dataType` (string): type of data (e.g., bias, dark, flat, science).
- `taiObs` (string): the time of observation (HH:MM:SS.sss)
- `pfiDesignId` (int): the top-end configuration design identifier
- `slitOffset` (float): the applied x offset of the slit

[1] We would like to get rid of `visit`, but it is baked into the current version of the data butler.
[2] `filter` doesn't make much sense for a spectrograph, but it is required by the current version of the data butler.

CHAPTER

SIX

CALIBRATIONS

"Calibs" are versioned calibration products wherein the behavior of the instrument is modeled (often using dedicated observations) and recorded for use in removing the instrumental signature of science data. The design for the flow of the calib construction pipeline is shown in Figure 6.1.

Note: This diagram represents the intended design rather than the actual current state. The `constructDetectorMap` module is currently named `reduceArc`, and the `constructPsf` module does not exist.

Calibs are created in the following order:

1. Bias
2. Dark
3. Flat
4. Fiber trace
5. Wavelength solution

Once the pipeline evolves further, there will likely be additional calibs to be constructed, such as the PSF parameters.

After each calib is constructed, it needs to be ingested into the calib repository for use. This involves specifying a validity range (in days), which is the time before and after the epoch of the calib that it will be used. The calib repository can live anywhere, but we normally put it inside the data repository.

The construction of bias, dark, flat and fiber trace uses an MPI process pool, so can operate on a cluster over multiple nodes (via Slurm or PBS); however, the current small data volumes mean this is not helpful at the moment.

6.1 Preparation

Before constructing calibs, make the calib repository:

```
mkdir -p /path/to/calibRepo
```

Next, we ingest a model detector map[1] into the calib repository. This will be used to identify the fibers being used, and bootstrap the wavelength solution:

```
ingestCalibs.py /path/to/dataRepo --calib /path/to/calibRepo $OBS_PFS_DIR/pfs/camera/
↪detectorMap-sim-1-r.fits --mode=copy --validity 1000
```

[1] This particular detector map was constructed from the instrument simulator. and the LAM fibers (2,65,191,254,315,337,400,463,589,650) have been updated with wavelength solutions from simulated arc spectra, with an RMS ~ 0.01nm; the other fibers have not been updated, and will not have good wavelength solutions.

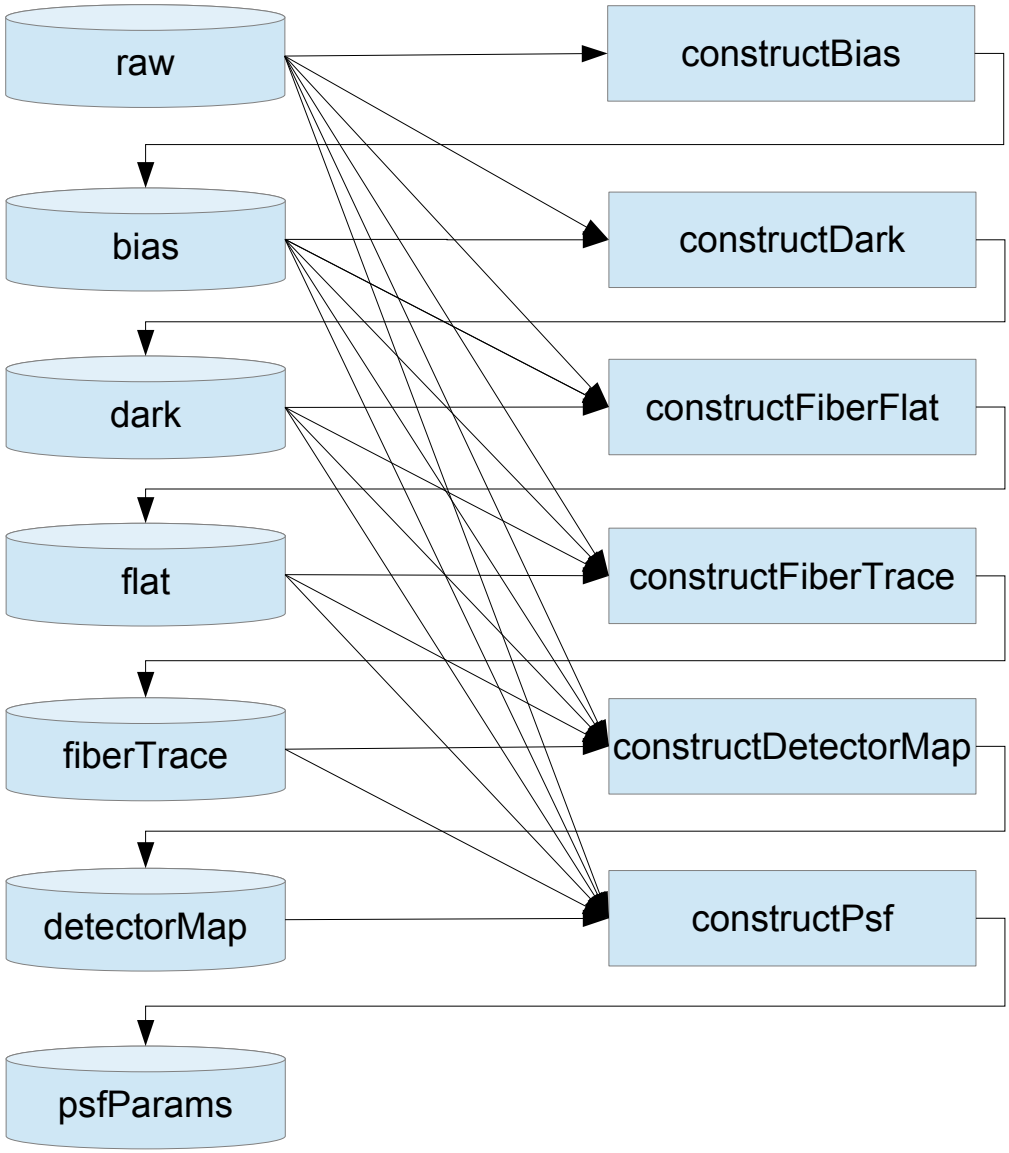

Figure 6.1: **The calib construction components of the pipeline design.** Each component on the left constructs the products on the right, which are used for subsequent components. Note: this is the intended design, not the current state.

6.2 Bias

The "bias" is the response of the detector to an exposure of zero length. The inputs are zero-length exposures, usually designated as `BIAS`. Bias calibs are constructed using `constructBias.py`[2]:

```
constructBias.py /path/to/dataRepo --calib /path/to/calibRepo --rerun calib/bias --id␣
↪field=BIAS
ingestCalibs.py /path/to/dataRepo --calib /path/to/calibRepo /path/to/dataRepo/rerun/
↪calib/BIAS/*.fits --validity 1000
```

Note that here we select the inputs by `field` instead of by `visit` numbers; but any method that selects the correct exposures is sufficient. We use a very long validity range because we assume that the calibrations will be stable[3], and we want to use these calibrations for all data we process.

6.3 Dark

The "dark" is the response of the detector to unit exposure time, with the shutter closed. The inputs are long exposures taken with the shutter closed, usually designated as `DARK`. Dark calibs are constructed using `constructDark.py`[4]:

```
constructDark.py /path/to/dataRepo --calib /path/to/calibRepo --rerun calib/dark --id␣
↪field=DARK
ingestCalibs.py /path/to/dataRepo --calib /path/to/calibRepo /path/to/dataRepo/rerun/
↪calib/DARK/*.fits --validity 1000
```

6.4 Flat

The "flat" is the response of the detector to a uniform light source. For fiber spectroscopy, we don't have a uniform light source with which to illuminate the detector, so instead we dither the slit head in a direction parallel to the slit head (i.e., the spatial dimension), filling in the gaps between the spectra. The inputs are quartz lamp observations with multiple slit positions. Flat calibs are constructed using `constructFiberFlat.py`[5]:

```
constructFiberFlat.py /path/to/dataRepo --calib /path/to/calibRepo --rerun calib/flat␣
↪--id field=QUARTZ
ingestCalibs.py /path/to/dataRepo --calib /path/to/calibRepo /path/to/dataRepo/rerun/
↪calib/FLAT/*.fits --validity 1000
```

6.5 Fiber trace

The "fiber trace" specifies the location and profile of each fiber's trace on the detector;. The input is a quartz lamp observation with the slit at the same position as will be used for science observations[6]. Fiber traces are constructed using `constructFiberTrace.py`[7]:

[2] This is an MPI-based script; use `--cores` instead of `-j` for parallelism.
[3] Of course, this needs to be verified. Also, some shortcomings in the calibs handling in the LSST butler need to be fixed.
[4] This is an MPI-based script; use `--cores` instead of `-j` for parallelism.
[5] This is an MPI-based script; use `--cores` instead of `-j` for parallelism.
[6] It's possible this will have to be done independently for each science observation since the location and profile can have subtle changes with changes in the cobra position. Also, when the slit is fully populated the fiber profiles will overlap, and we will need to use two input exposures: one for the odd fibers and one for the even fibers. At the moment, we aren't dealing with these details, as our current slit head is not fully populated and we aren't feeding them through cobras.
[7] This is an MPI-based script; use `--cores` instead of `-j` for parallelism.

```
constructFiberTrace.py /path/to/dataRepo --calib /path/to/calibRepo --rerun calib/
↪fiberTrace --id field=QUARTZ slitOffset=0.0
ingestCalibs.py /path/to/dataRepo --calib /path/to/calibRepo /path/to/dataRepo/rerun/
↪calib/FIBERTRACE/*.fits --validity 1000
```

6.6 Wavelength solution

The "detector map" provides a mapping from fiber and wavelength to position on the detector, essentially a wavelength solution[8]. The input is one or more arc observations with the slit at the same position as will be used for science observations[9]. Wavelength solutions are constructed using `reduceArc.py`[10]:

```
reduceArc.py /path/to/dataRepo --calib /path/to/calibRepo --rerun calib/arc --id␣
↪field=ARC
sqlite3 /path/to/calibRepo/calibRegistry.sqlite3 'DELETE FROM detectormap; DELETE␣
↪FROM detectormap_visit'
ingestCalibs.py /path/to/dataRepo --calib /path/to/calibRepo/path/to/dataRepo/rerun/
↪calib/arc/DETECTORMAP/*.fits  --validity 1000
```

Note that before ingesting the resultant detector map, we remove the one we used for bootstrapping. The need for this should be removed in the future, but currently it is necessary to prevent the two detector maps clashing.

[8] The `DetectorMap` is more than just a wavelength solution and hopefully this will soon become much more apparent, but currently this is its primary purpose.

[9] Like the fiber trace, it's possible this will have to be done independently for each science observation, since the line centroid might have subtle changes with changes in the cobra position. At the moment, we aren't dealing with these details, since we aren't using cobras.

[10] This is a regular script; use `-j` for parallelism.

CHAPTER

SEVEN

PIPELINE OPERATIONS

The flow of science data through the pipeline is shown in Figure 7.1. Raw science exposures are processed through the `reduceExposure` procedure, which removes the instrumental signatures, models and subtracts the bright sky lines from the 2D image (`subtractSky2d`), and extracts the spectra, producing `pfsArm` files. The `mergeArms` procedure takes all the `pfsArm` files from a single exposure, subtracts any residual sky emission from the 1D spectra, and merges the spectra from the individual arms, producing `pfsMerged` files. These `pfsMerged` files are fed to the `calculateReferenceFlux` procedure, which fits physical flux models to the flux standards, and writes these as `pfsReference` files. The `fluxCalibrate` procedure uses the `pfsMerged` and `pfsReference` files to calculate and apply the flux calibration, writing the flux calibrated single-exposure spectra as `pfsObject` files. Finally, the `coaddSpectra` procedure reads the `pfsArm` files from multiple exposures, applies the various calibrations measured throughout the pipeline, coadds the multiple observations of individual targets, and writes the coadded spectra as `pfsCoadd` files.

The pipeline has been implemented with placeholder algorithms that are sub-optimal and that we do not expect will be used on real science data. Use of these placeholder algorithms has allowed the development of the full end-to-end pipeline before the details of the final algorithms are known. Moving forward, our principle goal is to improve the quality of these algorithms in preparation for processing real science data.

7.1 detrend

In addition to the pipeline described above, we also provide a script that simply removes the instrumental signature from images; this is intended to support instrumental development. Here is an example usage:

```
detrend.py /path/to/dataRepo --calib /path/to/calibRepo --rerun detrending --id␣
↪visit=33
```

The output is a `calexp` file: an image with the instrumental signatures removed (or "detrended").

7.2 reduceExposure

`reduceExposure.py` reads raw images, removes the instrumental signatures (using the outputs of the calib pipeline), subtracts the bright sky lines from the image, and extracts the spectra, writing `pfsArm` files:

```
reduceExposure.py /path/to/dataRepo --calib /path/to/calibRepo --rerun pipeline --id␣
↪field=OBJECT
```

The `pfsArm` files will likely not be of great interest for most science users. The exception is when looking for potential contamination of one spectrum by its neighbours, as the `pfsArm` preserves the sampling from the image (i.e., no interpolation has been applied).

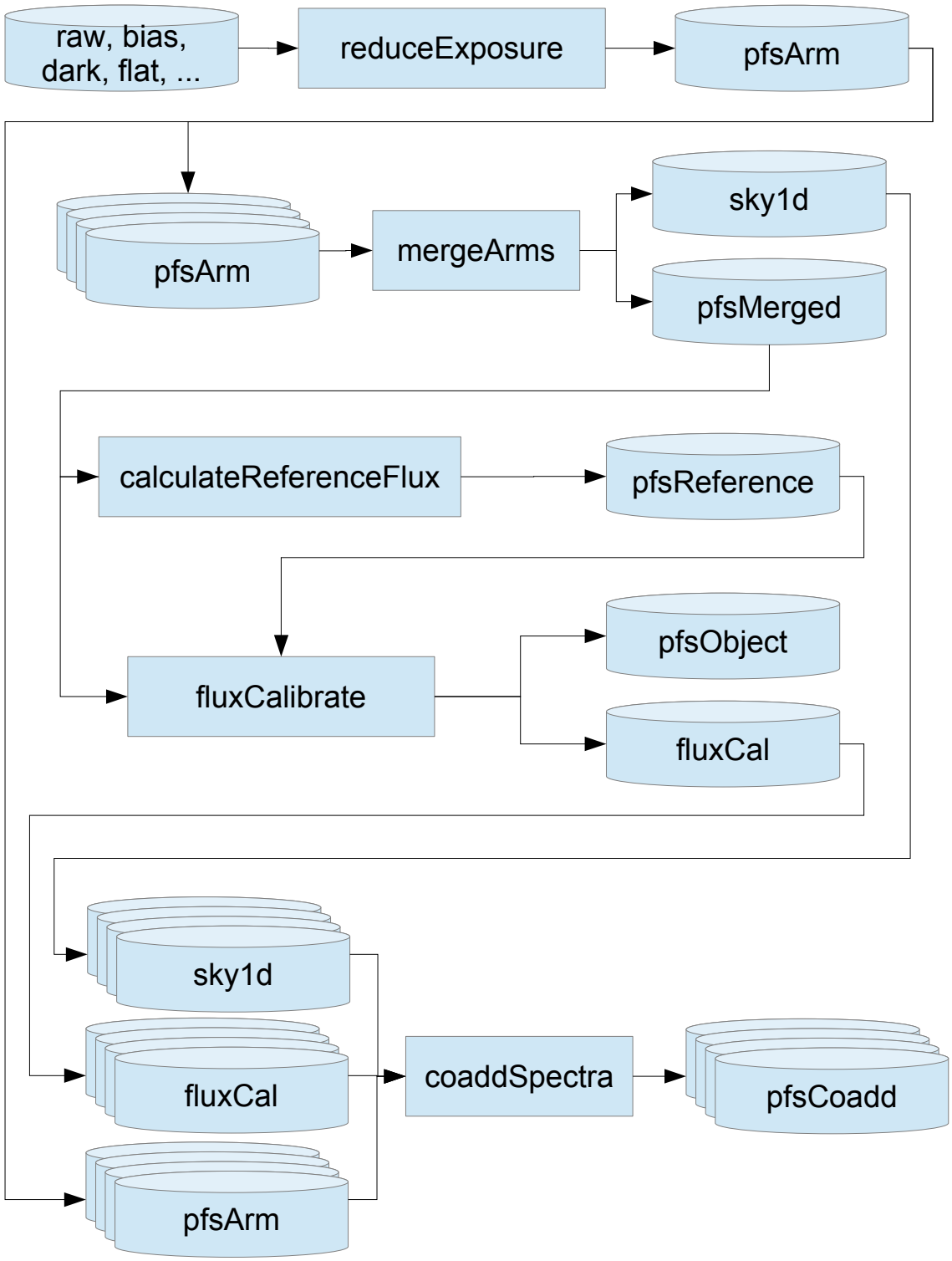

Figure 7.1: **The components of the pipeline for processing science observations.**

7.3 mergeArms

`mergeArms.py` reads the `pfsArm` files, subtracts any residual sky from the 1D spectra, and merges the spectra from the multiple arms[1], writing the `sky1d` and `pfsMerged` files:

mergeArms.py /path/to/dataRepo –calib /path/to/calibRepo –rerun pipeline –id field=OBJECT

The `sky1d` files will be used by `coaddSpectra`. The `pfsMerged` files will be used by `calculateReferenceFlux` and `fluxCalibrate`.

Science users may be interested in the `pfsMerged` files, since they contain the full-wavelength spectra from a single observation; however, note that they are not flux-calibrated.

7.4 calculateReferenceFlux

`calculateReferenceFlux.py` reads the `pfsMerged` files, and fits model spectra with physical fluxes to the flux calibration targets, writing `pfsReference` files:

```
calculateReferenceFlux.py /path/to/dataRepo --calib /path/to/calibRepo --rerun␣
↪pipeline --id field=OBJECT
```

The `pfsReference` files will be used by `fluxCalibrate`; we don't anticipate science users having much interest in them.

7.5 fluxCalibrate

`fluxCalibrate.py` reads the `pfsMerged` and `pfsReference` files, fits a flux calibration model, and writes `fluxCal` and `pfsObject` files:

```
fluxCalibrate.py /path/to/dataRepo --calib /path/to/calibRepo --rerun pipeline --id␣
↪field=OBJECT
```

The `fluxCal` files will be used by `coaddSpectra`. The `pfsObject` files are not used by the pipeline, but may be of interest to science users: since they contain the full-wavelength, flux-calibrated spectra from single observations, they will be of use for those interested in spectra variability.

7.6 coaddSpectra

`coaddSpectra.py` reads the `pfsArm`, `sky1d` and `fluxCal` files[2], coadds all spectra of repeat observations, and writes `pfsCoadd` files:

```
coaddSpectra.py /path/to/dataRepo --calib /path/to/calibRepo --rerun pipeline --id␣
↪field=OBJECT
```

The `pfsCoadd` files are the principal science product of the 2D pipeline, since they are the full-wavelength, flux-calibrated, coadded spectra from multiple observations.

[1] We don't currently have any data with multiple arms, but this is part of the pipeline because ultimately we will have data with multiple arms, and also because this module contains the 1D sky subtraction.

[2] The current implementation doesn't read or use the `sky1d` or `fluxCal` files. Whoops, sorry.

CHAPTER

EIGHT

BUILDING BLOCKS

The pipeline is constructed from a modest number of python classes[1] as the building blocks of the algorithmic modules. The classes are defined in both the `datamodel` and `drp_stella` packages; those in `datamodel` are of general purpose, intended for consumption by developers and science users, while those in `drp_stella` are more directed towards extracting the spectrum from the image. Here, we give an introduction to these classes. We regret that we do not currently have a full reference document with the various APIs, but we hope this will be useful for those seeking to use the outputs of the pipeline, or do development on the pipeline itself.

8.1 `pfs.datamodel.PfsSpectra`

`PfsSpectra` is a collection of spectra from a common source (e.g., an arm, or an entire exposure). This is the base class of the `pfsArm` and `pfsMerged` files. Useful attributes include:

- `wavelength`: wavelength array (nm), of dimension `NxM` for `N` spectra of length `M`.
- `flux`: flux array (counts or nJy), of dimension `NxM`.
- `mask`: mask array, of dimension `NxM`.
- `sky`: sky array, of dimension `NxM` (not currently used).
- `covar`: covariance array (central 3 diagonals of the full covariance matrix), of dimension `Nx3xM` (not currently set properly).
- `flags`: a mask interpreter.

Useful methods include:

- `plot`: plot the spectra using matplotlib.
- `resample`: resample the spectra to a common wavelength vector.
- `extractFiber`: pull out the spectrum from a single fiber into a `PfsSpectrum`.

8.2 `pfs.datamodel.PfsSimpleSpectrum`

`PfsSimpleSpectrum` represents the spectrum for a single object. This is intended to hold model spectra, and is the base class of the `pfsReference` files. Useful attributes include:

- `wavelength`: wavelength array (nm), of dimension `M`.
- `flux`: flux array (counts or nJy), of dimension `M`.
- `mask`: mask array, of dimension `M`.

[1] Some are implemented in C++ for efficiency and/or historical reasons, but have been wrapped into python.

- `flags`: a mask interpreter.

Useful methods include:

- `plot`: plot the spectrum using matplotlib.

8.3 pfs.datamodel.PfsSpectrum

`PfsSpectrum` is similar to *pfs.datamodel.PfsSimpleSpectrum*. This is the spectrum for a single object, but it is suitable for spectra from observations. This is the base class of the `pfsObject` and `pfsCoadd` files. Useful attributes include:

- `wavelength`: wavelength array (nm), of dimension `M`.
- `flux`: flux array (counts or nJy), of dimension `M`.
- `mask`: mask array, of dimension `M`.
- `sky`: sky array, of dimension `NxM`.
- `covar`: covariance array (central 3 diagonals of the full covariance matrix), of dimension `3xM` (not currently set properly).
- `covar2`: a low-resolution non-sparse covariance estimate (not currently set properly).
- `flags`: a mask interpreter.

Useful methods include:

- `plot`: plot the spectrum using matplotlib.

8.4 pfs.datamodel.PfsConfig

`PfsConfig` is the configuration of the top-end, including the mapping of fibers to objects. Useful attributes include:

- `fiberId`: array of fiber identifier.
- `ra`: Right Ascension (degrees) for each fiber.
- `dec`: Declination (degrees) for each fiber.
- `targetType`: target type (an integer, with values from `pfs.datamodel.TargetType`) for each fiber.
- `fiberMag`: magnitudes (an array, order matching that of the `filterNames`) for each fiber.
- `filterNames`: filter names (a list, order matching that of the `fiberMag`) for each fiber.
- `pfiNominal`: nominal position (x,y) for each fiber.
- `pfiCenter`: measured position (x,y) for each fiber.

Useful methods include:

- `selectByTargetType`: return indices for fibers matching a particular target type.
- `selectFiber`: return index for a particular fiber identifier.
- `getIdentity`: return a `dict` identifying a particular fiber identifier.
- `extractNominal`: extract the nominal positions for particular fibers.
- `extractCenter`: extract the center positions for particular fibers.

8.5 pfs.datamodel.TargetType

`TargetType` is an enumeration of target types. The mapping from the symbolic names to integers is an implementation detail, so code should always use the symbolic names rather than integers. The names are:

- `SCIENCE`: science target.
- `SKY`: empty sky.
- `FLUXSTD`: flux standard.
- `BROKEN`: fiber is broken.
- `BLOCKED`: fiber is blocked (hidden behind spot).

8.6 pfs.datamodel.MaskHelper

`MaskHelper` interprets the mask integers. The mapping from the symbolic names to mask integers is an implementation detail, so code should always use the symbolic names rather than integers. Use methods include:

- `get`: return the integer value given a list of symbolic names.

8.7 pfs.drp.stella.FiberTrace

`FiberTrace` tracks the position and profile of the fiber trace on the image. These are usually collected into a `FiberTraceSet`. Useful attributes include:

- `trace`: an image of the trace.
- `fiberId`: the fiber identifier.

Useful methods include:

- `extractSpectrum`: extract a spectrum from the image.
- `constructImage`: construct an image given a spectrum.

8.8 pfs.drp.stella.DetectorMap

`DetectorMap` provides a mapping between `(x,y)` position on the detector and `(fiberId,wavelength)`. Useful methods include:

- `findFiberId`: find the fiber at a position.
- `findPoint`: find the point on the detector for a fiber and wavelength.
- `findWavelength`: find the wavelength for a fiber and a row on the detector.
- `getWavelength`: retrieve the wavelength calibration for a fiber or all fibers.
- `getXCenter`: retrieve the column position for a fiber or all fibers.

CHAPTER

NINE

GETTING HELP

If you encounter problems installing the LSST stack, please first search the LSST pipelines installation documentation, and the LSST Community site, and if your question is not answered there, please ask a question on the LSST Community site.

If you encounter problems installing or using the PFS 2D DRP software, please ask a question in the `#drp-2d` channel on Slack, or e-mail your question to `pfs_software@astro.princeton.edu`.

Because our project is distributed around the world, it is important for efficient communication and a speedy resolution of your problem that you include sufficient details that will allow us to diagnose and/or reproduce the problem, including:

- Your goal:
  - What you are trying to achieve.
  - How you are attempting to achieve that.
  - Why you chose that method.
- Your environment:
  - Operating system.
  - Shell.
  - Working directory.
  - Previous commands that modify the environment.
  - Output of running `eups list -s`
- What command you ran:
  - Interpolate any environment variables.
  - Make any input files accessible to us.
- The output of the command you ran:
  - Terminal output.
  - Log files.
- Your understanding of the problem:
  - What you expected to happen.
  - What actually happened, and why this doesn't match what you expected.
  - What you have tried to diagnose or work around the problem.
  - The impact of the problem and the urgency of finding a solution.