PFS 2D Pipeline Design Documentation

The PFS 2D Pipeline Team

Feb 15, 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 outlines a design for the 2D DRP, including the data flow, the support classes, the functional modules, and the datasets that will be produced.

The 2D DRP will be built following the same philosophy as the LSST stack and the Hyper Suprime-Cam pipeline. The pipeline will be written in Python, for ease of development, maintenance and reading, while classes and functions requiring the performance of a compiled language will be implemented in C++ and wrapped using pybind11. As far as is practical given the smaller size of our development team, we will follow the coding styles and policies in the LSST Developer Guide.

CHAPTER

TWO

PIPELINE DATA FLOW

Here we outline a general overview of the pipeline, tracing the flow of data through the pipeline from raw data to the delivered product. We will introduce the major components, but a detailed explanation of the subcomponents is deferred to the section on *Functional Modules*.

2.1 Data Repository

The LSST stack includes an I/O abstraction layer known as the "data butler", or just "the butler". The butler maps keyword-value pairs into file paths within the "data repository" using pre-defined templates. Input data are read from, and output products are written to, a directory within the data repository. Within a data repository, outputs are typically written within a "rerun" directory specified by a symbolic name (usually with the `--rerun` command-line argument). This serves to group products from a processing run together.

The butler allows data to be specified using keyword-value pairs (usually with the `--id key1=value1 key2=value2` command-line argument[1], which allows for flexibility and avoids the need for the user to keep track of filenames. For example, a set of bias exposures from a particular date can be specified using the individual exposure numbers, if known (`--id exp=123..132`), or by the type of exposure and the date (`--id object=BIAS dateObs=2018-11-05`).

2.2 Ingesting raw data

In order to access raw data, it must first be ingested into the data repository. The LSST stack provides a script to perform this, which we will use: `ingestImages.py`[2].

Example usage:

```
ingestImages.py /path/to/repo /path/to/raw/data/*.fits
```

2.3 Calib Construction

"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 flow of the calib construction is shown in Figure 2.1.

[1] On the command-line, values can be specified like `this^that` meaning both `this` *and* `that`; `123..234` meaning all values from `123` to `234`; and `123..234:7` meaning all values from `123` to `234` counting by `7`.

[2] As of November 2018, this script ingests only raw images. We will need to modify it to also ingest the `pfiConfig` files.

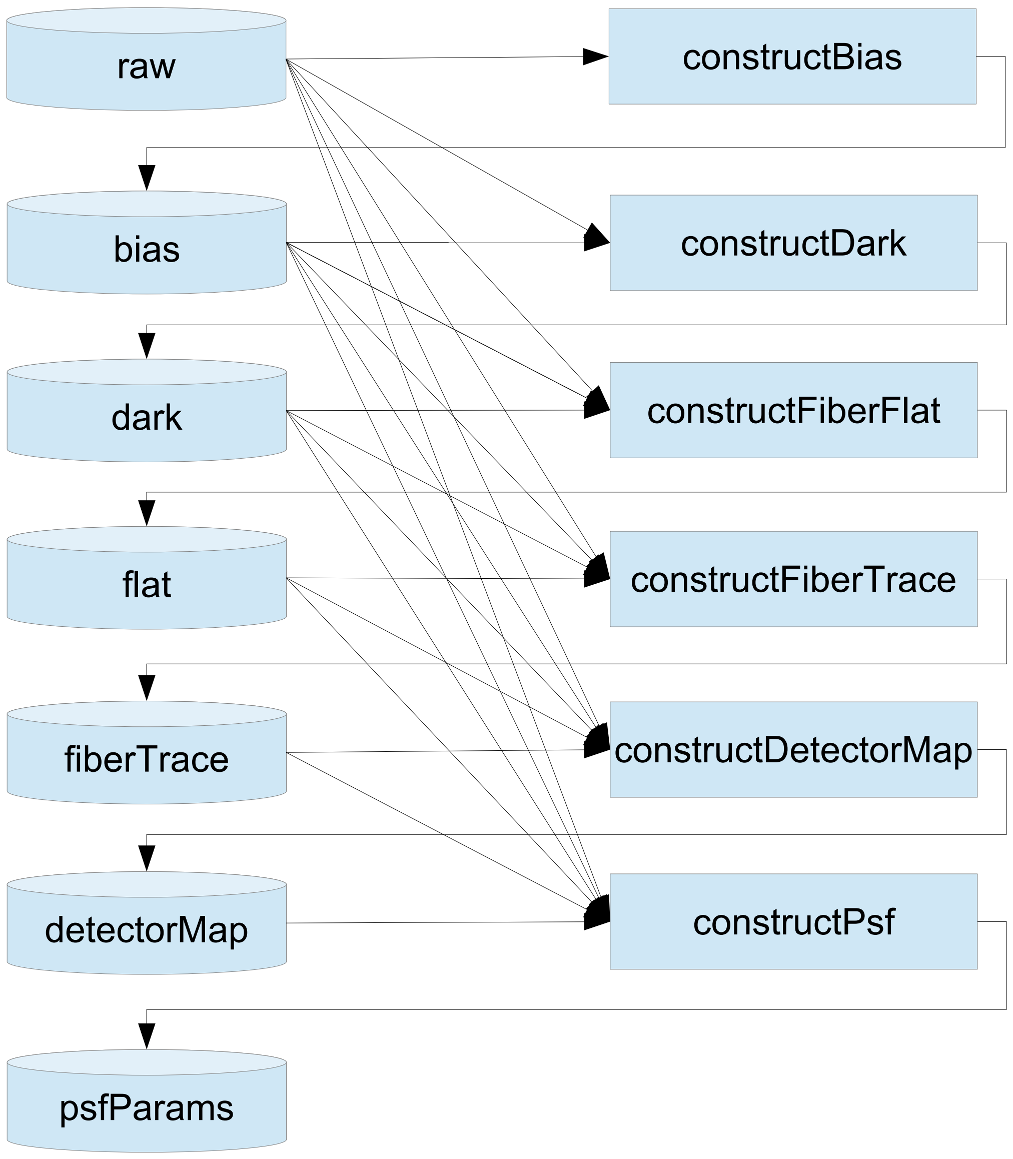

Figure 2.1: **The calib construction components of the pipeline.** Each component on the left constructs the products on the right, which are used for subsequent components.

The LSST stack includes a facility for constructing calibs and ingesting them into a calibration repository for later retrieval[3]. We will use the LSST scripts for constructing biases and darks, as these are constructed in the same way for spectroscopy as for imaging; and we will also use the LSST script, `ingestCalibs.py`, for ingesting the calibs into the calibration repository.

Example construction of a bias calib:

```
constructBias.py /path/to/repo --calib /path/to/calibs --rerun calib/bias --id␣
↪object=BIAS dateObs=2018-11-06 <operational arguments>
ingestCalibs.py /path/to/repo/ --calib /path/to/calibs /path/to/repo/rerun/calib/bias/
↪.../*.fits --validity 30
```

Example construction of a dark calib:

```
constructDark.py /path/to/repo --calib /path/to/calibs --rerun calib/dark --id␣
↪object=DARK dateObs=2018-11-06 <operational arguments>
ingestCalibs.py /path/to/repo --calib /path/to/calibs /path/to/repo/rerun/calib/dark/.
↪../*.fits --validity 30
```

At this point, we need to use our own calib construction modules, as our spectroscopic flat fields are observed and combined differently than for imaging. The inputs for our flat field construction script are quartz lamp observations with the slit position dithered in the dimension of the slit (i.e., `x` offsets). A new script, `constructFiberFlat.py` (see *constructFiberFlat*), will combine these:

```
constructFiberFlat.py /path/to/repo --calib /path/to/calibs --rerun calib/flat --id␣
↪object=QUARTZ dateObs=2018-11-06 <operational arguments>
ingestCalibs.py /path/to/repo --calib /path/to/calibs /path/to/repo/rerun/calib/flat/.
↪../*.fits --validity 1000
```

Next, we need to map the precise location and profile of each fiber's trace on the detector (see *constructFiberTrace*). It's possible[4] that 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[5]. Furthermore, 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. Because the fiber traces are obtained by all fibers observing the same quartz lamp, this also provides an opportunity to provide a relative flux calibration across all the fibers. Here is an example:

```
constructFiberTrace.py /path/to/repo --calib /path/to/calibs --rerun calib/fiberTrace␣
↪--id exp=123^124 <operational arguments>
ingestCalibs.py /path/to/repo --calib /path/to/calibs /path/to/repo/rerun/calib/
↪fiberTrace/.../*.fits --validity 1
```

We now need to map the wavelength solution over the detectors using arc exposures (see *constructDetectorMap*). Similar to the case for the fiber traces, this may need to be done independently for each science observation, but for now we will assume not. Here is an example:

```
constructDetectorMap.py /path/to/repo --calib /path/to/calibs --rerun calib/
↪detectorMap --id object=ARC dateObs=2018-11-06 <operational arguments>
ingestCalibs.py /path/to/repo --calib /path/to/calibs /path/to/repo/rerun/calib/
↪detectorMap/.../*.fits --validity 1
```

Finally, we need the PSF model parameters (see *constructPsf*). The exact contents of these PSF model parameters is yet to be determined, but it's clear that they are an important input to the pipeline and they will change with changes to the instrument, so it makes sense to treat them as a calib. Here's a possible example:

[3] The current calib system is crude, having grown organically, but it should serve our purposes. We expect it will mature over the next few years as the LSST team devotes more attention to it.

[4] Perhaps even likely.

[5] In this case, we can associate a fiber trace with a particular `pfsCconfigId`.

```
constructPsf.py /path/to/repo --calib /path/to/calibs --rerun calib/psf --id␣
↪object=DONUT dateObs=2018-11-06 <operational arguments>
ingestCalibs.py /path/to/repo --calib /path/to/calibs /path/to/repo/rerun/calib/psf/..
↪./*.fits --validity 1
```

2.4 Science observations

The flow of science data through the pipeline is shown in Figure 2.2.

The first operation when processing science observations is the most involved: the extraction of sky-subtracted spectra. `reduceExposure.py` (see *reduceExposure*) will operate on all arms of the same flavor (e.g., the blue arms from each spectrograph) so that the maximum information is available for modeling the sky. It will first perform the instrument signature removal (ISR), subtracting the bias and dark, and dividing by the flat. Then it will fit a PSF model to the sky lines, model the collection of sky line fluxes over the fibers, and subtract the sky lines from the images. Finally, the spectra will be extracted using the fiber trace and detector map. The product is a collection of sky-subtracted spectra for each spectrograph arm (`pfsArm`). Each will have been wavelength-calibrated (through the detectorMap, and perhaps tweaks using the sky lines) and a relative (across arms) flux calibration (through the fiber trace). Here is an example command-line:

```
reduceExposure.py /path/to/repo --calib /path/to/calib --rerun science --id exp=123␣
↪arm=r <operational parameters>
```

Next, we merge the arms within each spectrograph, so that subsequent operations can be done using all available spectral information for each object. This resamples and combines the spectra of each object from the separate arms. This also provides an opportunity to clean up any residuals from the 2D sky subtractions by fitting the sky residuals over the fibers and subtracting from the extracted spectra. The result is a set of spectra covering the entire spectral range, for the entire field-of-view. An example command-line is:

```
mergeArms.py /path/to/repo --calib /path/to/calib --rerun science --id exp=123
↪<operational arguments>
```

We now turn our attention to flux calibration of the extracted, merged spectra. The first thing we need to do for this is generate a set of reference spectra for the calibration (see *calculateReferenceFlux*). An example command-line is:

```
calculateReferenceFlux.py /path/to/repo --calib /path/to/calib --rerun science --id␣
↪exp=123 <operational arguments>
```

Now we can use the reference spectra to measure the flux calibration and apply it to the science targets (see *fluxCalibrate*). The result is wavelength-calibrated, flux-calibrated spectra from the exposure. An example command-line is:

```
fluxCalibrate.py /path/to/repo --calib /path/to/calib --rerun science --id exp=123
↪<operational arguments>
```

The final operation in the pipeline is to coadd spectra of the same object from multiple exposures (see *coaddSpectra*). This resamples and combines the spectra of each object from the separate arms of separate exposures. The result is wavelength-calibrated, flux-calibrated coadded spectra. An example command-line is:

```
coaddSpectra.py /path/to/repo --calib /path/to/calib --rerun science --id exp=123^234^
↪345 <operational arguments>
```

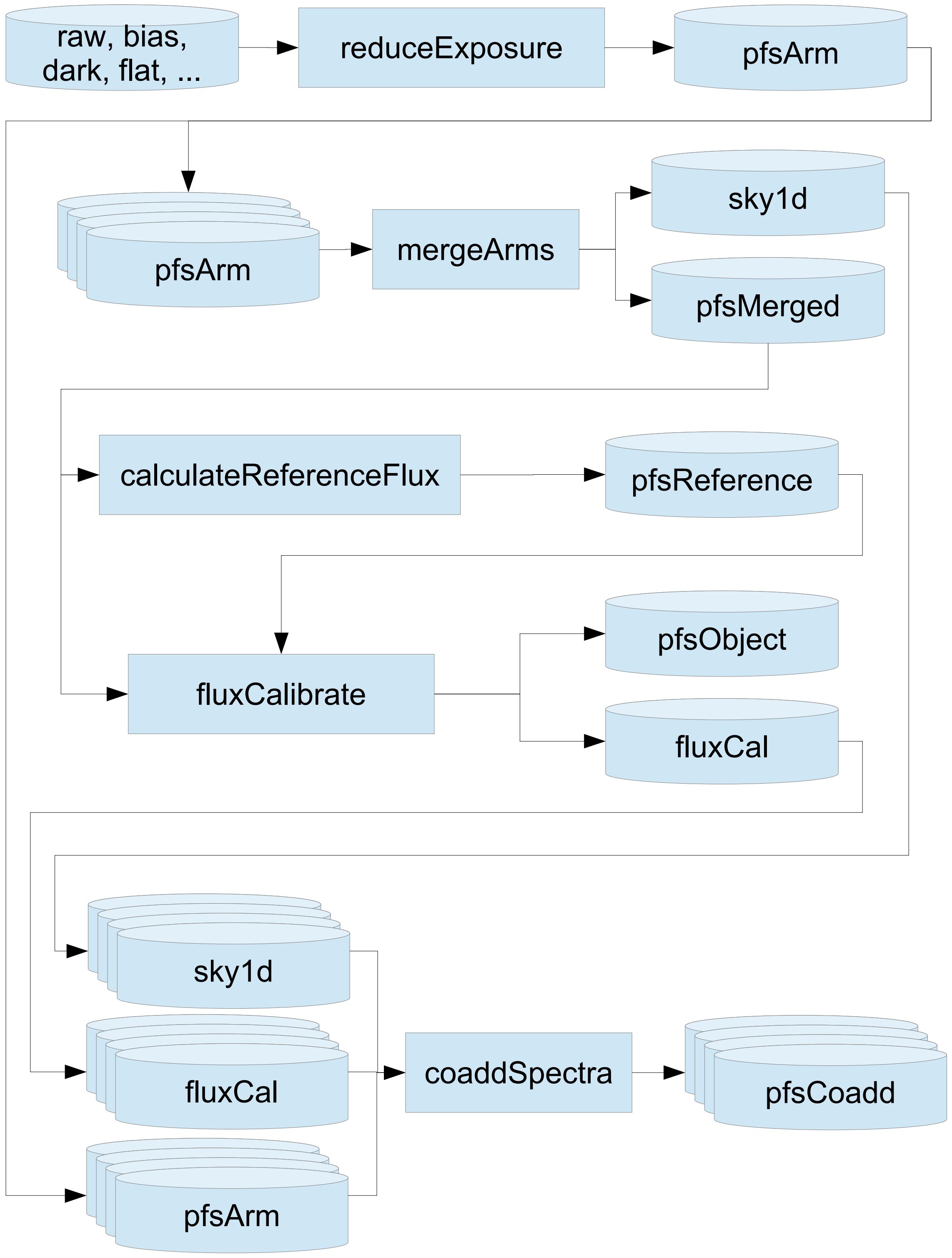

Figure 2.2: **The components of the pipeline for processing science observations.**

CHAPTER

THREE

SUPPORTING CLASSES

Besides the wide variety of primitives we can employ from LSST's Astronomical FrameWork (`afw`), we need some additional classes focussed on support for spectroscopy. We will not here attempt to reproduce the full API, but outline the intended capabilities and uses.

3.1 `DetectorMap`

This class models the fiber positions and wavelengths over the detector. As of November 2018, this class models the fiber positions and wavelengths independently, but the fiber positions and wavelengths are, in principle, a two-dimensional function: a quartz exposure shows lines of constant slit position (i.e., the fibers), while an arc exposure shows (broken) lines of constant wavelength (i.e., the emission lines).

The principal capabilities are:

- Identify the fiber at a point on the detector (`findFiberId`).
- Calculate the wavelength at a point on the detector (`findWavelength`).
- Calculate the point on the detector for a given fiber and wavelength (`findPoint`).
- Provides the wavelength solution for extracted spectra (`getWavelength`).
- Can serve as a foundation for identifying fibers on the image (`getXCenter`).

This class will be implemented in the `drp_stella` package in C++ (for maximum performance and so it can be used in other C++ modules) and wrapped into python.

3.2 `FiberTrace` and `FiberTraceSet`

A `FiberTrace` models the fiber positions and profiles as a function of detector row. A `FiberTraceSet` is a collection of `FiberTraces`.

As of November 2018, this class models the profile as a function of row as an image at native resolution which, because the profiles are undersampled, means they cannot be shifted to a new center; we hope to remedy this shortcoming in the future.

The principal capabilities of `FiberTrace` are:

- Provide an image of the trace (`getTrace`).
- Extract a spectrum from an image (`extractSpectrum`).
- Construct a model image given a spectrum (`constructImage`).

These classes will be implemented in the `drp_stella` package in C++ (for maximum performance) and wrapped into python.

3.3 `Spectrum` and `SpectrumSet`

A `Spectrum` is an measure of flux as a function of wavelength. A `SpectrumSet` is a collection of `Spectrums`.

`Spectrum` contains vectors for the flux, background, wavelength, mask and covariance. This is typically used to carry extracted spectra from observations, but it could also be used for reference spectra in physical units.

A `Spectrum` is persisted as a `PfsObject`, while a `SpectrumSet` is persisted as a `PfsSpectra`.

These classes will be implemented in the `drp_stella` package in C++ (for maximum performance) and wrapped into python.

3.4 `LineSpreadFunction`

A `LineSpreadFunction` is a model of the line profile of an extracted spectrum. It can be measured by extracting the two-dimensional PSF, and is an important ingredient for interpreting the science spectra produced by the pipeline.

The principal capabilities are:

- Calculate the spectrum of an emission line at a given wavelength.

This class will be implemented in the `drp_stella` package in C++ (for maximum performance) and wrapped into python.

3.5 `PfsSpectra`

A `PfsSpectra`[1] is a collection of spectra from a common source, which may be the arm of a spectrograph, or the entire instrument. It shall be the formal I/O representation of such in the PFS Datamodel.

The principal capabilities are:

- Read spectra from FITS (`read`).
- Write spectra to FITS (`write`).
- Plot spectra (`plot`).
- Carry data:
  - Wavelength arrays (`lam`)
  - Flux arrays (`flux`)
  - Mask arrays (`mask`)
  - Sky arrays (`sky`)
  - Covariance arrays (`covar`)

[1] This is called `PfsArm` as of November 2018, but that confuses the class and the use of the class. Renaming the class allows reuse of the class type for other cases where we want to group spectra. The dataset `pfsArm` shall now be of type `PfsSpectra`.

This class will be implemented in the `datamodel` package in python (for ease of use with no compilation or LSST dependencies required). While it carries the same information as the more-capable `SpectrumSet` class, it nevertheless is distinct, as the latter needs to be implemented in C++ for performance reasons. However, it will be used to persist data contained in `SpectrumSet`, and we will provide functions for converting between the two.

3.6 `PfsObject`

A `PfsObject` is a single spectrum of a particular object. It is the formal I/O representation of such in the PFS Datamodel.

The principal capabilities are:

- Read spectrum from FITS (`read`).
- Write spectrum to FITS (`write`).
- Plot spectrum (`plot`).
- Carry data:
  - Wavelength array (`lam`)
  - Flux array (`flux`)
  - Mask array (`mask`)
  - Sky array (`sky`)
  - Covariance array (`covar`)

This class will be implemented in the `datamodel` package in python (for ease of use with no compilation or LSST dependencies required). While it carries the same information as the `Spectrum` class, it nevertheless is distinct, as the latter needs to be implemented in C++ for performance reasons. However, it will be used to persist data contained in `Spectrum`, and we will provide functions for converting between the two.

3.7 `PfsConfig`

A `PfsConfig` carries data about the configuration of the prime-focus instrument, essentially tying the fiber IDs to their targets.

The principal capabilities are:

- Read data from FITS (`read`).
- Write data to FITS (`write`).
- Carry data for each fiber:
  - Object name (`str`)
  - RA (`float`; radians)
  - Dec (`float`; radians)
  - Fiber x, y position (`float`)
  - Nominal fiber x, y position (`float`)
  - Flag (`int`) indicating the fiber's use (e.g., `SCIENCE`, `SKY`, `FLUXSTD`, `BROKEN`, `BLOCKED`).
  - Bandpasses and corresponding magnitudes (`dict` mapping `str` to `float`).

This class will be implemented in the `datamodel` package in python (for ease of use with no compilation or LSST dependencies required).

CHAPTER

FOUR

FUNCTIONAL MODULES

Here we describe the individual modules comprising the pipeline: their inputs, components, algorithms, and outputs.

4.1 Top-level modules

The following top-level modules will be implemented as `lsst.pipe.base.CmdLineTasks` (possibly with MPI for scatter-gather operations). They may be run independently, or by a master pipeline driver script (which could be a `lsst.ctrl.pool.BatchPoolTask` or its successor).

4.1.1 _constructBias

`constructBias` operates on a set of bias exposures of a single spectrograph arm. For each input image, we apply the usual instrument signature removal (ISR) steps up to and not including bias subtraction. Then we average the individual exposures, with a mild clipping to reject deviant pixels. The final product is the combined master bias.

- Input datasets:
  - `raw`: exposures to combine.
- Output datasets:
  - `bias`: master bias; primary product.
  - `postISRCCD`: cached ISR-corrected exposures.

4.1.2 _constructDark

`constructDark` operates on a set of dark exposures of a single spectrograph arm. For each input image, we apply the usual instrument signature removal (ISR) steps up to and not including dark subtraction, and mask cosmic-rays on the basis of their morphology. Then we average the individual exposures, with clipping to reject deviant pixels. The final product is the combined master dark.

- Input datasets:
  - `raw`: exposures to combine.
- Output datasets:
  - `dark`: master dark; primary product.
  - `postISRCCD`: cached ISR-corrected exposures.

4.1.3 constructFiberFlat

`constructFiberFlat` operates on a set of quartz exposures of a single spectrograph arm, where the slit position has been dithered along the slit dimension. It can operate separately on individual spectrograph arms: there is no need to coordinate the operations for different arms or spectrographs, because this is solely a calibration of individual detectors and no flux calibration is taking place[1].

For each input image, we apply the usual instrument signature removal (ISR) steps up to and not including flat-fielding, and mask cosmic-rays on the basis of their morphology. Then we combine the individual images, normalizing the pixels within the row of each fiber by the total flux in that fiber row:

$$F(i, x, y) = \sum_j f_j(i, x, y) / \sum_x f_j(i, x, y)$$

where $F(i, x, y)$ is the combined flat-field with the flux in fiber i as a function of position, (x, y); and $f_j(i, x, y)$ is the j-th input image.

The final product is the combined master flat image.

- Input datasets:
  - `raw`: exposures to combine.
  - `bias`, `dark`: master bias and dark for ISR.
  - `detectorMap`: map of fiber positions, may be used for finding fibers.
- Output datasets:
  - `flat`: master flat; primary product.
  - `postISRCCD`: cached ISR-corrected exposures.

4.1.4 constructFiberTrace

`constructFiberTrace` operates on a set of quartz exposures of a single spectrograph arm, where the slit position is held constant and different fibers are illuminated in each exposure. It can operate separately on individual spectrograph arms: there is no need to coordinate the operations for different arms or spectrographs, because the quartz illumination of the screen (assumed constant) is sufficient to link them already.

For each input image, we apply the usual instrument signature removal (ISR) steps and mask cosmic-rays on the basis of their morphology. Then we find and centroid traces on each image, measure the trace profiles and extract the spectra. Next, the fiber traces are identified using the detectorMap, and the sets of fiber traces from the individual images are merged. Finally, each trace profile is normalized so that the extracted spectrum would match some reference spectrum:

$$t^*(i, \lambda) = t(i, \lambda) * R(\lambda) / S(i, \lambda)$$

where $t^*(i, \lambda)$ is the normalized trace of the i-th fiber for wavelength λ, $t(i, \lambda)$ is the raw trace, $R(\lambda)$ is the reference spectrum, and $S(i, \lambda)$ is the extracted spectrum of the i-th fiber.

The reference spectrum is arbitrary, and can be chosen for convenience or ease of flux calibration[2]. Some possibilities are:

- A smoothed version of the median extracted spectrum.
- A flat spectrum of unit value.

[1] A relative flux calibration occurs in constructFiberTrace. It would be difficult to do a flux calibration here, because the wings of the fibers overlap, so that a single pixel can contain flux from two fibers; therefore merely scaling the pixel values in the flat cannot flux-calibrate a single fiber.

[2] The reference spectrum should be smooth over the wavelength scale corresponding to any wavelength calibration errors.

- A 4500 K black body.

The final product is the normalized fiber trace.

- Input datasets:
  - `raw`: exposures to use.
  - `bias`, `dark`, `flat`: master bias, dark and flat for ISR.
  - `detectorMap`: map of fiber positions, used for finding fibers and rough wavelength calibration.
- Output datasets:
  - `fiberTrace`: trace of fibers; primary product.
  - `postISRCCD`: cached ISR-corrected exposures.

4.1.5 constructDetectorMap

`constructDetectorMap` operates on a set of arc exposures of a single spectrograph arm, where the slit position is held constant while different arc lamps are exposed in turn. It can operate separately on individual spectrograph arms, as each can be calibrated separately.

For each input image, we apply the usual instrument signature removal (ISR) steps and mask cosmic-rays on the basis of their morphology before extracting spectra for each fiber. Arc lines are identified in each fiber's spectrum, and the lines are centroided. For each fiber, the list of arc lines and their centroids for each input image is collected, and the wavelength solution is fit[3]; this solution is used to update the detectorMap.

This updated detectorMap is the final product.

- Input datasets:
  - `raw`: exposures to use.
  - `bias`, `dark`, `flat`: master bias, dark and flat for ISR.
  - `fiberTrace`: for extracting spectra.
  - `bootstrapDetectorMap`: a theoretical or average detectorMap for bootstrapping the specific detectorMap we're constructing.
- Output datasets:
  - `detectorMap`: map of fiber positions and wavelengths; primary product.
  - `postISRCCD`: cached ISR-corrected exposures.

4.1.6 constructPsf

`constructPsf` operates on a set of raw out-of-focus ("donut") arc exposures. It operates separately on individual spectrograph arms, as the camera in each arm is independent.

For each input image, we apply the usual instrument signature removal (ISR) steps and mask cosmic-rays on the basis of their morphology. Then, *with some sort of dark magic that I don't understand,* the donuts are fit with the model PSF.

The final product is the PSF model parameters.

- Input datasets:
  - `raw`: exposures to use.

[3] It's best to fit a function to the residuals of the wavelength solution provided by the detectorMap.

  - `bias`, `dark`, `flat`: master bias, dark and flat for ISR.
  - `detectorMap`: map of fiber position and wavelength, for identifying fibers and arc lines.
- Output datasets:
  - `psfParams`: PSF parameters; primary product.
  - `postISRCCD`: cached ISR-corrected exposures.

4.1.7 reduceExposure

`reduceExposure` operates on a set of raw science exposures for all arms of the same kind over the entire instrument, as it needs to fit models as a function of wavelength over the entire field of view in the two-dimensional sky subtraction.

For each input image, we apply the usual instrument signature removal (ISR) steps and mask cosmic-rays on the basis of their morphology. Then, if it found to be necessary, we can tweak the wavelength solution in the detectorMap by extracting the spectra and fitting the wavelengths of the sky lines.

Next we perform the two-dimensional sky subtraction (see *subtractSky2d* for details). Finally, for each arm the spectra are extracted and written as the final product.

- Input datasets:
  - `raw`: exposures to use.
  - `bias`, `dark`, `flat`: master bias, dark and flat for ISR.
  - `psfParams`: PSF parameters, for *subtractSky2d*.
  - `fiberTrace`: fiber profiles for extraction.
  - `detectorMap`: map of fiber position and wavelength, for wavelength calibration.
  - `pfiConfig`: top-end configuration, for identifying sky fibers.
- Output datasets:
  - `pfsArm`: sky-subtracted, wavelength-calibrated spectra from arm; primary product.
  - `postISRCCD`: ISR-corrected exposure.
  - `psf`: PSF model, from *subtractSky2d*.
  - `sky2d`: 2d sky model, from *subtractSky2d*.
  - `lsf`: line-spread function, from *subtractSky2d*.

4.1.8 mergeArms

`mergeArms` operates on all arms for the entire instrument, as it needs to fit models as a function of wavelength over the entire field of view in the one-dimensional sky subtraction, and it merges the arms within each spectrograph.

For all arms of the same kind, we perform a one-dimensional sky subtraction (see *subtractSky1d* for details). Now that we are done with corrections in the frame of the instrument, we can apply a barycentric wavelength correction. Finally, the spectra from the arms of each spectrograph are merged. The final product is the merged, sky-subtracted, wavelength-calibrated and barycentric-corrected spectra for the entire field of view.

- Input datasets:
  - `pfsArm`: sky-subtracted, wavelength-calibrated spectra from arm.
  - `lsf`: line-spread function.

  - `pfiConfig`: top-end configuration, for identifying sky fibers.
- Output datasets:
  - `pfsMerged`: Merged spectra for all spectrographs+arms; primary product.
  - `sky1d`: 1d sky model, from *subtractSky1d*.
- Algorithmic details:
  - We might do the merge using the Kirkby-Kaiser algorithm.

4.1.9 calculateReferenceFlux

`calculateReferenceFlux` operates on spectra from the entire field-of-view (i.e., the output of *mergeArms*).

For each spectrum that will be used for flux calibration (typically F-stars), we determine the most suitable reference spectrum from a grid of models. This reference spectrum should be scaled to the correct flux using broad-band photometry from the `pfiConfig`. The final product is the flux-corrected reference spectra.

- Input datasets:
  - `pfsMerged`: Merged spectra for all spectrographs+arms.
  - `pfiConfig`: top-end configuration, for identifying calibration fibers.
  - `refModels`: grid of reference models.
- Output datasets:
  - `pfsReference`: reference spectra; primary product.

4.1.10 fluxCalibrate

`fluxCalibrate` operates on spectra from the entire field-of-view (i.e., the output of *mergeArms*).

For each spectrum that will be used for flux calibration (typically F-stars) we measure the flux calibration vector. We model the ensemble of flux calibration vectors over the focal plane, and apply the flux calibration model to the science spectra. Finally, the science spectra can be tweaked to match the broad-band photometry in the `pfiConfig`. The final product is the wavelength-calibrated, flux-calibrated spectra for the entire field of view.

- Input datasets:
  - `pfsMerged`: Merged spectra for all spectrographs+arms.
  - `pfiConfig`: top-end configuration, for identifying calibration fibers.
  - `pfsReference`: reference spectra.
- Output datasets:
  - `pfsObject`: flux-calibrated object spectra; primary product.
  - `fluxCal`: flux calibration parameters.
- Algorithmic details:
  - When modeling the flux calibration over the field of view, we could consider weighting by the distance of the fiber position from the nominal position.

4.1.11 coaddSpectra

`coaddSpectra` operates on a set of spectra from the entire field-of-view.

First, we read the input `pfiConfig` files to determine the list of objects and their inputs, and then we coadd the input spectra of each object. In order to construct a coadd without correlated noise, we need to go back to the original extracted spectra (before merging arms). This requires re-applying the calibrations that were originally calculated from the merged spectra, specifically, the one-dimensional sky subtraction and flux calibration.

- Input datasets:
  - `pfiConfig`: top-end configuration, for identifying calibration fibers.
  - `pfsArm`: sky-subtracted, wavelength-calibrated spectra from arm.
  - `sky1d`: 1d sky model, from *subtractSky1d*.
  - `fluxCal`: flux calibration parameters, from *fluxCalibrate*.
- Output datasets:
  - `pfsCoadd`: coadded spectrum; primary product.
- Algorithmic details:
  - We coadd the original (un-resampled) spectra using the Kirkby-Kaiser algorithm.

4.2 Lower-level modules

The following modules support the top-level modules: they do not need to be stand-alone executables. They will be implemented as `lsst.pipe.base.Tasks` that return `lsst.pipe.base.Structs` with the necessary outputs. Multiple versions of these modules may be developed with increasingly sophisticated algorithms as the pipeline grows in functionality.

4.2.1 subtractSky2d

`subtractSky2d` subtracts sky lines from the two-dimensional images (i.e., before extracting the spectra). This is important because the sky lines from neighboring fibers overlap, especially when the lines are bright.

This module operates on all arms of the same kind for the entire instrument in a single exposure (e.g., all red arms in a single exposure). This is necessary because we will fit models as a function of wavelength over the entire field of view.

This module requires the following inputs:

- `exposureList` (`list` of `lsst.afw.image.Exposure`): a list of exposures for the arms; these shall be modified (the sky shall be subtracted).
- `pfiConfig` (`pfs.datamodel.PfiConfig`): configuration of the top-end, for identifying sky fibers.
- `fiberTraceList` (`list` of `pfs.drp.stella.FiberTraceSet`): a list of fiber traces for the arms (same order as for `exposureList`).
- `detectorMapList` (`list` of `pfs.drp.stella.DetectorMap`): a list of detectorMaps for the arms (same order as for `exposureList`).
- `psfParamsList` (`list` of `pfs.drp.stella.PfsPsfParams`): a list of PSF parameters for the arms (same order as for `exposureList`).
- `skyLineList` (`list` of `pfs.drp.stella.ReferenceLine`): a list of sky lines.

We will first remove the sky continuum so that we can measure the sky lines. In order to do so, we will extract the spectra and fit a continuum to the sky fibers. This continuum can be modelled as a function of RA,Dec (*fitFocalPlane*), and it is then subtracted from all fibers in two dimensions (using the fiber profiles in the `fiberTraceList` to construct images with the sky continuum spectra).

Next, we measure the sky lines. The details of this step have not been worked out yet, but it likely involves fitting a PSF (using the provided `psfParamsList`), fitting the PSF to the sky lines to measure their intensity, modelling the intensity as a function of focal plane position (*fitFocalPlane*), and then generating model images (using the PSF and sky line model) which can be subtracted from the input images.

The outputs of this module shall be:

- `psfList` (`list` of `pfs.drp.stella.PfsPsf`): the fit PSFs (same order as for `exposureList`).
- `continuumModel` (class TBD): the model for the sky continuum.
- `skyLineModel` (class TBD): the model for the sky lines.

4.2.2 subtractSky1d

`subtractSky1d` subtracts the sky from the one-dimensional spectra. This can be used to clean up the residuals after two-dimensional sky subtraction (*subtractSky2d*), or as the primary sky-subtraction technique.

This module requires the following inputs:

- `spectraList` (`list` of `pfs.drp.stella.SpectrumSet`): a list of spectra for the arms; these shall be modified (the sky shall be subtracted).
- `pfiConfig` (`pfs.datamodel.PfiConfig`): configuration of the top-end, for identifying sky fibers.
- `lsfList` (`list` of `pfs.drp.stella.Lsf`): a list of line-spread functions for the arms (same order as for `exposureList`).
- `skyLineList` (`list` of `pfs.drp.stella.ReferenceLine`): a list of sky lines.

This module consists of four parts:

1. Use the sky fibers to generate a model for the sky. Multiple models can be imagined for this:
   - A multiple of the average sky spectrum.
   - A linear combination of principal components.
   - A continuum plus discrete sky lines.
2. Fit the model to the sky fibers.
3. Fit the model parameters as a function of position on the focal plane (*fitFocalPlane*).
4. Subtract the model from all the fibers.

The outputs of this module shall be:

- `skyModel` (class TBD): the model for the sky.

4.2.3 fitFocalPlane

`fitFocalPlane` fits a set of vectors for fibers over the focal plane. These vectors might be a spectrum for each fiber, or the parameters of a model, but each needs to be modelled as a function of position on the focal plane.

This module requires the following inputs:

- `vectorList` (`list` of `numpy.ndarray`): Vectors to model over the focal plane.

- `fiberIdList` (`list` of `int`): List of corresponding fiber IDs (same order as `vectorList`).
- `pfiConfig` (`pfs.datamodel.PfiConfig`): configuration of the top-end, for mapping fiber ID to focal-plane position.
- `evalFiberIdList` (`list` of `int`): List of fiber IDs for which the model should be evaluated; may be `None` to indicate that the model should be evaluated for all fibers in the `pfiConfig`.

In addition to these inputs, a set of configuration parameters will govern how the fit is done:

- `raOrder` (`int`): Polynomial order in RA.
- `decOrder` (`int`): Polynomial order in Dec.
- `rejIter` (`int`): Number of rejection iterations.
- `rejThreshold` (`float`): Rejection threshold (standard deviations).
- `weighting` (`str`): Specifies how weighting is to be done:
  - `uniform`: no weighting.
  - `offset`: weight by distance between actual fiber position and nominal fiber position.

This should be a simple matter of fitting a two-dimensional polynomial, with optional rejection and weighting. Each vector index is fit independently.

The outputs of this module shall be:

- `modelList` (`list` of some polynomial class): Polynomial fit for each vector index (same order as `vectorList`).
- `chi2List` (`list` of `float`): The χ^2 value for each fit (same order as `vectorList`).
- `numList` (`list` of `int`): The number of values used for each fit (same order as `vectorList`).
- `evalList` (`list` of `numpy.ndarray`): The evaluated vectors for each of the fibers in the `evalFiberIdList` (same order as `evalFiberIdList`, or if `evalFiberIdList` is `None` then the same order as in the `pfiConfig`).

4.2.4 extractSpectra

`extractSpectra` extracts spectra from an image, given the fiber traces and detectorMap.

The module requires the following inputs:

- `image` (`lsst.afw.image.MaskedImage`): Image from which to extract spectra.
- `traces` (`pfs.drp.stella.FiberTraceSet`): Fiber traces, specifying the position and profile as a function of row.
- `detectorMap` (`pfs.drp.stella.DetectorMap`): Map of fiber position and wavelength on the detector; used for the wavelength solution.

The extraction could be done with one of a number of algorithms, the choice of which will be set by a configuration parameter:

1. Boxcar extraction: sum the data in pixels around the peak. This is the simplest possible algorithm, but doesn't maximize signal-to-noise; useful for testing.
2. "Optimal extraction": sum the data weighted by the fiber profile. This is a better algorithm for optimising the signal-to-noise, but it doesn't deal with neighboring fibers which may contaminate the fiber being extracted.

3. Simultaneous fit: solve the tri-diagonal matrix from least-squares fitting a linear combination of fiber profiles. This can be done in linear time, so it should be fast enough. This approach deals with neighbors, and is likely the ultimate algorithm we will use for science.
4. Iterative extractions: one can imagine an iterative approach whereby the optimal extraction is performed iteratively. We don't expect to use this algorithm.

These will be coded in C++ (as a method of the `FiberTrace` class) for speed.

The outputs of this module shall be:

- `spectra` (`pfs.drp.stella.SpectrumSet`): The extracted spectra.

CHAPTER

FIVE

DATASETS

Here we describe the various products of the pipeline. Some are intended for internal use, while others are intended for science users.

5.1 Raw products

The following products are provided by the observing system, and ingested into the data repository as the first step in pipeline operations:

- `raw` (`lsst.afw.image.ImageU`): the raw image from the data acquisition system.
- `pfiConfig` (`pfs.datamodel.PfsConfig`): top-end configuration, for mapping fibers to their targets.

5.2 Calib products

- `bias` (`lsst.afw.image.Exposure`): master bias frame.
- `dark` (`lsst.afw.image.Exposure`): master dark frame.
- `flat` (`lsst.afw.image.Exposure`): mask flat frame, constructed from dithered quartz exposures.
- `fiberTrace` (`pfs.drp.stella.FiberTraceSet`): position and profile of each fiber as a function of row.
- `detectorMap` (`pfs.drp.stella.DetectorMap`): mapping between fiber and wavelength to position on the detector.
- `bootstrapDetectorMap` (`pfs.drp.stella.DetectorMap`): a theoretical or average detectorMap for bootstrapping the specific detectorMap we're constructing.
- `psfParams` (type TBD): PSF parameters, determined from donut exposures.

5.3 Reference products

The following products provide bulk data for the operation of pipeline algorithms. They may be provided by the butler, or some other mechanism (e.g., files or directories in `obs_pfs`, a `git-lfs` repo, etc.).

- `refModels` (type TBD): a grid of reference models, used for *calculateReferenceFlux*.
- `arcLines` (type TBD): a list of arc lines: their wavelengths, identifications, strengths and flags indicating whether they should be used or not.

- `skyLines` (type TBD): a list of sky lines: their wavelengths, identifications, strengths and flags indicating whether they are resolved or not.

5.4 Temporary products

The following products are produced for convenience only, and can in general be deleted once their usefulness has been realised:

- `postISRCCD` (`lsst.afw.image.Exposure`): cached ISR-corrected exposure, for calib construction.

5.5 Operational products

The following products are produced by the pipeline in the course of operations, and may be useful for debugging or close inspection of data quality:

- `psf` (type TBD): PSF model, from *subtractSky2d*.
- `sky2d` (type TBD): sky model, from *subtractSky2d*.
- `lsf` (`pfs.drp.stella.LineSpreadFunction`): line-spread function, derived from the PSF model, from *reduceExposure*.
- `sky1d` (type TBD): sky model, from *subtractSky1d*.
- `fluxCal` (type TBD): flux calibration, from *fluxCalibrate*.
- `pfsReference` (`pfs.datamodel.PfsSpectra`): reference spectra, from *calculateReferenceFlux*.
- `pfsMerged` (`pfs.datamodel.PfsSpectra`): arm-merged spectra for the entire instrument, from *mergeArms*.

5.6 Science products

These are the main science products of the pipeline.

- `pfsArm` (`pfs.datamodel.PfsSpectra`): sky-subtracted, wavelength-calibrated spectra from a single arm of a single spectrograph, from *reduceExposure*. Since these spectra have not been resampled after extraction, this may be useful for identifying cosmic-ray hits masquerading as emission lines in the `pfsObject` or `pfsCoadd`.
- `pfsObject` (`pfs.datamodel.PfsObject`): flux-calibrated, barycentric wavelength-calibrated object spectrum from a single exposure, from *fluxCalibrate*. This is useful for investigating variations from exposure to exposure, or identifying cosmic-ray hits masquerading as emission lines in the `pfsCoadd`.
- `pfsCoadd` (`pfs.datamodel.PfsObject`): coadded spectrum from multiple exposures, from *coaddSpectra*. This is the main science product that most science users will want.

CHAPTER

SIX

CONCERNS AND/OR FUTURE DEVELOPMENT DIRECTIONS

Here we outline some concerns with the design as presented that might be remedied in a future version of the pipeline.

6.1 `DetectorMap` and `FiberTrace`

There is a degree of overlap between `DetectorMap` and `FiberTrace`: both track the center of the fibers as a function of row. This duplication is unnecessary, but is present in the current design for historical reasons. The functionality of `FiberTrace` could be subsumed into `DetectorMap` in a future design, at which point the `constructFiberTrace.py` and `constructDetectorMap.py` scripts will also be merged.

6.2 Bootstrapping the detectorMap

We need to know the detectorMap before we construct it with `constructDetectorMap.py`, because it is an input to both `constructFiberTrace.py` and `constructDetectorMap.py`. This can be done by using an additional calib product specifically for this purpose (we use `bootstrapDetectorMap` in `constructDetectorMap.py`).

6.3 Association of `DetectorMap` and `FiberTrace` with science exposures

If there is a one-to-one relationship between science exposures and calibration exposures[1], then it may not be convenient to treat the detectorMap and fiber trace used for science reductions as calibs. In that case, the quartz and arc exposures could be inputs to *reduceExposure*[2], which would first construct the fiber trace and detectorMap before operating on the science exposure.

6.4 NIR detectors

The NIR detectors produce multiple images as they read "up the ramp". While they should fit into the general flow of the pipeline design as outlined, the details are not yet clear, e.g., how they will be read by the butler, and what changes to ISR are necessary to support them. The specification of details is deferred until we get actual NIR data.

[1] This might occur if we "replay" the fiber positions used for the science exposures at the end of the night in order to take corresponding quartz (for the fiber trace) and arcs (for the detectorMap).

[2] The appropriate quartz and arc could be identified through having the same `pfsConfigId` as the raw exposure.