PFS 2D Pipeline Modules Documentation

The PFS 2D Pipeline Team

Jan 22, 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 the major algorithmic modules for the 2D DRP.

The data flow of the science pipeline is presented in the 2D DRP Design document, but here we give a summary for background. The flow of science data through the pipeline is shown in Figure 1.1. 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 general flow of the pipeline has been implemented with simple placeholder implementations for the algorithmic modules that perform the necessary functions in a sub-optimal manner. It is now our goal to improve the algorithmic quality of these modules. These will be developed under the drp_stella package, and written in Python[1]. 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.

These algorithmic modules will be implemented as subclasses of `lsst.pipe.base.Task`, as the configuration framework we use allows these to be substituted at runtime, so long as the call signature matches. It is the purpose of this document to specify these call signatures so that multiple algorithmic modules of each kind can be developed and employed as required. The call signatures specified in this documents are based on the current pipeline implementation which, because it currently uses only placeholder algorithms, may not be sufficient to implement more complicated algorithms (e.g., a data butler is required in order to load important data, but none is present in the current call signature). We therefore welcome proposals to extend or correct the call signatures in this document, but these proposals should be discussed by the DRP team and this document updated before the proposal is implemented.

Several modules are expected to return a "persistable object", which means an object of a custom class which can be persisted with the LSST data butler. We will use the `FitsCatalogStorage` storage type[2], which requires the following methods:

[1] Classes and functions requiring the performance of a compiled language can be implemented in C++ and wrapped using pybind11.

[2] There are other storage types that can be used, but `FitsCatalogStorage` has a clear API, and FITS allows portability. Despite the name, `FitsCatalogStorage` does not mean that a FITS table must be used (an image can be used as the format if desired); instead, it refers to the API used to read and write the file.

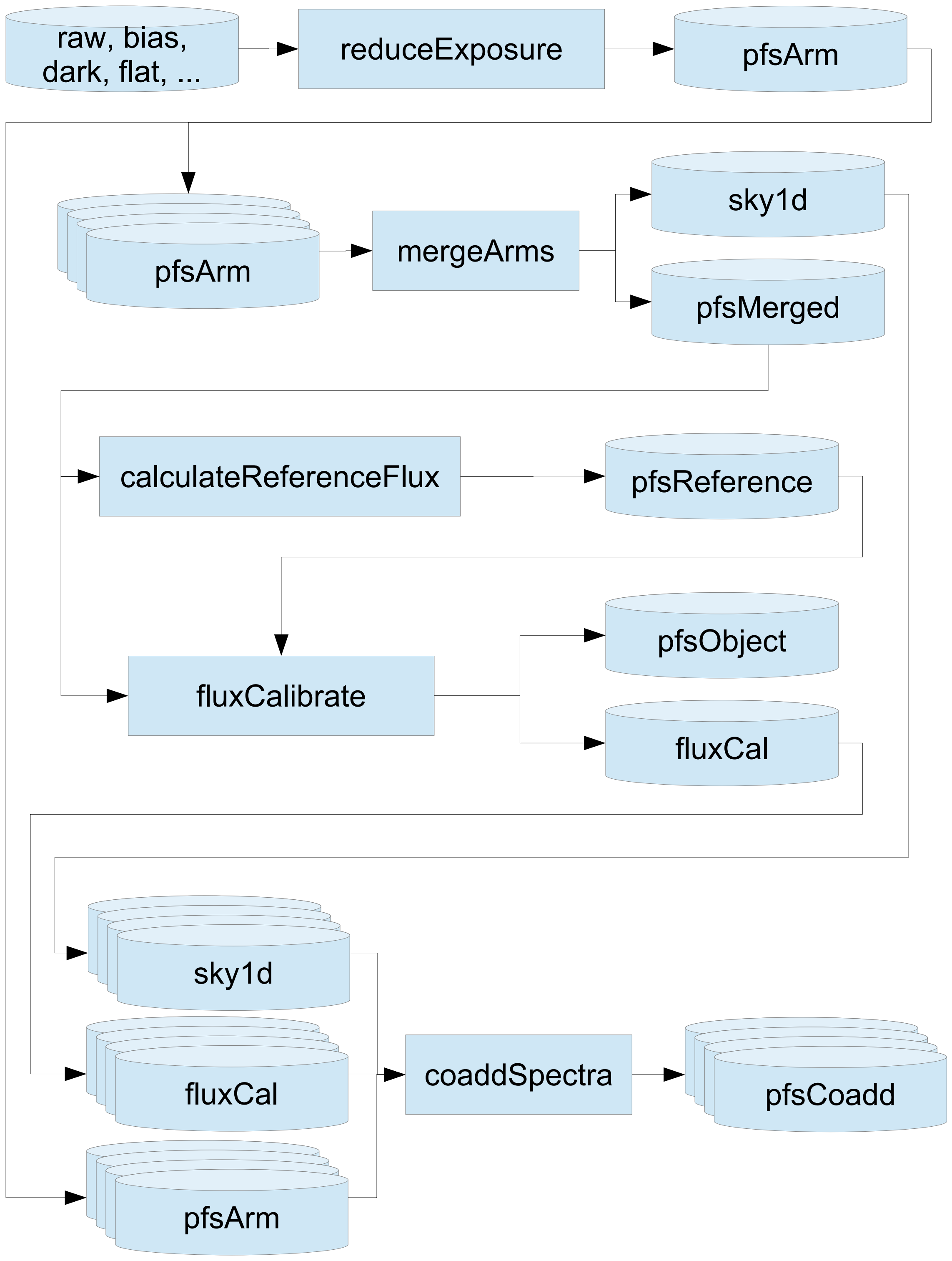

Figure 1.1: **The components of the pipeline for processing science observations.**

```
class SomePersistable:
    @classmethod
    def readFits(cls, filename):
        """Read from FITS file

        Parameters
        ----------
        filename : `str`
            Filename to read.

        Returns
        -------
        self : ``cls``
            Object read from FITS file.
        """
        # Use astropy.io.fits to open and read FITS file, then construct object
        return cls(stuff)

    def writeFits(self, filename):
        """Write to FITS file

        Parameters
        ----------
        filename : `str`
            Name of file to which to write.
        """
        # Use astropy.io.fits to write FITS file.
```

The following classes are useful building blocks of the algorithmic modules:

- `pfs.datamodel.PfsSpectra`: 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`.
- `pfs.datamodel.PfsSimpleSpectrum`: 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`.
  - `flags`: a mask interpreter.

Useful methods include:

  - `plot`: plot the spectrum using matplotlib.

- `pfs.datamodel.PfsSpectrum`: similar to `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.

- `pfs.datamodel.PfsConfig`: 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.

- `pfs.datamodel.TargetType`: 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).

- `pfs.datamodel.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.
- `pfs.drp.stella.FiberTrace`: 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.
- `pfs.drp.stella.DetectorMap`: 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

TWO

TWO-DIMENSIONAL SKY SUBTRACTION

Two-dimensional sky subtraction is part of `reduceExposure`. Because fibers can be separated by as few as 5 pixels, the flux in the wings of the PSF from bright sky lines in one fiber can leak into neighbouring fibers, contaminating the extracted spectra. The goal of this module is to model and subtract the bright sky lines from the image before extraction of the spectra.

This module will be provided multiple images, comprising all arms of the same kind within an exposure (e.g., the red arms from each of the spectrographs); this allows modelling of the spectrum over the entire focal plane. This module will be provided a 2D PSF model[1] for each exposure, although the class representing this is yet to be defined; it will likely look like the LSST `Psf` class. It will also be provided a `FiberTraceSet` and `DetectorMap` for each exposure so that it knows the location of sky lines. Finally, a `PfsConfig` allows identification of the different targets, including sky fibers and blocked or broken fibers. The module should subtract the sky from the input exposures, and return a persistable object that represents the sky that has been subtracted.

Here is an example definition:

```
class SubtractSky2DTask(lsst.pipe.base.Task):
    def run(self, exposureList, pfsConfig, psfList, fiberTraceList, detectorMapList):
        """Measure and subtract sky from 2D spectra image

        Parameters
        ----------
        exposureList : iterable of `lsst.afw.image.Exposure`
            Images from which to subtract sky.
        pfsConfig : `pfs.datamodel.PfsConfig`
            Top-end configuration, for identifying sky fibers.
        psfList : iterable of PSFs (type TBD)
            Point-spread functions.
        fiberTraceList : iterable of `pfs.drp.stella.FiberTraceSet`
            Fiber traces.
        detectorMapList : iterable of `pfs.drp.stella.DetectorMap`
            Mapping of fiber,wavelength to x,y.

        Returns
        -------
        sky2d : `SomePersistable`
            2D sky subtraction solution.
        """
        sky2d = self.measureSky(...)
        self.subtractSky(...)
        return sky2d
```

[1] Neven Caplar is working on measuring the 2D PSF using out-of-focus images, but we could imagine an alternative implementation using a classic image PSF code, such as PSFEx or *PIFF*.

CHAPTER

THREE

ONE-DIMENSIONAL SKY SUBTRACTION

One-dimensional sky subtraction is part of `mergeArms`. The goal of this module is to model and subtract the sky from the spectra. We will use it either to clean up correlated spectral residuals left by the two-dimensional sky subtraction or to perform sky subtraction entirely in the spectral domain (e.g., if the two-dimensional sky subtraction is too CPU-intensive to use for quick-look reductions).

This module will be provided the spectra from all arms of all spectrographs for the exposure; this allows modelling of the spectrum over the entire focal plane. This module will also be provided a line-spread function (LSF) model for each exposure, although the class representing this is yet to be defined; it will likely look like a one-dimensional version of the LSST `Psf` class. Finally, a `PfsConfig` allows identification of the different targets, including sky fibers and blocked or broken fibers. The module should subtract the sky from the input spectra, and return a persistable object that represents the sky that has been subtracted.

Here is an example definition:

```
class SubtractSky1DTask(lsst.pipe.base.Task):
    def run(self, spectraList, pfsConfig, lsfList):
        """Measure and subtract the sky from the 1D spectra

        Parameters
        ----------
        spectraList : iterable of `pfs.datamodel.PfsSpectra`
            List of spectra from which to subtract the sky.
        pfsConfig : `pfs.datamodel.PfsConfig`
            Configuration of the top-end, for identifying sky fibers.
        lsfList : iterable of LSF (type TBD)
            List of line-spread functions.

        Returns
        -------
        sky1d : `SomePersistable`
            1D sky model.
        """
        sky1d = self.measureSky(...)
        self.subtractSky(...)
        return sky1d
```

CHAPTER

FOUR

FOCAL PLANE FUNCTION FITTING

Fitting a vector function over the focal plane is a common operation, which can be used in the two-dimensional and one-dimensional sky subtractions, PSF measurement, and flux calibration modules.

The goal of this module is to fit a vector function over the focal plane. This vector function might be the strength of sky lines, the response of the instrument, or some other measurements made from the spectrum for a subset of fibers.

This module will be provided with `M` vectors of length `N`, along with corresponding errors and boolean masks, and a list of identifiers for the appropriate fibers. A `PfsConfig` will also be provided, from which can be obtained the position of the target on the sky or on the focal plane, and the intended position on the focal plane[1]. The module should return a persistable object that represents the fit.

In addition to the usual `run` method, the module's `Task` should also implement an `apply` method that takes the result from the `run` method and applies it to a list of fibers. Like the `run` method, this also requires a `PfsConfig` to get the required information on fiber positions.

Here is an example definition:

```python
class FitFocalPlaneTask(lsst.pipe.base.Task):
    def run(self, vectors, errors, masks, fiberIdList, pfsConfig):
        """Fit a vector function over the focal plane

        Parameters
        ----------
        vectors : `numpy.ndarray` of shape ``(M, N)``
            Measured vectors of length ``N`` for ``M`` positions.
        errors : `numpy.ndarray` of shape ``(M, N)``
            Errors in the measured vectors.
        masks : `numpy.ndarray` of shape ``(M, N)``
            Non-zero entries should be masked in the fit. If a boolean array,
            we'll mask entries where this is ``True``.
        fiberIdList : iterable of `int` of length ``M``
            Fibers being fit.
        pfsConfig : `pfs.datamodel.PfsConfig`
            Top-end configuration, for getting the fiber centers.

        Returns
        -------
        fit : `SomePersistable`
            Function fit to the data.
        """
        centers = pfsConfig.extractCenters(fiberIdList)
        return self.fit(vectors, errors, masks, centers)
```

(continues on next page)

[1] Early versions of this module might simply fit the vectors as a function of focal plane position, but one can imagine more sophisticated versions that might use the distance between the actual and intended fiber position to weight the fit, or even as a variable in the fit.

(continued from previous page)

```
    def apply(self, fit, fiberIdList, pfsConfig):
        """Apply the fit to fibers

        Parameters
        ----------
        fit : `SomePersistable`
            Function fit to the data.
        fiberIdList : iterable of `int` of length ``M``
            Fibers being fit.
        pfsConfig : `pfs.datamodel.PfsConfig`
            Top-end configuration, for getting the fiber centers.

        Returns
        -------
        result : `numpy.ndarray` of shape ``(M, N)``
            Function fit to the data.
        """
        centers = pfsConfig.extractCenters(fiberIdList)
        return func(centers)
```

PSF MEASUREMENT

PSF measurement is part of `reduceExposure`. The goal of this module is to construct a PSF model for each of the arms in an exposure, which will be used for the *subtractSky2d* module.

The module will operate on all arms of the same kind within an exposure (e.g., the red arms from each of the spectrographs); this allows modelling of quantities over the entire focal plane. It will be provided a list of butler data references (allowing it to load any persisted parameters used in the PSF fitting) and a list of images for the exposure. It should return a PSF for each of the exposures.

Here is an example definition:

```
class MeasurePsfTask(lsst.pipe.base.Task):
    def run(self, sensorRefList, exposureList):
        """Measure the PSF for an exposure over the entire spectrograph

        We operate on the entire spectrograph in case there are parameters
        that are shared between spectrographs.

        Parameters
        ----------
        sensorRefList : iterable of `lsst.daf.persistence.ButlerDataRef`
            List of data references for each sensor in an exposure.
        exposureList : iterable of `lsst.afw.image.Exposure`
            List of images of each sensor in an exposure.

        Returns
        -------
        psfList : `list` of PSFs (type TBD)
            List of point-spread functions.
        """
        ...
        return pfsList
```

CHAPTER

SIX

FLUX REFERENCE FITTING

Flux reference fitting is the main algorithmic content of `calculateReferenceFlux`. The goal is to fit a physical spectrum to an observed spectrum; this allows the subsequent flux calibration.

The module will be provided with the spectrum to be fit[1], and should return a reference spectrum.

Here is an example definition:

```
class FitReferenceTask(lsst.pipe.base.Task):
    def run(self, spectrum):
        """Fit a physical spectrum to an observed spectrum

        Parameters
        ----------
        spectrum : `pfs.datamodel.PfsSpectrum`
            Spectrum to fit.

        Returns
        -------
        spectrum : `pfs.datamodel.PfsReference`
            Reference spectrum.
        """
        ...
        return spectrum
```

[1] It may also need, either in the constructor or the `run` method, a data butler for loading the model spectra; but if the model spectra is small, we could simply place them in one of the git repositories (e.g., `obs_pfs`).

CHAPTER

SEVEN

MEASURE FLUX CALIBRATION

Flux calibration measurement is the main algorithmic content of `fluxCalibrate`. The goal is to measure and apply the calibration from counts on the detector to physical flux units.

The module will be provided with the arm-merged spectra and reference spectra for the flux calibration fibers. A `PfsConfig` will also be provided, from which can be obtained the position of the target on the sky or on the focal plane, and the intended position on the focal plane. The module should return a persistable object that represents the fit.

In addition to the usual `run` method, the module's `Task` should also implement an `apply` method that takes the result from the `run` method and applies it to spectra. Like the `run` method, this also requires a `PfsConfig` to get the required information on fiber positions.

Here is an example definition:

```
class MeasureFluxCalibrationTask(lsst.pipe.base.Task):
    def run(self, merged, references, pfsConfig):
        """Measure the flux calibration

        Parameters
        ----------
        merged : `pfs.datamodel.drp.PfsMerged`
            Arm-merged spectra.
        references : `dict` mapping `int` to `pfs.datamodel.PfsSimpleSpectrum`
            Reference spectra, indexed by fiber identifier.
        pfsConfig : `pfs.datamodel.PfsConfig`
            Top-end configuration, for getting fiber positions.

        Returns
        -------
        calib : `SomePersistable`
            Flux calibration.
        """
        ...
        return calib

    def apply(self, merged, pfsConfig, calib):
        """Apply the flux calibration

        Parameters
        ----------
        merged : `pfs.datamodel.drp.PfsMerged`
            Arm-merged spectra.
        pfsConfig : `pfs.datamodel.PfsConfig`
            Top-end configuration, for getting fiber positions.
```

(continues on next page)

(continued from previous page)

```
        calib : `SomePersistable`
            Flux calibration.

        Returns
        -------
        results : `list` of `pfs.datamodel.PfsObject`
            Flux-calibrated object spectra.
        """
        ...
        return results
```

CHAPTER

EIGHT

SPECTRA COMBINATION

The combination of spectra is part of both `mergeArms` and `coaddSpectra`: in `mergeArms`, we coadd observations of a target taken in the same exposure that have only a small wavelength overlap (in the dichroics); while in `coaddSpectra`, we coadd observations of a target taken in multiple exposures with substantial wavelength overlap.

Because we're supporting two different use cases with the same code, we need two separate interfaces. The usual `run` method will be used to support the combination of spectra from the same exposure. It shall be provided a list of spectra, a list of keywords for determining the identity of the combined spectra, and the class to hold the target spectra. It shall return the combined spectra. An additional method, `runSingle` will be used to support the combination of a single spectrum from multiple exposures. This method shall be provided a list of spectra, a list of the appropriate fiber for the object in each of the spectra, the class to hold the target spectra, the identity of the target and list of observations[1]. It shall return the combined spectrum. Both these methods should call an additional method that does the actual combination.

Here is an example definition:

```
class CombineTask(lsst.pipe.base.Task):
    """Combine spectra

    This can be done in order to merge arms (where different sensors have
    recorded measurements for the same wavelength, e.g., due to the use of a
    dichroic), or to combine spectra from different exposures. Both currently
    use the same simplistic placeholder algorithm.

    Note
    ----
    This involves resampling in wavelength.
    """
    def run(self, spectraList, identityKeys, SpectraClass):
        """Combine all spectra from the same exposure

        All spectra should have the same fibers, so we simply iterate over the
        fibers, combining each spectrum from that fiber.

        Parameters
        ----------
        spectraList : iterable of `pfs.datamodel.PfsSpectra`
            List of spectra to coadd.
        identityKeys : iterable of `str`
            Keys to select from the input spectra's ``identity`` for the
            combined spectra's ``identity``.
        SpectraClass : `type`, subclass of `pfs.datamodel.PfsSpectra`
            Class to use to hold result.
```

(continues on next page)

[1] This may also need be provided the flux calibration and one-dimensional sky subtraction as well.

(continued from previous page)

```
        Returns
        -------
        result : ``SpectraClass``
            Combined spectra.
        """
        ...
        return SpectraClass(...)

    def runSingle(self, spectraList, fiberId, SpectrumClass, target, observations):
        """Combine a single spectrum from a list of spectra

        Parameters
        ----------
        spectraList : iterable of `pfs.datamodel.PfsSpectra`
            List of spectra that each contains the spectrum to coadd.
        fiberId : iterable of `int`
            The fiber identifier for each of the spectra that specifies which
            spectrum is to be combined.
        SpectrumClass : `type`, subclass of `pfs.datamodel.PfsSpectrum`
            Class to use to hold result.
        target : `pfs.datamodel.TargetData`
            Target of the observations.
        observations : iterable of `pfs.datamodel.TargetObservations`
            List of observations of the target.

        Returns
        -------
        result : ``SpectrumClass``
            Combined spectrum.
        """
        ...
        return SpectrumClass(...)
```