Adapting the LSST Stack

Release 0.1

Aug 15, 2019

Contents

The aim of this documentation is to explain how to adapt the LSST software stack to process data from other telescope+detector combinations – hereafter referred to as a "camera".

The documentation takes the form of a tutorial based on the author's trial-and-error experience of adapting the LSST stack to a wide field camera. As such, while it will lead you through the steps needed to set up your own camera, you will have to adjust a wide range of parameters to suit your camera before it will produce reliable results. Also note that any description of the stack's packages are based on the author's own limited understanding which is, in turn, based on (currently) very limited official documentation for the LSST stack.

Accompanying this tutorial is a github repository, obs_necam (pronounced "any cam"), which contains templates of the scripts and files needed to adapt the LSST stack to an alternate camera. I recommend you refer to the contents of that repository while following this guide; indeed, feel free to fork and edit obs_necam as you progress.

Important Note August 2019: This tutorial is based on the Generation 2 ("Gen 2") stack. Currently, the stack is undergoing some major changes while transitioning to the next generation, Gen 3. While some of this tutorial will remain relevant for Gen 3, large parts will change. As such, while reading through this tutorial will introduce you to the broad concepts of developing your own obs package, we strongly recommend, if you can, that you hold off developing your own obs package until the stack has transitioned to Gen 3 with v.19 (due early 2020).

Finally, it is important to note that this documentation is in no way affiliated with nor endorsed by the LSST organisation.

CHAPTER 1

Table of contents

1.1 Introduction

A major component the LSST project is the development of a software stack that will process the raw data from the raft of CCDs that form the LSST's detector. The ultimate end product of the stack will be fully calibrated images of the entire southern sky and database containing the properties of the billions of astronomical sources contained within those images.

While the telescope itself will not see first light until the end of 2019, it is important that the software needed to process the first observations is in a reasonable state of readiness prior to that time. As such, considerable work has already been carried-out in developing this software - known collectively as the LSST software stack. This stack is freely available here on github on an open source licence.

Despite the stack being primarily developed for the LSST camera, since most astronomical imaging systems fundamentally similar (i.e., detectors consisting of one or more CCDs plus a set of filters), there is no reason why it cannot be adapted to work with other cameras, especially those carrying out wide-field astronomical surveys. Indeed, it is already being used as the primary data analysis pipeline for the Subaru Telescope's wide-field camera, HyperSuprimeCam.

1.1.1 Why adapt the LSST stack for another camera?

As software already exists to process astronomical imaging data, it is reasonable to ask why we should bother adapting and using the LSST stack to analyse data from other cameras. Here are my reasons, but you may have your own:

- it leverages the significant experience and expertise of the people developing the stack for the processing of our data;
- Speed: while the higher level tasks are performed with Python, the stack does all its ‘‘heavy lifting” using much faster pre-compiled C++ code. In addition, a number of the stack's top-level tasks can be parallelised, providing further speed-ups compared to other pipelines;
- the end stack will include many features that are not normally or easily implemented in other pipelines (e.g., forced photometry, multi-epoch stacking). Although some of these are still under development, adapting the stack to other cameras now means they can be exploited as and when they become available;

- the LSST stack is still under active development; by adapting the stack for other cameras, we can test it and provide feedback to improve it prior to and during the LSST's operations.

1.1.2 The reason for this guide

While developing the processing pipeline of a new wide field camera, some colleagues and I decided to explore the prospect of using the LSST software stack for this task. We soon discovered that to achieve this goal, we would need to develop a set of Python scripts and configuration files (hereafter, referred to as an ‘‘obs_package’‘) that tells the stack how to read and process our data. We also discovered, however, that while obs_packages exist for other cameras, there was very little documentation describing how they were developed.

Cue many weeks of reverse-engineering the stack and the available obs_packages of other cameras (primarily, obs_sdss and obs_subaru) and a trial-and-error process of producing an our own obs_package. We like to think of this guide as what we wished we had at the start of this process.

1.1.3 What this guide is

This guide will explain how to develop a *basic* obs_package that will allow the stack to process data from an alternate camera. It covers:

- a brief description of how to install the stack,
- a section on how to ‘‘ingest” images into a format and filestructure that the stack can understand.
- a description of how to process the raw images, including bias subtraction, flat fielding etc., to produce a final catalogue of detected sources.

The goal of the guide is simply to achieve these tasks without the stack crashing.

As I played no role in developing the LSST stack, there are some parts of the obs_package that I don't fully understand. I just know that I had to include them in order to achieve the end goal.

1.1.4 What this guide isn't

Simply following this guide will not produce scientifically valid images or catalogues. There are a huge number of parameters and calibrations that need to be set, tweaked and re-tweaked depending on your data. For example, I cannot tell you the what numbers to set for the dimensions of your detector, nor its overscan region, let alone its gain, pixel size or where to set the detection threshold etc. etc. While the stack may not crash with your data using the numbers I include in the examples, the output will certainly be garbage.

1.2 Installing the LSST software stack

To use this guide you will need to install the LSST software stack. Instructions on how to do this have been written by members of the LSST project. These instructions are available at: https://pipelines.lsst.io/install.

As you will be developing your own science pipeline, you will need to perform the lsstsw installation. Once you have done this you must also test your installation by performing the demo. Finally, I recommend that you run the `lsstsw/bin/setup.sh` script each time you open a new shell by adding the command to you .bashrc file:

```
. path/to/lsstsw/bin/setup.py
```

1.2.1 A note on the EUPS versioning system

The LSST stack consists of many packages that are currently being edited by more than one person on a daily basis. It easy to conceive that an edit in one package may not be compatible with a simultaneous edit by another person in a different package. To make sure this doesn't cause problems, the LSST stack uses the Extended Unix Product System (EUPS) to enable you to specify which version of each package to use. That way, if someone makes an edit in package A that isn't compatible with your edit of package B, you can still test your edits of B using a previous version of package A.

The reason this is important for this guide is that before you can use a package that it already available in the installed stack, for example, obs_subaru, you must set it up by issuing

```
setup obs_subaru
```

Of course, the obs_package you will develop using this guide is *not* already available in the stack. Therefore, before you can use it you must `declare` it to EUPS. Instructions explaining how to do this are provided later in the tutorial, once your obs_package is at a suitable level of readiness to be declared, setup, and used.

The official manual for the EUPS system is extensive and far more involved than what you will need to set up your own obs_package. Instead, if you wish to learn more about the EUPS system and syntax, I recommend you take a look at the EUPS tutorial at: https://developer.lsst.io/build-ci/eups_tutorial.html.

1.3 A camera's "obs" package

1.3.1 The EUPS table

To enable the stack to access your camera's obs_package, the EUPS system needs to know about it. To facilitate this, you need to declare it to the EUPS, then set it up (more on this later). When you do this, the EUPS looks for a table file in the `obs_<package>/v1/ups` directory you have made, named `obs_<package>.table` (e.g., `obs_necam.table`). This table file tells the EUPS what other packages in the stack your obs_package depends on, and thus also need setting up. It also modifies your paths to include the directories in which your scripts are contained.

To create your table file, within the `ups` directory open a file named `obs_necam.table` in an editor and add the following lines:

```
setupRequired(pipe_tasks)
envPrepend(PYTHONPATH, ${PRODUCT_DIR}/python)
```

This tells the EUPS that when your obs_package is set up, it also needs to set up the `pipe_tasks` package. The `pipe_tasks` package contains most (but not all - we'll add more later) of the scripts you'll call to process your data. Thankfully, you don't then need to figure out all the packages that `pipe_tasks` depend on; the ups table file for the `pipe_tasks` includes all the packages that *it* depends on to work, and so on and so forth. As such, you only need to list the very top-level packages you'll be using within your table file (similar to how apt-get or HomeBrew takes care of dependencies when installing new packages).

1.3.1.1 Declaring your obs_package

So that the EUPS knows where to find your ups table, you need to declare its location within the filesystem to the EUPS. You also need to declare the version of the package. This is done using the `eups declare` command, for example:

```
eups declare obs_necam v1 -r $stack/obs_necam/v1
```

Here, I have named the packaged `obs_necam` and given it the version `v1`. The input following `-r` tells the EUPS the path to your obs_package.

You can check whether EUPS is aware of your obs_package by issuing:

```
eups list
```

and checking whether it appears in the resulting list.

Once a package has been declared, it is saved within the EUPS database and does not need to be declared in any future sessions.

While your obs_package has been declared, it has not been set up. This means that, while EUPS is "aware" of it, the LSST stack cannot yet access its contents. To explain why this is the case, we need to consider package versions.

1.3.1.2 Setting up package versions

Imagine a situation where your obs_package is working, but you want to add a new feature. You don't want to edit the working version in case you break it, so you make a new directory: `obs_<package>/v2/` and copy over the contents of `obs_<package>/v1/`. You make your edits to add the new feature and declare it to EUPS:

```
eups declare obs_necam v2 -r $stack/obs_necam/v2
```

But, how does EUPS know which version to use? To tell it, you need to set up a declared version. To use your `v1` to analyse some data, you would issue, for example:

```
setup obs_necam v1
```

then, to switch to `v2` to test your new features, you would issue:

```
setup obs_necam v2
```

This makes keeping separate versions and switching between them quick and easy.

If you wish, you can tell the EUPS the version it should default to by telling it which is the current version with the following command:

```
eups declare -t current obs_necam v1
```

Unlike with declare (which you only need to do once), you must _setup_ your obs_package every time you start a new session. Of course, you can automate this by adding the `setup` command to your `.bashrc` file.

While your obs_package doesn't contain any scripts yet, since you included the `pipe_tasks` package in your ups table, you can check whether it has been set up correctly by testing whether you can call the pipe_tasks commands. To do this, try to issue:

```
setup obs_necam v1
processCcd.py
```

at the command line. `processCcd.py` is a `pipe_task` that (among other thnigs) calibrates and detects sources on your raw images. At this stage, `pipe_tasks` does not have sufficient information to run properly and should instead give you some information about various configurable parameters. However, if you get the following:

```
bash: processCcd.py: command not found
```

then it means that your obs_package has not been set up correctly. If this is the case, try declaring your obs_package again. If that still doesn't work, I suggest you start over with a clean obs_package.

1.3.2 The obs mapper

When you execute a task, such as `processCcd.py` to perform basic data reduction and source detection, the first thing it will do is look for the "mapper" script. This python script tells the task what obs_package to use, where to find the file structure of the data, where information on the physical properties of the detector is stored, how to read the data, etc. As such, the mapper script is one of the most important components of a camera's obs_package.

1.3.2.1 Telling an LSST task the location of the mapper file

The location of the mapper script is provided to a task via a file named (reasonably enough) `_mapper`, which is contained within the topmost level of your input data directory. For example, if your telescopes raw data is held in:

```
/path/to/my/data/raw/Night1/
```

then you may wish to put `_mapper` in the `data/` directory. In that instance, the input directory given to the LSST task would then be `/path/to/my/data/`.

To enable the LSST stack to work with our `obs_necam` package, `_mapper` would need to contain a single line entry:

```
lsst.obs.necam.necamMapper.NecamMapper
```

and make sure that the single line is followed with a carriage return.

1.3.2.2 The mapper script - Part 1

Because it plays a central role in the way an LSST task accesses data, an obs_package's mapper script can become rather extensive. As such, in this tutorial we will build up the mapper script in separate stages, returning to it as the obs_package develops. In this first part, we will use the mapper script to tell the executed LSST task where it can find a configuration file containing a set of instructions and parameters that tell the task what to do.

The part of the mapper script described here can be found in the obs_necam GitHub repository (https://github.com/jrmullaney/obs_necam) and is called `necamMapper.py` (the filename must be the same as the last-but-one of the period-separated strings in the `_mapper` file, described above, but with the addition of `.py` at the end). At the beginning of the script a number of LSST modules are imported. I won't describe these modules here; it makes more sense if they are described as we encounter them in the script. After these imports, a python class is defined called `NecamMapper` that inherits the `CameraMapper` class (note that `NecamMapper` is the last of the period-separated strings in the `_mapper` file), which is followed by a line that defines the `packageName`:

```python
class NecamMapper(CameraMapper):
        packageName = 'obs_necam'
```

This is the first time the task has been explicitly told (via `packageName = 'obs_necam'`) which package to use to access our data. As you'd expect, this *must* match an obs_package that has been setup in the eups system, and in almost all cases will be the current obs_package.

Now the task you have executed knows the `packageName`, it can look for a corresponding configuration file, which is described next.

1.3.3 Task config files

When you execute an task, such as processCcd, it first goes to the obs mapper script (introduced on the previous page), then looks for an associated configuration (hereafter, config) file within the obs_package. All config files are contained within the `config` directory of the obs_package (e.g., `obs_necam/v1/config/`) and are named after their associated tasks; for example, the config file for the processCcd task has the filename `processCcd.py`.

The config file tells the executed task what processes it should carry-out and provides it with parameter values. For example, to configure the source detection component of the processCcd task, the config file may instruct the task to measure the PSF and perform source deblending, while also providing the flux cut used to select sources to define the PSF. In this specific case, the config file plays a similar role as the .sex configuration file used by SourceExtractor.

Each LSST task has its own set of configuration parameters that can be set in the config file. If you execute a particular task at the command prompt and with the option `--show config`, e.g.:

```
processCcd.py . --show config
```

then rather than executing the task, it will instead spit-out a list of all the configurable parameters and their default values for that task, together with short (mostly single-line) descriptions of what each parameter does. If you have already set up obs_necam you can run the above command and get a list of the thousands of parameters you can set for processCcd.

Once you have used `--show config` to see what parameters can be set for a given task, then changing their values from the default is as straightforward as putting, for example, the following line

```
config.charImage.repair.cosmicray.nCrPixelMax = 1000000
```

in the task's respective config file.

Since each task requires a different config file, we'll edit the config files as required when we start working with different tasks. Before that, however, your obs_package needs to tell the LSST stack how to organise your input and output data. It does this via the policy file, which I'll describe next.

1.3.4 The policy file

One of the most fundamental jobs of an obs_package is to provide executed LSST tasks with the locations of data files within the local filesystem. These data files can include input raw images, as well as output processed images and data tables. The locations of these files are held within `policy` files, which are contained within the `policy` directory in the obs_package.

There is a policy file in the policy directory of obs_necam; it is called `NecamMapper.yaml`. Before the bulk of the file that describes the local filesystem, there are a couple of lines which tell the stack whether there is a registry of calibration files and where to look for a physical description of the camera. We'll cover those in more detail later, but for now we'll just add the following two lines to the policy file:

```
needCalibRegistry: true
camera: "../camera"
```

Next come a number of text blocks that describe the locations and contents of the stack's various input and output files. Surrounding groups of these blocks are classifiers called things like `exposures`, `calibrations`, `datasets`, etc. Considering the first block within the `exposures` classifier:

```
raw:
        python:       "lsst.afw.image.DecoratedImageU"
        persistable:  "DecoratedImageU"
        template:     "SCI/%(dateObs)s%(filter)s/Sci_%(frameId)04d.fts"
```

The first line `raw:` provides the stack with a shortname by which this type of file is identified withing the stack. The first two lines (labelled `python` and `persistable`) tell the stack what type of data is contained within the file; for example, whether it is an image file (with or without a header: `DecoratedImage` vs. `Image`, respectively) containing integer or float values (`DecoratedImageU` vs. `DecoratedImageF`, respectively), or a data table etc. (N.B., I've tried looking for an online list of all possible file types, but to no avail. I'll post it here if I ever do find it).

The next line `template: ...` describes the location of the file within the filesystem *once it has been ingested/processed by the stack* - the policy file does not include a reference to how your data is stored prior to ingestion (see the Ingest section for a description of the ingest process). The `template` line includes variables in the form of `%(<value>)s`. The purpose of the variables is to allow the desired file to be set either at run-time, or by the stack referring to a database (referred-to as a registry). For example, if you wanted to process the science file located at `SCI/2016-01-01/V/Sci_0001.fts`, you would refer to it at run-time with the ID:

```
dateObs=2016-01-01 filter=V frameId=1
```

Note the variable format descriptor after the closing bracket (e.g., the `s` in `%(<value>)s`), which tells the stack the format of each ID in the filepath. Here, `s` refers to string, whereas `04d` means a number four characters long, padded with 0's if necessary.

We'll keep adding to the policy file as we develop the obs_package as we progress through the guide.

1.3.4.1 The mapper script - Part 2

With our policy file now created (although far from complete), we need to tell our obs_package how to find it. We do this via the mapper script (in `python/lsst/obs/necam/necamMapper.py`).

After declaring our `packageName` within the mapper script, we start an `__init__` function (for now, we'll skip over the `MakeRawVisitInfoClass` variable in `necamMapper.py` since it is not needed just yet), then declare a variable that tells the mapper the name of our policy file:

```
def __init__(self, inputPolicy=None, **kwargs):

        #Declare the policy file:
        policyFile = Policy.defaultPolicyFile(self.packageName, "NecamMapper.yaml",
↪"policy")
        policy =Policy(policyFile)
```

This ensures that the stack knows where to find the filename conventions. Recall, however, that the policy file also tells the stack where to find information about the physical properties of the camera (via `camera: "../camera"`), which we consider next.

1.3.5 The camera files

To be able to process your data, the stack needs to know a number of things about the physical properties of your camera. For example, it needs to know how many pixels are on the CCD(s), whether it has an overscan region that needs trimming, what the saturation limit is (so it can mask saturated sources), etc. This information is provided to the stack by the files that reside in directory declared in your policy file (via `camera: "../camera"`).

I consider the camera files to be the most complex component of the obs_package, and there are many components within them that I don't fully understand. I have, however, been able to get the stack working with my camera's data by emulating the camera files in other obs_packages published in the LSST stack's github repositories.

The camera files consist of two main components:

- A python script called `camera.py` in which a large number of parameters are defined.
- A fits table containing a values that describe the physical properties of your camera's CCD(s).

Wide-area astronomical cameras often consist of multiple CCDs, and each one must have an associated fits table. Furthermore, a single CCD may consist of two or more sections, each driven by a different amp. The parameters describing each section are contained within separate rows within the fits table.

To create the fits table(s) describing the CCDs, I have written a python script called `buildDetector.py` and put it in the camera directory of obs_necam repository. Feel free to use this script to generate your own fits tables for your CCDs.

The camera files contained within obs_necam are set to work with the simulated data that we will process throughout later in this guide. Clearly, you'll need to adjust a wide number of parameters to suit your own camera. Since there are so many parameters to set within the camera files (including within `buildDetector.py`), and since there are a number that I don't fully understand myself, I haven't written an exhaustive list of them here. Instead, I refer you to the comments contained within the `camera.py` and `buildDetector.py` scripts.

With our obs_package now containing all the files that it needs (if not yet complete), we can now start running our first LSST stack task which will *ingest* our data.

Adapting the LSST stack to a new camera involves writing your own obs_package. I like to think of an obs_package as an interface between the rest of the LSST stack and the data from the camara. Contained within an obs_package is a set of Python scripts and configuration files that tells the rest of the stack information such as:

- the properties of your detector (e.g., dimensions, overscan region) and its filters,
- the file structure where your data is held,
- how to remove instrument signatures from your science frames (also known as bias subtraction, dark correction, flat fielding etc.)
- what tasks you'd like the stack to do, such as source detection, deblending, astrometric correction, photometric calibration etc.

Provided you only want to use the stack in its current form (i.e., you don't want to add new modules to the stack proper to add extra functionalities) then making your own obs_package is the *only* thing you will need to do to enable the stack to process the data from your camera. Having said that, developing your own obs_package is not a trivial task.

If you have already installed the LSST stack, you will note that it already contains a number of obs_packages; for example, `obs_sdss`, `obs_cfht` and, the most developed and most useful as a reference, `obs_subaru`. You can have a look around and familiarise yourself with the contents of these packages, either within your own install or in the LSST github repository (which is my preference, though bear in mind that the github repository is constantly being updated, so it won't be long until your installed stack is different to that on github).

You will notice that the various obs_packages already installed are just directories. So, to create your own obs_package, the first think you will need to do is `mkdir` a new directory named accordingly; I use `obs_necam` throughout. In the following, I assume that the `$stack` variable points to the directory containing installed LSST stack packages (e.g., in bash shell: `export stack=~/Workstuff/lsstsw/stack/Linux64/`):

```
cd $stack
mkdir -p obs_necam/v1
```

Here, the `-p` option causes the parent directory (i.e., `obs_necam`) to be created as well. Here, I have created the `v1` to contain the first version of the obs_package.

Next, within `obs_necam/v1`, you will need to `mkdir` five other directories called:

```
mkdir camera
mkdir config
mkdir policy
mkdir ups
mkdir -p python/lsst/obs/necam
```

The next few pages will describe in detail the files that these directories must contain to process your camera's data. Here, I just provide a brief description of the contents of each directory:

- **camera:** Files containing information that describe the properties of your camera, such as its dimensions, gain etc.
- **config:** Configuration files that tell the various stack process that access your data how to behave.
- **policy:** Files describing the file structure and type of input and output data (e.g., image, table etc).
- **ups:** A file telling the eups system what other packages need to be set up to use this obs_package.
- **python/lsst/obs/necam:** Various Python scripts that perform tasks such as instrument signature removal.

Before populating these directories with the files described above, you will need to add some `__init__.py` files to the `python/lsst` and `python/lsst/obs` directories:

```
cd $stack/obs_necam/v1/python/lsst/
echo 'import pkgutil, lsstimport' > __init__.py
echo '__path__ = pkgutil.extend_path(__path__, __name__)' >> __init__.py
```

Note the >> in the last line, which appends to the file. Next, do the same in `$stack/python/lsst/obs` . If you make a mistake in the above commands, edit the file using your favourite editor (vim, emacs, VS Code, etc.).

1.4 Ingesting: Declaring file locations and types

Before the stack can process your data, it first needs to *ingest* it. Ingesting is performed by the `ingestImages.py` task, the execution of which will be described later. Ingesting achieves two main goals:

- it copies/moves/links (depending on your preference) your raw data to the location in your filesystem specified within your policy file;
- it places entries within a database that contain selected information about the observations (such as date of observation, filter, etc.)

Since we have already set up the policy file to tell the stack where to place our raw input data within our filesystem, `ingestImages.py` can achieve its first goal. To achieve its second goal, however, `ingestImages.py` needs to know what values must be held within the database in order to uniquely identify a given observation. This information is provided to `ingestImages.py` by the obs_package via its own dedicated config file, which resides in the `config` directory and is called `ingest.py`.

1.4.1 The ingest config file

The ingest config tells the ingest task what information it needs to put into the data and where it can find this data (and how to parse it, where mecessary).

The first few lines in the ingest config file imports and retargets a module contained within your `python/lsst/obs/necam` directory that tells ingest how to *parse* the various values it eventually puts into the database (more on parsing later).

Next is a *translation* block

```
config.parse.translation = {'dataType':'IMGTYPE',
                            'expTime':'EXPTIME',
                            'frameId':'RUN-ID',
                            'filter':'FILTER',
                            'field':'OBJECT'}
```

which tells ingest which header keywords to look up in your raw data files to find the values that need to go into the database. For example, the exposure time of the observation is stored in the "expTime" column in the database, but is associated with the "EXPTIME" keyword in your raw file's fits header.

The translation block works well for values that are already stored in the appropriate format within the fits header of your raw data files. However, in many cases this will not be the case. For example, the date in the fits header may be in YYYY-MM-DDTHH:MM:SS.FFFF ISO format, yet the database only accepts YYYY-MM-DD format. To accomodate such cases requires some degree of *parsing*, whereby a small python function takes the value from the header and parses it into the correct format. These parsing functions are contained within the `python/lsst/obs/necam/ingest.py` file as part of a `NecamParseTask` class and have appropriate names such as `translateDate`. You'll need to write your own parsing functions where necessary, so feel free to use necams's translation functions as a guide. Having done that, you must tell the ingest config file which translator functions to use for each of the database entries that require it. This is done by the following block in the ingest config file

```
config.parse.translators = {'dateObs':'translateDate',
                            'taiObs':'translateDate',
                            'visit':'translateVisit',
                            'ccd':'translateCcd'}
```

which, again, you may wish to edit to suit your own requirements.

The ingest task now has everything it needs to know to extract and parse information from the fits headers of the input data. Next, it needs to know the column names this information should be stored under within the database and the format of the data (e.g., string, integer, etc.). It also needs to know which columns describe a `raw_visit` (it seems, however, this is may be a bit of a relic; see this LSST community forum thread). Finally, it also needs to know what combination of data uniquely identifies an exposure – for example, lots of exposures from a raft of CCDs may share a single visit number, but it may be the case that the combination of a visit number *and* a CCD number uniquely identifies an exposure. These three pieces of information are provided by the following three blocks in the ingest config file

```
config.register.columns = {'frameId':'text',
                           'visit':'int',
                           'ccd':'int',
                           'filter':'text',
                           'dataType':'text',
                           'expTime':'double',
                           'dateObs':'text',
                           'taiObs':'text',
                           'field':'text' }

config.register.visit = ['visit', 'ccd', 'filter', 'dateObs', 'taiObs']

config.register.unique = ['visit', 'ccd']
```

While you can add any information you want into the registry, there are some that are strongly recommended. For example, the stack will find it difficult/impossible to process data if it doesn't know what type of data each frame represents (e.g., dark, flat, science, etc. frame; given by the ‘‘dataType" column in the above example). Exposure times are needed to calculate count rates, and you should also include any parameters that you expect to select frames on (i.e., if you expect to select frames based on data range, or filter, then you'll need to include those).

This completes all the information that the `ingestImages.py` task needs to parse the necessary data from the headers of the raw images and ingest it into its database of images.

1.5 Processing a single science frame

1.5.1 Combining and ingesting master calibration frames

With the raw exposure now ingested, we can now start to process them. The first thing that needs to be done is to generate master calibration frames by combining the individual bias, dark, and flat frames. These master calibration frames will be used to correct the raw science frames for so-called ‘‘instrument signatures’‘.

As part of the data ingestion step, the LSST stack

1.5.2 Processing your data

Among the first and most fundamental tasks that you will want the LSST stack to do is to process your imaging data to remove instrument signatures (e.g., overscan removal, bias subtraction, flat-fielding) and perform source detection. Both these tasks are performed by `processCcd.py`.

Like all of the data processing tasks in the LSST stack, `processCcd.py` resides in the `pipe_tasks` repository. This means that, provided you have (a) setup the eups table in `obs_necam` following the instructions in The EUPS table section and (b) executed `setup obs_necam` command, then `processCcd.py` will be callable from the command line. To use `processCcd.py`, we need to tell it where the input data is, and where it should put any output data:

With the stack now installed and your data files ingested, we can start to process data. In this tutorial, we will use the stack to undertake the following processing steps:

- combine a set of calibration frames to create master bias, flat, and dark frames;
- correct a science frame using the master calibration frames;
- calibrate and characterise the science frame, which involves sky subtraction, PSF modelling, astrometric correction, and photometric calibration;
- source detection and measurement.

The first item on the above list is performed by the `constructBias.py`, `constructFlat.py` and `constructDark.py` pipe tasks, which we will look at next. Following ingestion of the master calibration frames, the remaining items are all performed by a single task: `processCcd.py`. This latter task will likely require more configuring than any other pipe task you will encounter.

Before continuing, it is worth pointing out that pipe tasks run on a single core, and will thus process your data serially. Most pipe tasks, however, have multi-core equivalents known as `pipe_drivers`, which farm out the pipe tasks to multiple cores. Configuring a pipe driver is similar to configuring a pipe tasks, so I’ve included a brief section on how to configure and run the equivalent pipe driver after describing its associated pipe task.